МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Инновационная образовательная программа
«Наукоемкие технологии и экономика инноваций»
Московского физико-технического института
(государственного университета)
на 2006–2007 годы

Е.М. Лаврищева, В.А. Петрухин

МЕТОДЫ И СРЕДСТВА ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Методы программирования

Методы доказательства и тестирования

Методы интеграции, преобразования и изменения

Инженерия приложений и предметных областей

Инженерия требований и повторного использования

Методы управления проектом, рисками, конфигурацией

Москва 2007

Лаврищева Е.М. ,   Петрухин В.А.

МЕТОДЫ И СРЕДСТВА ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Методы проектирования программных систем

Методы доказательства и тестирования

Методы интеграции, преобразования и изменений

Инженерия приложений и предметных областей

Инженерия требований и ПИК

Методы управления проектом, рисками, конфигурацией

Средства и инструменты

Москва 2007

УДК 004.4'2(075.8)
ББК 32.973-018.2я73
Л13

Р е ц е н з е н т ы:
Кафедра кибернетики Московского инженерно-физического института
(государственного университета)
(Зав. кафедрой кандидат технических наук *С.В. Синицын*)

Доктор технических наук *Б.А. Позин*

Лаврищева Е.М., Петрухин В.А.
Л13 Методы и средства инженерии программного обеспечения:
Учебное пособие. – М.:МФТИ, 2007. – 415 с.

Систематически изложено ядро знаний программной инженерии – SWEBOK, методы и средства программирования, их теория, практика, а также рекомендации стандартов программной инженерии на разработку программного обеспечения. Рассмотрены базовые понятия методов прикладного и теоретического проектирования, дан анализ методов доказательства, верификации и тестирования программ, методов интеграции и взаимодействия разноязыковых программ, преобразования программ и данных для разных сред и платформ. Определены основы инженерной дисциплины – управление проектом, риском, качеством. Описана инженерия приложений и предметной области на основе повторного использования компонентов, рассмотрены подходы и методы их аннотации для накопления в репозитариях и оценки их применимости в новых программных проектах. Дано краткое описание набора современных инструментов.

УДК 004.4'2(075.8)
ББК 32.973-018.2я73

ISBN 5-7417-0200-7

© Лаврищева Е.М., Петрухин В.А., 2007
© Московский физико-технический институт
(государственный университет), 2007

ПРЕДИСЛОВИЕ....... 7
ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ....... 9
ВСТУПЛЕНИЕ....... 10
Тема 1....... 13
ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ И ЖИЗНЕННЫЙ ЦИКЛ ПО....... 13
1. Анализ и характеристика областей знаний SWEBOK....... 17
1.1. Основы программных требований (Software Requirements)....... 17
1. 2. Проектирование ПО (Software design)....... 19
1.3. Конструирование ПО (Software Construction)....... 21
1.4. Тестирование ПО (Software Testing)....... 23
1.5. Сопровождение ПО (Software maintenance)....... 25
1. 6. Управление конфигурацией ПО (Software Configuration Management–SCM)....... 27
1.7. Управление инженерией ПО (Software Engineering Management)....... 28
1. 8. Процесс инженерии ПО (Software Engineering Process)....... 30
1. 9. Методы и средства инженерии ПО (Software Engineering Tools and Methods)....... 32
1. 10. Качество ПО (Software Quality)....... 33
2. Введение в жизненный цикл ПО стандарта ISO\IEC 12207 и связь его с ядром знаний программной инженерией SWEBOK....... 35
3. Обучение специальности – программная инженерия....... 40
3.1. Анализ системы знаний у ИТ–специалистов....... 42
3.2. Подходы к обучению программной инженерии....... 44
3.3. Анализ результатов дистанционного обучения....... 46
Контрольные вопросы и задания....... 47
Литература к теме 1:....... 47
Тема 2....... 49
МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ДЛЯ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ....... 49
2.1. Каскадная модель ЖЦ....... 50
2.2. Инкрементная модель ЖЦ....... 51
2.3. Спиральная модель....... 53
2.4.Эволюционная модель ЖЦ....... 54
2.5. Стандартизованная модель системы....... 55
2.6. Сопоставление модели ЖЦ стандарта ISO/IEC 12207 и областей –процессов SWEBOK....... 56
2.6.1. Характеристика процессов стандарта....... 57
2.6.2. Характеристика модели процессов в ядре SWEBOK....... 58
Контрольные вопросы и задания....... 60
Литература к теме 2....... 61
Тема 3....... 62
МЕТОДЫ ОПРЕДЕЛЕНИЯ ТРЕБОВАНИЙ В ПРОГРАММНОЙ ИНЖЕНЕРИИ....... 62
3.1 Определение понятий и видов требований....... 62
Виды требований....... 62
3.1.2. Анализ и сбор требований....... 63
3.1.3. Инженерия требований ПО....... 66
3.1.4. Верификация и формализация требований....... 67
3.2. Объектно-ориентированная инженерия требований....... 68
3.2.1. Метод инженерии требований А. Джекобсона....... 71
3.2.2. Модель анализа требований. Определение объектов....... 76
3.3. Классификация требований....... 78
3.4. Трассирование требований....... 79
Контрольные вопросы и задания....... 81
Литература к теме 3....... 82
Тема 4....... 83
МЕТОДЫ АНАЛИЗА И ПОСТРОЕНИЯ МОДЕЛЕЙ ПрО....... 83
4.1. Объектно–ориентированные методы анализа и построения моделей ПрО....... 83
4.1.1. Основные понятия анализа ПрО....... 84
4.1.2. Метод анализа и построения моделей С.Шлаер и С.Меллора....... 86
4.1.2.1. Информационная модель....... 86
4.1.2.2. Модель состояний....... 89
4.1.3. Модель процессов....... 91
4.2. Методы проектирования архитектуры ПО....... 94
4.2.1. Стандартный подход к проектированию системы....... 94

4.2.2. Общесистемный подход к проектированию архитектуры системы ... 97
4.2.3. Техническое проектирование ... 101
Контрольные вопросы и задания ... 102
Литература к теме 4. ... 102
Тема 5 ... 104
МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ ... 104
5.1. Методы систематического программирования ... 104
5.1.1. Структурный подход ... 104
5.1.2. Объектно–ориентированный метод проектирования ... 107
5.1.3. Метод моделирования UML ... 109
5.1.4. Компонентный подход к проектированию ... 112
5.1.4.1. Типы компонентных структур ... 115
5.1.4.2. Методология компонентной разработки систем ... 117
5.1.5. Аспектно–ориентированное программирование ... 118
5.1.6. Генерирующее (порождающее) программирование ... 121
5.1.7 Агентное программирование ... 125
5.2. Методы теоретического программирования ... 127
5.2.1. Алгебраическое программирование (АП) ... 128
5. 2.2. Экспликативное программирование (ЭП) ... 130
5.2.3. Алгоритмика программ ... 131
5.2.4. Формальные методы ... 134
Контрольные вопросы и задания ... 137
Литература к теме 5. ... 138
Тема 6. ... 140
ИНЖЕНЕРИЯ ПРИЛОЖЕНИЙ И ИНЖЕНЕРИЯ ПРЕДМЕТНОЙ ОБЛАСТИ ... 140
Введение ... 140
6. 1. Инженерия ПИК ... 141
6.2. Спецификация ПИК ... 145
6.3. Репозитарий компонентов ... 148
6.4. Описание интерфейса объектов-компонентов в распределенной среде ... 150
6.5. Инженерия приложений и предметной области ... 152
6.6. Инженерия оценивания стоимости реализации ПрО из компонентов ... 154
Литература к теме 6. ... 155
Тема 7 ... 157
МЕТОДЫ ВЕРИФИКАЦИИ И ТЕСТИРОВАНИЯ ПРОГРАММ И СИСТЕМ ... 157
7. 1. Методы доказательства программ ... 157
7.1.1. Методы доказательства правильности программ ... 158
7.1.1.1. Общая характеристика формальных методов доказательства ... 158
7.1.1.2 Модель формального доказательства конкретности программы ... 160
7.1.2. Техника символьного выполнения ... 163
7.1.3. Методы просмотра структуры программы ... 164
7.1.2. Верификация и аттестация программ ... 167
7.1. 3. Методы верификации объектно–ориентированных программ ... 168
7.2. Методы тестирования программ ... 169
7. 2. 1. Статические методы тестирования ... 169
7.2.2. Динамические методы тестирования ... 170
7.2.3. Функциональное тестирование ... 172
7. 3. Организационные аспекты процесса тестирования ... 173
7.3.1. Организация подготовки тестов ... 180
7.3.2. Команда тестировщиков ... 182
7.3.3. Организация процесса тестирования ... 184
Контрольные вопросы и задания ... 185
Литература к теме 7 ... 185
Тема 8 ... 187
МЕТОДЫ ИНТЕГРАЦИИ, ПРЕОБРАЗОВАНИЯ И ИЗМЕНЕНИЯ ... 187
КОМПОНЕНТОВ И ДАННЫХ ... 187
8.1. Методы интеграции (композиции) компонентов ... 187
8.2. Методы преобразования программ и данных ... 189
8.2.1. Парадигма преобразования данных ... 190
8.2.2. Формальное описание данных в ЯП и их преобразование ... 191
8.2.3. Средства стандарта ISO/IEC 11404–1996 для преобразования данных ... 192
8.3. Преобразование данных БД и замена БД ... 194
8.3.1. Основные этапы преобразования данных в БД ... 195

8.3.2. Унифицированные файлы для передачи данных между разными БД 196
8. 4. Методы внесения изменений в компоненты и в ПС 198
8.4.1. Реинженерия программных систем 200
8.4.2. Рефакторинг компонентов 201
8.4.3. Реверсная инжеиерия 202
Контрольные вопросы и задания 204
Литература к теме 8. 204
Тема 9 206
МОДЕЛИ КАЧЕСТВА И НАДЕЖНОСТИ В ПРОГРАММНОЙ ИНЖЕНЕРИИ 206
9.1. Модель качества ПО 206
9.1.1. Метрики качества программного обеспечения 211
9.1.2. Стандартный метод оценки значений показателей качества 214
9.1.3. Управление качеством ПС 216
9.2. Модели оценки надежности 219
9.1.1. Основные понятия в проблематике надежности ПС 220
9.2.2. Классификация моделей надежности 221
9.2.3. Модели надежности Марковского и Пуассоновского типов 224
Контрольные вопросы и задания 228
Литература к теме 9. 228
Тема 10 230
МЕТОДЫ УПРАВЛЕНИЯ ПРОЕКТОМ, РИСКОМ И КОНФИГУРАЦИЕЙ 230
10.1. Методы управления проектами 230
10.1.1. Методы управления программным проектом 231
10.1.1.1. Метод критического пути СРМ 231
10.1.1.2. Метод анализа и оценки PERT 232
10.1.2. Планирование проекта 234
10.1.3. Организационные аспекты управления в проекте 237
10.1.4. Оценивание проекта 241
10.2. Методы управление рисками 243
10.3. Управление конфигурацией программной системы 246
10.3.1. Управление конфигурацией 247
10.3.2. Планирование УК 249
10.3.3. Идентификация элементов конфигурации 250
10.3.4. Управление версиями 250
10.3.5. Конфигурационный контроль 251
10.3.6. Учет статуса конфигурации 252
10.3.7. Конфигурационный аудит 253
Контрольные вопросы и задания 253
Литература к теме 10. 254
Тема 11 255
СРЕДСТВА И ИНСТРУМЕНТЫ В ПРОГРАММНОЙ ИНЖЕНЕРИИ 255
11.1. Языковые средства описания компонентов и методов интеграции 255
11.1.1. Средства ЯП JAVA для описания и интеграции компонентов 256
11.1.2. Типы компонентов и средства их интеграции в JAVA 258
11.1.2. Система CORBA и средства описания объектов и компонентов 261
11.1.2.1. Язык описания интерфейсов в системе CORBA 263
11.1.2.2. Язык описания интерфейсов объектов 264
11.1.2.3. Интегратор объектов – брокер объектных запросов 266
11.1.3. Средства унифицированного процесса RUP 267
11.2. Энциклопедия инструментов создания ПС из объектов и компонентов 271
11.3. Средства и методы разработки архитектуры MSF 274
Контрольные вопросы и задания 279
Литература к теме 11. 279
ПРИЛОЖЕНИЕ 1 280
Словарь терминов программной инженерии 280
ПРИЛОЖЕНИЕ 2 286
Характеристика стандартов разработки автоматизированных систем (АС) 286
2.1 Характеристика стандарта ГОСТ 34.601–90 для разработки АС 286
2.2. Стандарт разработки документации на АС – ГОСТ 34.201–89 288
ПРИЛОЖЕНИЕ 3 291
Жизненный цикл компонентной разработки ПС 291
3.1. Этап разработки требований 291
3.2. Этап анализа поведения ПС 292

3.3. Этап спецификации интерфейсов и взаимодействия компонентов ... 293
3.4. Этап интеграции ... 293
3.5. Этап развертывания компонентов системы ... 296
3.6. Этап сопровождения ... 297
ПРИЛОЖЕНИЕ 4 ... 301
Кодекс этики программной инженерии ... 301
Литература ... 301
ПРИЛОЖЕНИЕ 5 ... 302
Стандарты программной инженерии ... 302

ПРЕДИСЛОВИЕ

Цель данного учебника – представить методы и средства программной инженерии (Software engineering) в систематизированном виде для их применения на процессах проектирования, тестирования и оценки качества программных систем.

Современные университетские курсы по информатике предусматривают обучение основам программирования, объектно-ориентированному подходу, UML–моделированию, параллельному программирования и др. Больше уделяется внимание современным языкам программирования (C++, JAVA) для современных компьютеров. В результате студенты получают подготовку по этим методам и средствам и недостаточные знания по инженерии проектирования и управления проектами, качеству, конфигурации и соответствующим стандартам.

В некоторых университетах проводятся лекционные курсы по теория алгоритмов, автоматов, математической логике, дискретной математике и другим формальным дисциплинам. Эти курсы основываются на математических дисциплинах (логика, алгебра, комбинаторика) и способствуют развитию математического мышления при проведении анализе предметной области, осмыслении постановок задач и разработке программ для получения на компьютере математического результата.

Производство и использование компьютерных программ в настоящее время является массовой деятельностью, разработкой программ занимаются почти семь миллионов человек, а их используют в своей профессиональной деятельности по специальности десятки миллионов. В связи с постоянно возрастающими объемами программных разработок требуется готовить кадровый потенциал, способный решать проблемы создания новых программных продуктов на инженерной основе, используя накопленный запас знаний в области программирования и управления системами.

Сложившуюся структуру и содержание подготовки специалистов надо расширить методами управления, планирования и регулирования работ, адаптируя их к условиям коллективной разработки программных систем с гарантированным качеством. Предпосылками этого является становление новой специальности, получившей название программной инженерии или инженерии программного обеспечения (Software Engineering), впитавшей в себя накопленный запас знаний в практике и теории программирования за последние десятилетия, а также обогатившейся инженерной дисциплиной выполнения процессов ЖЦ программного обеспечения.

В связи с этим предметом обучения современных студентов, будущих разработчиков программного обеспечения, менеджеров программных проектов, тестировщиков, верификаторов, контролеров качества и др. должны стать не только теоретические и прикладные методы проектирования, а и инженерные методы управления коллективом, планирования и оценивания качества выполняемых работ и укладывания в заданные сроки и стоимость проекта.

Данный учебник посвящен систематическому описанию накопленных знаний в области программирования, отражает аспекты теории и практики программирования. Для применения в лекционных курсах окажется полезным представленное в учебнике изложение современных методов программирования и обеспечения правильности программ, а также инженерии программирования (планирование, управление и оценка продуктов и процессов), сформировавшейся под влиянием развития программной инженерии – SE (Software Engineering) и стандартизации процессов программирования.

В нем также представлены современные средства и инструменты поддержки процессов создания проектов (Project Management, Rational Rose, MSF, RUP, CORBA, DCOM и др.).

Авторы надеются, что учебник поднимет уровень знаний разработчиков ПО, поможет им овладеть не только представленными знаниями в области теории и практики программирования, но инженерным программированием, включая методы планирования, управления и оценивания результатов своей деятельности.

Материал учебника апробирован при чтении лекций в Киевском национальном университете имени Тараса Шевченко (1985-1997 гг.) и в МФТИ (2000-2006 гг.), а также на международных конференциях и семинарах.

Авторы

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ACM – Association for Computing Machinery
API – Application Progpam Interface
ER – Entity – Relationship
IEC – International Electrotechnical Commission
ISO – International System Organization
IDL – Interface Definition Language
ISO – International Standard Organization
OM – объектная модель
ООП– объектно-ориентированный подход
OMA –Object Management Architecture
OOM – объектно-ориентированная методология
СОД – система обработки данных
ОС – операционная система
ORB – Object Request Broker
ПС – программная система
CMM – Capability Maturity Model
COM – Component Object Model
CORBA – Common Object Request Broker Architecture
COP – Component-Oriented Programming
CBSE – Component-Based Software Engineering
СРР – структура разбивки работ в проекте
CPM – Critical Path Method (метод критического пути)
PERT – Program Evaluation and Review Technique (анализ за методом PERT)
PMBOK – Project Management Body of Knowledge (ядро знаний в области управления проектами)
SQA – Software Quality Assurance (гарантирование качества)
V&V – Verification and Validation (верификация и валидация)
ПИК – повторного использования компонент
ПИ – программная инженерия
ПО – программное обеспечение
РП – распределенное приложение
ЖЦ – жизненный цикл
SWEBOK – SoftWare Engineering of Body Knowledge
RMI – Remote Method Invocation

ВСТУПЛЕНИЕ

Термину «программная инженерия» (Software engineering) уже более 30 лет. К моменту его появления компьютерные программы проникли во все сферы человеческой деятельности, а их разработка стала массовым занятием. Практически нет ни одной х сферы человеческой деятельности (медицина, экономика, коммерция, промышленность и т.д.), где бы не применялись компьютерные программы.

Примерно каждые 10 лет происходит смена языков программирования и ОС. Это приводит к необходимости изменять ранее изготовленные и функционирующие программы применительно к новым языкам и ОС. Например, преобразованием Фортран и Кобол программ в современные языки (С, JAVA и др.) занимаются огромные коллективы программистов из третьих стран и СНГ.

Эффективность разработчиков в зависимости от квалификации колеблется в отношении 20:200, отсюда требуется повышать уровень их знаний. На сегодня ядро стабильных знаний по программной инженерии составляет 75% от тех знаний, что используются в практической программисткой деятельности.

Эти условия поставили перед теоретиками и умудренными опытом программистами разработку новых инженерных методов создания и управления процессами проектирования, а перед прикладниками создание стандартов, регламентируюших эти процессы.

Знания разработчиков ПО отличаются большим разнообразием, являются не согласованными и разнородными, ориентированными на разные предметные области, поэтому мировая компьютерная общественность пришла к необходимости систематизировать знания в области программной инженерии, создав ядро знаний SWEBOK (Software Engineering Body Knowledge).

Программная инженрия (Software Engineering) является отраслью компьютерной науки, изучает вопросы построения программ для компьютеров, отражает закономерности развития в ней знаний, обобщает накопленный опыт программирования в виде комплексов общих знаний и правил регламентации инженерной деятельности разработчиков ПО.

Как инженерная дисциплина охватывает все аспекты создания ПО, начиная от разработки требований до создания, сопровождения и снятия с эксплуатации ПО, а также оценку трудозатрат, производительности и качества.

Инженерия изменений программных продуктов выполняется методами реинжинерии, реверсной инженерии (перепрограммирование) и рефакторинга программных компонентов и интерфейсов. Применение готовых продуктов (модулей, программ, систем и т.п.) в новых разработках привело к их инженерии, при которой компоненты становятся коммерческим продуктом, приносят прибыль разработчикам и сокращают затраты при создании новых систем за счет их накопления в репозитариях или электронных библиотеках.

Программостроение больших программных проектов становится инженерным по своей сути. В нем, кроме программистов, участвуют:

– менеджеры, которые планируют и управляют проектом, отслеживают сроки и затраты;
– инженеры службы хранения готовых компонентов;
– технологи, которые определяют инженерные методы и стандарты, регламентирующие и регулирующие процесс построения программных проектов;
– тестировщики, которые проверяют правильность выполнения процессов, сбор данных при тестировании и оценку качества компонентов и системы в целом.

Инструменты поддержки разработки ПО совершили гигантский скачок в своем развитии и теперь обычной практикой стало создание ПС с использованием современных визуальных и диаграммных средств UML.

Таким образом, возникновение программной инженерии определено следующими факторами:
– появление разнообразных сложных методов анализа и моделирования ПО;
– большое количество ошибок в ПО;
– необходимость в эффективной организации работы коллективов разработчиков ПО;
– повторное использование готовых программных компонентов и высокотехнологических средств разработки и управления ПО;
– реинженерия и рефакторинга отдельных компонентов систем, их адаптация к постоянно изменяющимся условиям и средам функционирования.

Материал данного учебного курса по программной инженерии разработан с учетом методов и средств ядра знаний SWEBOK. Он предназначен студентам, изучающим компьютерное дело, менеджерам коллективов программных проектов, желающих повысить уровень своей квалификации по планированию и управлению, а также профессионалам различных предметных областей.

Данный учебник состоит из тем по методам и средствам программной инженерии, которые необходимы для овладения профессией по инженерному созданию компьютерных систем. В конце каждой темы приводятся и контрольные вопросы и задания, а также используемая литература.

Тема 1. Введение в программную инженерию и процессы жизненного цикла. Дано определение программной инженерии, ее место в инженерной деятельности специалистов при создании компьютерных систем и общее описание десяти областей знаний профессионального ядра знаний SWEBOK.

Тема 2. Модели жизненного цикла для разработки программных систем. Описываются основные модели ЖЦ, которые используются в практике проектирования программных систем. Дана характеристика фундаментальных моделей ЖЦ (водопадной, спиральной, инкрементной, эволюционной) и стандартной модели.

Тема 3. Методы определения требований в программной инженерии. Приведены методы и инженерия требований к системе. Рассмотрен процесс сбора, накопления и спецификации требований. Дана классификация требований и характеристика функциональных и нефункциональных требований.

Тема 4. Методы анализа и построения моделей ПрО. Приведены методы анализа предметной области и построения моделей. Рассмотрены объектно–ориентированные и стандартизованные, традиционные методы проектирования архитектуры системы.

Тема 5. Методы проектирования программных систем. Представлено описание базовых основ методов систематического (структурного, компонентного, аспектно–ориентированного и др.) и теоретического (алгебраического, композиционного и алгеброалгоритмичекого) программирования для ознакомления студентов с теорией и практикой проектирования ПС.

Тема 6. Инженерия приложений и инженерия предметной области. Излагаются современные тенденции и направления развития инженерии приложений в плане производства одиночных ПС из ПИК и инженерии ПрО с многоразовым применением используемых решений для семейства ПС.

Тема 7. Методы верификации и тестирования программ и систем. Посвящена описанию методов проверки правильности программ: формальным методам доказательства, основанных на аксиомах и утверждениях, верификации и тестирования ПС на этапах ЖЦ.

Тема 8. Методы интеграции, преобразования и изменения компонентов и данных. Рассмотрены основы интеграции и преобразования программ и данных, а также методы изменения (реинженерия, реверсная инженерия и рефакторинга) компонентов и систем.

Тема 9. Модели качества и надежности в программной инженерии. Посвящена представлению моделей качества ПС, метрикам и методам достижения и измерения качества ПС. Рассмотрен основной показатель качества – надежность, дано описание математических моделей надежности и способов их применения на практике.

Тема 10. Методы управления проектом, риском и конфигурацией. Проведен анализ и дано описание инженерии программирования, принципов и методов планирования и управления программным проектом, рисками и формированием версий ПС.

Тема 11. Средства и инструменты программной инженерии. Дан обзор современных средств программирования и характеристика широко используемых CASE-средств (Project Management, Rational Rose, MSF, RUP, CORBA, DCOM и др.), при объектно–ориентированном проектировании ПС.

Приложение 1. Набор основных терминов, используемых в программной инженерии.
Приложение 2. Характеристика стандартов разработки автоматизированных систем.
Приложение 3. Жизненный цикл компонентной разработки ПС.
Приложение 4. Кодекс этики в программной инженерии.
Приложение 5. Стандарты в программной инженерии.

Тема 1
ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ И ЖИЗНЕННЫЙ ЦИКЛ ПО

Разработка и использование компьютерных программ в настоящее время стало массовой деятельностью. Более семи миллионов человек занимаются их разработкой, а сотни миллионов активно используют их в своей профессиональной деятельности [1]. Практически нет ни одной сферы деятельности (экономика, медицина, бизнес, коммерция, промышленность и т.д.), где бы программное обеспечение (ПО) не использовалось для автоматизации и улучшения этой деятельности. Спрос на ПО постоянно увеличивается, его сложность растет, а ошибки в ПО остаются.

Знания разработчиков ПО отличаются большим разнообразием, и, как правило, они являются не полными, не согласованными и разнородными, ориентированными на реализацию разных предметных областей, начиная от ОС и кончая современными прикладными бизнес-системами. И самое главное, что знания в процессе инженерной деятельности постепенно уточняются, видоизменяются и пополняются, за которыми не поспевают программисты.

Примерно каждые 10 лет происходит смена языков программирования и операционных сред для описания и функционирования программ. Это предполагает изменение ранее изготовленных и функционирующих программ в новые языки и операционные среды. На это тратятся огромные людские и финансовые ресурсы. Так, при изменении формата даты (2000 год) в программах и микросхемах на десятках млн. компьютеров участвовало более 2 млн. программистов, а затраты составили сотни млн. долларов. Переделка (реинженерия) ранее созданных прикладных Фортран программ в новые языки (С, Java и др.) и условия функционирования требует больших капиталовложений и привлечения программистов из третьих стран и СНГ.

В связи с появлением огромного количества новых видов и типов персональных компьютеров, а также Фреймворков с усовершенствованными ОС и средствами интероперабельности в программистском мире возникли большие сложности, связанные с адаптацией и переводом ранее действующих готовых прикладных систем и программ. Наиболее важные и необходимые из них преобразуются, тиражируются и поступают на рынок, как продукт для продажи (сидиромы, кассеты и т.п.).

Процесс формирования знаний в информатике и Computer Science послужил толчком для формирования самого претенциозного проекта 90-х годов в истории развития вычислительной техники – японский проект ЭВМ 5 поколения, который должен был сделать переворот в области проектирования и программирования ПО [2]. В этом проекте предполагалось, что в суперкомпьютеры будут заложены огромные интеллектуальные знания в виде экспертных систем, баз знаний, систем вывода новых знаний и т.п. Общение с такими суперкомпьютерами будет осуществляться голосом, образами, речью, а из постановок задач будут выводиться готовые решения без написания соответствующих программ. Этот проект не был реализован, так как знания специалистов–разработчиков таких компьютеров не были достаточно полными и совершенными и, кроме того, уровень интеллектуализации знаний на то время не был достаточно глубоко разработанным, чтобы идеи японского проекта воплотить в реальную аппаратуру и встроить в нее высоко интеллектуальные системы автоматизации постановок задач.

Проектирование и создание новых компьютерных систем, в том числе из готовых компонентов (Reusing) и систем, теоретически и практически осуществляется с учетом современных возможностей платформ и распределенных сред, в которых компоненты распределяются по разным узлам сети и взаимодействуют между собой через сетевые протоколы. Появились новые методы и подходы к разработке ПО: структурный, объектно-ориентированный, компонентный, аспектный, визуальный – UML, агентно-ориентированный, сервисный и др. [3-13].

Разработано огромное количество разнообразных инструментальных средств поддержки процесса проектирования ПО и методов оценки качества, производительности, стоимости и т.п. Процесс разработки ПО и методы оценки продукта, процессов ЖЦ стандартизованы (ISO/IEC 12207 [14], 15504 [15], ISO 9126[16-18] и др.). Все это способствует повышению уровня проектирования, тестирования, прогнозирования надежности и оценки качества ПО.

Вместе с тем, новый программный проект разрабатывается 1-2 года, а эволюционирует 6-7лет. На его сопровождение тратиться 61% затрат против 39% на его разработку. Эффективность разработчиков в зависимости от квалификации колеблется в отношении 20:200, отсюда требуется повышать уровень знаний разработчиков ПО. На сегодня ядро стабильных знаний по программной инженерии составляет 75% от тех знаний, что используются в практической программисткой деятельности.

В связи с этим мировое компьютерное сообщество пришло к необходимости систематизации накопленных знаний и общие из них зафиксировать в виде ядер знаний (Body of Knowledge – BOK) для разных областей информатики [19]. Для создания ядра знаний ПО был создан международный комитет при американском объединении компьютерных специалистов ACM (Association for Computing Machinery) и институте инженеров по электронике и электротехнике IEEE Computer Society. В комитет вошли специалисты мирового уровня в области информатики и разработки ПО, которые внесли свой опыт и знания, а также систематизировали накопленные разнородные знания и определили (1999г., 2001г., 2004г.) ядро профессиональных знаний SWEBOK (Software Engineering Body Knowledge) программной инженерии [20], как основы проектирования ПО. Ядро включает сумму знаний, распределенную по 10 специализированным областям, которые отражают отдельные процессы проектирования ЖЦ ПО и методы их поддержки.

Программная инженерия (Software Engineering) является отраслью Computer science, изучает вопросы построения компьютерных программ, отражает закономерности ее развития, обобщает опыт программирования в виде комплекса общих знаний и правил регламентации инженерной деятельности разработчиков ПО. В этом определении важно рассмотреть два основных аспекта.

1. Инженерная дисциплина, по которой инженеры применяя теоретические идеи, методы и средства для разработки ПО, проводят создание ПО, согласно стандартов, регламентирующих процессы проектирования и разработки.
2. Аспекты создания ПО. Программная инженерия рассматривает такие аспекты ПО как управление проектом ПО и разработка средств, методов и теорий, необходимых для создания качественных программных систем. Эта инженерная дисциплина предоставляет всю необходимую информацию и стандарты для выбора наиболее подходящего метода проектирования практических задач. Не исключается и творческий неформальный подход к созданию ПО.

Как инженерная дисциплина, она охватывает все аспекты создания ПО, начиная от формирования требований до создания, сопровождения и снятия с эксплуатации ПО, а также включает инженерные методы оценки трудозатрат, стоимости, производительности и качества. Т.е. речь идет именно об инженерной деятельности в программировании, поскольку ее сущность близка к определению инженерной деятельности в толковом словаре [2]:

1) инженерия есть применение научных результатов, что позволяет получать пользу от свойств материалов и источников энергии;
2) как деятельность по созданию машин для предоставления полезных услуг.

В программной инженерии, инженеры – это специалисты, выполняющие практические работы по реализации программ с применением теории, методов и средств компьютерной науки. Компьютерная наука (computer science) охватывает теорию и методы построения вычислительных и программных систем, тогда как программная инженерия рассматривает вопросы практического построения ПО. Знание компьютерной науки необходимо специалистам в области программного обеспечения так же, как знание физики – инженерам-электронщикам. Если для решения конкретных задач программирования не существует подходящих методов или теории, инженеры применяют свои знания, накопленные ими в процессе конкретных разработок ПО, а также используя опыт работы на соответствующих инструментальных программных средствах. Кроме того, инженеры должны работать в условиях заключенных контрактов и выполнять задачи с учетом этих условий.

В отличие от науки, целью которой есть получение знаний, для инженерии знание – это способ получения некоторой пользы. Как говорил известный специалист в области программой техники Ф.Брукс], «ученый строит, чтобы научиться, инженер учится, чтобы строить».

Таким образом, разработка программных систем можно считать инженерной деятельностью, имеющей значительные отличия от традиционной инженерии, в которой:
– ветви инженерии имеют высокую степень специализации, а у программной инженерии специализация заметна только в довольно узких применениях (например, операционные системы, трансляторы);
– объекты хорошо определены и манипуляции с ними происходят в узком контексте типичных проектных решений и деталей, которые отвечают типовым требованиям заказчиков и касаются отдельных деталей, а не общих вопросов, тогда как у программной инженерии подобная типизация отсутствует;
– отдельные готовые решения классифицированы и каталогизированы, а в программной инженерии каждая новая разработка - это новая проблема, в которой довольно тяжело рассмотреть аналогию с ранее разработанными системами.

Указанные отличия требуют проведения организационных и технических работ для превращения ее в специальность. В настоящее время мировая компьютерная общественность объединились в профессиональные комитеты и проводят такие работы: разработка ядра знаний SWEBOK, этического кодекса программиста [13], учебных курсов подготовки соответствующих специалистов, обучение специальности, сертификация специалистов в области программной инженерии и др.

Следующим шагом деятельности этих организаций и комитетов является создание общей компьютерной программы обучения – Curricula 2001 [8]. В ней содержатся рекомендации по структуре и преподаванию 15 учебных курсов по информатике (дискретные системы, программирование, теория сложности, ОС и др.) в том числе и по программной инженерии, как дисциплины, изучающей теорию, знания и практику эффективного построения ПО на всех этапах ЖЦ, которым соответствуют области знаний в SWEBOK.

В разработке больших программных проектов, кроме программистов, принимают участие:
– менеджеры, которые планируют и руководят проектом, отслеживают сроки и затраты;
– инженеры службы хранения готовых компонентов в библиотеках и репозитариях;
– технологи, которые определяют инженерные методы и стандарты, регламентирующие и регулирующие процесс реализации проекта;
– тестировщики (контролеры), которые проверяют правильность выполнения процесса проектирования и продуктов процессов, на основе собранных данных проводят измерения разных характеристик качества, включая оценку надежности ПО.

Таким образом, возникновение программной инженерии как дисциплины разработки ПО определено следующими важными факторами:
– накопленным значительным объемом интеллектуальных знаний в области создания ПО;
– появлением новых разнообразных методов анализа, моделирования и проектирования ПО;
– необходимостью совершенствования методов обнаружения ошибок в ПО;
– потребностями эффективной организации коллективов разработчиков ПО и оценки их деятельности;
– использованием готовых программных компонентов, высоко технологических средств и инструментов разработки ПО;
– реинженерией компонентов и систем для их адаптации к новым изменяющимся условиям сред и сетей.

Программная инженерия, как инженерная дисциплина, делает главный акцент на повышение качества и производительности ПО за счет применения новых и усовершенствованных: методов проектирования ПО; готовых компонентов и методов их генерации; методов эволюции ПО; методов верификации и тестирования ПО; инструментальных средств поддержки; методов управления проектами, методов оценки качества, производительности, стоимости и т.п.; стандартизации процессов разработки ПО (ISO/IEC 12207, ISO/IEC 15504, ISO 9126 и др.), регламентирующих этапы ЖЦ; подходов к оценке продуктов и процессов.

В данном разделе темы лекций дается систематическое изложение следующих взаимосвязанных аспектов в инженерии проектировании ПО:

– теоретический и интеллектуальный базис (методы, принципы, средства и методологии и др.) проектирования, представленный в ядре SWEBOK, способствующий созданию высококачественных программных продуктов и удовлетворяющих заданным заказчиком функциональных и нефункциональных требований;
– связь теоретических аспектов программирования с готовыми стандартами в области программной инженерии, которые регламентируют деятельность специалистов-

разработчиков ПО и организационные мероприятия по выполнению процессов верификации, валидации, тестирования, метрического анализа и оценки промежуточных и конечного результатов этой деятельности;
– концепции обучения разработчиков ПО методам и средствам программной инженерии, стандартным процессам ЖЦ с целью научить их методологии проектирования ПО, использующей знания в области программной инженерии и положений современных стандартов, регламентирующих процессы разработки, планирования, управления программным проектом, качеством, рисками и ресурсами проекта.

1. Анализ и характеристика областей знаний SWEBOK

Ядра знаний SWEBOK [20] является основополагающим документом, отображает мнение многих зарубежных и отечественных специалистов в области программной инженерии [3-13] и согласуется с современными регламентированными процессами ЖЦ ПО стандарта ISO/IEC 12207. В этом ядре знаний содержится описание 10 областей, каждая из которых представлена согласно принятой всеми участниками создания этого ядра общей схемы описания, включающей определение понятийного аппарата, методов и средств, а также инструментов поддержки инженерной деятельности. Описание каждой области вносит определенный запас знаний, который должен практически использоваться на соответствующих процессах ЖЦ с учетом приведенного стандарта.

Для наглядного представления понятийного аппарата областей SWEBOK проведено условное разбиение областей (рис. 1.а, б.) на основные (пять процессов проектирования ПС) и дополнительные, организационные методы и подходы, которые отображают инженерию управления проектированием ПС (конфигурацией, проектами, качеством и т.д.). В каждой области приведены ключевые понятия, подходы и методы проектирования разных типов ПС. Данное разбиение областей на главные и вспомогательные области соответствует структуре процессов стандарта ISO/IEC 12207 (см. подраздел 2), выполнение которых определяется знаниями, содержащимися в ядре SWEBOK и изученными разработчиками ПС.
Далее приводится изложение каждой в отдельности области знаний ядра знаний SWEBOK, их назначение и роль при проектировании и реализации программных продуктов.

В некоторых разделах данной главы показана связь с положениями соответствующих стандартов, которые регламентируют и регулируют выполнение процессов проектирования ПС разных видов программных систем.

1.1. Основы программных требований (Software Requirements)

Требования – это свойства, которыми должно обладать ПО для адекватного задания функций, а также условия и ограничения на ПО, данные, среду выполнения и технику.

Требования отражают потребности людей (заказчиков, пользователей, разработчиков), заинтересованных в создании ПО. Заказчик и разработчик совместно проводят сбор требований, их анализ, пересмотр, определение необходимых ограничений и документирование. Различают требования к продукту и к процессу, а также функциональные и нефункциональные требования, системные требования. Программные требования определяют требования к процессу, ОС, режиму выполнения ПО, выбору платформы и т.п.

Функциональные требования задают назначение системы, а нефункциональные – условия выполнения ПО. Системные требования описывают требования к программной системе, состоящей из взаимосвязанных программных и аппаратных подсистем и разных приложений. Требования могут оцениваться количественно (например, количество запросов в сек., средний показатель ошибок не должен превышать 1.5% от объема вводимой информации и т.п.). Значительная часть требований относится к атрибутам качества: безотказность, надежность и др.
Область знаний «Требования к ПО (Software Requirements)» состоит из следующих разделов:
– инженерия требований (Requirement Engineering),
– выявление требований (Requirement Elicitation),
– анализ требований (Requirement Analysis),
– спецификация требований (Requirement Specification).
– проверка требований (Requirement validation),
– управление требованиями (Requirement Menegement).

Инженерия требований к ПО – это дисциплина анализа и документирования требований к ПО, которая заключается в преобразовании предложенных заказчиком требований к системе в описание требований к ПО, их спецификация и верификация. Она базируется на модели процесса определения требований, процессах актеров – действующих лиц, обеспечивающих управление и формирование требований, а также на процессах повышения качества.

Модель процесса – это схема процессов ЖЦ, которые выполняются от начала проекта и до тех пор, пока не будут определены и согласованы требования. При этом процессом может быть маркетинг и проверка осуществимости требований в данном проекте.

Управление требованиями к ПО заключается в планировании и контроле выполнения требований и проектных ресурсов в процессе разработки компонентов на этапах ЖЦ.

Качество и процесс улучшения требований – это процесс формулировки характеристик и атрибутов качества (надежность, реактивность и др.), которыми должна обладать система и ПО, методы их достижения на этапах ЖЦ и адекватности процессов работы с требованиями.

Выявление требований – это процесс извлечения информации из разных источников заказчика (договоров, материалов аналитиков по задачам и функциям системы и др.), проведения технических мероприятий (собеседований, собраний и др.) для формирования отдельных требований на разработку. Требования согласуются с заказчиком и исполнителем.

Анализ требований – процесс изучения потребностей и целей пользователей, классификация и их преобразование к требованиям системы, аппаратуре и ПО, установление и разрешение конфликтов между требованиями, определение приоритетов, границ системы и принципов взаимодействия со средой функционирования. Требования могут быть функциональные и нефункциональные, которые определяют соответственно внешние и внутренние характеристикам системы. Функциональные требования характеризуют функции системы или ее ПО, способы поведения ПО в процессе выполнения функций и методы передачи и преобразования входных данных в результаты. Нефункциональные требования определяют условия и среду выполнения функций (например, защита и доступ к БД,

секретность, взаимодействие компонентов и др.). Разработка требований и их локализация завершается на этапе проектирования архитектуры и отражается в специальном документе, по которому проводится *согласование требований* для достижения взаимопонимания между заказчиком и разработчиком.

Спецификация требований *к ПО* – процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартом качества ISO/IEC 9126-94, которые будут отрабатываться на этапах ЖЦ ПО. В спецификации требований отражается структура ПО, требования к функциям, качеству и документации, а также задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные требования, нефункциональные требования и требования к взаимодействию с другими компонентами и платформами (БД, СУБД, маршаллинг данных, сеть и др.).

Валидация (аттестация) требований - это проверка требований, изложенных в спецификации для того, чтобы убедиться, что они определяют данную систему и отслеживание источников требований. Заказчик и разработчик ПО проводят экспертизу сформированного варианта требований с тем, чтобы разработчик мог далее проводить разработку ПО. *Верификация требований* – это процесс проверки правильности спецификаций требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие стандартам. В результате проверки требований делается согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, а также возможность продолжить проектирование ПО. Одним из методов аттестации является прототипирование, т.е. быстрая отработка отдельных требований на конкретном инструменте и исследование масштабов изменения требований, измерение объема функциональности и стоимости, а также создание моделей оценки зрелости требований.

Управление требованиями – это руководство процессами формирования требований на всех этапах ЖЦ, которое включает управление изменениями и атрибутами требований, отражающими программный продукт, а также проведение мониторинга – восстановления источника требований. Неотъемлемой составляющей процесса управления является т*рассирование требований* для отслеживания правильности задания и реализации требований к системе и ПО на этапах ЖЦ и обратный процесс отслеживания от полученного продукта к требованиям.

Таким образом, приведено краткое изложение сущности определения требований к ПО, базовых понятий и процессов их формирования, подходов к их оценке на предмет установления их соответствия потребностям заказчика, а также подходов для достижения их качества.

1. 2. Проектирование ПО (Software design)

Проектирование ПО – процесс определения архитектуры, компонентов, интерфейсов, других характеристик системы и конечного результата.

Область знаний «Проектирование ПО (Software Design)» состоит из следующих разделов:
– базовые концепции проектирования ПО (Software Design Basic Concepts),
– ключевые вопросы проектирования ПО (Key Issue in Software Design),
– структура и архитектура ПО (Software Structure and Architecture),

– анализ и оценка качества проектирования ПО (Software Design Quality Analysis and Evaluation),
– нотации проектирования ПО (Software Design Notations),
– стратегия и методы проектирования ПО (Software Design Strategies and Methods).

К базовым концепциям проектирования ПО относятся процессы ЖЦ (стандарт ISO/IEC 12207), процесс проектирования архитектуры с использованием разных принципов (объектного, компонентного и др.) и техник: абстракции, декомпозиции, инкапсуляции и др. Автоматизируемая система декомпозируется на отдельные компоненты, выбираются необходимые артефакты (нотации, методы и др.) программной инженерии и строится архитектура ПО.

К ключевым вопросам проектирования ПО относятся: декомпозиция на функциональные компоненты для независимого и параллельного их выполнения, принципы распределения компонентов в среде выполнения и их взаимодействие между собой, механизмы обеспечения качества и живучести системы и др.

При проектировании структуры ПО используется архитектурный стиль проектирования, основанный на определении основных элементов структуры – подсистем, компонентов и связей между ними.

Архитектура проекта – высокоуровневое представление структуры, задаваемое с помощью паттернов, компонентов и их идентификация. Описание архитектуры содержит описание логики отдельных компонентов системы, достаточное для проведения работ по кодированию, и связей между ними. Существуют и другие виды структур, основанные на проектировании образцов, шаблонов, семействе программ и их каркасов.
Паттерн – это конструктивный элемент ПО, который задает взаимодействие объектов (компонентов) проектируемой системы, определение ролей и ответственности исполнителей. Основным языком задания этого элемента является UML.

Паттерн может быть: *структурным*, в котором определяются типовые композиции структур из объектов и классов диаграммами классов, объектов, связей и др.; *поведенческим*, определяющим схемы взаимодействия классов объектов и их поведение диаграммами активностей, взаимодействия, потоков управления и др.; *креативным*, отображающим типовые схемы распределения ролей экземпляров объектов диаграммами взаимодействия, кооперации и др.

Анализ и оценка качества проектирования ПО включает мероприятия по анализу сформулированных в требованиях атрибутов качества, оценки различных аспектов ПО – размера и структуры ПО, функций и качества проектирования с помощью формальных метрик (функционально-ориентированных, структурных и объектно-ориентированных), а также проведения качественного анализа результатов проектирования путем статического анализа, моделирования и прототипирования.

Нотации проектирования позволяют представить артефакты ПО и его структуру, а также поведение системы. Существует два типа нотаций: структурные, поведенческие и множество различных их представлений.

Структурные нотации являются графическими, они используются для представления структурных аспектов проектирования, компонентов и их взаимосвязей, элементов архитектуры и их интерфейсов. К ним относятся формальные языки спецификаций и

проектирования: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity–Relation Diagrams), IDL (Interface Description Language), классы и объекты, компоненты и классы (CRC Cards), Use Case Driven и др. Нотации включают языки описания архитектуры и интерфейса, диаграммы классов и объектов, диаграммы сущность-связь, компонентов, развертывания, а также структурные диаграммы и схемы.

Поведенческие нотации отражают динамический аспект поведения систем и их компонентов. Таким нотациям соответствуют диаграммы: Data Flow, Decision Tables, Activity, Colloboration, Pre-Post Conditions, Sequence, таблицы принятия решений, формальные языки спецификации, языки проектирования PDL и др.

Стратегия и методы проектирования ПО. Данный раздел знаний представляет различные стратегии и методы, которые используются при проектировании. К общим стратегиям относятся: снизу-вверх, сверху–вниз, абстракции, паттерны и др. Функционально–ориентированные (структурные) методы базируются на структурном анализе, структурных картах, Dataflow-диаграммах и др. Они ориентированы на идентификацию функций и их уточнение сверху–вниз, после чего проводится разработка диаграмм потоков данных и описание процессов. В обьектно–ориентированном проектировании ключевую роль играет наследование, полиморфизм и инкапсуляция, а также абстрактные структуры данных и отображение объектов [30] Подходы, ориентированные на структуры данных, базируются на методе Джексона (Jackson) [8] и используются для задания входных и выходных данных структурными диаграммами.

Компонентное проектирование ориентировано на использование и интеграцию компонентов (особенно компонентов повторного использования) и на их интерфейс, обеспечивающий взаимодействие компонентов; является базисом других видов программирования, в том числе сервисно-ориентированного, в котором группы компонентов обеспечивают функциональный сервис. К другим методам относятся: формальные, точные и трансформационные методы, а также UML для моделирования архитектурных решений с помощью диаграмм [31].

Таким образом, предложенные в данной области знаний подходы, стратегии и методы проектирования ПО, средства распределения и взаимодействия компонентов в разных средах являются основными при разработке проекта с применением разных элементов (шаблонов, сценариев, диаграмм и др.) и стилей проектирования структуры ПО, а также мероприятий по проведению анализа полученных на этапе проектирования атрибутов качества ПО.

1.3. Конструирование ПО (Software Construction)

Конструирование ПО – создание работающего ПО с привлечением методов верификации, кодирования и тестирования компонентов. К инструментам конструирования ПО отнесены языки программирования и конструирования, а также программные методы и инструментальные системы (компиляторы, СУБД, генераторы отчетов, системы управления версиями, конфигурацией, тестированием и др.). К формальным средствам описания процесса конструирования ПО, взаимосвязей между человеком и компьютером и с учетом среды окружения отнесены языки конструирования.

Область знаний «Конструирование ПО (Software Construction)» включает следующие разделы:

– снижение сложности (Reduction in Complexity),
– предупреждение отклонений от стиля (Anticipation of Diversity),
– структуризация для проверок (Structuring for Validation),
– использование внешних стандартов (Use of External Standards)

Основу данной области составляют задачи понижения сложности конструирования программного продукта, предупреждение отклонений от стиля (лингвистического, формального, визуального и др.), которое обеспечивается применением наиболее подходящих стилей конструирования, структуризация ПО и использование внешних стандартов.

Лингвистический стиль основан на использовании словесных инструкций и выражений для представлений отдельных элементов (конструкций) программ. Он используется при конструировании несложных конструкций и приводится к виду традиционных функций и процедур, логическому и функциональному их программированию и др.

Формальный стиль используется для точного, однозначного и формального определения компонентов системы. В результате его применения обеспечивается конструирование сложных систем с минимальным количеством ошибок, которые могут возникнуть в связи с неоднозначностью определений или обобщений при конструировании ПО неформальными методами.

Визуальный стиль является наиболее универсальным стилем конструирования ПО. Он позволяет разработчикам проекта представлять в наглядном виде сложные программные конструкции. Например, графический интерфейс пользователя освобождает разработчика от подбора необходимых координат и свойств объектов интерфейса. Визуальный язык проектирования UML представляет разработчику набор удобных диаграмм для задания статической и динамической структуры ПО [31].

При применении визуального стиля конструирования создается текстовое и диаграммное описание структуры ПО, которое выводится на экран дисплея не только для их рассмотрения, но и корректировки.

В процессе конструирования должны использоваться внешние стандарты ЯП (Ада 95, С++ и др.), языков описания данных (XML, SQL и др.), средств коммуникации (COM, CORBA и др.), интерфейсов компонентов (POSIX, IDL, APL) [33], сценариев UML [31] и др.
Управление конструированием базируется на моделях конструирования, планировании и внесении изменений.

Модели конструирования включают набор операций, последовательность действий и результаты. Виды моделей определяются стандартом ЖЦ, методологиями и практиками. Основные стандарты ориентированы на экстремальное программирование и RUP [32].

Планирование состоит в определении порядка создания компонентов и методов обеспечения качества. Измерение в конструировании ориентировано на количественную оценку объема кода, степени использования ПИК, вероятности появления дефектов и количественных показателей качества ПО.

Внесение изменений проводится с целью сохранения функциональной целостности системы и рефакторинга кода на основе проведенного метрического анализа необходимости проведения изменений в конструируемое ПО.

Тестирование в конструировании. Проводится две формы тестирования созданного кода – модульное и интеграционное. Виды тестирования описаны в специальной области знаний (см. ниже). При этом используются два стандарта (IEEE 829-1996 и IEEE 1008-1987) тестирование элементов ПО и документации. Обеспечение качества конструирования базируется не только на тестировании и отладке отдельных программ, а и на просмотрах, инспектировании, анализе и оценках результатов тестирования.

Таким образом, рассмотренные механизмы конструирования позволяют разработчику проекта принять решение об использовании методов конструирования или проектирования. Наиболее современным считается метод моделирования UML.

1.4. Тестирование ПО (Software Testing)

Тестирование ПО – это процесс проверки работы программы в динамике, основанный на выполнении конечного набора тестовых данных и сравнения полученных результатов с запланированными вначале.

Область знаний «Тестирование ПО (Software Testing)» включает следующие разделы:

– основные концепции и определение тестирования (Testing Basic Concepts and definitions),
– уровни тестирования (Test Levels),
– техники тестирования (Test Techniques),
– метрики тестирования (Test Related Measures),
– управление процессом тестирования (Managing the Test Process).

Основная концепция тестирования базируется на терминологии, теории и инструментах подготовки и проведения процесса тестирования ПО, а также оценке данных статистического анализа процесса тестирования. При тестировании выявляются недостатки: отказы (faults) и дефекты (defects), как причины нарушения работы системы, сбои (failures), как нежелательные ситуации, ошибки (errors), как последствия сбоев и др. Базовым понятием тестирования является тест, который выполняется в заданных условиях и на наборах данных. Тестирование считается успешным, если найден дефект или ошибка, и они устраняются. Степень тестируемости зависит от задания критериев покрытия системы тестами и вероятности появления сбоев. Данные базовые понятия зависят от уровня, видов и техник тестирования ПО.

Уровни тестирования это:
– *тестирование отдельных элементов,* которое заключается в проверке отдельных, изолированных и независимых частей ПО;
– *интеграционное тестирование,* которое ориентировано на проверку связей и способов взаимодействия (интерфейсов) компонентов друг с другом, включая компоненты, расположенные на разных архитектурных платформах распределенной среды;
– *тестирование системы* предназначено для проверки правильности функционирования системы в целом, с обнаружением отказов и дефектов в системе и их устранение. При этом контролируется выполнение сформулированных нефункциональных требований (безопасность, надежность и др.) в системе,

правильность задания и выполнения внешних интерфейсов системы со средой окружения и др.

К видам тестирования относятся:
– *функциональное тестирование*, которое заключается в проверке соответствия выполнения специфицированных функций;
– *регрессионное тестирование* – тестирование системы или ее компонентов после внесения в них изменений;
– *тестирование эффективности* – проверка производительности, пропускной способности, максимального объема данных и системных ограничений в соответствии со спецификациями требований;
– *нагрузочное (стресс) тестирование* – проверка поведения системы при максимально допустимой нагрузке или при превышении;
– *альфа и бета-тестирование* – внутреннее и внешнее тестирование системы. Альфа – без плана, бета с планом тестирования;
– *тестирование конфигурации* – проверка структуры и идентификации системы на различных наборах, а также проверку работы системы в различных конфигурациях.

К видам тестирования относятся также подходы и методы проверки поведения системы на этапе испытания ПО и приемки в соответствии с требованиями и заданными параметрами относительно состава ПО, количества и типа компьютеров, среды и ОС.

Техники тестирования бывают таких видов:
– *«белый (стеклянный) ящик»,* основанный на задании информации о структуре ПО или системе;
– *«черный ящик»,* основанный на задании тестовых наборов данных для проверки правильности работы компонентов и системы в целом без знания их структуры;
– основанные на спецификациях, анализе граничных значений, таблицах принятия решений, критериев потоков данных, статистики отказов и др.;
– основанные на использовании блок–схем, по которым строятся программы и наборы тестов для покрытия всех условий выполнения частей системы и системы в целом;
– на основе обнаруженных дефектов, условий использования, природы и особенностей приложения и др.

Управление тестированием это:
– планирование процесса тестирования (составление планов, тестов, наборов данных) и измерение показателей качества ПО;
– проведение тестирования reuse-компонентов и паттернов, как основных объектов сборки ПО;
– генерация необходимых тестовых сценариев, соответствующих среде выполнения ПО;
– верификация правильности реализации системы и валидация реализованных требований к ПО;
– сбор данных об отказах, ошибках и др. непредвиденных ситуациях при выполнении программного продукта;
– подготовка отчетов по результатам тестирования и оценка характеристик системы.

Стандарт ISO/IEC, ГОСТ 12207 не выделяет деятельность по тестированию в качестве самостоятельного процесса, а рассматривает тестирование, как необъемлемую часть ЖЦ.

Измерение результатов тестирования. Измерение, как часть планирования и разработки тестов, базируется на размере программ, их структуре и количестве обнаруженных дефектов. Метрики тестирования обеспечивают измерение процесса планирования, проектирования и тестирования; а также результатов тестирования на основе таксономии отказов и дефектов, покрытия границ тестирования, проверки потоков данных и др. Документация на тестирование включает, согласно стандарту IEEE 829-98, описание тестовых документов, их связи между собой и с процессом тестирования. Без документации по процессу тестирования, невозможно провести сертификацию продукта и оценку модели CMM1 [22]. После завершения тестирования рассматриваются вопросы стоимости и рисков, связанных с появлением сбоев и недостаточно надежной работой системы. Стоимость тестирования является одним из ограничений, на основе которого принимается решение о прекращении или продолжении тестирования.

Таким образом, данная область знаний SWEBOK представляет разработчику методы проверки правильности ПО: верификация, валидация, тестирование. Определяются типы, уровни и техники тестирования ПО, методы планирования процесса и тестовых наборов данных для прогонки ПО в режиме испытания конкретного модуля или системы в целом, а также методы измерения результатов тестирования.

1.5. Сопровождение ПО (Software maintenance)

Сопровождение ПО – совокупность действий по обеспечению работы ПО, а также по внесению изменений в случае обнаружения ошибок в процессе эксплуатации, по адаптации ПО к новой среде функционирования, а также по повышению производительности или других характеристик ПО. В связи с решением проблем 2000года сопровождение стало рассматриваться как более важный процесс, который должен строго обеспечиваться и обновляться участниками разработчиков. Новая версия системы должна решать те же задачи, иметь план переноса информации БД и учет стоимости сопровождения. Сопровождение (согласно стандартов ISO/IEC 12207 и ISO/IEC 14764) считается модификацией программного продукта в процессе эксплуатации при условии сохранения целостности продукта.

Область знаний «Сопровождение ПО (Software maintenance)» состоит из следующих описаний разделов:
– основные концепции (Basic Concepts),
– процесс сопровождения (Process Maintenance),
– ключевые вопросы сопровождения ПО (key Issue in Software Maintenance) ,
– техники сопровождения (Techniques for Maintenance).

Сопровождение рассматривается с точки зрения удовлетворения требований в данном ПО, корректности его выполнения, процессов обучения и оперативного учета процесса сопровождения.

Основные концепции включают базовые определения и терминологию, подходы к эволюции и сопровождению ПО, а также к оценке стоимости сопровождения и др.

К основным определениям относится ЖЦ ПО (стандарт ISO/IEC 12207) и документация. Эта область трактуется, как процесс выполнения, анализа необходимости модификации, оценки стоимости работ по изменению функций. Рассматриваются проблемы, связанные с увеличением сложности продукта при большом количестве изменений и преодоления этого.

Процесс сопровождения включает: модели процесса сопровождения и планирование деятельности людей, которые проводят запуск ПО, проверку правильности его выполнения и внесения в него изменений. Процесс сопровождения согласно стандарту ISO/IEC 14764 проводиться путем:

– корректировки, т.е. изменения продукта при реализации обнаруженных ошибок и нереализованных задач;
– адаптации, т.е. настройки продукта к изменившимся условиям эксплуатации или новой среды выполнения данного ПО;
– улучшения, т.е. изменения продукта для повышения производительности или уровня сопровождения;
– проверки ПО для поиска и исправления скрытых ошибок, обнаруженных при эксплуатации системы.

Ключевые вопросы сопровождения ПО. Основными из этих вопросов являются управленческие, измерительные и стоимостные. Сущность управленческих вопросов состоит в контроле ПО в процессе модификации, совершенствовании функций и недопущении снижения производительности системы. Оценка стоимости затрат на сопровождение зависит от типа ПО, квалификации персонала, платформы и др. Знание этих факторов позволяет не только их сохранить, но и уменьшить. Вопросы измерения относятся к оценке характеристик системы после ее модификации, например, после тестирования оценка качества полученного ПО.

Эволюция ПО. Известный специалист в области ПО Леман (1970г.) предложил рассматривать сопровождение как эволюционную разработку программных систем, поскольку сданная в эксплуатацию система не всегда является полностью завершенной, ее надо изменять в течение срока эксплуатации. В результате программная система становиться более сложной и плохо управляемой, возникает проблема уменьшения ее сложности. К техникам эволюции ПО относятся реинженерия, реверсная инженерия и рефакторинг.

Реинженерия – это улучшение возможностей, функций в устаревшем ПО путем его реорганизации и реструктуризации, перепрограммирования или настройка на другую платформу или среду с обеспечением удобства его сопровождения

Реверсная инженерия состоит в восстановлении спецификации (графов вызовов, потоков данных и др.) по полученному коду системы (особенно, когда в нее внесено много изменений) для наблюдения за ней на более высоком уровне. Восстанавливается идентификация программных компонентов и связей между ними для обеспечения перестройки системы к новой форме.

Рефакторинг ориентирован на улучшение структурных характеристик и качественных показателей объектно-ориентированных программ без изменения их поведения. Этот процесс реализуется путем изменения отдельных операций над текстами, интерфейсами, средой программирования и выполнения ПО, а также настройки или внесения изменений в инструментальные средства поддержки ПО. Если сохраняется форма существующей системы при изменении, то рефакторинг – один из вариантов обратной инженерии.

Таким образом: рассмотренные проблемы сопровождения позволяют специалистам узнать весь круг вопросов сопровождения, эволюции и унаследования старых программных систем.

1. 6. Управление конфигурацией ПО (Software Configuration Management– SCM)

Управление конфигурацией – дисциплина идентификации компонентов системы, определения функциональных и физических характеристик аппаратного и программного обеспечения для проведения контроля внесения изменений и трассирования конфигурации на протяжении ЖЦ. Это управление соответствует одному из вспомогательных процессов ЖЦ (ISO/IEC 12207), выполняется техническим и административным руководством проекта и заключается в контроле указанных характеристик конфигурации системы и их изменении; составления отчета о внесенных изменениях в конфигурацию и статус их реализации; проверки соответствия внесенных изменений заданным требованиям.

Конфигурация системы – состав функций, программных и физических характеристик программ или их комбинаций, аппаратного обеспечения, обозначенные в технической документации системы и реализованные в продукте.

Конфигурация ПО включает набор функциональных и физических характеристик ПО, заданных в технической документации и достигнутых в готовом продукте. Т.е это сочетание разных элементов продукта вместе с заданными процедурами сборки и отвечающие определенному назначению. Элемент конфигурации – график разработки, проектная документация, исходный и исполняемый код, библиотека компонентов, инструкции по установке системы и др.

Область знаний «Управление конфигурацией ПО» состоит из следующих разделов:

– управление процессом конфигурацией (Management of SMC Process),
– идентификация конфигурации ПО (Software Configuration Identification),
– контроль конфигурации ПО (Software Configuration Control),
– учет статуса конфигурации ПО (Software Configuration Status Accounting),
– аудит конфигурации ПО (Software Configuration Auditing),
– управление релизами (версиями) ПО и доставкой (Software Release Management and Delivery).

Управление процессом конфигурации. Это деятельность по контролю эволюции и целостности продукта при идентификации, контроле изменений и обеспечении отчетности информации, касающейся конфигурации. Включает:

– систематическое отслеживание вносимых изменений в отдельные составные части конфигурации и проведение аудита изменений и автоматизированного контроля за внесением изменений в конфигурацию системы или ПО;
– поддержка целостности конфигурации, ее аудит и обеспечение внесения изменений в один объект конфигурации, а также в связанный с ним другой объект;
– ревизия конфигурации на предмет проверки разработки необходимых программных или аппаратных элементов и согласованности версии конфигурации с требованиями;
– трассировка изменений в конфигурацию на этапах сопровождения и эксплуатации ПО.

Идентификация конфигурации ПО проводится путем выбора элемента конфигурации ПО и документирования его функциональных и физических характеристик, а также оформления технической документация на элементы конфигурации ПО.

Контроль конфигурации ПО состоит в проведении работ по координации, утверждению или отбрасыванию реализованных изменений в элементы конфигурации после формальной ее идентификации, а также оценке результатов.

Учет статуса конфигурации ПО проводится в виде комплекса мероприятий для определения уровня изменений в конфигурацию, аудита конфигурации в виде комплекса мероприятий по проверке правильности внесения изменений в конфигурацию ПО. Информация и количественные показатели накапливается в соответствующей БД и используются при управлении конфигурацией, составлении отчетности, оценке качества и выполнении других процессов ЖЦ.

Аудит конфигурации – это деятельность, которая выполняется для оценки продукта и процессов на соответствие стандартам, инструкциям, планам и процедурам. Аудит определяет степень удовлетворения элемента конфигурации заданным функциональным и физическим характеристикам системы. Различают функциональный и физический аудит конфигурации, который завершается фиксацией базовой линии продукта.

Управление релизами (версиями) ПО это: отслеживание имеющейся версии элемента конфигурации; сборка компонентов; создание новых версий системы на основе существующей путем внесения изменений в конфигурацию; согласование версии продукта с требованиями и проведенными изменениями на этапах ЖЦ; обеспечение оперативного доступа к информации относительно элементов конфигурации и системы, к которым они относятся. Управление выпуском охватывает идентификацию, упаковку и передачу элементов продукта и документации заказчику. При этом используются следующие основные понятия.

Базис (baseline) – формально обозначенный набор элементов ПО, зафиксированный на этапах ЖЦ ПО.

Библиотека ПО – контролируемая коллекция объектов ПО и документации, предназначенные для облегчения процесса разработки, использования и сопровождения ПО.

Сборка ПО – объединение корректных элементов ПО и конфигурационных данных в единую исполняемую программу.

Таким образом, описание данной области показывает, что процесс управления конфигурации является важным процессом идентификации элементов, формирования версии системы и их управления.

1.7. Управление инженерией ПО (Software Engineering Management)

Управление инженерией ПО (менеджмент) – руководство работами команды разработчиков ПО в процессе выполнения плана проекта, определение критериев и оценка процессов и продуктов проекта с использованием общих методов управления, планирования и контроля работ.

Как любое управление, менеджмент ПО предполагает планирование, координацию, измерение, контроль и отчет по процессу управления проектом; представляет собой системную, дисциплинированную и измеряемую разработку ПО. Координацию людских, финансовых и технических ресурсов при реализации задач проекта

выполняет менеджер проекта, аналогично тому, как это делается в технических проектах. В его обязанности входит соблюдение запланированных бюджетных и временных характеристик и ограничений, стандартов и сформулированных требований. Общие вопросы управления проектом содержится в ядре знаний PMBOK [19] в разделе Management Process Activities, а также в стандарте ISO/IEC 12207 – Software life cycle processes [14], где управление проектом рассматривается как дополнительный и организайионный процесс ЖЦ,

Область знаний «Управление инженерией ПО (Software Engineering Management)» состоит из следующих разделов:

– организационное управление (Organizational Management),
– управление процессом и проектом (Process/Project Management),
– измерение инженерии ПО (Software Engineering Measurement).

Организационное управление – это планирование и составление графика работ, оценка стоимости работ, подбор и управление персоналом, контроль за выполнением работ согласно принятых стандартов и планов. Главными проблемами организационного управления проектом являются: управление персоналом (обучение, мотивация и др.), коммуникации между сотрудниками и средой (сценарии, встречи, презентации и др.), а также риски (минимизация риска, техники определения риска, выбор решений по их предотвращению и др.). Для управления проектом создается такая структура коллектива, специалисты которой могут выполнить проект, они распределяются по работам и все вместе реализуют задачи проекта под руководством менеджеру проекта.

Задачам проекта сопоставляется оборудование, средства и исполнители. Проводится распределение обязанностей специалистов и их выполнение с учетом заданной стоимости и срока разработки проекта.

Процесс управления проектом включает: составление плана проекта, построение графиков работ (сетевых или временных диаграмм) исходя из имеющихся ресурсов, а также распределение персонала по работам с учетом сроков и стоимости их выполнения и др.; анализ финансовой, технической, операционной и социальной политики организации для выбора правильной стратегии выполнения плана; контроль процесса управления планами и продуктами на процессах.

Управление продуктом заключается в уточнении требований и проверки (валидацию) их на соответствие, в просмотре и ревизии требований на соответствие заданным спецификациям качества, а также в проверке (верификации) правильности реализованных функций в отдельных продуктах проекта. Процесс управления проектом базируется на заданных сроках выполнения работ, их начала и окончания. Результаты планирования отображаются в сетевых диаграммах (PERT – Program Evaluation and Review Technique, CPM – Critical Path Method и др.), предназначенных для отображения полного комплекса работ, времени их выполнения и зависимостей между разными работами.

Сетевая диаграмма PERT является графом, в вершинах которого располагаются работы, а дуги задают взаимные связи между этими работами. Такой граф является наиболее распространенным представлением сети для управления разными видами работ на сегодняшний день. Другой тип сетевой диаграммы является событийным, когда в ее вершинах указываются события, а работы задаются линиями между двумя

узлами–событиями. Ожидаемое время выполнения работы для сетевых диаграмм оценивается с помощью среднего весового значения трех оценок: оптимистической, пессимистической и ожидаемой – вероятностной. Эти оценки берутся из заданного времени на разработку и заключений экспертов, оценивающих отдельные работы и в комплексе. Есть и другие методы оценок.

После составления плана решается вопрос управления и контроля проекта согласно плану, выбранного процесса и сущности проекта. Корректно составленный план обеспечивает выполнение требований и целей проекта. Процесс контроля более всего направлен на внесение изменений в проект, оценку риска и оценка принимающих решений.

Управление рисками является важной проблемой выполнения проекта и представляет собой процесс определения рисков и разработки мероприятий по уменьшению их влияния на ход выполнения проекта.

Риск – вероятность проявления неблагоприятных обстоятельств, которые могут повлиять негативно на реализацию качества проекта и на управление разработкой (например, увольнение сотрудника и отсутствие замены для продолжения работ и др.). Для управления риском проводится идентификация и анализ риска, оценка критических рисков и планирование непредвиденных ситуаций, касающихся рисков. Предотвращение риска заключается в выполнении действий, которые снимают риск (например, увеличение времени разработки и др.), уменьшают вероятность появления нового риска при реорганизации проекта, БД или транзакций, а также при выполнении ПО.

Измерение в инженерии ПО проводится для определения отдельных характеристик объектов инженерии (продуктов, процессов и т.п.) и их измерение. Проводится комплекс работ, связанных с выбором метрик для оценки качества процессов и продуктов, а также обстоятельств и зависимостей, влияющих на их измерение. К ним относятся: совершенствование процессов управления проектом; оценки временных затрат и стоимости ПО, их регулирование; определение категорий рисков и отслеживание факторов для регулярного расчета вероятностей их возникновения; проверка заданных в требованиях показателей качества отдельных продуктов проекта и проекта в целом [15].

Проведение разного рода измерений является важным принципом любой инженерной деятельности, а в программном проекте оно положительно влияет на результат выполнения и создания ПО, необходимого заказчику и потребителям. Без измерений в инженерии ПО процесс управления становится неэффективным и превращается в самоцель.

Таким образом, рассмотрение основных принципов формирования команды разработчиков, методов планирования работ и оценки процессов создания ПО и самого продукта позволяет менеджеру коллектива разработчиков проекта сосредоточится на планировании работ, оценках трудозатрат, распределения ресурсов и управления изменениями и рисками.

1. 8. Процесс инженерии ПО (Software Engineering Process)

Процесс инженерии ПО включает концепции, инфраструктуру, методы определения и измерения этапов ЖЦ, поиск ошибок и внесение изменений, а также анализ и оценку качества продукта.

Область знаний «Процесс инженерии ПО (Software Engineering Process)» состоит из следующих разделов:

– концепции процесса инженерии ПО (Software Engineering Process Concepts),
– инфраструктура процесса (Process Infrastructure),
– определение процесса (Process Definition),
– оценка процесс (Process Assessments),
– количественный анализ процесса (Qualitative Process Analysis),
– выполнение и изменение процесса.(Process Implementation and Change).

Данная область знаний связана со всеми элементами управления процессами ЖЦ ПО, изменения которых проводятся в связи с их совершенствованием. Цель управления в применении лучших процессов, соответствующих реальной практике выполнения конкретного проекта.

Инфраструктура процесса базируется на основных положениях стандартов IEEE/IEC 12207 и 15504, а также на видах ресурсов (групп разработчиков, технических средств, программных продуктов и др.) и процессе инженерии ПО (групповом или по типу экспериментальной фабрики (Experience Factory– EF), базирующейся на моделях проекта и продукта, моделях качества и риска. Инфраструктура включает уровни управления, отношения в коллективе, инженерные методы организации и интеграции программного продукта. Основной задачей EF является совершенствование ПО после получения опыта и уроков его разработки.

Определение процесса основывается на: типах процессов и моделей (водопадная, спиральная, итерационная и др.); моделях ЖЦ процессов и средств, стандартах ЖЦ ПО ISO/IEC 12207 и 15504, IEEE std 1074-91 и 1219-92; а также методах и нотациях задания процессов и автоматизированных средствах их поддержки. Основной целью процесса является повышение качества получаемого продукта, улучшение различных аспектов программной инженерии, автоматизация процессов и др.

К нотациям определения процессов диаграммы потоков данных, диаграммы переходов и состояний. Ряд нотаций используются в рамках RUP. Есть специальные нотации для графического представления бизнес-процессов (Business Process Management Notation). Автоматизация заключается в использовании тех или иных диаграмм и нотаций (RUP, MSF, IBM, Rational Rose и др.).

Оценка процесса проводится с использованием соответствующих моделей и методов оценки. Например, оценка потенциальной способности специалиста для выполнения соответствующей работы. SWEBOK обращает внимание на необходимость проведения оценки возможности организации проводить разработку ПС на основе модели оценки зрелости (CMM [22]) и процессов, согласно которым проводится разработка ПО.
Оценки относятся также и к техническим работам программной инженерии, к управлению персоналом и качеству ПО. Для этого проводится наблюдение за выполнением процесса, сбор информации, моделирование, классификация полученных дефектов, а также статический контроль и оценка процесса.

Качественный анализ процесса состоит в идентификации и поиске слабых мест в процессе создания ПО до начала его функционирования. Рассматривается две техники анализа: обзор данных и сравнение данного процесса с основными положениями стандарта ISO/IEC–12207; сбор данных о качестве процессов; анализ главных причин отказов в функционировании ПО, откат назад от точки

возникновения отклонения до точки нормальной работы ПО для выяснения причин изменения процесса.

Выполнение и изменение процесса. Существует ряд фундаментальных аспектов измерений в программной инженерии, лежащих в основе детальных измерений процесса. Оценка совершенствования процессов проводится для установления количественных характеристик процессов и продуктов. После процесса развертывания ПО, выполняются вычисления, после чего проводится инспекция результатов выполнения, анализ и принятие решений, в частности возникновение необходимости изменения процесса, организационной структуры проекта и некоторых инструментов управления измерениями. Результаты процесса могут касаться оценки качества продукта, продуктивности, трудозатрат на процессе и др.

Таким образом, рассмотрение моделей ЖЦ и их особенностей, анализа процесса изменения ПО и подходов к его моделированию, методов контроля, сбора данных о дефектах и моделях оценки процессов ПО позволят разработчикам овладеть рассмотренными задачами данной области знаний SWEBOK.

1. 9. Методы и средства инженерии ПО (Software Engineering Tools and Methods)

Методы и средства включают среду разработки, средства и методы разработки, используемые на процессах ЖЦ. Средства обеспечивают спецификацию требований, конструирование и сопровождение ПО. Методы обеспечивают проектирование, реализацию и выполнение ПО на процессах, а также достижение качества процессов и продуктов.

Область знаний «Методы и средства инженерии ПО (Software Engineering Tools and Methods)» состоит из разделов:

– инструменты (Software Tools),
– методы (Software Methods).

Инструменты ПО подразделяются на инструменты:
– работы с требованиями,
– проектирования ПО (редакторы схем и диаграмм),
– конструирования ПО (редакторы текстов, компиляторы, отладчики),
– тестирования (генераторы тестов, среды исполнения тестов),
– автоматизации процесса инженерии ПО,
– контроля качества,
– управления конфигурацией ПО (управление версиями, учет дефектов и решения этих проблем),
– управления инженерией ПО (планирование проектов, управление рисками).

Методы инженерии ПО включают эвристические (неформальные) методы – структурные, объектно-ориентированные, ориентированные на данные и на прикладную область, а также формальные методы проектирования и прототипирования.

Таким образом, данная область знаний SWEBOK предоставляет разработчикам и пользователям ПО информацию о современных методах и инструментах его проектировании и дает возможность выбрать методы и инструменты, наиболее подходящие для использования в соответствующем типе программного проекта.

1. 10. Качество ПО (Software Quality)

Качество ПО – набор характеристик продукта или сервиса, которые характеризуют его способность удовлетворить установленным или предполагаемым потребностям заказчика. Понятие качества имеет разные интерпретации в зависимости от конкретной системы и требований к программному продукту. Кроме того, в разных источниках таксономия и модели качества отличаются. Каждая модель имеет различное число уровней и общее число характеристик качества.

Стандарт ISO 9126-01 рассматривает *внешние и внутренние характеристики* качества. Первые отображают требования к функционирующему программному продукту. Для того чтобы количественно определить критерии качества, по которым будет осуществляться проверка и подтверждение соответствия ПО заданным требованиям, определяются соответствующие внешние измеряемые свойства (внешние атрибуты) ПО и метрики, к которым относятся . модели оценки атрибутов и диапазоны изменения значений соответствующих атрибутов. Метрики, применение которых возможно только для *работающего* на компьютере ПО, используются на стадии тестирования или функционирования, называются *внешними метриками*.

Внутренние характеристики качества и внутренние атрибуты ПО используются при составлении плана достижения необходимых внешних характеристик качества для конечного программного продукта. Для квантификации внутренних характеристик качества используются метрики проверки соответствия промежуточных продуктов внутренним требованиям к качеству, которые получаются на этапах предыдущих тестированию (определение требований, проектирование, кодирование).

Внешние и внутренние характеристики качества отображают свойства самого ПО (работающего или не работающего), а также взгляд заказчика и разработчика на это ПО. Непосредственного конечного пользователя ПО интересует эксплуатационное качество ПО – совокупный эффект характеристик качества, который измеряется в сроках использования результата, а не свойств самого ПО. Это понятия шире, чем любая отдельная характеристика (например, удобство использования или надежность). Окончательная оценка качество проводится в соответствии со стандартом ISO 15504-98 [15]. Качество может повышаться за счет постоянного улучшения используемого продукта, в связи с процессами обнаружения, устранения и предотвращения сбоев/ дефектов в ПО.

Область знаний «Качество ПО (Software Quality)» состоит из следующих разделов:
– концепция качества ПО (Software Quality Concepts),
– определение и планирование качества (Definition & Planning for Quality),
– деятельности и техники гарантии качества и V&V(Activities and Techniques for Software Quality Assurance, Validation –V & Verification – V),
– измерения в анализе качества ПО (Measurement in Software Quality Analysis).

При рассмотрении данной области знаний представлен обширный материал по изучению проблемы качества ПО и путей его достижения в процессе проектной деятельности групп разработчиков.

Концепция качества ПО включает внешние и внутренние характеристики качества, их метрики, а также модели качества, определенные на множестве внешних и внутренних характеристик, которые определены в стандартах качества [16] – это шесть характеристик и для каждого из них 4-5 атрибутов. К характеристикам качества относятся:

– функциональность,
– надежность,
– удобства использования,
– эффективность,
– сопровождаемость,
– переносимость.

Базовая модель качества включает эти характеристики и относится к любому типу программных продуктов. При разработке требований заказчик формулирует те требования к качеству, которые наиболее подходят для заказываемого программного продукта.

Определение и планирование качества ПО основывается на положениях стандартов в этой области, составлении планов графиков работ и процедурах проверки и др. План обеспечения качества включает набор действий для проверки процессов обеспечения качества (верификация, валидация и др.) и формирование документа по управлению качеством.

Управления качеством применяется к процессам, продуктами и ресурсам, а также включает требования к процессам и их результатам. Планирование качества включает:

– определение продукта в терминах заданных характеристик качества;
– планирование процессов для получения требуемого качества;
– выбор методов оценки планируемых характеристик качества и установления соответствия продукта сформулированным требованиям.

В стандарте 12207 определены специальные процессы: обеспечения качества, верификации, аттестации (валидации), совместного анализа, аудита. Области ядра знаний SWEBOK (пп.1.7 и 1.8) предлагают методы разработки программных продуктов в организациях, занимающихся ПО.

Деятельности и техники гарантии качества включают: инспекцию, верификацию и валидацию ПО.

Инспекция ПО – анализ и проверка различных представлений системы и ПО (спецификаций, архитектурных схем, диаграмм, исходного кода и др.) и выполняется на всех этапах ЖЦ разработки ПО.

Верификация ПО – процесс обеспечения правильной реализации ПО, которое соответствует спецификациям, выполняется на протяжении всего жизненного цикла. Верификация дает ответ на вопрос, правильно ли создана система.

Валидация – процесс проверки соответствия ПО функциональным и нефункциональным требованиям и ожидаемым потребностям заказчика.

Верификация и валидация в целом начинаются выполняться на ранних стадиях ЖЦ и ориентированы на качество. Они планируются и обеспечиваются определенными ресурсами с четким распределением ролей. Проверка основывается на использовании соответствующих техник тестирования для обнаружения тех или иных дефектов и сбора статистики. В результате собранных данных проводится оценка правильности реализации требований и работы ПО в заданных условиях.

Измерение в анализе качества ПО основывается на: сборе данных при выполнении процессов создания продукта на заданных ресурсах; определении метрик оценки процессов, ПО и моделей их измерения; документировании измерений и др. Для оценки фактических характеристик качества продукта проводится *тестирование ПО* путем исполнения кода на тестовых данных, сбора статистики и проведения анализа выходных результатов и полученных рабочих характеристик ПО.

Тесты разрабатываются для имитации работы системы в режиме тестирования с реальными входными данными для проверки правильности работы ПО и сбора данных о числе отказов, дефектах, ошибках и т.п.

В процессе тестирования ПО обнаруживаются разного рода ошибки, которые могут повлиять на получение правильного результата. Исходя из полученных в ПО ошибок устанавливается несоответствие количества реализованных функций, заданных в спецификациях на систему, а также оцениваются нефункциональные характеристики системы, заданные в требованиях (производительность, надежность и др.) с помощью данных, собранных на этом процессе. Проводятся также следующие типы оценок: управление планам, инспекциями, прогонами, аудитами. Сущность управления состоит в принятии решений о необходимости внесения изменений для устранения ошибок, определении адекватности планов и требований, оценки рисков и др.

Целью инспекций является обнаружение различных аномальных состояний в ПО независимыми специалистами команды экспертов и с привлечением авторов промежуточного или конечного продукта. Эксперты инспектирует выполнение требований, интерфейсы, входные данные и т.п., а затем документируют обнаруженные отклонения в проекте.

Назначением аудита является независимая оценка продуктов и процессов на соответствие регулирующим и регламентирующим документам (планам, стандартам и др.), формулирование отчета о случаях несоответствия и предложений для их корректировки.

Таким образом, данная область знаний SWEBOK представляет методологию проведения мероприятий по достижению высокого качества ПО. Рассматриваются характеристики и атрибуты качества, согласно стандарта ISO 9126-98, и приведены способы их достижения на процессах ЖЦ ПО. Определяются виды и техники анализа ПО, прогонки системы на тестах и методы оценки показателей качества.

2. Введение в жизненный цикл ПО стандарта ISO/IEC 12207 и связь его с ядром знаний программной инженерией SWEBOK

Программная инженерия, как инженерная дисциплина охватывает все аспекты создания ПО от начальной стадии разработки системных требований, создания ПО и его использования. Эталонная модель программной инженерии включает три взаимосвязанных фактора: процессы, программные продукты, ресурсы (человеческие, технические и финансовые).

Каждая ПС на протяжении своего существования проходит определенную последовательность *процессов* (этапов), начиная от постановки задачи, до ее воплощения в готовую программу, эксплуатации и изъятия. Такая последовательность этапов называется *жизненным циклом (ЖЦ)* разработки ПС. На каждом этапе ЖЦ выполняется определенная совокупность процессов и/или подпроцессов, каждый из

которых порождает соответствующий промежуточный продукт, используя результаты предыдущего.

Все продукты процессов программной инженерии представляют собой некоторые описания, а именно тексты требований к разработке, согласование договоренностей с заказчиком, архитектура, структура данных, тексты программ, документация, инструкции по эксплуатации и т.п.

Главными *ресурсами* разработки ПС в программной инженерии являются сроки, время и стоимость. Правильное использование этих ресурсов на процессах ЖЦ определяет эффективность этой разработки.

Разновидности действий и задач, представленные в процессах ЖЦ ПС, отображены в международном стандарте ISO\IEC 12207 (таблица 1) и связаны содержательно с областями знаний SWEBOK.

Данный стандарт устанавливает архитектуру верхнего уровня ЖЦ ПО, начиная от разработки концепции до утилизации системы. Архитектура представляет собой множество процессов, взаимосвязей между ними и определяет действия и задачи, т.е, он определяет, что надо делать, а не как надо выполнять действия или задачи процессов.

Стандарт не обязывает использовать определенную модель ЖЦ ПО или конкретную методологию разработки ПО и не предъявляет требования к формату и содержанию создаваемых документов. Поэтому организации–пользователю этого стандарта потребуются для своей работы дополнительные стандарты или процедуры, определяющие разные детали процесса (ISO выпускает руководства и процедуры, дополняющие стандарт 12207).

Основная идея данного стандарта состоит в том, что разработка и сопровождение ПО должно осуществляться так, как этого требует инженерная дисциплина. Следуя этой идее, разрабатывается каркас (framework), имеющий четкие связи с окружением системной инженерии - ПО, техническим обеспечением, исполнителями и деловой практикой.
Все процессы в данном стандарте разделены на три категории:
– основные процессы;
– обеспечивающие (поддерживающие) процессы;
– организационные процессы.

Для каждого из процессов определены виды деятельности (действия - activity) и задачи и определяется совокупность результатов (выходов) видов деятельности и задач, а также некоторые специфические требования. Стандарт дает перечень работ для основных обеспечивающих и организационных процессов.

Процессы ЖЦ в стандарте ISO/IEC 12207 Таблица 1

№ п/п	Наименование процессов (подпроцессов)
	Категория "Основные процессы"
1.1	Заказ (договор)
1.1.1	Подготовка заказа, выбор поставщика
1.1.3	Мониторинг деятельности поставщика, прием потребителем
1.2	Поставка (приобретение)
1.3	Разработка
1.3.1	Выявление требований
1.3.2	Анализ требований к системе
1.3.3	Проектирование архитектуры системы
1.3.4	Анализ требований к ПО системы
1.3.5	Проектирование ПО
1.3.6	Конструирование (кодирование) ПО
1.3.7	Интеграция ПО
1.3.8	Тестирование ПО
1.3.9	Системная интеграция
1.3.10	Системное тестирование
1.3.11	Инсталляция ПО
1.4	Эксплуатация
1.4.1	Функциональное использование
1.4.2	Поддержка потребителя
1.5	Сопровождение
	Категория "Процессы поддержки"
2.1	Документирование
2.2	Управление конфигурацией
2.3	Обеспечение гарантии качества
2.4	Верификация
2.5	Валидация
2.6	Общий просмотр
2.7	Аудит
2.8	Решение проблем
2.9	Обеспечение применимости продукта
2.10	Оценивание продукта
	Категория "Организационные процессы"
3.1	Категория
3.1.1	Управление на уровне организации
3.1.2	Управление проектом
3.1.3	Управление качеством
3.1.4	Управление риском
3.1.5	Организационное обеспечение
3.1.6	Измерение
3.1.7	Управления знаниями
3.2	Усовершенствование
3.2.1	Внедрение процессов
3.2.2	Оценивание процессов
3.2.3	Усовершенствование процессов

К основным процессам относятся:

– *процесс приобретения* инициирует ЖЦ ПО и определяет действия организации-покупателя (или заказчика), которая приобретает автоматизированную систему, программный продукт или сервис. Этот процесс включает следующие виды деятельности: инициация; подготовка запроса, контракта и его актуализация; мониторинг поставщиков; приемка и завершение;

– *процесс поставки* определяет действия предприятия - поставщика, которое снабжает покупателя системой, программным продуктом или сервисом. Данный процесс включает в себя следующие виды деятельности: инициация; подготовка предложений (ответа на запрос); контракт; планирование; выполнение и контроль; анализ и оценка; поставка и завершение. Процесс поставки начинается тогда, когда устанавливаются договорные отношения на поставку ПО между заказчиком и поставщиком. В зависимости от условий договора процесс поставки может включать процесс разработки ПО, процесс эксплуатации для обеспечения служб эксплуатации ПО или процесс сопровождения для исправления и улучшения ПО;

– *процесс разработки* определяет действия предприятия - разработчика, которое разрабатывает программный продукт. Этот процесс включает в себя: внедрение процесса (implementation); анализ требований к системе; проектирование архитектуры системы; анализ требований к ПО; проектирование архитектуры ПО; детальное проектирование ПО; кодирование и тестирование ПО; интеграция ПО; интеграция системы; квалификационное тестирование; установка ПО; обеспечение приемки ПО;

– *процесс эксплуатации* определяет действия предприятия-оператора, которое обеспечивает обслуживание системы (ПО) в процессе ее эксплуатации пользователями (консультирование пользователей, изучение их потребностей с точки зрения удовлетворения их системой и т.д.). Этот процесс направлен на: внедрение процесса; функциональное тестирование; эксплуатацию системы; обеспечение пользователя документацией по проведению эксплуатации ПО;

– *процесс сопровождения* определяет действия организации, выполняющей сопровождение программного продукта (управление модификациями, поддержку текущего состояния и функциональной пригодности, инсталляцию и удаление программного продукта на вычислительной системе пользователя). Данный процесс ориентирован на: внедрение процесса; анализ проблем и модификация; реализация модификаций; анализ сопровождения; миграция (перемещение) ПО; удаление ПО.

К обеспечивающим процессам создания ПС относятся: документирование, управление версиями, верификация и валидация, просмотры, аудиты, оценивание продукта и др. Процессы управления версиями соответствуют управлению конфигурацией системы, которая также, как и продукты процессы, должны проверяться на правильность реализации целей проекта и соответствия требованиям заказчика. Задачи проверки рекомендуется выполнять специальные контролеры, обладающие знаниями методов и процессов.

К организационным процессам относятся процессы управления проектом (менеджмент разработки), качеством, риском и др. Эти процессы организационно поддерживаются специальными службами: контроля процессов, измерения продуктов, проверки качества, соблюдения стандартных положений и др. Предполагает проведение

обучения персонала, определение набора задач и ответственности каждого участника в реализации задач на процессах ЖЦ и др.

Процессы, определенные в этом стандарте, образуют полное множество. Пользователь стандарта может выбрать соответствующее подмножество для достижения своей конкретной цели. Процессы, действия и задачи приведены в стандарте в наиболее общей естественной последовательности. В зависимости от целей конкретного проекта процессы, действия и задачи выбираются, упорядочиваются и применяются итерационно или рекурсивно. Разработчик должен определить или выбрать модель ЖЦ ПО в зависимости от сложности, стоимости и ресурсов программного проекта.

Данный стандарт является главным документом, определяющим содержание деятельности в сфере технологии разработки, а знания, которые необходимы исполнителям для выполнения всех видов деятельности по проектированию и реализации поставленных задач перед проектом, определяют методы и средства ядра знаний SWEBOK.

Между стандартом ISO\IEC 12207 и ядром знаний SWEBOK существует связь и взаимовлияние друг на друга, тем более в разработке обоих документов примерно в одно время принимали участие высококвалифицированные специалисты в области программирования и информатики.

Общие идеи и методы программирования, сложившиеся в 90-х годах прошлого столетия, проникли в оба направления и оказали влияние на их структуру и содержание. Программисты–профессионалы систематизировали накопившиеся знания и создали 10 разделов, которые близки процессам ЖЦ по целям, задачам и видам деятельности. В ядре знаний SWEBOK они изложены, как фундаментальные знания и инженерные методы управления разработкой ПО, а в стандарте, как общие положения, структура и регламентированные процессы проектирования, начиная от процесса постановки требований до эксплуатации ПО. Процессы стандарт отвечают на вопрос, как надо делать, т.е. какие действия и задачи процессов ЖЦ надо выбрать, чтобы построить конкретное ПО. Ядро знаний SWEBOK отвечает на вопрос, какими методами, средствами и инструментами надо выполнять регламентированные действия и задачи процессов ЖЦ, чтобы построить ПО.

Таким образом, программная инженерии сформировалась как инженерная дисциплина, которая базируются на теоретических и прикладных методах и средствах разработки ПО, которые будут излагаться в данном учебнике более подробно, и стандартах (ISO/IEC 12207, 15404, ISO 9126 и др.), содержащих рекомендации, правила и методики управления разработкой ПО. Эти два базиса объединяет инженерия оценивания результатов на процессах ЖЦ, управление качеством ПО, оценка затраченных ресурсов на его создание и учета стоимости деятельности участников разработки.

Таким образом, инженерия программирования делает акцент на принципы, методы и подходы к управлению проектом, конфигурацией и качеством ПО, а стандарты регламентирует процессы организационной деятельности при инженерному проведению работ в процессе проектирования и разработки ПО.

Ядро знаний SWEBOK, а также многочисленные монографии и статьи по методам и средствам программной инженерии предоставляет всю необходимую информацию для выбора наиболее подходящего метода, средства, инструмента, а также процессов ЖЦ

для реализации конкретного программного проекта на инженерной и регламентированной, стандартизованной основе.

Естественно, что в небольших программных проектах всегда будет место творческим и неформальным подходам, вносимых отдельными личностями-профессионалами, при создании разного рода уникальных продуктов, процесс разработки которых не всегда вкладывается в общее стандартное русло.

Инженерная дисциплина проектирования включает организационные процессы – планирование, управление и сопровождение. Планирование ставит своей целью составить планы и графики работ по реализации проекта и распределить их между разными категориями специалистов с учетом их квалификации и уровня знаний проблематики программной инженерии. Второй процесс обеспечивает привнесение методов управления в процесс выполнения работ по программированию, а именно управление временем, стоимостью и сроками. Третий процесс рассматривается как процесс выполнения проекта, обнаружения и устранения найденных недостатков в изготовленной системе, а также внесения новых функций по заказу пользователей этой системы. Один из метров программной инженерии М.Джексон определил [8] золотое правило программирования: *всякая только что законченная программная система сразу требует изменений.*

3. Обучение специальности – программная инженерия

Инженерная деятельность в программировании близка по своей сущности к определению инженерной деятельности (например, приборостроение), определенной в толковом словаре:

1) инженерия есть применение научных результатов в практику, что позволяет получать пользу от свойств материалов и источников энергии;
2) деятельность по созданию машин для предоставления полезных услуг.

В программной инженерии, инженеры – это специалисты, выполняющие практические работы по реализации программ с применением теории, методов и средств компьютерной науки, которая охватывает теорию и методы построения вычислительных и программных систем. Знание компьютерной науки необходимо специалистам в области ПО так же, как знание физики – инженерам-электронщикам [5]. Если для решения конкретных задач программирования не существует подходящих методов или теории, инженеры применяют свои знания, накопленные ими в процессе конкретных разработок ПО, а также исходя из опыта работы на соответствующих инструментальных программных средствах. Кроме того, инженеры должны работать в условиях заключенных контрактов и выполнять задачи с учетом их условий.

В отличие от другой науки, целью которой есть получение знаний, в инженерии знание является способом получения некоторой пользы. Ф.Брукс [7] считает, что «ученый строит, чтобы научиться, инженер учится, чтобы строить».

Таким образом, разработку ПО можно считать инженерной деятельностью. Она имеет важные отличия с традиционной технической инженерией:

– традиционные ветви инженерии имеют высокую степень специализации, а у программной инженерии специализация заметна только в довольно узких применениях (например, операционные системы, трансляторы, редакторы и др.);
– объекты традиционной инженерии хорошо определены и манипуляции с ними происходят в узком контексте типичных проектных решений и деталей, которые отвечают типовым требованиям заказчиков и касаются отдельных деталей, а не общих вопросов, тогда как у программной инженерии подобная типизация отсутствует;
– отдельные готовые решения и изделия в традиционной инженерии классифицированы и каталогизированы, а в программной инженерии каждое новое решение и разработка некоторого элемента ПО - это новая проблема, для которой довольно трудно установить аналогию с ранее выполненными разработками таких продуктов, как программа, компонент, система и т.п.

Приведенные отличия требуют значительных усилий и доработок для превращения программной инженерии в специальность. Мировая компьютерная общественность признала целесообразность и своевременность таких усилий. Подтверждением этого является совместное создание ядра SWEBOK, разных программ обучения (Cirricula - 2001-2005) [23, 24], институтов и комитета международного профессионального объединения в области информатики. Их главной целью является проведение работ по преобразованию программной инженерии в специальность, которая имела бы зафиксированные признаки для ее распознания и официального признания в мировом сообществе специалистов [25-28].

Практика специализации профессиональной деятельности, которая сложилась в цивилизованном мире, позволяет считать профессию "зрелой", если для нее существуют:

– система начального обучения специальности;
– механизмы развития умений и навыков персонала, которые необходимы для его практической деятельности;
– квалификация персонала организована в рамках профессии;
– лицензирование специалистов организовывается под управлением соответствующих государственных органов (в частности, для систем с повышенным риском, например, для атомных станций и т.п.);
– системы профессионального усовершенствования квалификации персонала и отслеживания современного уровня знаний и технологий по специальности, чтобы специалисты могли выжить в условиях интенсивного развития специальности;
– этический кодекс специалистов;
– профессиональное объединение.

Одним из дополнительных отличий программной инженерии состоит в большой сложности и недостаточной специализации большинства видов деятельности по ее основным направлениям, отсутствие научной системы классификации и каталогизации готовых решений в этой области.

Отметим, что указанные профессиональные организации в 1999г. приняли этический кодекс специалистов по программной инженерии [13], разработали руководства для обучения программной инженерии, а также создали программу обучения Computing Curricula (CC) 2001 [24]. Кроме того, в США работает комитет по сертификации учебных заведений (Computing Accreditation Commission of the Accreditation Board for Engineering and Technology [29].

Таким образом, инженерная деятельность в программировании приблизилась к энциклопедическому определению, отмечается ее становление как специальности большинством преподавательского состава Вузов, связанных с информатикой. Программная инженерия в соответствии с SWEBOK имеет связь со смежными дисциплинами:

– компьютерные науки;
– управление проектом;
– электротехническая инженерия;
– математика;
– телекоммуникации и сети;
– менеджмент;
– когнитивные науки;
– другие инженерные дисциплины.

В результате, можно сделать вывод о том, что содержание новой инженерной дисциплины "программная инженерия" является сформированной, ей уделяется большое внимание как специалистов, совершенствующих ее разные аспекты, так и разработчиков, которые используют ее в практике программирования и реализации разных видов и типов программных систем.

3.1. Анализ системы знаний у ИТ–специалистов

Вопрос обучения специалистов, занимающихся разработкой и внедрением информационных технологий (ИТ- специалистов), дискутируется много лет [22-28]. Считается, что обучение должно быть поставлено таким образом, чтобы ИТ - специалисты обладали не только фундаментальными знаниями в области computer science, но и достигали баланса между эффективными и жесткими сроками, широтой функциональных возможностей продукта и его низкой ценой, многообразием потребностей пользователей и ограничениями архитектур, постоянно повышали свою квалификацию. Во всем мире сохраняется высокая потребность в квалифицированных специалистах в области ИТ (информационных технологий). Сфера образования должна готовить специалистов, которые могли бы приступать к производственной деятельности сразу после получения диплома для удовлетворения этой потребности.

Одной из наиболее серьезных проблем, является подбор специалистов на руководящие должности: директоров информационных служб, руководителей проектов и технологов. Руководители бизнес–проектов отмечают, что выпускники технических специальностей университетов не имеют навыков управления и знаний современных технологии, что не позволяет им управлять проектами в области ИТ. Специалисты, обладающие навыками управления, как правило, не обладают актуальными фундаментальными знаниями, не знают специфики информационных и компьютерных ресурсов и зачастую пытаются решить проблемы проекта с помощью стандартных инженерных или управленческих методов, что приводит к серьезным управленческим ошибкам. Как свидетельствует статистика [1], в итоге более 42% проектов проваливаются или не укладываются в заданную стоимость и сроки.

Другой серьезной проблемой является, как указывалось выше, существующие отличия программной инженерии от классических инженерных дисциплин. Постоянное изменение предмета изучения и большой объем фундаментальных знаний, для непосредственного создания информационных систем требуются знания. Современный ИТ-специалист, в первую очередь, сам использует богатый арсенал программных средств, в результате производительность труда возросла многократно, и значительно

возросла ответственность, лежащая на каждом специалисте и требования к его квалификации. С другой стороны информационные системы тоже изменились, они стали значительно сложнее, объем требуемых знаний для построения и использования настолько возрос, что потребовалась значительная специализация ИТ-специалистов для обеспечения достижения необходимых результатов. Как правило, большинство систем требует эффективной работы большого коллектива высококвалифицированных специалистов.

Таким образом, ИТ-специалисты должны знать основополагающие принципы и системные подходы к решению вопросов разработки, внедрения и вывода информационных систем из эксплуатации, управления требованиями [9], рисками, качеством и конфигурацией [13], интеграцией информации/компонентов [11], тестированием и метрическим анализом готовых систем [13]. ИТ-специалисты должны уметь эффективно работать в команде разработчиков, выбирать адекватные задачам процессы и технологии. Для распределенных проектов особые требования предъявляются к навыкам невербальной коммуникации.

ИТ-специалисты, занимающие руководящие должности, должны знать общие методы, стандарты ЖЦ ПО и качества, инструменты проектирования компьютерных систем [5–12], а также инфраструктуру организации-разработчика (оборудование, связи, интерфейсы и т.п.), уровни зрелости организации (коллектива) в соответствии с моделью CMM (Capability Maturity Model [29, 30]), базовые понятия программной инженерии SWEBOK [4] и другие действующие отечественные и международные стандарты.

Ключевыми дисциплинами получения знаний студентов на современных факультетах информатики, ориентированных на разработку ПО, являются: программирование и языки программирования, как формальные математические объекты, дискретная математика Кнута, основы математической логики, введение в формальную семантику Хоара, теория алгоритмов и основы построения ЯП. Получаемый такой базис знаний способствует формированию математического мышления и формального подхода к процессам разработки ПО.

С точки зрения когнитивной психологии требуется не только преподавание этих дисциплин, но и применение теоретических знаний на практике. Например, проведение силами студентов разработки некоторого прототипа проекта с использованием идей объектно-ориентированного или компонентного проектирования с повторным использованием готовых объектов и компонентов согласно ЖЦ создания программного продукта. Прототип может быть собран из готовых компонентов и на нем отработаны функции, определенные в требованиях к системе. При реализации долговременного проекта, в процессе разработки могут быть определены стандарты на компоненты, модели качества, новые методы проверки надежности компонентов, а также подходы к определению уровня зрелости по модели CMM, соответствующей оценки деятельности разработчиков этого проекта. Таким образом, в процессе обучения студентами приобретаются не только теоретические знания, но и умение их использовать при выполнении различных ролей в группе разработчиков ПО, создании отдельных версий проекта, проведении их качественного анализа и оценки заданных в требованиях показателей качества.

3.2. Подходы к обучению программной инженерии

В рамках работ по становлению программной инженерии опубликовано ряд учебников и монографий, отображающие разные ее аспекты, и сформировалось несколько подходов к обучению и подготовки соответствующих специалистов [5–13]:
1) введение в программы обучения отдельных элементов программной инженерии;
2) создание самостоятельной специальности «программная инженерия» и обучение ей студентов всех курсов;
3) сертифицированное обучение программной инженерии как профессии на курсах подготовки или переподготовки ИТ–специалистов.

Подход 1. Обучение этой дисциплине фактически уже проводиться на факультетах информатики в виде отдельных курсов, отражающих аспекты программной инженерии: модульное, объектно-ориентированное, компонентное программирование, управление данными, тестирование ПО и др. Дипломированные специалисты, прошедшие изучение некоторых из этих аспектов, не пользуются большим спросом на рынке труда, так как не имеют знаний и опыта в организации планирования и управления деятельностью разработчиков ПО, а также знаний по вопросам распределения ресурсов (людских, аппаратных, программных), оценки трудозатрат, повышения качества и др. важных моментов ведения крупных промышленных проектов. Они могут использоваться как программисты, либо повышать знания до уровня менеджера проекта или инженера в области программной инженерии на курсах повышения квалификации.

Подход 2. Наибольшее развитие в международной практике получил подход, ориентированный на создание самостоятельной специальности «программная инженерия» на факультетах информатики. Данный подход поднимает престиж учебного заведения, требует дополнительных вложений на его оборудование и привлечение соответствующего преподавательского состава. Учебный план факультета информатики предусматривает программы по информатике и программной инженерии. Согласно [24] эти учебные программы относятся как 50:50. При этом треть предметов факультета связаны с программной инженерией, а две трети – с информатикой.
Другая программа обучения CC–2001 [22] рекомендует это соотношение, как 90 к 10. На факультетах информатики курс программной инженерии должен занимать 10%, т.е. 24–30 учебных часов в семестр. При этом 10% преподавателей факультета должны учить студентов дисциплине “программная инженерия”, а 90%– 15 (пятнадцати) базовым курсам по специальности “Информатика” (дискретные системы, основы программирования, алгоритмы и теория сложности, ОС, теория алгоритмов, языки программирования и др.). Отведенный для программной инженерии диапазон часов соответствует не только базовым требованиям CC–2001, но и требованиям к курсам информатики, перечисленных в документах комитета по сертификации учебных заведений [23], готовящих инженерных специалистов. Программная инженерия, как дисциплина изучает теорию, сумму знаний, отображенных в ядре SWEBOK, и практику эффективного построения ПО на всех этапах ЖЦ. Если на факультете информатике работает 30 преподавателей, то по программной инженерии их должно быть не менее 5. CC–2001 рекомендует типовой факультативный учебный план по программной инженерии, включающий 12 тем:

1. Проектирование ПО.
2. Интерфейсы приложений.
3. Программные средства и окружение.
4. Процессы разработки ПО
5. Требования к ПО.

6. Проверка соответствия ПО.
7. Методы эволюции ПО.
8. Управление программными проектами.
9. Компонентно-ориентированная разработка (не обязательная).
10. Формальные методы.
11. Надежность и качество ПО.
12. Подходы к разработке специализированных систем (не обязательная).

Многие из этих тем, нашли отражение в ядре SWEBOK. Темы 2, 9, 12 относятся к важным в проблематике разработки ПО, они обобщают целые эпохи формирования и применения общих методов (от модульного до компонентного) программирования в процессе создания ПО и специализированных систем различного назначения с использованием средств (GUI, CORBA, COM – интерфейсы и др.). По этим темам к настоящему времени накоплен огромный опыт и знания, которые слабо представлены в ядре SWEBOK. Эти темы включены в программу обучения CC–2001 как не обязательные темы, хотя проблемы интерфейса и компонентного подхода сейчас являются перспективными направлениями дальнейшего развития современного программирования и производства ПО.

Подход 3. Обучение сформировавшихся специалистов программной инженерии на специальных курсах повышения квалификации (с отрывом и без отрыва от производства) представляет значительный интерес для действующих компаний, специализирующихся в разработке ПО и крупных программных проектов. Для усовершенствования квалификации персонала фирм объявляются или заказываются отдельные темы программной инженерии или целый курс. Самыми широко распространенными сертифицированными курсами являются: программная инженерия, управление требованиями, повторное использование, управление программными проектами, моделирование в UML и Case Rational Rose и др.

Рассмотренные подходы к преподаванию программной инженерии используются в США, Канаде, Великобритании и других европейских странах. Остановимся на анализе состояния дел по ее преподаванию на факультетах информатики в странах СНГ. До настоящего времени во многих Вузах ведется обучение ЯП (С++, Паскаль, Бейзик Lava и др.), теории алгоритмов, ОС, СУБД, информационным технологиям и системам и др. Обсуждение предмета программной инженерии как специальности проведено, например, в российской прессе [26-28, 34]. Общее мнение состоит в том, что системное преподавание этой дисциплины в основном отсутствует, в университетах ощущается недостаток профессорско-преподавательского состава и соответствующих учебно-методических пособий.

В частности, в [27] отмечается, что сложившаяся советская система образования в университетах бывшего СССР «постепенно все больше и больше отдаляется от требований современного мира»; программирование представляет фундаментальную науку и прикладную инженерную дисциплину, основанную на применение теоретических знаний в жестких ограничениях реального мира. Тем более, что в информатике отсутствует такие понятия, как ресурсы, трудозатраты, менеджмент, измерения и оценки продуктов, которые в программной инженерии играют ключевую роль. В связи с этим возникает естественная потребность в разносторонних и полноценных программах обучения профессиональных специалистов в области программной инженерии, основу которых может составлять CC-2001 [22].

Основу подготовки специалистов по программной инженерии составляют специальные программы обучения и учебники, с помощью которых студенты смогут усвоить суть данной инженерной дисциплины, получать навыки и овладеть методами создания и управления программными проектами с учетом требований индустрии, современных сред и заказчиков.

На основе огромного опыта программирования разных программных систем, преподавания спецкурса по этой дисциплине в Киевском Национальном университете им. Тараса Шевченко и учитывая международный опыт по созданию ядра знаний SWEBOK, разработан и опубликован учебник серии «высшее образование XXI столетия» [8], который рекомендован в качестве учебного пособия для обучения студентов в Вузах. Однако этот учебник вышел небольшим тиражом, его использует авторы при проведении курсов в Национальном университете имени Тараса Шевченко, в Киевском отделении МФТИ, в Национальном авиационном институте.

3.3. Анализ результатов дистанционного обучения

По инициативе и в рамках Украинской Ассоциации Производителей ПО (УАППО) было создано (март 2003г.) Украинский Учебно-Практический Центр Программной Инженерии [33], учебная программа [34], основная целью которой была подготовка менеджеров программных проектов, руководителей проектов за счет овладения знаниями, представленными в ядре знаний SWEBOK, а также знаний методам современного программирования .

Обучением было охвачено 35 человек. Первоначально в обучение включились 5 человек (руководители групп разработчиков ПО, инженеры, программисты и др.). Они не были знакомы с базисом дисциплины – программная инженерия и начали обучение именно с этого курса. Содержание тем учебной программы по замыслу составлялось очень коротким, чтобы учащийся самостоятельно изучал дополнительную литературу. В частности по этому курсу предлагалась в качестве основной литературы – оригинал материала SWEBOK – knowledge areas.

Анализ дистанционной коммуникации преподавателей и учащихся позволил выделить следующие факты:

– отсутствие необходимых коммуникативных навыков и базовых знаний у значительной части обучающихся;
– высокая эффективность в обучении формальному представлению пилотных проектов;
– многие учащиеся продемонстрировали качество выполнения заданий и в краткие сроки;
– выявление учащихся, не имеющих способностей к управленческой деятельности в ИТ сфере.

По результатам обучения можно сделать следующие выводы:

– предварительное повышение квалификации учащихся до уровня базового с постепенным переходом к изучению курсов основной программы.
– постепенное наращивание сложности практических заданий для выработки необходимых навыков.
– увеличение объема вводных лекций для каждой темы.
– предоставление шаблонов выполнения практических заданий.

Формирование электронной библиотеки материалов (включая выполненные практические задания) на курсам программы обучения.

Построение схемы обучения сотрудников компании, совместно с консалтингом по улучшению процессов разработки этой компанией программных проектов.

Значительную помощь в дистанционном процессе обучения оказали специальные фонды при общественных и коммерческих организациях, заинтересованных в подготовке квалифицированных ИТ–специалистов.

Контрольные вопросы и задания

1. Назовите цели и задачи программной инженерии.
2. Назовите признаки зрелой профессии. Какие из них присущи программной инженерии.
3. Назовите области знаний SWEBOK инженерии разработки ПО.
4. Приведите базовые понятия SWEBOK.
5. Определите цели и задачи области инженерии – управление проектом.
6. Определите цели и задачи области инженерии – управление качеством.
7. Дайте определение жизненного цикла разработки программного обеспечения.
8. Назовите три основные группы процессов жизненного цикла и перечислите процессы каждой из групп.
9. Назовите дополнительные процессы ЖЦ и перечислите их.
10. Дайте характеристику организационных процессов ЖЦ.
11. Какой международный стандарт определяет перечень и содержание процессов ЖЦа программного продукта?
13. Все ли процессы, указанные в стандарте, должны быть выполнены при каждой разработки программного обеспечения или дает ли стандарт такие возможности, которые могут быть актуальными для конкретного случая?
13. Какие разделы ядра знаний и стандарта наиболее необходимы при разработке программных систем.

Литература к теме 1.

1. Программы следующего десятилетия. Открытые системы.– Декабрь, 2001.–с.60-71.
2. *McConnel S., Tripp L.* Professional Software Engineering: Fact or Fiction ? //IEEE Software.-Nov.-Dec.-1999.-P.13-18
3. *Pfleeger S.L.* Software Engineering. Theory and practice. – Printice Hall: Upper Saddenle River, New Jersey 07458, 1998. – 576 p.
4. *Jacobson I.* Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing. – New York: Addison-Wesley Publ.Co., – 1994.– 529 p.
5. *Иан Соммервил* Инженерия программного обеспечения. 6 -издание.–Москва–Санкт–Петербург–Киев, 2002.–623 с.
6. *Е.М. Лаврищева* Проблематика программной инженерии// К.: Знання.–1991.–19с.
7. Бабенко Л.П., Лаврищева Е.М. Основы программной инженерии. Учебник (укр. язык). – Киев: Знання, 2001. –269 с.
8. *Jackson M.* Software requirement & specifications.– Wokingham, England: Addison–Wesley, ACM Press Books, 1995. –228 p.
9. Meyer. Object-oriented Software Construction. – 2nd. ed., Prentice Hall, 1997.–531 p.
10. Jacobson I., Griss M., Jonsson P. Software Reuse.–N.–Y.– Addison–Wesley, 1997. – 497 p.

11. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных систем, Киев, Изд. «Наукова думка», 1997г.–228 с.
12. *Андон Ф.И., Коваль Г.И., Коротун Т.М., Суслов В.Ю.* Основы инженерии качества программных систем.–К: Академпериодика, 2002.–502 с.
13. *Jotterbarn D., Miller K., Rogerson S.* Software Engineering CODE of Ethic is Approved//Com. of the ACM .v. 42.–N 10.–1999.–P. 102–107.
14. ISO/IEC 12207: 1995.– Information technology - Software life cycle processes) Информационные технологии - Процессы жизненного цикла программного обеспечения..
15. ISO/IEC TR 15504, Information Technology–Software Process Assessment (Part 1 – 9).
16. ISO/IEC 9126, Information Technology - Software quality characteristics and metrics (Part 1 – 4) 1997.
17. ISO-IEC 15288, System Life Cycle Processes.
18. ISO/IEC DIS 15026, Information technology – System and Software integrity levels.
19. *R.H. Thayer, ed.,* Software Engineering Project Management, 2nd.ed., IEEE CS Press, Los Alamitos, Calif. 1997.–391 p.
20. www.swebok.org, malto:sorlic@borland.ru
21. Capability Maturity Model for Software, version 1.1/M.Paulk, B Curtis at al// CMU–SEI–93–24, Soft.Engin. Institute, Pittsburg PA 15213, Feb.– Pittsburg.–82p.
22. D. Bagert et al., Guidelines for Software Engineering Education? Version 1.0, Carnegie Mellon Univ., Pittsburgh, pa., 1999.
23. http://computer.org/education/cc2001 или http://se.math.spbu.ru/cc2001- русский вариант.23.
24. Майер Б. Программная инженерия как предмет обучения. Открытые системы. Июль-август, 2001.–с.80–86.
25. M.Shaw. Software Engineering Education: A Roadmap. The future of Software Engineering, Finkkelstein, ed., ACM press, New York, 2000.
26. T.Lethbridge, What Knowledge is important to Software Professional?, Computer, vol33, № 5, May, 2000.
27. Е.М. Лаврищева. Пути стандартизации программной инженерии как специальности // Материалы межд. научно–практ. конф. “Теория и практика стандартизации образования” часть 1. Минск, 18–19 янв. 2001г.–с.8–13.
28. ABET – Criteria for According Computing Programs, 2002 (Computing Accreditation Commission of the Accreditation Board for Engineering and Technology). www.abet.org/criteria.htm
29. Буч Г. Объектно-ориентированный анализ и проектирование с примерам приложений на С++, 2-е изд. – М.: “Изд-во Бином”, 1998. – 560 с.
30. Богсс У., Богсс М. UML и Rational Rose.–Изд.-во Лори, 2000.-580с.
31. Jacobson I., Booch G., Rumbaugh J. The Unified Software Development Process, N.–Y.–Addison-Wesley, 1999. – 463 p.
32. Эммерих В. Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510 с.

Тема 2

МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ДЛЯ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ

За десятилетия опыта построения программных систем был наработан ряд типичных схем последовательности выполнения работ при проектировании и разработки ПС. Такие схемы получили название моделей ЖЦ.

Модель жизненного цикла – это схема выполнения работ и задач на процессах, обеспечивающих разработку, эксплуатацию и сопровождение программного продукта, и отражающая жизнь ПС, начиная от формулировки требований к ней до прекращения ее использования [1-5].

Исторически в эту схему работ включают:
– разработку требований или технического задания,
– разработку системы или технического проекта,
– программирование или рабочее проектирование,
– пробную эксплуатацию,
– сопровождение и улучшение,
– снятие с эксплуатации.

Выбор и построение модели ЖЦ ПС базируется на концептуальной идее проектируемой системы, ее сложности и стандартов, позволяющих формировать схему выполнения работ по усмотрению разработчика и заказчика. Модель ЖЦ разбивается на процессы реализации, которые должны включать отдельные работы и задачи, реализуемые в данном процессе, и при их завершении осуществлять переход к следующему процессу модели.

При выборе общей схемы модели ЖЦ для конкретной предметной области, решаются вопросы включения или невключения отдельных работ, очень важных для создаваемого вида продукта. На сегодня основой формирования новой модели ЖЦ для конкретной прикладной системы является стандарт ISO/IEC 12207, который включает полный набор процессов (более 40), охватывающий все возможные виды работ и задач, связанных с построением ПС.

Из этого стандарта можно выбрать только те процессы, которые более всего подходят для реализации данного ПС. Обязательными являются основные процессы, которые присутствуют во всех известных моделях ЖЦ. В зависимости от целей и задач предметной области они могут быть пополнены процессами из дополнительных или организационных процессов или подпроцессов этого стандарта. Например, решение вопроса включения в новую модель ЖЦ процесса обеспечение качества компонентов и системы в целом или определение набора проверочных (верификационных) процедур для обеспечения правильности и соответствия разрабатываемого ПС (валидация) заданным требованиям, а также процесса обеспечения возможности внесения изменений в требования или в компоненты системы и т.п.

Процессы, включенные в модель ЖЦ, предназначены для реализации уникальной функции ЖЦ и могут привлекать другие процессы для выполнения специализированных возможностей системы (например, защита данных). Интерфейсы между двумя любыми процессами ЖЦ должны быть минимальными и каждый из них привязан к архитектуре системы.

Если работа или задача требуется более чем одному процессу, то они могут стать процессом, используемым однократно или на протяжении жизни системы.. Каждый процесс должен иметь внутреннюю структуру, установленную в соответствии с тем, что должно выполняться на этом процессе.

Процессы модели ЖЦ ориентированы на разработчика системы. Он может выполнять один или несколько процессов и процесс может быть выполнен одним или несколькими разработчиками, при этом один из них является ответственным за один процесс или за все на модели, даже если отдельные работы выполняет другой разработчик.

Создаваемая модель ЖЦ увязывается с конкретными методиками разработки систем и соответствующими стандартами в области программной инженерии. Иными словами каждый процесс ЖЦ подкрепляется выбранными для реализации задач средств и методов.

Важную роль при формировании модели ЖЦ имеют организационные аспекты:
– планирование последовательности работ и сроков их исполнения,
– подбор и подготовка ресурсов (людских, программных и технических) для выполнения работ,
– оценка возможностей реализации проекта в заданные сроки и стоимость и др.

Внедрение модели ЖЦ в практическую деятельность по созданию программного продукта позволяет упорядочить взаимоотношения между субъектами процесса разработки ПС и учитывать динамику модификации требований к проектам и системе.

Эти и другие вопросы послужили источником формирования различных видов моделей ЖЦ, основанных на процессном подходе к разработке программных проектов. Основными среди них зарекомендовали себя в практике программирования являются следующие: каскадная, спиральная, инкрементная, эволюционная, стандартизованная.

2.1. Каскадная модель ЖЦ

Одной из первых начала применяться *каскадная или водопадная модель*, в которой каждая работа выполняется один раз и в том порядке, как они представлены в схеме модели ЖЦ. Т.е. делается предположение, что каждая работа будет выполнена настолько тщательно, что после ее завершения и перехода к следующему этапу возвращения к предыдущему не потребуется. Разработчик проверяет промежуточный результат разными известными методами верификации и фиксирует его в качестве готового эталона для следующего процесса. На рис.2.1. показана каскадная модель. В ней возвращение к начальному процессу работ предусматривается после сопровождения при возвращение на начальный процесс.

Согласно данной модели работы и задачи процесса разработки обычно выполняются последовательно, как это представлено в схеме. Однако вспомогательные и организационные процессы (контроль требований, показателей качества и др.) обычно выполняются параллельно с процессом разработки.

Ценность такой модели состоит в фиксации последовательных процессов разработки программного продукта. Недостатком этой модели является то, что в основу ее концепции положена модель фабрики, где продукт проходит стадии от замысла до

производства, затем передается заказчику как готовое изделие, изменение которого не предусмотрено, хотя возможна замена на другое подобное изделие в случае рекламации или некоторых ее деталей, вышедших из строя.

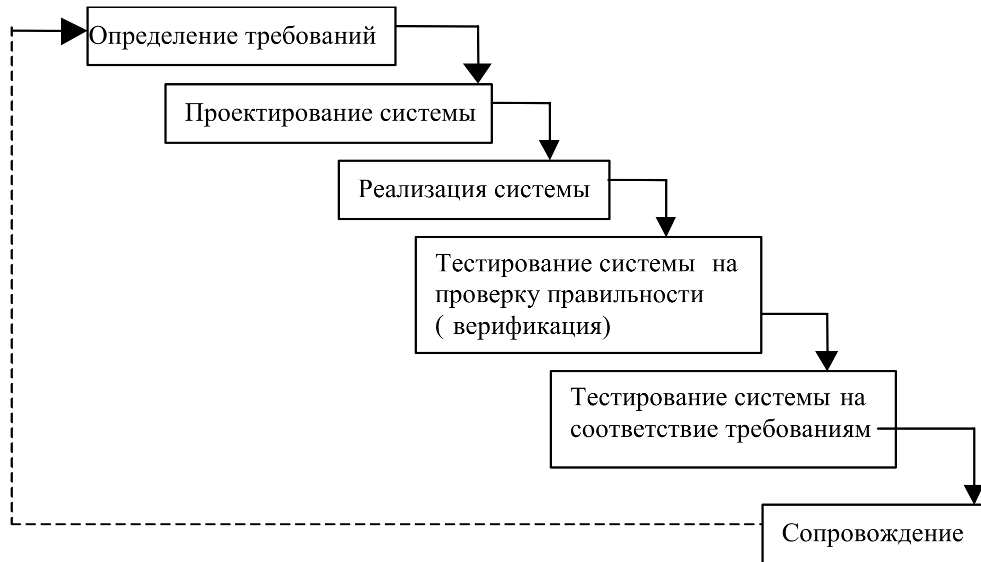

Рис. 2.1. Каскадная модель ЖЦ разработки программных систем

При таком подходе необходимо учитывать следующие факторы риска:

– требования недостаточно хорошо представлены;
– система слишком большая по объему, чтобы быть реализованной в целом;
– быстрые изменения в технологии и в требованиях;
– ограниченные ресурсы (людские, программные и др.);
– полученный продукт может оказаться непригодным для использования из-за неправильного понимания требований или функций системы, а также недостаточного тестирования.

Преимущества реализации системы с помощью каскадной модели следующие:
– все возможности системы реализуются одновременно;
– применяется в случае, если старая система должна быть полностью заменена.

Каскадную модель можно рассматривать как модель ЖЦ, пригодную для создания первой версии ПО для проверки реализованных в ней функций. При сопровождении и эксплуатации могут быть обнаружены разного рода ошибки, которые будут исправляться разработчиком, начиная с первого процесса данной модели.

2.2. Инкрементная модель ЖЦ

Эту заложена еще называют нарастающей моделью, суть которой состоит в возможности усовершенствования продукта (рис.2.2). Разработка начинается с предоставления набора требований и реализации системы путем последовательного конструирования и фиксации промежуточных продуктов (1, …, N) системы, постепенно приближающейся к итоговой системе (рис.2.2).
Первая создаваемая промежуточная структура системы реализует часть требований, в последующую структуру добавляют дополнительные требования и так до тех пор, пока не будет окончательно согласованы требования и соответственно разработка продукта системы. Над каждой промежуточной структурой выполняются необходимые процессы, работы и задачи, например, анализ требований и создание

архитектуры могут быть выполнены одновременно. Процессы разработки технического проекта ПС, его программирование и тестирование, сборка и квалификационные испытания ПС выполняются при создании каждой последующей структуре.

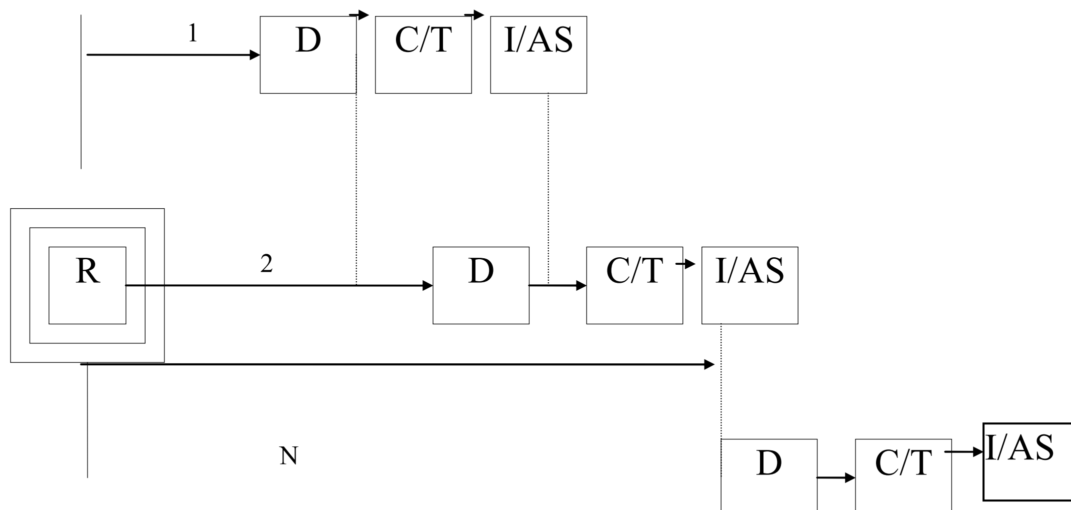

Рис.2.2. Пример инкрементной модели

В данном примере используются следующие обозначения
– R (Requirements) требования,
– C/T (Coding/Testing) кодирование, тестирование,
– D (Design) проектирование,
– I/AS (Installation/acceptance) инсталляция, сопровождение.

В соответствии с данной моделью ЖЦ, процессы которой практически такие же, что и в каскадной модели, ориентир делается на разработку некоторой законченной промежуточной структуры, а задачи процесса разработки выполняются последовательно или частично параллельно для ряда отдельных промежуточных структур.

Работы и задачи процесса разработки могут выполняться неоднократно в той же последовательности для всех промежуточных структур. Процессы сопровождения и эксплуатации могут быть реализованы параллельно с процессом разработки путем проверки частично реализованных требований в каждой промежуточной структуре и так до получения законченного варианта системы. Вспомогательные и организационные процессы обычно выполняются параллельно с процессом разработки и концу разработки будут собраны данные, на основании которых может быть устанавлен уровень надежности и качества изготовленной системы.

При применении данной модели необходимо учитывать следующие факторы риска:
– требования составлены непонятно для реализации;
– все возможности системы требуется реализовать с самого начала;
– быстро меняются технологии и требования к системе;
– ограничения в ресурсном обеспечении (люди, финансы), когда разработчики реализуют систему в течение длительного времени.

Данную модель разработки целесообразно использовать, в случае когда:
– желательно реализовать некоторые возможности системы быстро за счет создания промежуточного продукта;
– система разделена на отдельные составные части структуры, которые можно представлять как некоторый промежуточный продукт;
– возможно увеличение финансирования на разработку отдельных частей системы.

2.3. Спиральная модель

Исходя из возможности внесения изменений как в процесс, так и в создаваемый промежуточный продукт была создана *спиральная модель*.

.

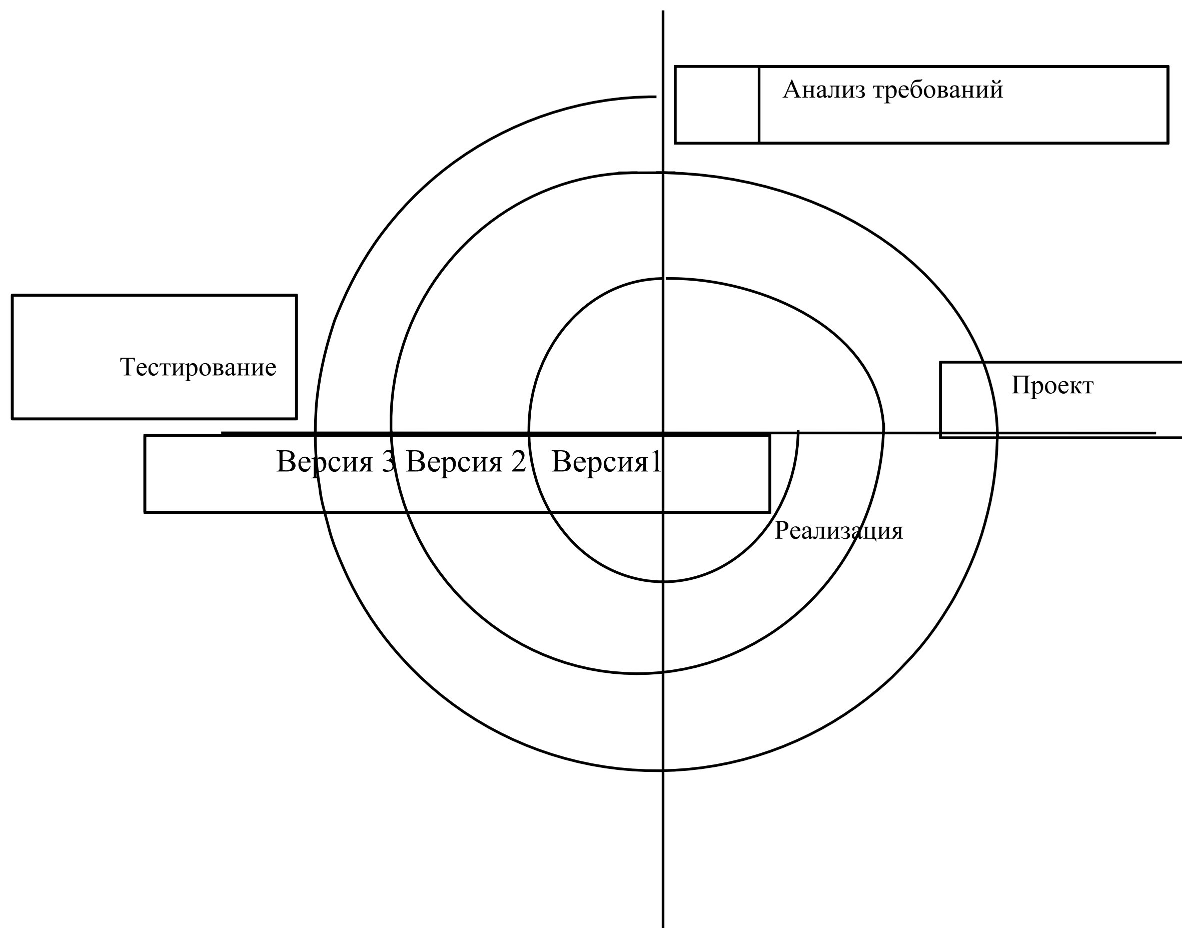

Рис.2.3. Спиральная модель ЖЦ разработки программных систем

Это допущение ориентировано на удовлетворение потребности изменений сразу, как только будет установлено, что созданные артефакты или элементы документации (описание требований, проекта, комментариев различного вида и т.п.), не соответствуют действительному состоянию разработки после внесения некоторых изменений
Данная модель ЖЦ допускает анализ продута на витке разработки, его проверку, оценку его правильности и принятия решения двигаться на следующий виток или опуститься на нижний для доработки.

Отличие этой модели от каскадной модели состоит в возможности спиральной модели обеспечивать многоразовое возвращение к процессу формулирования требований и к повторной разработке с любого процесса выполнения работ.

На изображенной спиральной модели (рис.2.3), каждый виток спирали соответствует одной из версий разработки системы. При необходимости внесения изменений в систему на каждом процессе витка для получения версии системы, обязательно вносятся изменения в предварительно зафиксированные требования, после чего происходит возврат на предыдущий виток спирали для продолжения реализации новой версии системы с учетом изменений.

2.4.Эволюционная модель ЖЦ

В случае эволюционной модели система разрабатывается в виде последовательности блоков структур (конструкций). В отличие от инкрементной модели ЖЦ, подразумевается, что требования устанавливаются частично и уточняются в каждом последующем промежуточном блоке структуры системы (рис.2.4.).

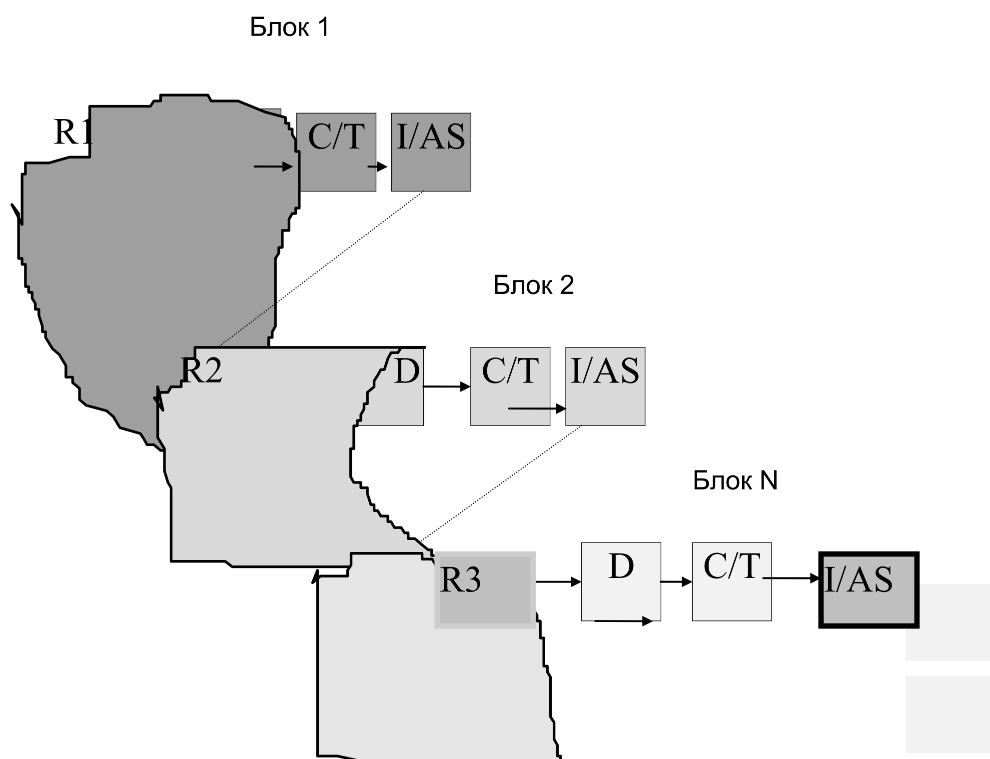

Рис.2.4. Пример эволюционной модели

На данном рисунке модели используются следующие обозначения
– R (Requirements) требования,
– C/T (Coding/Testing) кодирование, тестирование,
– D (Design) проектирование,
– I/AS (Installation/acceptance) инсталляция, сопровождение.

Работы и задачи процесса разработки в соответствии с данной моделью выполняются не однократно, но в той же последовательности, что для всех блоков структуры.

Так как промежуточные блоки структуры соответствуют реализации некоторых требований, то соответственно их реализацию можно проверять на процессе сопровождения и эксплуатации, т.е. параллельно с процессом разработки блоков структуры системы. При этом вспомогательные и организационные процессы могут

выполняться параллельно с процессом разработки и накапливать сведения по данным количественных и качественных оценок на процессах разработки.

При этом подходе учитываются такие факторы риска:
– реализация всех возможностей системы сразу;
– ограниченные ресурсы (людские, финансовые) заняты разработкой в течение длительного времени.

Преимущества применения данной модели ЖЦ состоит в следующем:
– проведение быстрой реализация некоторых возможностей системы;
– промежуточный продукт может использоваться на следующем процессе;
– в системе выделяются отдельные части для реализации их в отдельности;
– возможность увеличения финансирования системы;
– обратная связь устанавливается с заказчиком для уточнения требований;
– упрощение внесения изменений.

Модель развивается в направлении привлечения к разработке новых разработчиков и механизмов прототипирования для моделирования функциональности, нефункциональных требований (несанкционированного доступа, аутентификации и др.).

2.5. Стандартизованная модель системы

Типичный ЖЦ разработки системы начинается с формулировки идеи или потребности, проходит все процессы разработки, производства, эксплуатации и сопровождения системы. ЖЦ в практике программирования обычно делиться на этапы, процессы. Каждый процесс характеризуется видами деятельности и задачами, которые выполняются на нем. Переход от одного процесса к другому должен быть санкционирован (определены входные и выходные данные).

Модель общего стандартизованного ЖЦ, как правило, включает в себя следующие процессы:
– определение требований;
– разработка (проектирование);
– верификация, валидация, тестирование;
– изготовление;
– эксплуатация;
– сопровождение.

Данной модели соответствует все виды деятельности, которые начинаются с разработки идеи проблемы или концепции программного продукта и кончая его изготовлением. Стандарт ISO/IEC 12207 объединяет эти виды деятельности в основные, организационные и вспомогательные процессы, которые и составляют ЖЦ ПО.

Процессы ЖЦ приобретения, поставки и разработки используются для анализа и определения системных требований, решений верхнего уровня проектирования системы, и предварительного определения требований к компонентам системы, включая ПО. Процесс разработки может быть использован для анализа, демонстрации, валидации, тестирования, прототипирования требований и проектных решений.

На этом этапе проектирования разрабатывается техническое, программное, организационное обеспечение системы, а также проектируются, разрабатываются,

интегрируются, тестируются и оцениваются компоненты системы. Результатом этого процесса является система, которая соответствует нужному продукту.

Исходный стандарт разработан так, чтобы его можно было применить полностью или частично. Процессы, действия и задачи основных процессов отбираются, адаптируются и применяются для разработки или модификации ПО. Процесс проектирования может включать одну или более итераций процесса разработки. Результатом являются основные требования к ПО, проект и его реализация.

Если разрабатываемое ПО является частью системы, то могут понадобиться все действия процесса разработки, а если – автономное ПО, то все действия на уровне системы могут не понадобиться.

Во время процесса производство – изготовление спроектированная и разработанная система изготавливается для заказчика или для покупателей. Целью процесса является производство и установка работающей системы у заказчика для сопровождения. Для ПО данный процесс заключается в копировании изготовленного ПО и документации на соответствующие носители для различных пользователей. К видам деятельности на процессе относится – управление реализацией и конфигурацией. Другие вспомогательные процессы и действия могут применяться при необходимости.

Изготовленная система передается заказчику или продается покупателям. Период развертывания системы начинается с поставки первой работающей ее версии заказчику. Продажа системы начинается с первой версии системы, которая поставляется всем покупателям и заканчивается изъятием с рынка. Другие процессы (приобретения, поставки и разработки) могут использоваться при инсталляции и проверки разработанной или модифицированной системы.

Процесс эксплуатации включает использование системы пользователями и покупателями и заканчивается, когда система больше не удовлетворят пользователей и она удаляется из эксплуатации.

Во время сопровождения система модифицируется, вследствие обнаруженных ошибок и недостатков в ее разработке либо по требованиям пользователя, которому желает ее адаптировать к новой среде или усовершенствовать отдельные функции системы. Данный процесс включает обеспечение логической, технической и программной документации пользователю.

Процесс удаление системы означает снятие ее с обслуживания, удаление ее архивов и носителей кодов системы.

2.6. Сопоставление модели ЖЦ стандарта ISO/IEC 12207 и областей – процессов SWEBOK

Каждая область знаний SWEBOK по существу соответствует отдельным процессам, которые определены в стандарте 12207. В связи с этим проведен сравнительный анализ модели областей (процессов) SWEBOK и модели процессов ЖЦ этого стандарта с целью сопоставления задач процессов стандарта и основных задач областей знаний SWEBOK. В этих целях рассмотрим сначала соответственно их процессы.

2.6.1. Характеристика процессов стандарта

Процессы данного стандарта разбиты по группам: основные, вспомогательные и организационные процессы.

К основным процессам стандарта относятся:
– приобретения (acquisition),
– поставки (supply),
– разработки (development),
– эксплуатации (operation),
– сопровождения (maintenance).

Процесс приобретения инициирует ЖЦ ПО и определяет действия организации-покупателя (или заказчика), которая приобретает автоматизированную систему, программный продукт или сервис.
Процесс поставки определяет действия предприятия - поставщика, которое снабжает покупателя системой, программным продуктом или сервисом.
Процесс разработки определяет действия предприятия - разработчика, которое разрабатывает программный продукт.
Процесс эксплуатации определяет действия предприятия-оператора, которое обеспечивает обслуживание системы (ПО) в процессе ее эксплуатации пользователями (консультирование пользователей, изучение их потребностей с точки зрения удовлетворения их системой и т.д.)
Процесс сопровождения определяет действия организации, выполняющей сопровождение программного продукта (управление модификациями, поддержку текущего состояния и функциональной пригодности, инсталляцию и удаление программного продукта на вычислительной системе пользователя).

К вспомогательным процессам стандарта относятся стандарты:
– документирования (documentation),
– управления конфигурацией (configuration management),
– обеспечения качества (quality assurance),
– верификации (verification),
– валидации (validation),
– совместного анализа (оценки) (joint review),
– аудита (audit),процесс решения проблем (problem resolution).

Вспомогательные процессы поддерживают реализацию основных процессов и обеспечивают требуемое качество ПО. Они инициируются другими процессами.

К организационным процессам стандарта относятся процессы*:*
– управления (management),
– создания инфраструктуры (infrastructure),
– усовершенствования (improvement),
– обучения (training).

За каждый процесс стандарта отвечает определенный участник разработки или руководитель. Для каждого из процессов стандарта определены виды деятельности (действия - activity) и задачи, которые в него входят, определена совокупность результатов видов деятельности и задач, а также некоторые специфические требования. В табл.1 приведено общее количество определенных в стандарте процессов (17), действий (74) и задач (232).

Таблица 1

Общий перечень процессов ЖЦ стандарта 12207

Класс	Процесс	Действие	Задача
Основные процессы	5	35	135
Вспомогательные процессы	8	25	70
Организационные процессы	4	14	27
Итого	17	74	232

Из этого множества стандартов далее будут сравниваться только те процессы, которые имеют аналоги областям знаний в ядре знаний SWEBOK.

2.6.2. Характеристика модели процессов в ядре SWEBOK

В ядре знаний SWEBOK определено 10 областей знаний, пять из них по своим задачам и выполняемым действиям соответствуют *основным процессам* ЖЦ стандарта. Остальные пять областей ядра можно отнести к числу процессов обеспечения и управления разработкой программного продукта, в части верификации, сбора данных для оценки качества и др., начиная от разработки требований и кончая сопровождением программного продукта. И хотя ядро знаний явно не содержит названий процессов, функционально они соответствуют общепринятым процессам разработки и стандарту, а именно отдельным основным, вспомогательным и организационным процессам.

Первые пять областей ядра знаний SWEBOK по своему содержанию соответствуют следующим процессам:

– разработка требований;
– проектирование;
– конструирование;
– тестирования;
– сопровождение.

Эти процессы задают последовательность задач и действий при разработке разных типов ПС с применением современных методов и средств, которые представлены в ядре знаний.

В табл.2 приведен сопоставительный перечень основных процессов, их задач, приведенных в SWEBOK и ЖЦ стандарте. При этом процессы приобретения и поставки из состава основных процессов исключаются, поскольку они не относятся к процессам разработки программных систем.

Остальные пять областей, которые определены в ядре знаний SWEBOK, по своим функциям соответствуют отдельным вспомогательным и организационным процессам ЖЦ стандарта:

– управление конфигурацией;
– управление инженерией;
– управление качеством
– процесс инженерии;
– методы и средства инженерии ПО;
– управление качеством.

Данные процессы предназначены для управления программным проектом, конфигурацией и методами и средствами обеспечения инженерии программирования, а именно оценки качества процессов, промежуточных результатов, полученных на процессах, и конечного продукта.

Таблица 2

Задачи основных процессов в SWEBOK и ЖЦ

Области–процессы	**Задачи областей SWEBOK**	**Задачи процессов ЖЦ в стандарте**
Разработка Требований	Инженерия требований. Выявления требований. Анализ требований. Спецификация требований. Проверка требований. Управления требованиями.	Подготовка заказа Выявление требований Анализ требований к системе Анализ требований к ПО Описание документа .
Проектирование ПО	Разработка архитектура ПО Структура ПО. Нотация. Анализ качества проектирования. Стратегия и методы проектирования.	Проектирование архитектуры системы Проектирование архитектуры ПО Детальное проектирование ПО. Кодирование и тестирование ПО.
Конструирование ПО	Снижение сложности. Предупреждение отклонений от стиля. Структуризация системы для проверок. Использование внешних стандартов.	Конструирование структуры системы Кодирование элементов структуры и ПО Интеграция элементов. Применение стандартов программной инженерии.
Тестирование ПО	Уровни тестирования. Техники тестирования. Метрики тестирования. Управления тестированием.	Тестирование ПО. Интеграционное тестирование. Квалификационное тестирование. Интеграция системы. Системное тестирование. Установка ПО. Обеспечение приемки ПО.
Сопровождение ПО	Процесс сопровождения. Ключевые вопросы сопровождения. Техники сопровождения.	Инсталляция ПО Внедрение процесса. Анализ проблем и модификаций. Реализация модификаций. Анализ сопровождения. Миграция (перемещение) ПО. Удаление ПО.
Эксплуатация системы	Методы обеспечения эксплуатации системы	Внедрение процесса. Функциональное тестирование. Эксплуатация системы. Поддержка пользователя.

Эти процессы определяют также задачи поиска, обнаружения ошибок и внесения изменений в требования и продукты. В табл.3 приведен перечень областей ядра SWEBOK, их задач и соответствующих задач организационных процессов ЖЦ стандарта.

Таблица 3

Задачи организационных и дополнительных процессов в SWEBOK и ЖЦ

Области – процессы	**Задачи областей SWEBOK**	**Задачи процессов ЖЦ стандарта**
Управление конфигурацией	Процесс управления конфигурацией. Идентификация элементов. Контроль конфигурации. Учет статуса. Аудит. Управления версиями.	Внедрение процесса. Определение конфигурации. Контроль конфигурации. Учет состояния конфигурации. Оценка конфигурации. Управление реализацией и поставкой.
Управление проектом	Организационное управление. Управления процессами и проектом. Планирование проектом. Инженерия измерения ПО. Управление риском.	Инициация и определение области применения. Планирование. Выполнение и контроль. Анализ управления проектом: – технический анализ; – аудит (ревизия).
Управление качеством	Концепция качества ПО. Определение и планирование качеством. Верификация и валидация. Измерение в анализе качества ПО.	Внедрение процесса. Обеспечение производства. Обеспечение качества. Процесс верификации. Процесс валидации. Анализ и оценивание качества.
Методы и средства инженерии	Методы инженерии. Инструменты инженерии.	Процесс усовершенствования: – определение процесса; – оценка процесса; – улучшение процесса.
Процесс инженерии ПО	Инфраструктура процесса. Определения процесса. Измерения процесса. Анализ проекта. Выполнения изменений. Оценки стоимости и затрат.	Внедрение процесса. Создание инфраструктуры. Сопровождение инфраструктуры. Завершение.

Контрольные вопросы и задания

1. Охарактеризуйте понятие модели ЖЦ и назовите их виды.
2..Дайте характеристику каскадной модели.
3. Определите отличительную особенность спиральной модели ЖЦ.
4. Какие общие черты имеют инкрементная и эволюционная модели?
5. Дайте перечень процессов ЖЦ стандарта и назовите их назначение.
6. Как построить новую модель ЖЦ на основе стандарта?
7. Дайте классификацию процессов ЖЦ стандарта.

8. Назовите процессы управления проектом.
9. Назовите процессы управления качеством.
10. Проведите сравнительную оценку модели процессов ЖЦ стандарта 12207 и областей–процессов ядра знаний SWEBOK.

Литература к теме 2

1. *ISO\IEC 12207*: 1995-0801:Informational Technology - Software life cycle processes. // ГОСТ Р ИСО/МЭК 12207-99 Информационная технология. Процессы жизненного цикла программных средств.
2. *Андон Ф.И., Коваль Г.И., Коротун Т.М., Суслов В.Ю.* Основы инженерии качества программных систем.–К: Академпериодика, 2002.–502с.
3. *Иан Соммервил*. Инженерия программного обеспечения. 6 -издание.–Москва–Санкт–Петербург–Киев, 2002.–623с.
4. *С.А.Орлов.* Технологии разработки программного обеспечения. Учебник для Вузов. Питер.–2002.–463с.
5. *Васютович В., Самотохин С., Никифоров Г.* Регламентация жизненного цикла программных средств // iakimov@gost.ru

Тема 3

МЕТОДЫ ОПРЕДЕЛЕНИЯ ТРЕБОВАНИЙ В ПРОГРАММНОЙ ИНЖЕНЕРИИ

Каждая программная система представляет собой определенный преобразователь данных, поведение и свойства которого определяются в процессе создания этой системы, ориентированной на решение некоторой проблемы. К программной системе предъявляются требования в виде соглашений между заказчиком и исполнителем.

В общем случае под *требованиями* к ПС понимают свойства, которыми должна обладать эта система для адекватного выполнения предписанных ей функций. Примерами таких функций могут быть автоматизация присущих персоналу обязанностей, обеспечение руководства организацией информацией, необходимой для принятия решений и др. Т.е., программная система может моделировать достаточно сложную деятельность людей и их организацию, их взаимодействие как между субъектами автоматизируемой области, так с физическим оборудованием компьютерной системы и т.п.[1-5].

3.1 Определение понятий и видов требований

Одна из проблем индустрии программного обеспечения — это отсутствие общепринятых определений терминов, которыми пользуются для описания: требований пользователя, требований к ПО, функциональных требований, системных требований, технологических требований, требований к продукту и бизнес-требований. В разных источниках понятия требований определяются, исходя из разных условий и взглядов на то, что по ним создается. Назовем ряд определений в проблематике требований [2,3].

Требования — это «нечто такое, что приводит к выбору дизайна системы".
Требования – это свойства, которыми должен обладать продукт, чтобы представлять какую-то ценность для пользователей.
Требования – это спецификация того, что должно быть реализовано. В них охарактеризовано описание поведения системы, ее свойства и атрибуты. Они могут быть ограничены процессом разработки системы.

Согласно международного глоссария по терминологии [6] требования включают описание:
1) условий или возможностей, необходимых пользователю для решения поставленных проблем или достижения целей;
2) условий или возможностей, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворить стандартам, спецификациям или другим формальным документам;
3) документированное представление условий или возможностей проектирования системы.

Виды требований
Требования к продукту охватывают требования как пользователей (внешнее поведение системы), так и разработчиков (некоторые скрытые параметры).Термин *пользователи* относится ко всем заинтересованным лицам в создании системы.

Требования к ПО состоят из трех уровней — бизнес-требования, требования пользователей и функциональные требования. Каждая система имеет свои нефункциональные требования.

Требования пользователей (user requirements) описывают цели и задачи, которые пользователям позволит решить система. К способам представления этого вида требований относятся варианты использования, сценарии и таблицы «событие — отклик».

Системные требования (system requirements) обозначают высокоуровневые требования к продукту, которые содержат многие подсистемы или вся система. Из требований для всей системы главными являются функциональные требования к ПО.

Функциональные требования включают описание требований в видам и типам реализуемых функций и документируются в *спецификации требований к ПО* (software requirements specification, SRS), где описано и ожидаемое поведение системы. Спецификация требований к ПО используется при разработке, тестировании, гарантии качества продукта, управлении проектом и его функциями. В дополнение к функциональным требованиям спецификация содержит нефункциональные требования (защита данных, адаптивность, изменчивость и др.), где описаны цели и атрибуты качества.

Атрибуты качества (quality attributes) представляют собой дополнительное описание функций программного продукта, выраженных через описание его характеристик, важных для пользователей или разработчиков. К таким характеристикам относятся легкость и простота использования, легкость перемещения, целостность, эффективность и устойчивость к сбоям. Другие нефункциональные требования описывают внешние взаимодействия между системой и внешним миром, а также ограничения дизайна и реализации. *Ограничения* (constraints) касаются выбора возможности разработки внешнего вида и структуры продукта.

Бизнес-требования (business requirements) содержат высокоуровневые цели организации или заказчиков бизнес–системы. Как правило, их высказывают те, кто финансируют проект, покупатели системы, менеджер реальных пользователей и отдел маркетинга. *Бизнес-правила* (business rules) включают корпоративные политики, правительственные постановления, промышленные стандарты и вычислительные алгоритмы. Они не являются требованиями к ПО, потому что они находятся снаружи границ любой системы ПО. Однако они часто налагают ограничения на выполнение конкретных вариантов использования или на функции системы, подчиняющимся соответствующим правилам. Бизнес-правила становятся источником атрибутов качества, которые реализуются в функциональности.

3.1.2. Анализ и сбор требований

В современных информационных технологиях процесс ЖЦ, на котором фиксируются требования на разработку ПО системы, является определяющим для задания функций, сроков и стоимости работ, а также показателей качества и др.

Анализ и сбор требований является довольно нетривиальной задачей, поскольку в реальных проектах приходится сталкиваться с такими общими трудностями:

– требования не всегда очевидны и могут исходить из разных источников, их не всегда удается ясно выразить словами;

– имеется много различных типов требований на различных уровнях детализации и их число может стать большим, требующим ими управлять;

– требования связаны друг с другом, а также с процессами разработки ПС и постоянно меняются.

Требования имеют уникальные свойства или значения свойств. Например, они не являются одинаково важными и простыми для согласования.

Аналитик системы, который занимается анализом и составлением требований, должен иметь определенные знания ПрО и навыки, чтобы справиться с этими трудностями. Он должен уметь:
– анализировать проблему,
– понимать потребности заказчика и пользователей,
– определять функции системы и требования к ним,
– управлять контекстом проекта и изменением требований.

В требованиях к ПС должны отображаться проблемы системы и фиксироваться реальные потребности заказчика, касающиеся функциональных, операционных и сервисных возможностей разрабатываемой системы. В результате создается договор между заказчиком и исполнителем системы на ее создание.

Здесь цена ошибок и нечетких неоднозначных формулировок очень высока, поскольку время и средства могут расходоваться на ненужную заказчику систему или программу. Для внесения изменений в требования может потребоваться повторное проектирование и, соответственно, перепрограммирование всей системы или отдельных ее частей. Как утверждает статистика, процент ошибок в постановке требований и в определении задач системы превышает процент ошибок кодирования. Это является следствием субъективного характера процесса формулирования требований и отсутствия способов его формализации. К примеру, в США ежегодно расходуется до $ 82 млрд. на проекты, признанные после реализации не соответствующими требованиям заказчиков.

Существуют стандарты на разработку требований на систему и документы, в которых фиксируются результаты создания программного, технического, организационного и др. видов обеспечения автоматизированных систем (АС) на этапах жизненного цикла, включающие. В приложении 2, 3 дается краткое описание ГОСТ 34.601-90 «Стадии и этапы разработки АС» и ГОСТ 34.201-89 «Документация на разработку АС».

В процессе формулирования требований на систему принимают участие:

– представители от заказчика из нескольких профессиональных групп;
– операторы, обслуживающие систему;
– разработчики системы.

Процесс формулирования требований состоит из нескольких подпроцессов? Сбор и анализ требований.

Сбор требований. Источниками сведений о требованиях могут быть:

– цели и задачи системы, которые формулирует заказчик. Для однозначного их понимания разработчику системы необходимо их тщательно осмыслить и согласовать с заказчиком;

– действующая система или коллектив, выполняющий ее функции. Система, которую заказывают, может заменять собою старую систему, переставшую удовлетворять заказчика или действующий персонал. Изучение и фиксация ее функциональных

возможностей дает основу для учета имеющегося опыта и формулирования новых требований к системе. При этом имеется определенная опасность перенесения недостатков старой системы в новую, поэтому нужно уметь отделить новые требования к реализуемой проблеме от заложенных неудачных решений в старой системе.

Таким образом, требования к системе формулируются исходя из:
– знаний заказчика относительно проблемной области, формулирующего свои проблемы в терминах понятий этой области;
– ведомственных стандартов заказчика и требований к среде функционирования будущей системы, ее исполнительских и ресурсных возможностей.

Методами сбора требований являются:

– интервью с носителями интересов заказчика и операторами;
– наблюдение за работой действующей системы с целью отделения ее проблемных свойств от тех, которые обусловлены структурою кадров;
– сценарии (примеров) возможных случаев выполнении ее функций, ролей лиц, запускающих эти сценарии или взаимодействующих с системой во время ее функционирования.

Продукт процесса сбора требований – неформализованное их описание – основа контракта на разработку между заказчиком и исполнителем системы. Такое описание является входом для следующего процесса инженерии требований - анализа требований. Исполнитель этого процесса выполняет дальнейшее уточнение и формализацию требований, а также их документирование в нотации, понятной коллективу разработчиков системы.

Анализ требований. Это процесс изучения потребностей и целей пользователей, классификация и их преобразование к требованиям системы, к ПО, установление и разрешение конфликтов между требованиями, определение приоритетов, границ системы и принципов взаимодействия со средой функционирования. Требования классифицируются по функциональному и нефункциональному принципу, а также по отношению их в внешней или внутренней стороне системы.

Функциональные требования относятся к заданию функций системы или ее ПО, к , способам поведения ПО в процессе выполнения функций и методам преобразования входных данных в результаты.

Нефункциональные требования определяют условия и среду выполнения функций (например, защита и доступ к БД, секретность, взаимодействие компонентов функций и др.).

Разработанные требования специфицируются и отражается в специальном документе, по которому проводится *согласование требований* для достижения взаимопонимания между заказчиком и разработчиком.

Функциональные требования отвечает на вопрос "что делает система", а нефункциональные требования определяют характеристики ее работы (вероятность сбоя системы, защита данных и др.). К основным нефункциональным требованиям, существенным для большинства ПС и выражающих ограничения, актуальные для многих проблемных областей относятся:

– конфиденциальность;
– отказоустойчивость;
– несанкционированный доступ к системе;
– безопасность и защита данных;
– время ожидания ответа на обращение к системе;
– свойства системы (ограничение на память, скорость реакции при обращении к системе и т. п.).

Для большинства этих ограничений может быть зафиксирован спектр характерных понятий – *дескрипторов,* которые используются для наименования и раскрытия смыслового названия. Состав дескрипторов для ряда нефункциональных требований зафиксирован в соответствующих международных и ведомственных стандартах, которые позволяют избежать неоднозначности в их толковании.

Функциональные требования отражают семантические особенности проблемной области, а терминологические расхождения для них являются достаточно существенными. Имеется тенденция к созданию стандартизации понятийного базиса большинства проблемных областей, которые имеют опыт компьютеризации.

Следующий шаг анализа требований - установление их приоритетов и избежание конфликтов между ними.

Продукт процесса анализа – построенная модель проблемы, которая ориентирована на понимание этой модели исполнителем до начала проектирования системы.

К настоящему времени сложилось направление в инженерии программирования – инженерия требования, сущность которой достаточно подробно рассмотрена в соответствующей области знаний ядра SWEBOK и приводится ниже.

3.1.3. Инженерия требований ПО

Инженерная дисциплина анализа и документирования требований к ПО, которая заключается в преобразовании предложенных заказчиком требований к системе в описание требований к ПО, их спецификация и верификация. Она базируется на *модели процесса* определения требований, процессах актеров – действующих лиц, управлении и формировании требований, а также на процессах верификации и повышения их качества.

Модель процесса – это схема процессов ЖЦ, которые выполняются от начала проекта и до тех пор, пока не будут определены и согласованы требования. При этом процессом может быть маркетинг и проверка осуществимости требований в данном проекте.

Управление требованиями к ПО заключается в планировании и контроле выполнения требований и проектных ресурсов в процессе разработки компонентов системы на этапах ЖЦ.
Качество и процесс улучшения требований – это процесс формулировки характеристик и атрибутов качества (надежность, реактивность и др.), которыми должна обладать система и ПО, методы их достижения на этапах ЖЦ и адекватности процессов работы с требованиями.

Управление требованиями к системе – это руководство процессами формирования требований на всех этапах ЖЦ, которое включает управление изменениями и атрибутами требований, отражающими программный продукт, а также проведение мониторинга – восстановления источника требований. Неотъемлемой составляющей процесса управления является т*рассирование требований* для отслеживания правильности задания и реализации требований к системе и ПО на этапах ЖЦ и обратный процесс отслеживания от полученного продукта к требованиям.

При управлении требований выполняются процессы:
– управления версиями требований,
– управление рисками,
– разработка атрибутов требований,
– контроль статуса требований, измерение усилий в инженерии требований,
– другие.

Связь между разработкой и управлением требований представлена на рис.3.1.

Управление рисками состоит оценке, предотвращении и контроле появления риска определения отдельных требований. Проводиться планирование работ на проекте по управлению рисками в разработке требований.

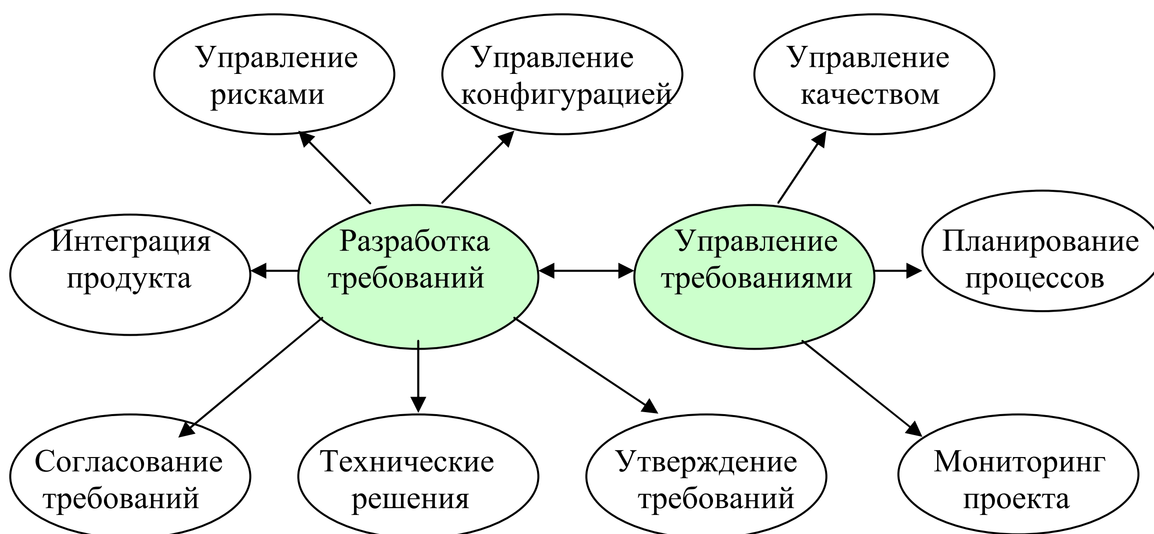

Рис.3.1. Связь между разработкой требований, управления требованиями и другими процессами проекта

3.1.4. Верификация и формализация требований

Сбор требований является начальным и неотъемлемым этапом процесса разработки ПС. Он заключается в определении набора функций, которые необходимо реализовать в продукте. Процесс сбора требований реализуется частично в общении с заказчиком, частично посредством мозговых штурмов разработчиков. Результатом является формирование набора требований к системе, именуемой в отечественной практике техническим заданием.

Фиксация требований (Requirement Capturing), с одной стороны, определяется желаниями заказчика в реализации того или иного свойства. С другой стороны в

процессе сбора требований может обнаружиться ошибка, которая приведет к определенным последствиям, устранение которых заберет непредвиденные ресурсы – дополнительное кодирование, перепланирование.

Ошибка может быть тем серьезней, чем позже она будет обнаружена, особенно если это связано с множеством спецификаций. Поэтому одной из составляющей этапа фиксации требований, наряду со сбором является верификация требований, а именно проверка их на непротиворечивость и полноту.

Автоматизированная верификация требований может производиться лишь после спецификации или формализации требований.

Спецификация требований к ПО – процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартом качества ISO/IEC 12119-94, которые будут учитываться при создании программного продукта на этапах ЖЦ ПО. В спецификации требований отражается структура ПО, требования к функциям, показателям качеству, которых необходимо достигнуть, и к документации. В спецификации задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные требования, нефункциональные требования и требования к взаимодействию с другими компонентами (БД, СУБД, протоколы сети и др.).

Формализация включает в себя определение компонентов системы и их состояний; правил взаимодействия компонентов и определения условий в формальном виде, которые должны выполняться при изменении состояний компонентов. Для формального описания поведения системы используются языки инженерных спецификаций, например, UML. В качестве формальной модели для описания требований используются базовые протоколы, которые позволяют использовать дедуктивные средства верификации в сочетании с моделированием поведения систем путем трассирования.

Валидация (аттестация) требований - это проверка требований, изложенных в спецификации для того, чтобы убедиться, что они определяют данную систему и отслеживание источников требований. Заказчик и разработчик ПО проводят экспертизу сформированного варианта требований с тем, чтобы разработчик мог далее проводить разработку ПО. Одним из методов аттестации является прототипирование, т.е. быстрая отработка отдельных требований на конкретном инструменте и исследование масштабов изменения требований, измерение объема функциональности и стоимости, а также создание моделей оценки зрелости требований.

Верификация требований – это процесс проверки правильности спецификаций требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие стандартам. В результате проверки требований делается согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, а также возможность продолжить проектирование ПО.

3.2. Объектно-ориентированная инженерия требований

Структурирование проблемы на отдельные компоненты и обеспечение способов их взаимодействия определяют понятность и "прозрачность" описания требований, полученных в результате процесса анализа.

Типы составляющих компонентов и правила их композиции определяются в программной инженерии как *архитектура системы*. Процесс декомпозиции проблемы, определяющей архитектуру системы, называют *парадигмой программирования,* базирующейся на двух широко распространенных моделях: функции-данные и объектная модель.

Модель функции-данные исторически появилась первой. Согласно этой модели проблема декомпозируется на последовательность функций и обрабатываемых с их помощью данных. Элементами композиции служат данные и функции над ними, их представления должны быть согласованы между собой. Если изменяются некоторые данные, то пересматриваются функции, которые их обрабатывают, и определяются пути их изменений. Т.е. внесение локальных изменений в постановку проблемы требует ревизии всех данных и функций для подтверждения того, что на них не повлияли внесенные изменения.

Объектно-ориентированный подход к разработке программных систем такого недостатка не имеет. В нем общее видение решения проблемы формирования требований осуществляется исходя из следующих постулатов:

– мир составляют объекты, которые взаимодействуют между собой;
– каждому объекту присущ определенный состав свойств или атрибутов, который определяется своим именем и значениями;
– объекты могут вступать в отношения друг с другом;
– значения атрибутов и отношения могут с течением времени изменяться;
– совокупность значений атрибутов конкретного объекта в определенный момент времени определяет его состояние;
– совокупность состояний объектов определяет состояние мира объектов;
– мир и его объекты могут находиться в разных состояниях и порождать некоторые события;
– события могут быть причиною других событий или изменений состояний.

Каждый объект может принимать участие в определенных процессах, разновидностями которых есть:

– переходы из одного состояния в другое под влиянием соответствующих событий;
– возбуждение определенных событий или посылка сообщений другим объектам;
– операции, которые могут выполнять объекты;
– возможные совокупности действий, которые задают его поведение;
– обмен сообщениями.

Объект это определенная абстракция данных и поведения. Множество экземпляров с общим набором атрибутов и поведением составляет класс объектов. Определение класса связано с известным принципом *сокрытия информации*, суть которого можно сформулировать так: сообщайте пользователю только то, что ему нужно. Этот принцип имеет ряд преимуществ:

– пользователь избавлен от необходимости знать лишнее;
– то, что ему не сообщили, он не испортит (защита от намеренных или случайных неправомерных действий;
– все, о чем не знает пользователь, можно изменять.

Таким образом, определение объектов в соответствии с данным принципом состоит из двух частей - видимой и невидимой. Видимая часть содержит все сведения, которые требуется для того, чтобы взаимодействовать с объектом и называется *интерфейса объекта.* Невидимая часть содержит подробности внутреннего устройства объекта, которые "инкапсулированы" (т.е. находятся словно бы в капсуле). Так, например, если объектом является некоторый прибор, который регистрирует показатели температуры, то к видимой его части относится операция показа значения температуры.

Другим важным свойством определения объектов является *наследование*. Один класс объектов наследует другой, если он полностью вмещает все атрибуты и поведение наследуемого класса, но имеет еще и свои атрибуты и (или) поведение. Класс, который наследуют свойства другого, называют *суперклассом*, а класс, которого наследует, называют *подклассом.* Наследственность фиксирует общие и отличающиеся черты объектов и позволяет явно выделять компоненты проблемы, которые можно использовать в ряде случаев при построении нескольких классов–наследников.

Классы могут образовывать иерархии наследников произвольной глубины, где каждый отвечает определенному уровню абстракции, являясь обобщением класса-наследника и конкретизацией класса, который наследует его самого. Например, класс "число" в качестве наследников имеет подклассы: "целые числа", "комплексные числа" и "действительные числа". Все эти подклассы наследуют операции суперкласса (сложения и вычитания), но каждый из них имеет свои особенности выполнения этих операций.

При объектно–ориентированном подходе модели определяются через взаимодействие определенных объектов. В модели требований фигурируют объекты, взаимодействие которых определяет проблему, решаемую с помощью программной системы, а в других моделях (модели проекта, моделях реализации и тестирования) заданный принцип взаимодействия объектов определяет сущность решения этой проблемы (модели проекта и реализации) или проверки достоверности решения (модель тестирования).

В настоящее время предложен ряд современных методов объектно-ориентированного анализа требований, объектно-ориентированного проектирования программ, объектно-ориентированного программирования (С++, JAVA). Наибольшую ценность среди них имеет проблема согласованности между ними.

Если удается установить соответствие между объектами указанных моделей на разных стадиях (процессах) жизненного цикла продукта, то они позволяют провести *трассирование требований*, т.е. проследить за последовательной трансформацией требований объектов на этих стадиях. Трассирование заключается в контроле трансформаций объектов при переходе от этапа к этапу с учетом внесения изменений во все наработанные промежуточные продукты разных стадий разработки и ее завершения.

Концептуальное моделирование проблемы системы происходит в терминах взаимодействия объектов:

– онтология домена определяет состав объектов домена, их атрибуты и взаимоотношения, а также оказываемые услуги;

– модель поведения определяет возможные состояния объектов, инциденты, события, инициирующие переходы из одного состояния в другое, а также сообщения, которые объекты посылают друг другу;
– модель процессов определяет действия, которые выполняют объекты.

Все объектные методы имеют в своем составе приведенные модели, отличающиеся своими нотациями и некоторыми другими деталями.

3.2.1. Метод инженерии требований А. Джекобсона

Данный метод является методом с последовательным выявлением объектов, существенных для домена проблемной области [3]. Все методы анализа предметной области в качестве первого шага выполняют выявление объектов. Правильный выбор объектов обуславливает понятность и точность требований. Рассматриваемый метод Джекобсона дает рекомендации относительно того, с чего начинать путь к пониманию проблемы и поиску существенных для нее объектов.

Автор назвал свой метод как базирующийся на вариантах (примерах или сценариях – use case driven) системы, которую требуется построить. В дальнейшем термин сценарий используется для обозначения варианта представления системы.

Сложность проблемы обычно преодолевается путем декомпозиции ее на отдельные компоненты с меньшей сложностью. Большая система может быть сложной, чтобы представить ее компоненты программными модулями. Поэтому разработка системы с помощью данного метода начинается с осмысления того, для кого и для чего создается система.

Сложность общей цели выражается через отдельные подцели, которым отвечают функциональные или нефункциональные требования и проектные решения. Цели являются источниками требований к системе и способом оценки этих требований, путем выявления противоречий между требованиями и установления зависимостей между целями.

Цели можно рассматривать, как точку зрения отдельного пользователя или заказчика или среды функционирования системы. Цели могут находиться между собою в определенных отношениях, например, конфликтовать, кооперироваться, зависеть одна от другой или быть независимыми.

Следующим шагом после выявления целей является определение носителей интересов, которым отвечает каждая цель, и возможные варианты удовлетворения составных целей в виде сценариев работы системы. Последние помогают пользователю получить представление о назначении и функциях системы. Эти действия соответствуют первой итерации определения требований к разработке.

Таким образом, производится последовательная декомпозиция сложности проблемы в виде совокупности целей, каждая из которых трансформируется в совокупность возможных вариантов использования системы, которые трансформируются в процессе их анализа в совокупность взаимодействующих объектов.

Определенная цепочка трансформации (проблема – цели – сценарии – объекты) отражает степень концептуализации в понимании проблемы, последовательного

снижения сложности ее частей и повышения уровня формализации моделей ее представления.

Трансформация обычно выражается в терминах базовых понятий проблемной области, активно используется для представления актеров и сценариев, или создается в процессе построения такого представления.

Каждый сценарий инициируется определенным пользователем, являющимся носителем интересов. Абстракция определенной роли личности пользователя–инициатора запуска определенной работы в системе, представленной сценарием, и обмена информацией с системой - называется *актером*. Это абстрактное понятие обобщает понятие действующего лица системы. Фиксация актеров можно рассматривать, как определенный шаг выявления целей системы через роли, которые являются носителями целей и постановщиками задач, для решения которых создается система.

Актер считается внешним фактором системы, действия которого носят недетерминированный характер. Этим подчеркивается разница между пользователем как конкретным лицом и актером–ролью, которую может играть любое лицо в системе.

В роли актера может выступать и программная система, если она инициирует выполнение определенных работ данной системы. Таким образом, актер – это абстракция внешнего объекта, экземпляр которого может быть человеком или внешней системой. В модели актер представляется классом, а пользователь – экземпляром класса. Одно лицо может быть экземпляром нескольких актеров.

Лицо в роли (экземпляр актера) запускает операции в системе, соотнесенные с поведением последовательности транзакций системы и называется *сценарием*. Когда пользователь, как экземпляр актера, вводит определенный символ для старта экземпляра соответствующего сценария, то это приводит к выполнению ряда действий с помощью транзакций, которые заканчиваются тогда, когда экземпляр сценария находится в состоянии ожидания входного символа.

Экземпляр сценария существует, пока он выполняется и его можно считать экземпляром класса, в роли которого выступает описание транзакции.

Сценарий определяет протекание событий в системе и обладает состоянием и поведением. Каждое взаимодействие между актером и системой это новый сценарий или объект. Если несколько сценариев системы имеют одинаковое поведение, то их можно рассматривать как класс сценариев. Вызов сценария – это порождение экземпляра класса. Таким образом, сценарии – это транзакции с внутренним состоянием.

Модель системы по данному методу основывается на сценариях и внесение в нее изменений должно осуществляться повторным моделированием актеров и запускаемых ими сценариев. Такая модель отражает требования пользователей и легко изменяется по их предложению. Сценарий – это полная цепочка событий, инициированных актером, и спецификация реакции на них. Совокупность сценариев определяет все возможные варианты использования системы. Каждого актера обслуживает своя совокупность сценариев.

Можно выделить базовую цель событий, существенную для сценария, а также альтернативные, при ошибках пользователя и др.

Для задания модели сценариев используется специальная графическая нотация, со следующими основными правилами:

– актер обозначается иконой человека, под которой указывается название;
– сценарий изображается овалом, в середине которого указывается название иконы;
– актер связывается стрелкой с каждым запускаемым им сценарием.

На рис. 3.2 представлен пример диаграммы сценариев для клиента банка, как актера, который запускает заданный сценарий. Для достижения цели актер обращается не к кассиру или клерку банка, а непосредственно к компьютеру системы.

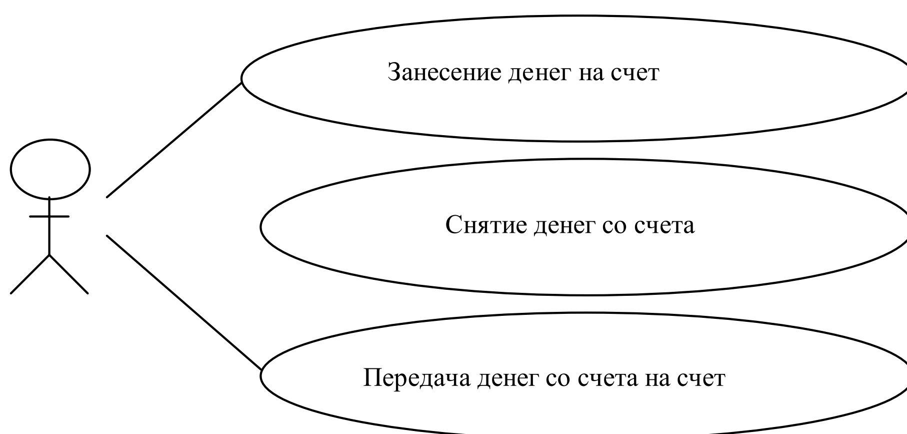

Рис.3.2. Пример диаграммы сценариев для клиента банка

Все сценарии, которые включены в систему, обведены рамкой, определяющей границы системы, а актер находится вне рамки, являясь внешним фактором по отношению к системе.

Отношения между сценариями. Между сценариями можно задавать отношения, которые изображаются на диаграмме сценариев пунктирными стрелками с указанием названия соответствующих отношений.

Для сценариев определены два типа отношений:

Отношение "расширяет" означает, что функции одного сценария является дополнением к функциям другого, и используется когда имеется несколько вариантов одного и того же сценария. Инвариантная его часть изображается как основной сценарий, а отдельные варианты как расширения. При этом основной сценарий является устойчивым, не меняется при расширении вариантов функций и не зависит от них.

Типовой пример отношения расширения: моделирование необязательных фрагментов сценариев приведен на рис.3.3. При выполнении сценария расширение прерывает

выполнение основного расширяемого сценария, причем последний не знает, будет ли расширение и какое. После завершения выполнения сценария расширение будет продолжено путем выполнения основного сценария.

Отношение "использует" означает, что некоторый сценарий может быть использован как расширение несколькими другими сценариями.

Оба отношения – инструменты определения наследования, если сценарии считать объектами. Отличие между ними состоит в том, что при расширении функция рассматривается как дополнение к основной функции и может быть понятной только в паре с ней, тогда как в отношении "использует " дополнительная функция имеет самостоятельное определение и может быть использована всюду.

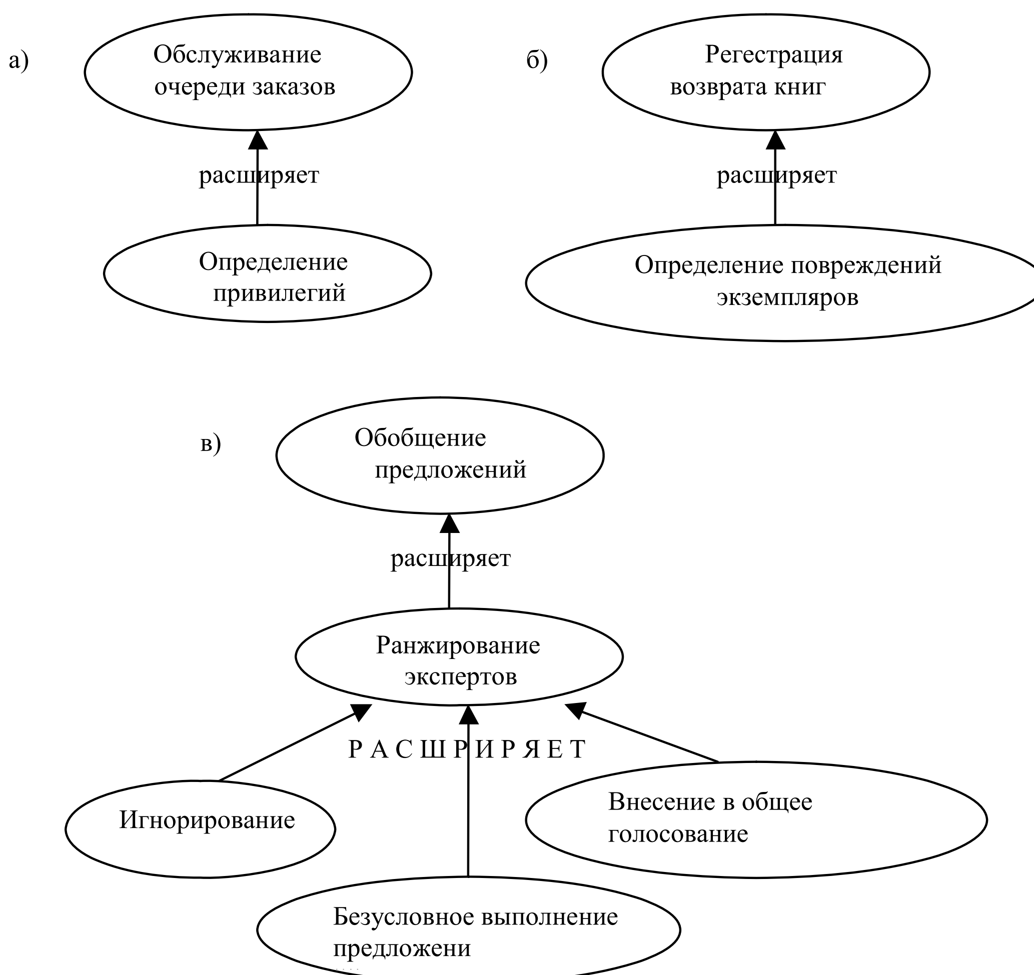

Рис.3.3. Примеры расширения сценариев

На рис. 3.4. показано, что сценарий "сортировать" связан отношением "использует" с несколькими сценариями.

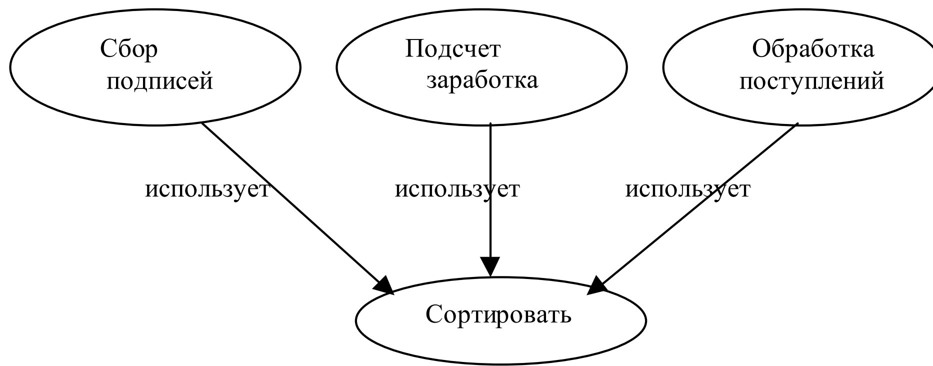

Рис.3.4. Пример отношения "использует "

Таким образом, продуктом первой стадии инженерии требований (сбора требований) является модель требований, которая состоит с трех частей:

1) онтология домена;
2) модель сценариев, называемая диаграммой сценариев;
3) описание интерфейсов сценариев.

Модель сценариев сопровождается неформальным описанием каждого из сценариев. Нотация такого описания не регламентируется. Как один из вариантов, описание сценария может быть представлено последовательностью элементов:

– название, которое помечает сценарий на диаграммах модели требований и служит средством ссылки на сценарий;
– аннотация (краткое содержание в неформальном представлении);
– актеры, которые могут запускать сценарий;
– определение всех аспектов взаимодействия системы с актерами (возможные действия актера и их возможные последствия), а также запрещенных действий актера;
– предусловия, которые определяют начальное состояние на момент запуска сценария, необходимое для успешного выполнения;
– функции, которые реализуются при выполнении сценария и определяют порядок, содержание и альтернативу действий, выполняемых в сценарии;
– исключительные или нестандартные ситуации, которые могут появиться при выполнении сценария и потребовать специальных действий для их устранения (например, ошибка в действиях актера, которую способна распознать система);
– реакции на предвиденные нестандартные ситуации;
– условия завершения сценария;
– постусловия, которые определяют конечное состояние сценария при его завершении.

На дальнейших стадиях сценарий трансформируется в сценарий поведения системы, т.е. к приведенным элементам модели добавляются элементы, связанные с конструированием решения проблемы и нефункциональными требованиями:

– механизмы запуска сценария (позиции меню);
– механизмы ввода данных;
– поведение при возникновении чрезвычайных ситуаций.

Следующим шагом процесса проектирования является преобразование сценария в описание компонентов системы и проверка их правильности.

Описание интерфейсов делается неформально, они определяют взгляд пользователя на систему, с которым необходимо согласовать требования, собранные на этапе анализа системы и определения требований. Все вопросы взаимодействия отмечаются на панели экрана для каждого из актеров и их элементы (меню, окна ввода, кнопки - индикаторы и т.п.).

Построение прототипа системы, моделирующего действия актера, помогает отработать детали и устранить недоразумения между заказчиком и разработчиком.

3.2.2. Модель анализа требований. Определение объектов

Модель требований дает обобщенное представление о будущих услугах системы для ее клиентов (актерам). Полученное представление является предметом анализа с целью дальнейшего структурирования проблемы. Основу составляет объектная архитектура, результатом структурирования которой должна быть совокупность объектов, полученная путем последовательной декомпозиции каждого из сценариев на объекты с действиями сценария, а также их взаимодействие, определяемое функциональностью системы.

Таким образом, стратегия выбора объектов в системе базируется на следующих принципах:

– изменение требований неизбежно;
– объект модифицируется вследствие изменения соответствующих требований к системе;
– объект должен быть устойчивым к модификации системы (локальные изменения отдельного требования, должны повлечь за собой изменения как можно меньшего количества объектов).

Исходя из этих принципов, в данном методе различаются типы объектов в зависимости от их способности к изменениям, система структурируется согласно следующих критериев:

– наличие информации для обработки системы (для обеспечения ее внутреннего состояния);
– определение поведения системы;
– презентация системы (ее интерфейсов с пользователями и другими системами).

Выбор критериев обусловлен экспертными исследованиями динамики изменений действующих систем.

Для каждого критерия функциональности системы имеется совокупность объектов, с помощью которых локализуются требования к наиболее стабильным фрагментам.

Рассматривается три типа объектов:
– объекты-сущности;
– объекты интерфейса;
– управляющие объекты.

Объекты-сущности моделируют в системе долгоживущую информацию, хранящуюся после выполнения сценария. Обычно, им соответствуют реальные сущности, которые находят свое отображение в БД. Большинство из них может быть выявлено из

анализа онтологии проблемной области, но во внимание берутся только те из них, на которые ссылаются в сценариях.

Объекты интерфейса включают в себя функциональности, зависимые непосредственно от окружения системы и определенных в сценарии. С их помощью актеры взаимодействуют со сценариями системы, их задачей является трансляция информации, которую вводит актер, в события, на которые реагирует система, и обратная трансляция событий, вырабатывающая системой, в вывод для актера. Такие объекты определяются путем анализа описаний интерфейсов сценариев модели требований и анализа действий актеров по запуску каждого из соответствующих ему сценариев.

Управляющие объекты - это объекты, которые превращают информацию, введенную объектами интерфейса и представленную объектами-сущностями, в информацию, что выводится интерфейсными объектами. Они являются своеобразным "клеем" для соединения объектов, связывая цепочки событий и задавая взаимодействие объектов.

Такое распределение служит целям локализации изменений в системе. При преобразовании модели требований в модель анализа каждый сценарий разбивается на эти три типа объектов. При этом один и тот же объект может присутствовать в нескольких сценариях, и важно распознать такие объекты, чтобы унифицировать их функции и определить их как единый объект. Критерий распознавания состоит в том, что если в различных сценариях имеется ссылка на один и тот же экземпляр объекта, то это один и том же объект.

После выделения объектов, формируется базис архитектуры системы как совокупности взаимодействующих объектов, для каждого из которых можно установить связь с соответствующим сценарием модели требований. После чего провести трассирование требований, идя от модели требований к модели анализа.

Атрибуты объектов представлены прямоугольниками, объединенными прямой линией с символом объекта, при этом на линии указывается название атрибута, а в прямоугольнике – его тип.

Между объектами определяются ассоциации, которые изображаются одной– или двунаправленной стрелкою, на которых указываются названия ассоциаций типа:

– взаимодействует,
– составляется с,
– выполняет роль,
– наследует,
– расширяет,
– использует.

Эти ассоциации существенно отличаются от ассоциаций в моделях данных. Последние нацелены преимущественно на осуществление навигации в БД, тогда как ассоциации определяют взаимодействие объектов.

Исходя из известного метода анализа требований И. Джекобсона на стадии анализа определяются:

– онтология домена;

– модель сценариев;
– неформальное описание сценариев и актеров;
– описание интерфейсов сценариев и актеров;
– диаграммы взаимодействия объектов сценариев.

Полученные требования трассируются, после чего проводится реализация функций системы на следующих этапах ЖЦ (см. тема 4, 5).

3.3. Классификация требований

Формируемые требования к системе разделены на две категории:
– функциональные требования, которые отображают возможности проектируемой системы;
– нефункциональные требования, которые отображают ограничения, определяющие принципы функционирования системы и доступа к данным системы пользователей.

Первая из приведенных категорий дает ответ на вопрос "что делает система", а вторая определяет характеристики ее работы, например, что вероятность сбоев системы на некотором промежутке времени, доступ до ресурсов системы разных категорий пользователей и др.

Функциональные требования характеризуют функции системы или ее ПО, а также способы поведения ПО в процессе выполнения функций и методы передачи и преобразования входных данных в результаты. *Нефункциональные требования* определяют условия и среду выполнения функций (например, защита и доступ к БД, секретность и др.). Разработка требований и их локализация завершается на этапе проектирования архитектуры и отражается в специальном документе, по которому проводится окончательное согласование требований для достижения взаимопонимания между заказчиком и разработчиком.

Функциональные требования связаны с семантическими особенностями ПрО, для которой планируется разработка ПС. Важным фактором является проблема использования соответствующей терминологии при описание моделей ПрО и требований к системе. Одним из путей ее решения является стандартизации терминологии для нескольких ПрО (например, для информационных технологий, систем обеспечения качества и др.). Тенденция к созданию стандартизированного понятийного базиса для большинства ПрО отражает важность этой проблемы в плане обеспечения единого понимания терминов, используемых в документах, описывающих требования к системе и к ПО.

Нефункциональные требования могут иметь числовой вид (например, время ожидания ответа, количество обслуживаемых клиентов, БД данных и др.), а также содержать числовые значения показателей надежности и качества работы компонентов системы, период смены версий системы и др. Для большинства ПС, с которыми будут работать много пользователей, требования должны выражать такие ограничения на работу системы:
– конфиденциальность;
– отказоустойчивость;
– одновременность доступа к системе пользователей;
– безопасность;
– время ожидания ответа на обращение к системе;

– ограничения на исполнительские функции системы (ресурсы памяти, скорость реакции на обращение к системе и т.п.);
– регламентации действующих стандартов, упрощающих процессов организации формирования требований и менеджмента.

Иными словами, для каждой системы формулируются нефункциональные требования, относящиеся к защите от несанкционированного доступа, к регистрации событий системы, аудиту, резервному копированию, восстановлению информации. Эти требования на этапе анализа и проектирования структуры системы должны быть определены и формализованы аналитиками. Для обеспечения безопасности системы должны быть определены категории пользователей системы, которые имеют доступ к тем или иным данным и компонентам.

Определяются объекты и субъекты защиты. На этапе анализа системы решается вопрос, какой уровень защиты данных необходимо предоставить для каждого из компонентов системы, выделить критичные данные, доступ к которым строго ограничен. Для этого применяется система мер по регистрации пользователей, т.е. идентификации и аутентификации, а также реализуется защита данных, ориентированная на регламентированный доступ к объектам данных (например, к таблицам, представлениям). Если же требуется сделать ограничение доступа к данным (например, к отдельным записям, полям в таблице), то в системе должны предусматриваться определенные средства (например, мандатная защита).

Для обеспечения восстановления и хранения резервных копий БД, архивов баз данных изучаются возможности, предоставляемые СУБД, и рассматриваются способы обеспечения требуемого уровня бесперебойной работы системы, а также организационные мероприятия, отображаемые в соответствующих документах по описанию требований к проектируемой системе.

В целях описания нефункциональных (защита, безопасность, аутентификация и др.) требований к системе и фиксации сведений о способности компонента обеспечивать несанкционированный доступ к нему неавторизованных пользователей, языки спецификаций компонентов дополнились средствами описания нефункциональных свойств. Эта группа свойств целиком связана с сервисом среды функционирования компонентов, ее способностью обеспечивать меры борьбы с разными угрозами, которые поступают извне от пользователей, не имеющих прав доступа к некоторым данным или ко всем данным системы.

Процесс формализованного описания функциональных и нефункциональных требований, а также требований к характеристикам качества с учетом стандарта качества ISO/IEC 9126–94 будет уточняться на этапах ЖЦ ПО и специфицироваться. В спецификации требований отражается структура ПО, требования к функциям, качеству и документации, а также задается архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные, нефункциональные требования и требования к взаимодействию с другими компонентами и платформами (БД, СУБД, сеть и др.).

3.4. Трассирование требований

Одной из главных проблем сбора требований является проблема изменения требований. Требования появляются в процессе общения между группой заказчиков и аналитиками системы в виде интервью, обсуждений, которые не приносят желаемого результата. Это объясняется тем, что составляющих элементов требований

последовательно изменяются, благодаря чему их содержание и форма постепенно становятся более точными и полными, т.е. соответствуют действительности [6].

Инструменты трассировки поддерживают развитие и обработку требований с сохранением их описания и внутренних связей между ними. Трассировка помогает проверять особенности системы на спецификациях требований, выявлять источники разнообразных ошибок и управлять изменениями требований. Если требования разрабатывались в объектной ориентации, то объектами трассировки являются классы и суперклассы и поддерживаемые отношения между ними.

Некоторые методы трассировки базируются на формальных спецификациях отношений (фреймы, соглашения сотрудничества и др.), другие ограничиваются описаниями действий, ситуаций, контекста и возможных решений.

Трассировку можно описать исходя из следующего:

1) требования изменяются во время функционирования системы;
2) возникновение требований и их расположение зависит от деталей практической ситуации и контекста их возникновения (требования можно изменить, изменяя эти детали);
3) трассировка требований должна поддерживаться и изменятся на протяжении всего ЖЦ программного продукта (т.к. изменяются сами требования, необходимо проводить изменение и промежуточных результатов, полученных при анализе, спецификации, кодировке и т.д.);
4) для удобства трассировки использовать иерархическую структуру связей между требованиями, основу которой составляет информация об атрибутах требований.

Чтобы принять решение о возможных модификациях, необходимо иметь достаточно информации о частях и связях между ними. Более того, различные аспекты требований могут быть по-разному представлены и изменены их контексты путем персонального вмешательства аналитиков или заказчика.

Механизмы трассировки должны учитывать следующее:
1) вместо простых связей вводить более сложные отношения (например, транзитивное отношение для выделения цепочек связей) или вводить специфические отношения;
2) использовать сложные и гибкие пути трассировки;
3) поддержать базы данных объектов трассировки и отношений между ними.

Желательно наличие механизмов поддержки возможностей объектно-ориентированного программирования, операций над классами, а также механизмов унификации функций и отношений (1:1, 1:N, N:M), уничтожение и изменение связей, установка стандартных связей.

Трассировка может быть выборочной для определенных объектов, беспорядочной проверкой объектов, связанных с другими объектами, а также с возможными переходами от одного объекта к другим. Она проводится с использованием механизмов поддержания контекста и отношений, соответствующих данной ситуации (например, трассировка с регулярными выражениями, когда контекст может быть изменен только при модификации соответствующего регулярного выражения).
Человеческий фактор. Любой процесс постановки требований напрямую связан с умением людей контактировать друг с другом, кооперироваться или эффективно

распределять разноплановые производственные функции между собой. Необходимо уметь достигнуть соглашения в спорных вопросах, касающихся дизайна или технических решений. Организационные функции не должны выходить за рамки понимания проблемы. Важно чтобы лица, работающие над проектом на разных уровнях, имели возможность эффективно общаться друг с другом.

Технологии проекта должны обращать внимание на динамику людских отношений в коллективе. Они должны способствовать эффективному достижению согласия и управления разногласиями, давать возможность снижать сложность отношений в группе сотрудников, работающих над проектом, особенно в повседневной работе при создании высококачественного продукта.

Наиболее привычной и результативной моделью отношений между заказчиком и проектировщиком является модель типа «учитель – ученик». На практике заказчик наблюдает за работой проектировщика системы. Когда заказчик его не беспокоит, проектировщик обращает внимание на примеры и сам отвечает себе на возникающие вопросы. В порядке исключения, заказчик может остановить свою работу, обдумать поставленный вопрос для пояснения и удовлетворения проектировщика.

Это даже помогает разработчику разобраться в деталях своей работы и избавляет его от описания излишних абстракций. Проектировщик должен сформировать свои независимые идеи относительно проектировки будущей системы и постоянно консультироваться с заказчиком, что избежать ошибок на самых ранних этапах проектирования системы Использование данной модели на практике возможно только при хороших взаимоотношениях между заказчиком и исполнителем проекта.

Контрольные вопросы и задания

1. Как называется этап ЖЦ разработки ПО, на котором фиксируется контракт между заказчиком и исполнителем разработки?
2. Назовите действующих лиц процесса формирования требований.
3. Назовите источники сведений о требованиях.
4. Какова последовательность шагов по использованию действующей системы в новой разработке?
5. Назовите категории классификации требований.
6. Цели и составляющие концептуального моделирования проблемы.
7. Что определяет онтология концептуального моделирования проблемы?
8. Объясните суть отношений, с помощью которых строятся понятия: обобщение, декомпозиция, абстракция, ассоциация.
10. Назовите элементы объектно-ориентированного моделирования программных систем.
11. В чем состоит принцип сокрытия информации?
12. Определите концепция модели сценариев для сбора требований.
13. Дайте пояснения для нотации диаграммы сценариев и базовых отношений в них.
14. Назовите основные типы объекты модели.
15. Приведите задачи трассировки требований.
16. Расскажите о принципах взаимоотношений между заказчиком и разработчиком требований к системе.

Литература к теме 3

1. *Леонов И.В.* Введение в методологию разработки программного обеспечения при помощи Rational Rose. Требования к системе и способы использования// igorvleonov@esc.ru

2. *Вигерс К.И.* Разработка требований к ПО. Москва, 2004.– Русская редакция Microsoft.–575с.
3. *Pamela Zave, Michael Jackson,* «Four Dark Corners of Requirements Engineering», ACM Transactions on Software Engineering, January 1997, №1.
4. *Jacobson I., Griss M., Jonsson P*. Software Reuse. - N.-Y. - Addиson-Wesley, 1997. - 497р.
5. http:/www.rational.com.uml.html
6. *Francisco A. C. Pinheiro, Joseph A. Goguen,* «An Object - Oriented tool for Tracing Requirements», «Software», Mach 1996, № 3.

Тема 4

МЕТОДЫ АНАЛИЗА И ПОСТРОЕНИЯ МОДЕЛЕЙ ПрО

Разработка ПС начинается с анализа предметной области (ПрО), которая автоматизируется, для выделения в ней объектов, отношений между ними и операций пополнения объектов модели ПрО новыми детализированными свойствами и характеристиками. Наиболее разработанным методом программирования является метод проектирования на основе стандарта и многообразие методов, построенных на основе объектно-ориентированного подхода и предназначенных для определения объектных моделей (ОМ), близких к концептуальным моделям, и позволяющих на их основе проводить проектирование структуры или архитектуры системы.

На этапе анализа ПрО при построении модели ОМ проводится выявление функциональных задач и требований к их реализации и выполнению. Требования формируются разработчиком и заказчиком совместно, они документируются и утверждаются, являясь основой реализации архитектуры системы.

Далее рассматривается два направления проектирования архитектуры системы, основанные на применении:

1) объектно-ориентированных методов (например, OOAS, OOA, OMT и др.);
2) стандартных и общесистемных методов (Гост ГОСТ 34.601-90, ER-моделирование и др.).

4.1. Объектно–ориентированные методы анализа и построения моделей ПрО

Наиболее распространенными методами объектно-ориентированного анализа ПрО, широко используемые в практике программирования являются следующие:

– метод OOAS, позволяющий выделить объекты реального мира ПрО, определить сущности, свойства и отношения объектов и из них построить информационную модель, модель состояний объектов и процессов представления потоков данных (dataflow) [1];
– метод OOA позволяет провести анализ, определить требований к ПрО, специфицировать потоки данных в ПрО в виде диаграммной модели [2];
– метод SD структурного проектирования структуры системы, данных и программы преобразования входных данных в выходные с помощью структурных карт Джексона [3-5];
– методология OOAD позволяет построить модели ПрО с помощью ER-моделирования, понятий и их отношений с использованием структурных диаграмм, диаграмм «сущность-связь» и матрицы информационного управления [6, 7];
– объектное OMT моделирование объектной, динамической, функциональной моделей и взаимодействия объектов [8, 9];
– метод Г.Буча, включающий классы, суперклассы и операции наследования, полиморфизма и упрятывания информации об объектах, дополненный вариантами использования Джекобсона для задания сценариев работы системы и задач ПрО и диаграммными средствами Румбаха, в результате имеем UML-метод для анализа ПрО и представления архитектуры системы с помощью набора диаграмм взаимодействующих объектов [10, 11];

– метод построения объектной эталонной модели в CORBA и предоставления набора сервисных системных компонентов общего пользования для обеспечения функционирования объектных компонентов распределенных приложений [12, 13];
– метод генерации частей систем из готовых компонентов (generative programming), объединивший в себе возможности ООП, компонентного и аспектного проектирования и развитый средствами многоразового использования отдельных членов семейства программ ПрО, функциональные и нефункциональные требования представляются в модели характеристик и др. [12].

Наиболее используемым практическим методом проектирования о*бъектной модели* ПрО*,* является метод, реализованный в системе CORBA. Основным элементом модели является объект, который именуется и инкапсулирует некоторую сущность ПрО. Объекту соответствует одна или несколько операций обращения к методу объекта. Объекты группируются в типы, а их экземпляры в подтипы/супертипы. Объекты инкапсулируют методы реализации, которые невидимы во внешнем интерфейсе, операции объектов вызывают этот метод для выполнения. Под влиянием этих операций объект получает некоторое состояние. Специализация типа определяется постепенно на этапах стратегии, анализа, проектирования и реализации объектов. Взаимодействие объектов осуществляется брокером объектных запросов, операций.

Общая характеристика разновидностей объектно–ориентированных методов показывает, что они имеют много общих черт (например, ER-моделирование), и свои специфические особенности. Их разработчики вводили в метод необходимые новые понятия, которые семантически совпадали с другими, ранее определенными.

4.1.1. Основные понятия анализа ПрО

Предлагаемый метод основан на объектно-ориентированном подходе, теории множеств и предназначен для выявления сущностей ПрО, формализации представления объектов и их отношений. При этом при построении концептуальной модели используются следующие понятия.

Объекты ПрО - это абстрактные образы ПрО с множеством свойств и характеристик, их определение зависит от уровня абстракции и совокупности полученных о них знаний. Спецификация объекта включает:

<имя объекта > <концепт>,

где <имя объекта> – идентификатор, строка из литер и десятичных чисел;
< концепт > – некоторый денотат, определяющий объект реального мира в соответствии с интерпретацией сущности моделируемой ПрО.

Предметная область (домен) – это совокупность объектов и связей, которые представляются описанием свойств и характеристик, специфических для ПрО, и задач, которые выполняются в системе. Пространство ПрО делится на пространство задач и решений. Пространство задач – это сущности, концепты, понятия ПрО, а пространство решений – это множество функциональных компонентов, которым соответствуют задачи ПрО, описанные с помощью понятий и концептов.

Модель ПрО строится с использованием словаря терминов, точных определений терминов этого словаря, характеристик объектов и процессов, которые протекают в системе, а также множества синонимов и классифицированных логических взаимосвязей между этими терминами.

Концептуальная модель – это модель ПрО из сущностей и отношений, разработанная без ориентации на программные и технические средства выполнения задач ПрО, которые будут добавляться в дальнейшем в процессе проектирования системы.

Под концептом понимается абстрактное собрание сущностей ПрО, которые имеют одни и те же характеристики. Все сущности ПрО, которые объединяет концепт согласуются с характеристиками, которые в системе ассоциируются с атрибутами. Каждый концепт в модели обозначается уникальным именем и идентификатором. Группа подобных концептов – это родительский концепт, который определяется заведомо определенным набором общих атрибутов для данной группы концептов.

Атрибут - это абстракция, которой владеют все абстрагированные концепты сущности. Каждый атрибут обозначается именем, уникальным в границах описания концепта. Множество объединенных в группу атрибутов, имеет идентификатор группы атрибутов. Множество идентификаторов групп могут быть объединены в класс и иметь идентификатор класса.

Концепт вместе со своими атрибутами в информационной концептуальной модели представляется графически или в текстовом виде.

Отношение - это абстракция набора связей, которые имеют место или возникают между разными видами объектов ПрО, абстрагированные как концепты. Каждая связь имеет уникальный идентификатор. В информационной модели отношения могут быть текстовыми или графическими. Для формализации отношений между концептами добавляются вспомогательные атрибуты, ссылки на идентификаторы отношений. Некоторые отношения образуются как следствие существования других отношений.

Выделение сущностей ПрО проводиться с учетом отличий, определяемых соответствующими понятийными структурами. Объекты, как абстракции реального мира, обладают поведением, обусловленным свойствами и отношениями с другими объектами, структурно упорядочиваются посредством применения теоретико-множественных операций (принадлежности, объединения, пересечения, разности и др.) к множеству (классу) объектов для установления отношений между объектами.. Объекты могут находиться в отношениях, или связях между собой.

Различаются статические (постоянные) связи, которые не изменяются или изменяются редко, и динамические связи, которые имеют определенные состояния и изменяться на протяжении сеанса функционирования системы.

Статические связи реализуются путем добавления специальных атрибутов для объектов, которые принимают в них участие. Преобладающей моделью представления данных является реляционная модель, в которой не разрешается иметь множественные (повторяемые) значения атрибутов, и добавление выполняется по таким правилам:

а) в случае связи 1:1 дополнительный атрибут может определяться для одного из объектов связи и содержать идентификатор экземпляра, который принимает участие в связи;
б) в случае связи 1:N дополнительный атрибут предоставляет объекту N экземпляров, принимающих участие в связи;
в) в случае связи N:M создается ассоциативный объект, который фиксирует пару экземпляров (по одному для каждого из объектов), принимающих участие в связи.

Такой объект, кроме своего названия, имеет первым атрибутом идентификатор первого из связанных экземпляров объектов, а вторым атрибутом - идентификатор экземпляра второго.

Связи между объектами могут эволюционировать с течением времени, и состояния эволюции может существенно влиять на ход решения соответствующей задачи. Для таких случаев связи обязательно строится ассоциативный объект и определяется модель состояний (см. тема 3.). Для представления состояния ассоциативного объекта в состав его атрибутов добавляется атрибут, который фиксирует его текущее состояние.

Среди определенных действий, которые сопровождают переходы в состояния для модели состояний связей, должны быть операции создания нового экземпляра ассоциативного объекта (если новая пара экземпляров вступает в связь) и его уничтожения (если связь прерывается).

4.1.2. Метод анализа и построения моделей С.Шлаер и С.Меллора

Программная система по данному методу создается как совокупность определенного множества доменов проблемных областей, каждый из которых представляет собой отдельный мир со своими объектами, которые анализируется независимо друг от друга. Продуктом анализа домена есть три модели:

– информационная модель системы или онтология домена;
– модель состояний объектов, определенных в составе информационной модели (или онтологии);
– модель процессов, обеспечивающих переходы из одного состояния объекта в другое.

Ниже рассматриваются эти модели.

4.1.2.1. Информационная модель

Под *информационной моделью* понимается совокупность объектов предметной области и связи (отношения) между ними. Представление информационной модели в данном методе базируется на известной реляционной модели данных. Для построения этой модели проводится выявление существенных объектов домена и предоставление им уникальных и значимых (мнемонических) названий. При этом учитываются следующие возможные категорий объектов:

– реальные предметы мира, имеющие физическое воплощение;
– абстракции физических предметов этого мира;
– абстракции или цели использования этих предметов определяются абстрактными действиями, изменяющими их состояния;
– взаимодействия - это отношение между объектами;
– спецификации – это представление правил, стандартов, критериев качества и ограничений на использование системы.

Для классов объектов выбираются уникальные имена, устанавливаются атрибуты и устанавливаются связи между объектами.

Атрибуты объектов. Для списка выявленных объектов определяются их характерные признаки или свойства, называемые атрибутами. Каждый атрибут есть абстракция

одной характеристики объекта, которая присуща всем представителям класса объектов. За атрибутом закрепляется имя, уникальное в границах этого класса.

Для каждого из выбранных атрибутов определяются возможные значения (типы значений) одним из следующих способов:

– задание числового диапазона;
– перечисление возможных значений;
– ссылка на документ, в котором определены возможные значения;
– правила генерации значений.

Идентификаторы объектов. Для объекта задается идентификатор в виде одного или нескольких атрибутов, значение или совокупность значений которых однозначно выделяют экземпляр объекта среди других в классе. Примером идентификатора может быть название или имя объекта, табельный номер сотрудника, номер паспорта, код плательщика налогов и др. Совокупность таких атрибутов может зависеть от области определения объекта.

Ссылка на атрибут может уточняться именем класса, задаваемым через точку. Атрибуты объектов представляются как атрибуты отношений согласно следующих правил:

– каждый экземпляр объекта обязательно имеет одно значение (значение не может быть неопределенным или отсутствующим);
– атрибут одномерен и не имеет нескольких значений одновременно;
– если идентификатор составляется из нескольких имен атрибутов, все указанные имена атрибутов, кроме первого, относятся к первому указанному имени объекта.

Связи объектов. После определения состава классов объектов домена и присущих им атрибутов, рассматриваются связи между объектами этого домена. Объекты одного класса могут участвовать в бинарных, то есть в по-парных связях с объектами другого или одного и того же класса. Рассмотрим несколько видов связи:

1) проект имеет исполнителей, которые заняты в проекте;
2) руководитель управляет исполнителями, т.е. исполнитель подчинен руководителю;
3) исполнитель занимает комнату, т.е. комната занята исполнителем.

На каждой стадии проектирования информационной модели фиксируется возможность экземпляра определенного класса объектов находиться в отношениях (связях) с экземпляром другого класса или одного и того же класса. Существенной особенностью связей является количество экземпляров объектов, которые одновременно могут принимать в них участие.

В методе различаются три фундаментальных вида связи между объектами:
– один к одному (1:1), в связи принимают участие по одному экземпляру с каждой стороны (пример, в некоторой организации руководитель занимает отдельный кабинет и руководит лично только одним проектом);
– один ко многим (1:n), один экземпляр объекта некоторого класса может поддерживать отношения одновременно с несколькими экземплярами объектов другого или того же класса (пример, руководитель может иметь несколько подчиненных, но у каждого из них один шеф);

– много ко многим (m: n), в связи могут принимать участие несколько экземпляров объектов с каждой стороны.

Метод С.Шлаер и С.Меллора предусматривает специальную графическую нотацию для фиксации связей, базирующихся на диаграммах метода Чена сущность - связь (entity–relations) [3] для представления информационной модели проблемной области, суть которого заключается в следующем.

Связи между объектами изображаются стрелками, указывающими направление связи. Возле рамки объекта, принимающей участие в связи, на линии стрелки указывается роль, которая этот объект поддерживает в данной связи. Связь 1:1 обозначается двунаправленной стрелкой, имеющей по одному "наконечнику" стрелки с каждой стороны, связь 1:n обозначается стрелкой, имеющей два "наконечника" со стороны объекта, для которого в связи могут принимать участие несколько экземпляров, и, наконец, по два "наконечника" с каждой стороны имеет стрелка, означающая связь вида n : m.

Над стрелкой может указываться название (имя) связи. Связи могут быть безусловными, т.е. каждый экземпляр объекта заданного класса принимает участие в связи. Условные связи, когда отдельные экземпляры объектов класса не принимают участия в связи, и обозначаются буквой "у" в конце стрелки.

Отношение, имеющее особый вес для представления онтологии и выражающее общность и различие между классами объектов, является отношением наследования . Оно представляется с помощью, так называемой, диаграммы классов. На рис.4.1. приведен пример такой диаграммы.

Рис.4.1. Пример диаграммы класса

Построенная диаграмма информационной модели сопровождается неформальным описанием всех объектов, их атрибутов и связей, в которых объекты принимают участие.

4.1.2.2. Модель состояний

Данная модель предназначена для отображения динамики изменений, происходящих в состоянии каждого из объектов класса, т.е. динамику их поведения. Все экземпляры одного класса объектов согласно понятия класса имеют одинаковое поведение. Базовыми понятиями модели динамики поведения объектов являются:

– состояние объекта, которое определяется текущими значениями отдельных его атрибутов;
– состояние объекта изменяется в результате произошедших действий или стимулов;
– состояние домена, которое определяется совокупностью состояний его объектов;
– изменение состояния объекта сопровождается некоторыми процессами, которые определены для каждого состояния.

Для фиксации динамических аспектов требований как отражения поведения объектов в рассматриваемом методе предложены две альтернативные нотации: графическая, которая называется диаграммой переходов состояний (ДПС) и табличная, которая называется таблицей переходов состояний (ТПС).

Построение модели состояний начинается с выделения среди определенных в информационной модели классов объектов тех, которые имеют динамическое поведение (т.е. изменяют свое состояние с течением времени), или, как говорят, имеют жизненный цикл от создания экземпляра объекта и до его разрушения.

Для каждого из выделенных объектов определяются:

1) множество состояний, в которых объект может находиться;
2) множество инцидентов или событий, которые побуждают экземпляры класса изменять свое состояние;
3) правила перехода для каждого из зафиксированных состояний, как указание на новое состояние экземпляра данного класса, если произойдет некоторое событие из множества событий тогда, когда объект находится в данном состоянии;
4) действия или процессы для каждого из определенных состояний, которые выполняются при переходе в данное состояние.

Для представления этой информации в нотации диаграммы перехода состояний предусматривается следующее:

– каждое состояние, определенное для класса объектов, получает название и порядковый номер, уникальную метку и название;
– состояние обозначается рамкой, содержащей номер и название состояния;
– переход от состояния к состоянию изображается направленной дугой, помеченной меткою и названием события, обусловившего переход;
– начальное состояние обозначается стрелкой, направленной к соответствующей рамке и является состоянием, которое экземпляр объекта приобретает после его создания (или инициализации);
– заключительное состояние задает конец жизненного цикла экземпляра объекта, если экземпляр продолжает существовать или разрушается, обозначается оно пунктирной рамкой;
– указание на действия, которые должны быть выполнены экземпляром объекта, когда он приобретает некоторое состояние.

Для изменения состояния экземпляра класса объектов выполняются действия:

– обработка информации, переданной в сообщении о событии;
– изменение определенного атрибута объекта;
– вычисления;
– генерация операции для некоторого экземпляра класса;
– генерация события, сообщение о котором должно передается внешнему по отношению к данному домену объекту (например, человеку-оператору другой системе);
– передача сообщения о событии от внешних объектов;
– взаимодействие с двумя специфическими объектами – таймером и системными часами, где *таймер* служит для измерения интервала времени и встроен системным образом в данный метод.

Атрибутами таймера являются:

– уникальный идентификатор экземпляра таймера;
– интервал времени, через который будет подан сигнал о наступлении некоторого события;
– метка наступающего события, при условии, что остаток времени равен нулю;
– идентификатор экземпляра объекта, для которого устанавливается таймер.

Таймер устанавливается для отдельного экземпляра некоторого управляемого объекта (например, духовой шкаф печи) для определения наступления события, данными которого есть значения атрибутов таймера. Предусмотрены события для установки таймера в нуль и его уничтожения.

Системные часы представляют собой также встроенный в метод объект, для которого можно читать его атрибуты - показатели системного времени (минуты, часы, день, месяц, год).

Альтернативой для графической нотации диаграммы перехода состояний является соответствующая табличная нотация, каждое из возможных для класса объектов состояний представляется строкою этой таблицы, а каждое из воздействующих на объект событий - столбцом. Клетка таблицы перехода состояний определяет состояние, которое приобретает объект, если соответствующее столбику событие произойдет, когда он находился в состоянии, соответствующему строке. При этом допускается, чтобы некоторые комбинации событие / состояние не приведут к изменению состояния экземпляра объекта. Такие клетки таблицы содержат указание "событие игнорируется".

Отдельные комбинации событие / состояние могут быть запрещены, тогда в соответствующих клетках помещается текст "не может быть". Заметим, что при использовании таблицы перехода состояний действия, соответствующие состояниям, определяются отдельной нотацией.

При выборе диаграммы или таблицы состояний перехода аргумент будет в пользу диаграммы из–за ее наглядности и определенности действий, тогда как табличная форма служит для фиксации всех возможных комбинаций состояние/событие, чем обеспечивается полнота и непротиворечивость заданных требований. Все сказанное

касается отдельных объектов как базовых составляющих (компонентов) архитектуры системы в целом.

Важным принципом объединения компонентов в систему является наличие у компонентов общих событий, причем чаще всего один из компонентов порождает событие, а другие на его реагируют. На этом принципе базируется способ объединения отдельных объектов и компонентов в систему.

Программная система в целом рассматривается как взаимодействие объектов с помощью механизма обмена сообщениями (между объектами системы и внешними объектами) для задания определенных событий и данных к ним. При этом считается, что внешние события, которых система не запрашивает, приводят к запуску системы. Внешние события, ожидание которых предусмотрено в системе, представляются в виде сообщений, посылаемых системою внешним объектам, которые в ответ направляют сообщение о наступлении или отсутствии событий.

Поскольку поведение отдельного объекта представлено диаграммой перехода состояний, то поведение системы в целом представляется как схема взаимодействия отдельных таких диаграмм, каждая из которых получает название и изображается на схеме овалом с этим названием. Овалы связаны между собою стрелками, отвечающими сообщениям о событиях, связывающих отдельные диаграммы перехода состояний. На стрелке указывается метка события, а направление стрелки соответствует направлению передачи сообщения. Внешние объекты обозначаются прямоугольниками с названиями. Пример взаимодействия моделей поведения объектов приведен на рис. 4.2.

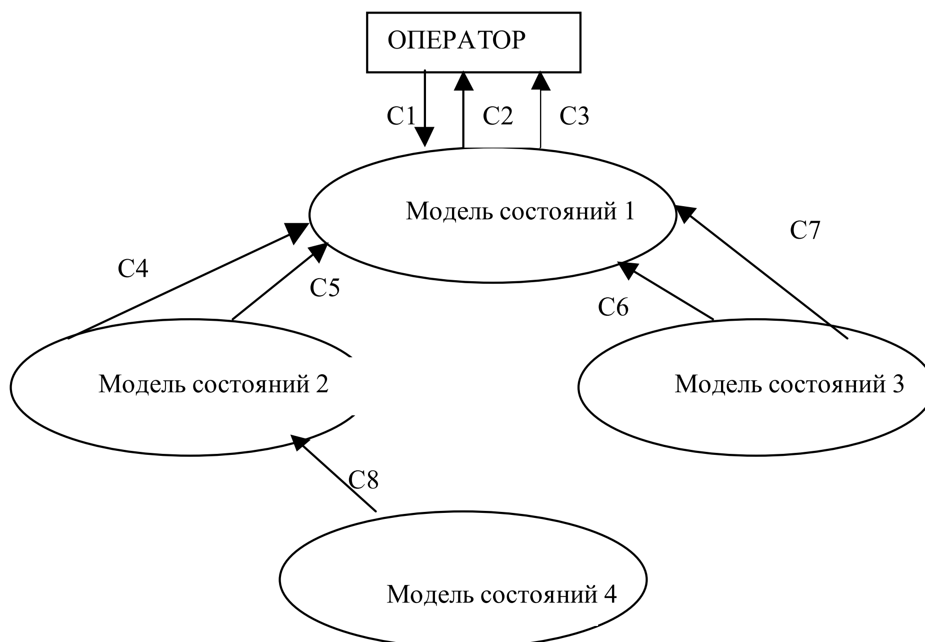

Рис. 4.2. Схема взаимодействия моделей поведения объектов

4.1.3. Модель процессов

Модель состояний объектов задает требования к поведению системы, предусматривает определение действий, которые сопровождают изменение состояний объектов. Действия – это алгоритмы, которые выполняются системой как реакции на события и функции системы. Понимание требований к системе предусматривает и понимание указанных выше действий, иногда достаточно сложных. Для преодоления сложности

понимания действий используется декомпозиция их на отдельные составляющие, названные процессами.

Последовательность выполняемых процессов образует поток управления. Процессы обмениваются данными, образуя потоки данных. Два указанных выше типа потоков предлагается использовать как модели алгоритмов действий системы. Для их представления в данном методе предусмотрена специальная нотация, получившая название диаграммы действий потоков данных.

В качестве источников данных могут быть:

– атрибуты объектов, которые продолжают существовать после завершения работы системы;
– системные часы, как показатель времени;
таймеры;
– данные событий;
– сообщения внешних объектов (людей, операторов и т.п.).

Правила построения диаграмм действий потоков данных:

– каждой из диаграмм перехода состояний может отвечать только одна диаграмма действий потоков данных;
– процесс изображается овалом, в середине которого указано содержание или название процесса;
– потоки данных изображены стрелками, на которых указываются идентификаторы данных, передаваемых от процесса к процессу; стрелка к овалу процесса указывает на входные данные процесса, направление от овала процесса - на выходные данные;
– источники данных изображены как прямоугольники или рамками с открытыми сторонами;
– данным, имеющим своими источниками архивные объекты, соответствуют потоки с названиями атрибутов объектов, которые передаются потоками (при этом название объекта может не указываться);
– потоки данных от таймера маркируются названием таймера;
– потоки данных от системных часов маркируются показателями времени (час, минута, и т.п.);
– событие, сообщение о котором получает процесс, изображается как стрелка, маркируемая названиями данных событий;
– процесс, породивший событие и процесс от приема сообщения о событии, относятся к одной диаграмме действий потоков данных и связывается потоком;
– если событие, созданное процессом некоторой диаграммой действий потоков данных, вызывает передачу сообщения процессу, другая диаграмма действий потоков данных, для первого из указанных выше процессов указывается стрелкой, ведущей от процесса в "никуда", а для второго - стрелкой, ведущей к процессу из "ниоткуда", причем в обоих случаях стрелки маркируется данными передаваемого события.

Различаются следующие типы процессов:

– аксессор, осуществляющий доступ к архивам;
– генератор событий;
– преобразователь данных (вычисления);
– проверки условий.

Потоки управления на диаграмме действий потоков данных обозначаются пунктирными стрелками. Если процесс представляет собой проверку определенного условия, при выполнении которого осуществляется передача управления другому процессу, то соответствующий поток управления изображается перечеркнутой пунктирной стрелкой. Пример диаграммы действий потоков данных приведен на рис. 4.3.

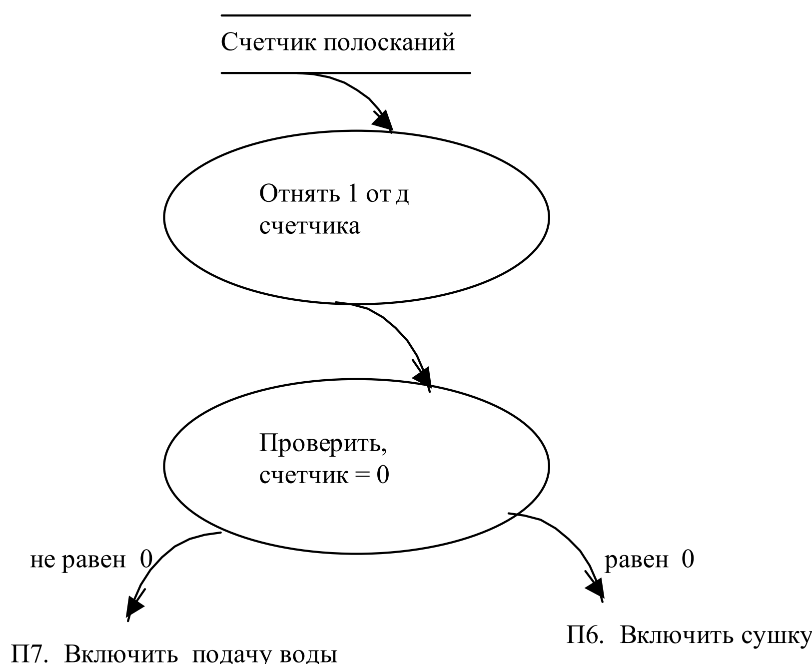

Рис. 4.3. Пример диаграммы действий потоков данных

К диаграммам действий потоков данных добавляется неформальное описание функций процессов, которое входят в ее состав. Для описания подробностей действий процессов нотация не регламентируется.

После завершения описания диаграммы действий потоков данных для всех объектов системы составляется общая таблица процессов, состоящая из следующих колонок:

– идентификатор процесса;
– тип процесса;
– название процесса;
– название состояния, для которого определен процесс;
– название действия состояния.

Такая таблица дает возможность проверить: непротиворечивость названий и идентификаторов процессов; полноту определенных событий и процессов; события генерируются или обрабатываются соответствующим процессом. Кроме того, наличие такой таблицы дает возможность выявить процессы, общие для нескольких действий или состояний и унифицировать их.

Можно представить таблицы трех видов, упорядоченных по идентификаторам процессов или по модели состояний или по типу процесса.

Результатами метода инженерии требований С.Шлаер и С.Меллора являются три модели.

1. Информационная модель системы, состоящая из:
– диаграммы сущность – связь;
– описания объектов и их атрибутов (неформальное);
– описания связей между объектами (неформальное).

2. Модель поведения объектов системы, представленная в виде:
– диаграммы переходов в состояния диаграмм перехода состояний или таблицы перехода состояний;
– неформальное описания действий диаграммами перехода состояний;
– неформальное описания событий диаграммами перехода состояний.

3. Модель процессов состояний объектов, представленная в виде:
– диаграмм действий потоков данных;
– таблицы процессов состояний;
– неформальное описание процессов.

Совокупность перечисленных моделей считается достаточной для перехода к процессу проектирования системы.

4.2. Методы проектирования архитектуры ПО

Проектирование ПО – это процесс разработки, следующий за этапом анализа и формирования требований. Задачей проектирования является преобразование требований к системе в проектные решения, требования к ПО и построение архитектуры системы.

Проектирование архитектуры системы проводится разными методами, основанными на известных подходах (структурный, объектно-ориентированный, компонентный и др.), каждый из которых предлагают свой путь решения проблемы проектирования, включая соответствующие им конструктивные элементы и методы задания структуры моделей (концептуальной, объектной и др.) системы. Среди методов проектирования наиболее часто использовался стандартный, процессы и задачи которого регламентированы стандартом.

При проектировании используется объектно-ориентированная парадигма, по которой любая система рассматривается как совокупность взаимодействующих объектов. Все подпроцессы проектирования будем рассматривать относительно тех объектов, требования к которым определены на этапе инженерии требований.

4.2.1. Стандартный подход к проектированию системы

Разработка автоматизированных систем (АС) выполнялась и выполняется на основе положений, представленных в стандарте ГОСТ 34.601-90 (см. Приложение 2). Он состоит из стадий и этапов разработки АС, которые, в зависимости от особенностей автоматизируемой системы, можно объединять друг с другом или вообще не включать в процесс разработки. Основанием для разработки АС является договор между разработчиком системы и ее заказчиком.

Этапами стандарта, ориентированным на разработку архитектуры АС, являются: формирование требований к АС, разработка концепции АС и проектирование технического проекта, в котором на основе сформулированных требований и концепций их реализации задаются конкретные задачи системы и пути их решения.

Эти этапы заканчивается созданием и утверждением отчета о научно-исследовательской работе, в которой дается оценка необходимых для реализации АС ресурсов, вариантов разработки АС и порядка проведения оценки качества системы.

На этапе разработки эскизного проекта используются проектные решения ко всей системе или ее части и определяется перечень задач системы, концепция информационной базы, функции и параметры основных программных компонентов системы, а также основные алгоритмы обработки информации.

Этап разработки технического проекта предусматривает разработку проектных решений относительно системы и ее частей, разработку документации на АС и комплектацию АС, а также определение технических требований на систему и проектирование задач для смежных частей проекта.

Проектные решения определяют организационную структуру, функции персонала АС, структуру технических средств, языка программирования, СУБД, общие характеристики ПО, систему классификации и кодирования, а также варианты ведения информационной базы системы.

Таким образом, данный стандарт обеспечивает:
– *концептуальное проектирование,* которое состоит в уточнении и согласовании деталей требований;
– *архитектурное проектирование,* которое состоит в определении главных структурных особенностей создаваемой системы;
– *техническое проектирование,* которое состоит в отображении требований на среду функционирования системы и определении задач и принципов их реализации;
– *детальное проектирование,* которое состоит в определении алгоритмов задач, построения БД и программного обеспечения системы.

При концептуальном проектировании определяются:

– источники поступления данных от заказчика, который несет ответственность за их достоверность;
– объекты системы и их атрибуты;
– способы материализации связей между объектами и виды организации данных.
– интерфейсы с потенциальными пользователями системы на ранних стадиях ЖЦ цикла разработки системы для оказания им помощи при формулировки целей и функций системы;
– правила взаимодействия пользователя и системы с точки зрения понимания, эффективности и скорости реакции системы.

Организация интерфейсов базируется на определенных ключевых элементах, определение которых предшествует проектированию конкретных экранов и форматов обмена данными, в частности:

1) значащие термины, образы и понятия, которые являются для пользователя понятными и распространенные в домене;
2) ментальная модель организации и представления данных, функций и ролей;
3) правила навигации (просмотра) данных, функций и ролей;
4) визуальные приемы демонстрации пользователю элементов системы;
5) методы взаимодействия, которые отвечают требованиям пользователей будущей системы.

При архитектурном проектировании системы принимаются во внимание конкретные интерфейсы, учитывающие такие аспекты, как традиции, обычаи, особенности домена и присущие ему действующие лица. В частности, пользователю определенный психологический комфорт предоставляют языковых форматы в меню и иконах интерфейсов, а также метрические показатели систем измерений и др.
При желании сделать будущую систему "поликультурной", то есть способной адаптироваться к разным языкам, выделяются текстуально чувствительные к культурной среде элементы, которые предполагают замену стандартных икон, введения новых текстов и др. Количество выдаваемых знаков в текстах определяется лаконизмом избранного языка общения (английский наиболее лаконичней). Длина текстовых сообщений и окон ввода текстов должна определяться как параметр настройки будущей системы.

Правила навигации должны учитывать традиции чтения слева - направо или наоборот.
Таким образом, общей рекомендацией концепций проектирования является построение полностью всех экранных форм системы и "проигрывание" с пользователем для выбора разных вариантов, отвечающих его вкусам. Такой выбор часто входит в в противоречие с заданными характеристиками интерфейса (например, удобство доступа и обеспечение конфиденциальности, быстродействие и сложность обработки и др.).

Для построения интерфейсов существует широкий выбор методов и средств. Большинство из них базируется на фиксации определенных классов объектов интерфейса (выбор из меню, заполнение экранных форм и др.) и средствах монтирования их в программную систему в виде интегрированных блоков или автономных подсистем интерфейса с пользователем.

При техническом проектировании системы на основе модели представления требований и понятий объектно–ориентированного подхода проводится уточнение состава и содержания функций, присущих им операций (методов), а также схем взаимодействия объектов.

Содержание операций, которые способны выполнять объекты, может быть раскрыто с помощью диаграмм потоков данных (см. тема 3.).

Взаимодействие объектов организовывается путем обмена сообщений, в ответ на которые они выполняют соответствующие операции и изменяют свое состояние, или посылают сообщения другим объектам.

Для уточнения поведения объектов можно использовать ряд диаграмм, отображающих различные аспекты взаимодействия объектов. Такие диаграммы входят в состав метода UML.

Все уточнения, которые делаются относительно данных, интерфейсов и поведения объектов в сценарии, могут привести к необходимости пересмотра моделей анализа требований и даже состава объектов. Изменения надо начинать с продуктов этапа инженерии требований, а именно модели анализа требований и др. Трудозатраты на поиск мест локации необходимых изменений в упомянутых моделях уменьшается при применении метода трассирования требований.

Определяются нефункциональные требования, задающие определенные ограничения структурой организации системы или среды использования. К отдельным

разновидностям нефункциональных требований относятся требования секретности, безопасности, отказоустойчивости, корпоративной работы над общими ресурсами и др.

При построении моделей требований для автоматизированных систем учитывается их роль и место в прикладных приложениях системы. Для них разработано немало национальных, корпоративных и ведомственных стандартов, которые фиксируют правила обеспечения защиты данных, безопасности системы и др.

Результатом формирования нефункциональных требований может быть расширение модели требований дополнительными сведениями или соответствующими объектами, выполняющими требования взаимодействия, безопасности, защиты данных и др.

4.2.2. Общесистемный подход к проектированию архитектуры системы

Одним из путей архитектурного проектирования является определение структуры создаваемой системы, состав компонентов, способов их представления и объединения. Фактически строится архитектура, которая состоит из четырех уровней, для каждого из которых определен набор компонентов, выполняюших определенную прикладную или общесистемную функции в системе (рис.4.4.).

Прикладные программные системы
Специфические для бизнеса компоненты
Общесистемные компоненты *Интерфейс с универсальными системами программной инженерии*
Системные компоненты *Интерфейс с оборудованием*

Рис.4.4. Поуровневая архитектура системы

Первый уровень – **системные компоненты**. Они осуществляют взаимодействие с периферийными устройствами оснащения компьютеров (принтеры, клавиатура, сканеры, манипуляторы и и т.п.), используются при построении операционных систем и не попадают в поле зрения разработчиков приложений.

Ко второму уровню относятся **общесистемные компоненты.** Они обеспечивают взаимодействие с универсальными сервисными системами, такими, как операционные системы, системы баз данных и знаний, системы управления сетями и т.п. Компоненты данного слоя используются во многих приложениях как их составляющие.

К третьему уровню относятся **специфические компоненты** определенной проблемной области. Они являются составляющими программных систем, предназначенных для решения ее задач (бизнес-задач). Их еще называют семейством программных систем.

И, наконец, к четвертому слою относятся **прикладные** программные системы, которые построены для решения конкретных задач отдельных групп потребителей информации (офисные системы, системы бухгалтерского учета и др.) Они могут использовать компоненты упомянутых нижних уровней.

Компоненты любого из выделенных уровней используются, как правило, на своем уровне или на более верхнем. Для каждого уровня определен соответствующий набор профессиональных знаний, умений и навыков, необходимых для создания и использования его компонент, Этот набор определяет соответствующее разделение профессионалов в программной инженерии (системщики, прикладники, программисты и др.).

На уровне архитектурного проектирования система рассматривается как композиция компонент третьего уровня, имеющая доступ до компонентов первого и второго уровней.

Т.е. архитектурное проектирование – это разработка компонентов третьего уровня, определение входных и выходных данных, слоев иерархии компонентов и их связей.

Известными архитектурными схемами, определяющими стиль работы программной системы, являются распределенная, клиент-сервер и многоуровневая.

Распределенная схема обеспечивает взаимодействие компонентов системы, расположенных на разных компьютерах через стандартные интерфейсы и механизмы вызова, выполняемые промежуточными средами (COM/DCOM, CORBA, OLE, Java и др.): RPC (Remote Procedure Calls), RMI (Remote Method Invocation), tuple spaces, aplets и др.

В трехуровневой архитектурной схеме типа клиент/сервер сервер или брокеры объектных запросов (ORB) предоставляют общесистемный сервис и различные ресурсы, а также управляют распределенными объектами (компонентами). Архитектура такой системы может быть многоуровневой, если объекты предоставляют услуги и сами пользуются услугами других объектов, расположенных на разных уровнях этой схемы. Данная архитектурная схема отображает объектный стиль проектирования ПО, стиль моделирования проблемы с помощью UML и унифицированного процесса RUP [13, 14].

Объектный стиль проектирования ПО заключается в представлении объектной модели, абстрагировании ее сущностей объектами и в иерархической организации объектов с инкапсуляцией отдельных их возможностей.

Стиль моделирования UML заключается в декомпозиция проблемы на отдельные подсистемы (пакеты), каждая из которых отвечает проектным решениям и функциональным (и нефункциональным) требованиям и построении модели предметной области. Определяются носители интересов (акторов) и возможных прецедентов - последовательности действий акторов для получения результатов. В пакете задается модель прецендентов, описывающих важные функциональные возможности и представление требований. Разрабатываются варианты использования, в которых определяется состав объектов и принципы их взаимодействия. Для объектов определяются их структурные особенности – атрибуты и ассоциации, представляемые в диаграммах классов. Поведенческие аспекты системы отражаются диаграммами последовательности, в которых задается: последовательность взаимодействий объектов во времени, правила перехода от состояния к состоянию (диаграммы состояний) и к действию (диаграммы действий). Особенности кооперативного поведения объектов отражаются диаграммами кооперации.

Объекты проблемы и соответствующие им диаграммы использования задают общую архитектурную схему проблемы, в рамках которой осуществляется описание ее структуры и специфики поведения компонентов, для понимания того, как построена архитектура системы.

Стиль проектирования архитектуры в рамках унифицированного процесса RUP состоит в том, чтобы предоставить все виды деятельности, которые команда разработчиков системы использует на фазах процессов при построении моделей, способных охватить систему, определить ее структуру и поведение в нотации UML.

В зависимости от модели проектирования ПО (каскадная, спиральная, иерархическая и др.) созданная на начальном этапе архитектура ПО может расширяться и уточняться итеративно (например, при изменении требований заказчиком) на последующих этапах, что способствует получению полной и надежной архитектуры.

Результат проектирования – архитектура, т.е ее каркас и архитектурная инфраструктура, содержащая набор компонентов, из которых можно формировать некоторый конкретный вид архитектурной схемы для конкретной среды выполнения системы. Заканчивается проектирование описанием архитектуры ПО, в котором отображены зафиксированные проектные решения, принятые в ходе работы архитекторов системы, в том числе описание логической и физической структуры и данных, а также способов взаимодействия ее объектов.

В состав архитектура системы входят статические и динамические объекты, их связи и интерфейс между компонентами других архитектурных уровней. В ней представлены результаты анализа, декомпозиции системы на отдельные подсистемы и компоненты, а также набор справочников, словарей и т.д. Описание архитектуры имеет множество представлений, отражающих отдельные аспекты реализации системы. Каждое представление детализирует проблему и отдельные ее части, а также их связи и интерфейсы

Современные программные системы являются довольно сложными композициями разнообразных функций, вместе с тем имеются тысячи модулей, выполненных как готовые программные продукты, которые могут быть включены в любые программные системы для выполнения соответствующих им функций. При этом примитивные функции могут составлять композиции, которые выполняют определенные обобщенные функции. Они могут связываться в новые композиции и т.п. Для того, чтобы совокупность современных готовых к использованию средств можно было обозреть, введена определенная послойная их структуризация, суть которой состоит в следующем

На этапе инженерии требований создается совокупность объектов, которые привязываются к определенному сценарию для реализации определенных функций в этом сценарии.

В сложных программных системах количество выделенных объектов может насчитывать сотни, их композиции не будут иметь выразительного представления, даже с учетом того, что объекты разных сценариев могут совпадать, и потребуется дополнительный анализ для их отождествления.

Основными рекомендациями для декомпозиции сложной системы на компоненты или модули являются:

– четкое определение цели и возможность проверки их выполнимости;
– обязательное определение входные и выходные данных;
– задание иерархии, каждый уровень которой отвечает уровню абстракции системы и позволяет скрывать те детали, которые будут отработаны на следующих уровнях.

Такая пошаговая детализация принятия решений не только сводит решение сложной задачи к нескольким более простым, но и сосредоточиться на решении общих задач разработки компонентов отдельными членами команды с применением разных инструментальных средств, влияющих на эффективное их функционирование. При этом интерфейсы между компонентами должны быть согласованными для интеграции их в единую структуру.

Полученные совокупности объектов объединяются в подсистемы с учетом таких требований:

1) каждая создаваемая подсистема должна ассоциироваться с определенными элементами продукта инженерии требований (как, например, актер, сценарий, объект и т.п.);
2) необязательные функции или часто изменяемые функции выделятся в подсистемы так, чтобы каждая функция, для которой прогнозируются изменения требований, была как отдельная подсистема, связанная с одним актером (изменения вызывают актером);
3) интерфейс подсистемы понятен и имеет взаимосвязи с другими подсистемами. Каждая подсистема должна выполнять минимум услуг или функций и иметь фиксированное множество параметров интерфейса.

После отделения изменяемых и производных подсистемы проводится анализ связей и зависимостей, которые существуют между объектами с целью образования подсистем с внутренними связями и прозрачными внешними интерфейсами.

К способам объединения объектов в подсистему можно отнести:

– . сборка объектов в подсистему, которые ничем не связаны между собой;
– логическое объединение объектов в подсистему, которые являются функционально независимыми, имеют общее свойство, для которых можно установить определенное логическое отношение (например, одна и та же функция реализована для разных сред, как ввод данных для дисков и порта сети);

– объединение по времени, т.е. сборка объектов в подсистему, которые активизируются в общий промежуток времени;
– коммуникативное объединение, т.е. собираются объекты, которые имеют общий источник данных;
– процедурное объединение, т.е. в подсистему собираются объекты, которые последовательно передают друг другу управление;
– функциональное объединение объектов, каждый из которых входит в подсистему, выполняет часть работ для выполнения общей функции, которую выполняет подсистема.

Анализ приведенных выше способов объединения объектов в подсистемы с точки зрения стойкости к изменениям, показывает, что каждое изменение требует соответствующей корректировки минимального количества архитектурных компонент. Отсюда можно сделать вывод, что все способы не отличаются легкостью модификации

требований. Что касается функционального объединения, то ему соответствуют определенные требования в модели.

Если вновь создаваемая система используется в готовой системе (унаследование системы), то ее целесообразно считать подсистемой создаваемой системы. Использование готовых компонентов или систем повторного использования снимает проблему дублирования и сокращает объем работ при проектировании архитектуры системы.

Повторным объектом может стать объект, поведение которого частично используется в нескольких подсистемах.

При выделении таких объектов и подсистем в большом проекте учитываются некоторые критерии. Например, если разработку будут ведут несколько групп разного уровня компетентности, или разного уровня обеспеченности ресурсами и они разъединены географически, то разделение на подсистемы ведется с учетом этих обстоятельств.

Результаты архитектурного проектирования представляются в нотациях, которые представлены в модели анализа требований средствами диаграмм (сущность-связь, переходов в состояния, потоков данных и действий, классов и т.п.). В указанных диаграммах задействованы объекты проекта, которые детализируют заданные требования к разработке и отображают решения, которые оказывают влияние на реализацию этих требований

4.2.3. Техническое проектирование

Техническое проектирование состоит в отображении архитектуры системы в среду функционирования и определении всех конструкций композиций компонентов архитектуры.

На этом этапе происходит привязка проекта к техническим особенностям платформы реализации: СУБД, ОС, коммуникации, скорость реагирования системы на внешние условия т.п.

Объекты модели анализа требований согласовываются с учетом перечисленных выше особенностей, формализуются все стимулы, которые посылают или получают объект, и все операции, которые являются ответом на указанные стимулы.

Любой аспект привязки может потребовать построения вспомогательных интерфейсных или управляющих объектов и корректировки существующих. Более того, может оказаться возможность использования готовых подсистем, устройство которых отличается от подсистем, которые были до сих пор определены на основании анализа требований. В этом случае вносятся соответствующие корректировки в модель анализа требований и в архитектуру системы.

Следующим шагом проектирования может быть учет определенных свойств, которые определяют возможность системы функционировать в другой среде, и обладать качеством, сформулированным заказчиком.

Переносимость системы. Под этим термином понимают возможность изменять определенные используемые сервисные системы (ОС, системы коммуникаций в сетях,

СУБД, и т.п.) путем локальной настройки соответствующих модулей. Обычно речь идет о переносимости относительно конкретного типа сервисных систем, например, переносимость относительно СУБД, переносимость относительно системы файлов и т.п.

Для реализации таких свойств определяются объекты, которые взаимодействуют с сервисными системами, относительно которых декларируется переносимость. Любой определенный таким образом объект заменяется на объект, который взаимодействует не непосредственно с сервисной системой, а с некоторым абстрактным объектом-посредником, который осуществляет трансформацию абстрактного интерфейса в интерфейс конкретной сервисной системы. Объект-посредник при этом имеет свойство настраиваться на конкретную сервисную систему.

Контрольные вопросы и задания

1. Определите задачи анализа предметной области и процессов проектирования архитектуры системы.
2.Сформулируйте задачи концептуального проектирования моделей ПрО.
3. Назовите продукты анализа домена в методе Шлаер и Меллора.
4. Назовите модели метода Шлаер и Меллора и их суть.
5.Какие еще модели ПрО Вы знаете?
6. Перечислите ключевые факторы, влияющие на проектирование интерфейсов.
7. Назовите примеры нефункциональных требований, которые требуется учитывать на стадии проектирования архитектуры.
8. Какие уровни выделяются в архитектуре системы?
9. Какие известны способы объединения объектов в подсистемы?
10. Назовите приемы обеспечения переноса системы в другую среду.

Литература к теме 4.

1. С.Шлеер и С.Меллор, Объектно-ориентированный анализ: моделирование мира в состояниях //.-К.-Диалектика, 1993.-240с.
2. Coad P.,Yourdan E. Object-oriented analysis.-Second Edition.-Prentice Hall.– 1991.– 296p.
3. Yourdan E. Modern Srtucrured Analysis. --New York: Yourdan Press /Prentice Hall, 1988.–297p.
4. DeMarko D.A., McGovan R.L. SADT: Structured Analysis and Design Technique. New York: Mcgray Hill, 1988.– 378 c.
5. Yourdan E., Constantine L. Structured Design. Yourden Press. Engwood Cliffs.N.J.–1983.
6. Martin J., Odell J.J. Object-oriented analysis and design.-Prentice Hall.–1992.–367p.
7. Barker R. CASE-method. Entity Relatiionship Modeling.–Copyrigth ORACLE Corporation UK Limited New York: Publ., 1990. – 312 p.
8. Schardt J.A. Assentials of Distributed Object Design M.S.E. Advanced Concepts Center.– 1994.–p.225–234
9. Rumbaugh J., Blaha V., Premerlani W. Object- Oriented Modelling and Design, Englewood Cliffs, NJ: Prentice Hall.– 1991.– 451p.
10. Гради Буч. Объектно-ориентированное проектирование.- 3-е издание. –М.:"Бином", 1998.-560 с.
11. Jacobson I. Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing.– New York: Addison-Wesley Publ.Co. – 1994.– 529 p.
12. К.Чернецки, У.Айзенекер. Порождающее программирование. Методы, инструменты, применение.– Издательский дом «Питер».– Москва– Санкт-Петербург… Харьков, Минск.– 2005.–730с.

13. Орфали Р., Харки Д. Эдвардс Дж. Основы CORBA. Из.-во “Малип”, М.: 1999.– 317с.
14. Эммерих В. Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.
13. Рамбо Дж., Джекобсон А , Буч Г. UML: специальный справочник.– СПб.: Питер.– 2002.– 656с.
14. Кендалл Скотт. Унифицированный процесс. Основные концепции.–Москва–С–Петербург–Киев.–2002.– 157с.

Тема 5

МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

В последние годы наибольшее развитие и использование получил объектно–ориентированный, компонентный, сервисно–ориентированный подходы. Для их поддержки Были разработаны и инструментальные средства (Rational Rose, Rational Software, RUP, Demral , OOram и др.). Инструменты рассматриваются в теме 10.

Эти подходы составляют основу методам проектирования ПС, которые практически применяются разными разработчиками, использующими подходящие для своих целей возможности этих методов. Получили дальнейшее развитие теоретические методы (алгебраическое, алгеброалгоритмическое, композиционное и др.), которые основываются на математических, алгебраических и логико–алгоритмических подходах и методах формального построения и доказательства программ.

В данной главе представлено описание основ методов систематического и отдельных методов теоретического программирования для ознакомления студентов с теорией и практикой проектирования ПС.

5.1. Методы систематического программирования

К методам систематического программирования отнесены следующие методы:

– структурный,
– объектно–ориентированный,
– моделирование в UML,
– компонентный,
– аспектно–ориентированный,
– генерирующий,
– агентный и др.

Каждый метод имеет свое множество понятий и операций для проведения процесса разработки компонентов или ПС. Вновь появившиеся методы, например, генерирующее программирование использует возможности объектно–ориентированного, компонентного и аспектно–ориентированного методов.

Дадим краткую характеристику возможностей этих методов, необходимую студентам для ознакомления с общими аспектами проектирования ПС в рамках приведенных методов.

5.1.1. Структурный подход

Сущность структурного подхода разработки ПС заключается в декомпозиции (разбиении) системы на автоматизируемые функции, которые в свою очередь делятся на подфункции, на задачи и так далее. Процесс декомпозиции продолжается вплоть до определения конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимосвязаны.

В основу структурного подхода положены такие общие принципы:
– разбивка системы на множество независимых задач, легких для понимания и решения;

– иерархическое упорядочивание, т.е. организация составных частей проблемы в древовидные структуры с добавлением новых деталей на каждом уровне.

В основе этих принципов лежат операции:
– абстрагирования, т.е. выделения существенных аспектов системы и отвлечения от несущественных;
– формализации, т.е строгое методологическое решение проблемы;
– непротиворечивости, состоящей в обосновании и согласовании элементов системы;
– структуризации данных (т.е. данные должны быть структурированы и иерархически организованы).

При структурном анализе применяются в основном три вида наиболее распространённых моделей проектирования ПС:

SADT (Structured Analysis and Design Technique) модель и соответствующие функциональные диаграммы [1];
SSADM (Structured Systems Analysis and Design Method) – метод структурного анализа и проектирования [2];
IDEF0 (Integrated Definition Functions) метод создания функциональной модели, IDEF1 – информационной модели, IDEF2 – динамической модели и др. [3].

На стадии проектирования эти модели расширяются, уточняются и дополняются диаграммами, отражающими структуру программного обеспечения: архитектуру ПО, структурные схемы программ и диаграммы экранных форм.

Метод функционального моделирования SADT. На основе метода SADT, предложенного Д. Россом, разработана методология IDEF0 (Icam DEFinition), которая является основной частью программы ICAM (Интеграция компьютерных и промышленных технологий), проводимой по инициативе ВВС США.

Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им функции и действия, а также связи между ними.

Основные элементы этого метода основываются на следующих концепциях:

– графическое представление блочного моделирования. Графическая диаграмма отображает функцию в виде блока, а интерфейсы (вход/выход) представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описываются посредством интерфейсных дуг, выражающих "ограничения", которые в свою очередь определяют, когда и каким образом функции выполняются и управляются;
– строгость и точность. Правила SADT строги и имеют ограничения на количество блоков на каждом уровне декомпозиции (от 3 до 6 блоков) и связность диаграмм через номера блоков;
– уникальность меток и наименований;
– разделение входов и управлений (определение роли данных).
– отделение организации от функции, т.е. исключение влияния организационной структуры на функциональную модель.

Метод SADT может использоваться для моделирования широкого круга систем и определения требований и функций, а затем для разработки системы, которая удовлетворяет требованиям и реализует эти функции. Для действующих систем SADT может быть использоваться при анализе функций, выполняемых системой, а также для определения механизмов, посредством которых они осуществляются.

Результатом применения метода SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы – главные компоненты модели, все функции и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, показана с левой стороны блока, а результаты выхода показаны с правой стороны. Механизм (человек или автоматизированная система), осуществляющий операцию, представляется дугой, входящей в блок снизу (рис.5.1).

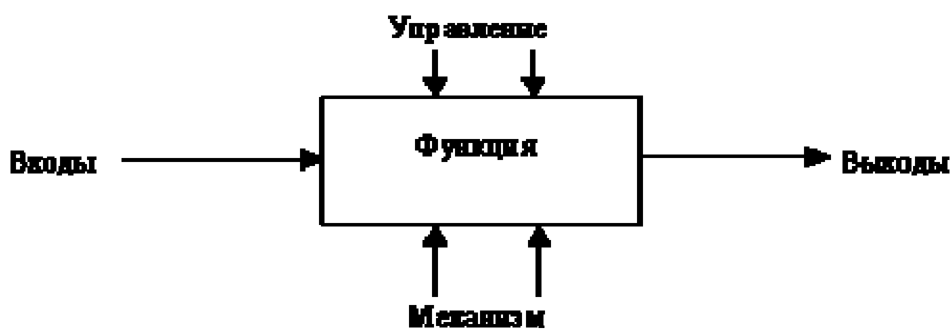

Рис. 5.1. Структура модели

Одной из наиболее важных особенностей метода SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

Метод SSADM базируется на таких структурных диаграммах: последовательность, выбор и итерация. Моделируемый объект задается их сгруппированной последовательностью, следующих друг за другом, операторами выбора элемента из группы и циклическим выполнением отдельных элементов.

Базовая диаграмма является иерархической и включает в себя следующие: список всех компонентов описываемого объекта; идентифицированные группы выбранных и повторяемых компонентов, а также последовательных компонентов. Модель процесса проектирования включает в себя:

– определение функций;
– моделирование взаимосвязей событий и сущностей;
– логическое проектирование данных;
– проектирование диалога;
– логическое проектирование БД;
– физическое проектирования.

В основе стратегического проектирования лежит анализ и определение требований. Определяется область действия приложения, существующие информационные потоки, формирование общего представления о затратах на разработку и потверждение

возможности дальнейшего использования приложения. Результатом является спецификация требований, которая применяется для логической спецификации системы.

Логическое проектирование включает в себя в себя проектирование диалога и процесса обновления БД. Проектирование состоит в создании логической модели и спецификации, в которой отображены входные и выходные данные, процессы выполнения запросов и
процессов обновления.на основе логической БД. Одной из целей логического проектирования является минимизация дублирования трудозатрат при физическом проектировании, обеспечение целостности на основе установления взаимосвязей между событиями и сущностями.

Физическое проектирование состоит в определении типа СУБД, необходимого для представления сущностей и связей между ними при соблюдении спецификации логической модели данных и учета ограничений на память и время обработки. Предусматривается реструктуризация проекта в целях изменения механизмов доступа, повышения производительности, объема логической структуры, добавление связей и документирования. Физическая спецификация включает в себя:

– спецификацию функций и схемы реализации компонентов функций,
– описание процедурных и непроцедурных компонентов и интерфейсов,
– определение логических и физических групп данных с учетом ограничений оборудования и стандартов на разработку,
– определение групп событий, которые обрабатываются как единое целое с выдачей сообщений о завершении обработки и др.

Проект системы представляется структурной моделью, задающей работы и взаимосвязи между этими работами в виде потоков проектных документов, отображенных в сетевом графике, а также модули, стадии и шаги проектирования, которые соответствуют ЖЦ.

5.1.2. Объектно–ориентированный метод проектирования

Объектно–ориентированное проектирование (ООП) [4, 5] представляет собой стратегию, в рамках которой разработчики системы вместо операций и функций мыслят в понятиях *объеков.* Объект – это нечто, способное пребывать в различных состояниях и имеющее определенное множество операций. Состояние определяется как набор атрибутов объекта. Операции, связанные с объектом, предоставляют сервисы другим объектам (клиентам) для выполнения определенных вычислений.

Объекты создаются в соответствии с определением класса объектов, который служит шаблоном для создания объектов. В него включены описания всех атрибутов и операций, связанных с объектом данного класса.

Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять набор операций, определяемый состоянием объекта. Объекты скрывают информацию о представлении состояний и ограничивают к ним доступ.

Под *процессом* объектно–ориентированного проектирования подразумевается проектирование классов объектов и взаимоотношений между этими классами. Когда

проект реализован в виде исполняемой программы, все необходимые объекты создаются динамически с помощью определений классов. Этот подход подразумевает выполнение трёх этапов при проектировании:

1. *Объектно–ориентированный анализ.* Создание объектно–ориентированной модели предметной области приложения. Здесь объекты отражают реальные объекты–сущности и операции, выполняемые этими объектами.
2. *Объектно–ориентированное проектирование.* Разработка объектно–ориентированной модели системы ПО (системной архитектуры) с учётом требований. В этой модели определение всех объектов подчинено решению конкретной задачи.
3. *Объектно–ориентированное программирование.* Реализация архитектуры (модели) системы с помощью объектно–ориентированного языка программирования (C++, Java) для определения объектов и средств определения классов объектов.

Данные этапы могут ”перетекать” друг в друга, т.е. они могут не иметь четких рамок и на каждом этапе применяется одна и та же система нотации. Переход к следующему этапу приводит к усовершенствованию результатов предыдущего этапа путём более детального описания определенных ранее классов объектов и определения новых классов.

Объектно–ориентированные системы можно рассматривать как совокупность автономных и независимых объектов. Изменение реализации какого–нибудь объекта или добавление ему новых функций не влияет на другие объекты системы. Четкое соответствие между реальными объектами (например, аппаратными средствами) и управляющими объектами программной системы облегчает понимание и реализацию проекта.

Объекты могут быть повторно используемыми компонентами, они независимо инкапсулируют данные о состоянии и операциях. Архитектуру ПО можно разрабатывать проект на базе объектов, ранее созданных в предыдущих проектах. Это снижает стоимость проектирования, программирования и тестирования ПО. Кроме того, возможность использования стандартных объектов уменьшает риск, связанный с разработкой ПО.

Модель окружения системы и модель использования системы представляют собой две взаимно дополняющие друг друга модели взаимоотношений системы и с ее средой:

Модель окружения системы – это статическая модель, которая описывает другие системы из пространства разрабатываемого ПО.

Модель использования системы – динамическая модель, которая показывает взаимодействие данной системы со своим окружением (средой).

Когда взаимодействия между проектируемой системой ПО и ее окружением определены, эти данные можно использовать как основу для разработки архитектуры системы. При этом необходимо применять знания об общих принципах проектирования системных архитектур и данные о конкретной предметной области.

Существует два типа моделей системной архитектуры:
– *статические модели*, которые описывают статическую структуру системы в терминах классов объектов и взаимоотношений между ними. Основными взаимоотношениями, которые документируются на данном этапе, являются отношения обобщения, отношения «используют–используются» и структурные отношения.

– *динамические модели*, которые описывают динамическую структуру системы и показывают взаимодействия между объектами системы (но не классами объектов).

Документируемые взаимодействия содержат последовательность запросов к сервисам объектов и описывают реакцию системы на взаимодействия между объектами.

Язык моделирования UML поддерживает большое количество возможных статических и динамических моделей, в том числе модель подсистем и модель последовательностей. Модель последовательностей – одна из наиболее полезных и наглядных моделей, которая в каждом узле взаимодействия документирует последовательность происходящих взаимодействий между объектами.

5.1.3. Метод моделирования UML

Метод UML (United Modeling Language – унифицированный язык моделирования) является результатом совместной разработки специалистов программной инженерии и инженерии требований [6, 7]. Он широко используется ведущими разработчиками программного обеспечения как метод моделирования программных продуктов на всех стадиях жизненного цикла разработки программных систем.

В основу метода положена парадигма объектного подхода, при котором концептуальное моделирование проблемы происходит в терминах взаимодействия объектов:
– онтология домена определяет состав классов объектов домена, их атрибутов и взаимоотношений, а также услуг (операций), которые могут выполнять объекты классов ;
– модель поведения определяет возможные состояния объектов, инциденты, инициирующие переходы с одного состояния к другому, а также сообщения, которыми обмениваются объекты;
– модель процессов определяет действия, которые выполняют объекты.

UML концептуальная модель требований рассматривается как совокупность нотаций – диаграмм, которые визуализируют основные элементы структуры системы.

Метод UML обеспечивает эффективное представление взаимодействий между разными участниками разработки с заданием комментариев. Допускаются два вида комментариев: традиционный неформальный текст или стереотип.

Стереотип – это средство метаклассификации элемента в UML, т.е. стандартизированный комментарий элемента, который представлен какой–нибудь из диаграмм и характеризует содержание элемента диаграммы или его назначение. Они задаются на диаграммах названием, которое приводится в двойных угловых скобках (например, <<треугольник>>,<<кривая>>).

Некоторые стереотипы являются фиксированными в UML и имеют стандартные названия, например, <<актер>>, <<система>>, <<подсистема>>, <<исключительная ситуация>>, <<интерфейс>> и т.п.

Кроме того, разработчик может вводить стереотипы, типичные для своего домена, т.е. UML разрешает расширять и адаптировать стереотипы к конкретным областям применения и облегчают понимание конкретных моделей.

Совокупность диаграмм метода отображает наиболее важные случаи функционирования системы. Рассмотрим более подробно основные диаграммы.

Диаграмма классов (Class diagram) отображает онтологию домена и по содержанию эквивалентна информационной модели метода С.Шлаера и С.Меллора. Она определяет состав классов объектов и их взаимоотношения.

Диаграмма задается иконами и связями между ними. Икона –стандартизированное и фиксированное визуальное изображение определенного понятия, имеет вид прямоугольника, который может быть разделен на две или три части. Верхняя часть – обязательная, определяет имя класса. Вторая и третья части определяют соответственно – список атрибутов класса и список операций класса.

Атрибутами могут быть типы значений в UML:
– public (общий), что означает операцию класса, вызванную из любой части программы любым объектом системы,
– protected (защищенный) означает операцию, вызванную лишь объектом того класса, в котором она определена или наследована,
– private (частный) означает операцию, вызванную только объектом того класса, в котором она определена.

В то же время пользователь может определять специфические для него атрибуты. Под операцией понимается сервис, который экземпляр класса может выполнять, если к нему будет произведен соответствующий вызов. Операция имеет название и список аргументов.

Классы могут находиться в следующих отношениях или связях.

Ассоциация – взаимная зависимость между объектами разных классов, каждый из которых является равноправным ее членом. Она может обозначать количество экземпляров объектов каждого класса, которые принимают участие в связи (0 – если ни одного, 1 – если один, * – если много).

Зависимость между классами, при которой класс–клиент может использовать определенную операцию другого класса; классы могут быть связаны отношением трассирования, если один класс трансформируется во второй в результате определенного процесса ЖЦ.

Экземпляризация – зависимость между параметризированным абстрактным классом–шаблоном (template) и реальным классом, который инициирует параметры шаблона (например, контейнерные классы языка C++).

Диаграмма сценариев. Содержание и нотация этой диаграммы полностью совпадает с той, которая описана в методе Э.Джекобсона (см.п.3.).

Диаграммы моделирования поведения системы. Под поведением системы понимается множество объектов, которые обмениваются сообщениями с помощью следующих диаграмм:

– последовательность, упорядочивающей взаимодействие объектов при обмене сообщениями;
– сотрудничество, определяющей роль объектов во время их взаимодействия;
– активность, показывающей потоки управления при взаимодействии объектов;
– состояний, указывающих на динамику изменения объектов.

Диаграмма последовательности применяется для задания взаимодействия объектов. Любому из объектов сценария ставится в соответствие его линия жизни, которая отображает ход событий от его создания до разрушения. Диаграмма представляет все объекты, которые принимают участие во взаимодействии. Их порядок не имеет принципиального значения и выбирается произвольно, руководствуясь наглядностью взаимодействия.

Взаимодействие объектов контролируется событиями, которые происходят в сценарии и стимулируют посылки сообщений другому объекту.

Диаграммы сотрудничества представляют совокупность объектов, поведение которых имеет значение для достижения целей системы, и взаимоотношения тех ролей, которые важны в сотрудничестве. На этой диаграмме можно моделировать статическое взаимодействие объектов без учета фактора времени. При параметризации диаграмма представляет абстрактную схему сотрудничества – *паттерн*, для которого может быть создано определенное множество конкретных схем сотрудничества.

Диаграмма деятельности представляет поведение системы в виде определенных работ, которые может выполнять система или актер и эти работы могут зависеть от принятия решений в зависимости от условий, которые сложились. Эта диаграмма напоминает блок–схемы алгоритмов и программ, но в отличие от них, предусмотрена возможность выполнять параллельно несколько деятельностей и точки синхронизации их завершения.

Диаграмма состояний базируется на использовании расширенной модели конечного автомата и определяет:
– условия переходов и действия при переходе;
– действия при входе в состояние, его активности и при выходе из состояния;
– вложенные и параллельно действующие состояния.

Переход по списку данных инициирует некоторое событие. Состояние зависит от условий перехода, что подобно тому, как взаимодействуют две параллельно работающие машины, если изменение состояния одной машины влияет на другую машину.

Диаграмма реализации состоит из диаграммы компонента и диаграммы размещения.

Диаграмма компонента отображает структуру системы как композицию компонентов и связей между ними. Это граф, узлами которого являются компоненты, а дуги отображают отношения зависимости.

Диаграмма размещения задает состав физических ресурсов системы (узлов системы) и отношений между ними. Разрабатывается не только программная часть, но и необходимые аппаратные устройства, которые должны взаимодействовать с программными компонентами. В диаграмме обычно используются стереотипы: <<процессор>>, <<устройство>>, <<дисплей>>, <<память>>, <<диск>> и др.

Пакеты в UML Предусмотрен общий механизм организации некоторых элементов (объектов, классов, подсистем и т.п.) в группы. Группирование возможно начиная от системы к подсистемам разного уровня детализации, вплоть до классов. Результат группирования называется *пакетом*.

Пакет определяет название пространства, занимаемого элементами, которые являются его составляющими и средством ссылки на это пространство. Это важно для больших систем, которые насчитывают сотни, а иногда и тысячи элементов, и потому требуют иерархического структурирования.

Подсистема в UML рассматривается как случай пакета, который имеет самостоятельную функцию. Пакет может быть вложенным, то есть состоять из пакетов, классов, подсистем и т.п.

Объединение элементов в пакеты может происходить из разных соображений, например, если они используются совместно или созданы одним автором, или касаются определенного аспекта рассмотрения, как например, интерфейс с пользователем, устройства ввода/вывода и т.п. На стадии реализации к одному пакету могут быть отнесены все подсистемы, которые в диаграмме размещения связаны с одним узлом.

Пакет может быть элементом конфигурации, как элемент композиции при построении системы, на которую можно ссылаться в разных диаграммах. Термином *конфигурация* обозначается структура программной системы из отдельных модулей или из заведомо определенного состава их вариантов. Так, например, ОС может включать конфигурацию модулей, которые позволяют взаимодействие с разнообразными устройствами, но лишь отдельные из них подключаться к данному компьютеру с созданной версией ОС в виде конкретной конфигурации.

Среди фиксированных стереотипов для обозначения разновидностей пакета введены такие: система, прикладная система, подсистема, элемент конфигурации, составная системы, охватывающая система и др. Например, стереотип <<унаследовано>> может прибавляться к пакету, который является элементом старой версии системы и без переработки включен в новую версии системы.

5.1.4. Компонентный подход к проектированию

По оценкам экспертов, 75 % работ по программированию в мире дублируются (например, программы складского учета, начисления зарплаты, расчета затрат на производство продукции и т.п.). Большинство из этих программ типовые, но каждый раз находятся особенности, которые влияют на их повторное разработку. Компонентное проектирование сложных программ из готовых компонентов является наиболее производительным [8–12].

Переход к компонентам происходил эволюционно: от подпрограмм, модулей, функций. При этом усовершенствовались элементы, методы их композиции и накопления для дальнейшего использования (рис.5.2).

Компонентный подход дополняет и расширяет существующие подходы в программировании, особенно ООП. Объекты рассматриваются на логическом уровне проектирования программной системы, а компоненты – это непосредственная физическая реализация объектов.

Один компонент может быть реализацией нескольких объектов или даже некоторой части объектной системы, полученной на уровне проектирования. Компоненты конструируются как некоторая абстракция, которая состоит из трех частей: информационной, внешней и внутренней.

Информационная часть представляет собой информацию о компоненте: назначение, дата изготовления, условия применения (ОС, среда, платформа и т.п.); уровень повторного использования; контекст или окружение; способ взаимодействия между собою компонентов.

Элемент композиции	**Описание элемента**	**Схема взаимодействия**	**Представле–ние, хранение**	**Результат композиции**
Процедура, подпрограмма, функция	Идентификатор	Непосредственное обращение, оператор вызова	Библиотеки подпрограмм и функций	Программа
Модуль	Паспорт модуля, связи	Вызов модулей, интеграция модулей	Банк, библиотеки модулей	Программа с модульной структурой
Объект	Описание класса	Создание экземпляров классов, вызов методов	Библиотеки классов	Объектно–ориентиро–ванная программа
Компонент	Описание логики (бизнес), интерфейсов (APL,IDL), схемы развертывания	Удаленный вызов в компонентных моделях (COM, CORBA, OSF, …)	Репозитарий компонентов, серверы и контейне–ры компонентов	Распреде–ленное компонентно–ориенти–рованное приложение
Сервис	Описание бизнес–логики и интерфей–сов сервиса (XML, WSDL, …)	Удаленный вызов (RPC, HTTP, SOAP, …)	Индексация и каталогизация сервисов (XML, UDDI, …)	Распреде–ленное сер–висо–ориен–тированное приложение

Рис.5.2. Схема эволюции элементов компонентов

Внешняя часть определяет взаимодействие компонента со средой и с платформой, на которой он будет выполняться. Эта часть имеет следующие основные характеристики:

– интероперабельность как способ взаимодействия с другими компонентами, с клиентом или сервером, а также обеспечения переносимости на другую платформу;
– способ интеграции (композиции) компонентов;
– нефункциональные сведения (безопасность, аутентификация, надежность и др.);
– технология проектирования (например, объектно–ориентированная среда и т.п.) и повторное использования компонентов.

Внутренняя часть – это некоторый артефакт (кластер, системная или абстрактная структура, фрагмент кода и др.) и вид его представления: проектный компонент, проектная спецификация, вычисляемая часть, код и др.

Разработана новая архитектура компонента – контейнер, в котором определяются функции, порядок их выполнения, вызываемые события и сервисные свойства. Выполнение контейнера осуществляется через его интерфейс с помощью провайдера..

Компоненты наследуются в виде классов и используются в модели или композиции, а также в каркасе интегрированной среды. Управление компонентами проводится на трех уровнях: архитектурном, компонентном и на уровне инфраструктуры интерфейса. Между этими уровнями существует взаимная связь.

Внутренняя часть компонента состоит из (рис.5.3): интерфейса (interfaces), реализации (implementation), схемы развертки (deployment).

ХАРАКТЕРИСТИКИ		
Интерфейс	**Реализация**	**Схемы развертывания**
♦ Один или несколько; ♦ Уникальность именования в пределах системы; ♦ Клиентский или серверный (входной или выходной); ♦ определенная сигнатура; ♦ описание методов взаимодействия	♦ одна или несколько; ♦ ориентация на конкретную платформу и операционное окружение ♦ выбор конкретной реализации; ♦ поддержка интерфейсов компонента	♦ типовость процедуры развертывания; ♦ управляемость; ♦ настраиваемость на операционную среду; ♦ модифицируемость

Рис.5.3. Основные составные элементы компонента

Интерфейсы отображают взгляд пользователя на компонент, то есть что компонент будет делать, когда к нему обращаются.

Реализация – это код, который будет использоваться при обращении к операциям, которые определены в интерфейсах компонента. Компонент может иметь несколько реализаций, например, в зависимости от операционной среды или от модели данных и соответствующей системы управления базами данных, которая необходима для функционирования компонента.

Развертка – это физический файл или архив, готовый к выполнению, который передается пользователю и содержит все необходимые способы для создания, настройки и функционирования компонента.
Компонент описывается в языке программирования, не зависит от операционной среды (например, от среды виртуальной машины JAVA) и от реальной платформы (например, от платформ в системе CORBA), где он будет функционировать.

Расширением понятия компонента есть паттерн – абстракция, которая содержит описание взаимодействия совокупности объектов в общей кооперативной деятельности, для которой определены роли участников и их ответственность. Представляется повторяемой

частью программного элемента, как схемы или взаимосвязи контекста описания решения проблемы.

Интерфейс компонентов. Для объединения компонентов в компонентную модель необходимым условием является наличие формально определенных интерфейсов в языках IDL и APL, а также механизмов динамического контроля связей между компонентами в современных средах.

Спецификация интерфейса в API и IDL включает описание функциональных свойств компонентов, их типов и порядка задания операций передачи аргументов и результатов для взаимодействия компонентов. То есть, компонент – физическая сущность, которая реализует определенную совокупность интерфейсов. Сами интерфейсы являются понятием, которое связывает логическую и физическую модели. Для описания самих компонентов, как правило, применяется ООП и его свойства: наследование, инкапсуляция, полиморфизм. В языке JAVA понятие интерфейса и класса являются базовыми. Компонентные модели – Javabeans и Enterprise Javabeans, а также модель CORBA используют объектно–ориентированные свойства.

5.1.4.1. Типы компонентных структур

Компонентная модель – отражает многочисленные проектные решения по композиции компонентов, определяет типы паттернов компонентов и допустимые между ними взаимодействия, а также снижает время развертывания программной системы в среде функционирования.

Каркас – представляет собой высокоуровневую абстракцию проекта программной системы, в которой отделены функции компонентов от задач управления ими. То есть, бизнес–логика – это функции компонентов, а каркас задает правильное и надежное управление ими. Каркас объединяет множество взаимодействующих между собою объектов в некоторую интегрированную среду для решения заданной конечной цели. В зависимости от специализации каркас называют "белым или черным ящиком".

Каркас типа "белый ящик" включает абстрактные классы для представления цели объекта и его интерфейс. При реализации эти классы наследуются в конкретные классы с указанием соответствующих методов реализации. Использование такого типа каркаса более характерно для ООП.

Для каркаса типа «черный ящик» в его видимую часть выносятся точки, разрешающие изменять входы и выходы.

Композиция компонентов включает четыре возможных класса:

– *композиция компонент–компонент* обеспечивает непосредственное взаимодействие компонентов через интерфейс на уровне приложения;
– *композиция каркас–компонент* обеспечивает взаимодействие каркаса с компонентами, при котором каркас управляет ресурсами компонентов и их интерфейсами на системном уровне;
– *композиция компонент–каркас* обеспечивает взаимодействие компонента с каркасом по типу «черного ящика», в видимой части которого находится спецификация

для развертывания и выполнения определенной сервисной функции на сервисном уровне;
– *омпозиция каркас–каркас* обеспечивает взаимодействие каркасов, каждый из которых может разворачиваться в гетерогенной среде и разрешать компонентам, входящим в каркас, взаимодействовать через их интерфейс на сетевом уровне.

Компоненты и их композиции, как правило, запоминаются в репозитарии компонентов, а их интерфейсы к репозитарии интерфейсов.

Повторное использование в компонентном программирование в общем случае представляет собой любые порции формализованных знаний, добытые во время реализации программных систем и используемых в новых разработках [13–15].

Повторно используемые компоненты (ПИК} – элементы знаний о минувшем опыте разработки систем, которые могут использовать не только их разработчиками, но и путём адаптации к новым условиям. ПИК упрощает и сокращает сроки разработки новых программных систем. Высокий уровень стандартизации и распространение электронных коммуникаций (сети Интернет) обеспечивает довольно простое получение и широкое использование готовых компонентов в разных проектах за счет:

– отражения фундаментальных понятий приложения;
– скрытия способа представления и предоставления операций обновления и получения доступа;
– обработки исключительных операций в приложении.

При создании компонентов, предназначенных для повторного использования, общий интерфейс должен содержать операции, которые обеспечивают разные способы использования компонентов. Возможность повторного использования приводит к усложнению компонента, а значит к уменьшению понятности. Поэтому требуется некоторый компромисс между ними.

Как и любые элементы промышленного производства, компоненты должны отвечать определенным требованиям, иметь характерные свойства, структуру, механизмы использования и др. В UML все компоненты делятся на три типа:

1) для развертывания в компьютерной среде;
2) как рабочие продукты (файлы текстов с программами и данными, элементы с описаниями архитектуры и правилами генерации конечного кода и др.);
3) для среды выполнения (временные программные объекты, файлы, таблицы базы данных и др.).

Главным преимуществом создания программных систем из компонентов является уменьшение затрат на разработку за счет:

– выбора готовых компонентов с подобными функциями, пригодными для практического применения;
– настраивания готовых компонентов на новые условия, которые связаны с меньшими усилиями, чем разработка новых компонентов.

Поиск готовых компонентов основывается на их классификации и каталогизации. Метод классификации предназначен для представления информации о компонентах с целью быстрого поиска и отбора. Метод каталогизации обеспечивает физическое

размещение компонентов в репозитариях для непосредственного доступа к ним в процессе интеграции.

Интероперабельность компонентов. Сложились ряд подходов для решения этой проблемы, наиболее часто они связаны с анализом кода для внесения изменений при переносе его в другую среду. Механизмы обеспечения интероперабельности имеют разные концепции и реализации, т.е. приведение представления одного компонента к форме, понятной второму либо к некоторому промежуточному состоянию.

Стандартный механизм интероперабельности для связи между Java и C/C++ компонентами использует Java Native Interface (JNI) при реализации обращения из Java–классов к функциям и библиотекам на других ЯП. Механизм реализации включает: анализ Java–классов для поиска прототипов обращений к функциям на C/C++; генерацию заглавных файлов для использования их при компиляции C/C++ программ. Java–класс "знает", что в нем помещается обращение не к Java–методу. Схема связи Java → C/C++ [16].

Другой вариант реализации этой задачи предлагает технология Bridge2Java фирмы IBM alpha Works. В ней обеспечивается обращение из Java–классов к COM–компонентам [5]. Для COM–компонента генерируется оболочка, которая включает прокси–класс для преобразования данных с помощью стандартной библиотеки типов и вызов COM–функций. Данная схема не требует изменений в исходном Java–классе, а COM–компоненты могут быть написаны на разных ЯП.

Механизм интероперабельности предлагает и платформа .Net [17] с помощью промежуточного языка Common Language Runtime (CLR), в который транслируются коды, написанные в ЯП (C#, Visual Basic, C++, Jscript) и интегрируются эти компоненты. При этом используется библиотека стандартных классов независимо от языка реализации и доступа к готовым компонентам без ориентации на эту платформу (например, к COM–компонентам). С этой целью используются стандартные средства генерации оболочки для COM–компонента в представление .Net–компонента, а также для приведенных выше ЯП.

Особенности ОС и архитектур компьютеров учитывает среда CORBA, в которой реализована иерархия механизмов интероперабельности – от самого верхнего уровня (поддержка разных ЯП) к самому нижнему (с учетом архитектуры).

5.1.4.2. Методология компонентной разработки систем

Эта методология включает в себя две основные фазы процесса разработки.

1. *Разработка отдельных компонентов* исходя из следующих требований:
- формализованное определение спецификаций интерфейсов, поведения и функциональности компонента;
- использование системы классификации для поиска и отбора необходимых компонентов, а также для их физического размещения;
- обеспечение принципа повторности.

Интеграция (композиция) компонентов в более сложные программные образования:

- разработка требований (Requirements) к программной системе;
- анализ поведения (Behavioral Analysis) программной системы;

- спецификация интерфейсов и взаимодействия компонентов (Interface and Interaction Specification);
- интеграция разработанных компонентов и компонентов повторного использования (Application Assembly and Component Reuse) в единую среду;
- тестирование компонентов и среды;
- развертывание (System Deployment) программной системы у пользователя;
- поддержка и сопровождение программной системы (System Support and Maintenance).

Описание этапов ЖЦ приведено в Приложении 3.

5.1.5. Аспектно–ориентированное программирование

Аспектно–ориентированное программирование (АОП) [18–20] – это парадигма построения гибких к изменению ПС за счет добавление новых функций, средств безопасности и взаимодействия компонентов с другой средой, синхронизации одновременного доступа частей ПС к данным, вызова общесистемных средств и др.

Аспектом может быть некоторая функция, ПИК, элемент или часть готовой программы, отдельный компонент, концепция взаимодействия, защиты и др. Созданная средствами АОП ПС из отдельных программ семейства может включать набор ПИК, объекты, небольшие методы и аспекты, как средство дополнения ПС необходимыми концепциями взаимодействия или защиты для новой среды, которые пересекают (переплетают) компоненты и тем самым значительно усложняют процесс вычислений.

Реализация аспектов в различных частях программного кода ПС решается путем установления перекрестных ссылок и точек соединения, через которые осуществляется связь аспекта с транзакциями, защитой данных и т.п.

В основе АОП лежит метод разбиения задач ПрО на ряд функциональных компонентов с применением аспектов (синхронизации, взаимодействия, защиты и др.), которые встраиваются в отдельные компонентов в некоторые их точки для выполнения соответствующих нефункциональных требований к организации выполнения компонента с другими компонентами или средами.

Кроме того, в качестве аспекта может использоваться некоторая задача, которая интересует нескольких заинтересованных лиц проекта и представленная с помощью вариантов использования, функции для компонента или программы. Некоторые аспекты могут реализовываться на этапах ЖЦ процесса разработки, способствуя улучшению результат разработки ПС.

Создание конечного продукта ПС в АЛП выполняется по технологии, соответствующей разработке компонентных систем, с той разницей, что здесь используются аспекты, которые задают условия выполнения компонентов (безопасность, защиту, взаимодействие и др.) в среде функционирования. В процессе разработки аспекты отображают разные роли взаимодействующих лиц, что приближает аспект к роли программного агента в плане выполнения некоторых функций при определении архитектуры системы, управления проектом и повышения качества ПС.

Для использования аспекта в проектных решениях в АОП введен дополнительный механизм фильтрации входных сообщений, с помощью которых выполняется

изменение параметров и имен текстов аспектов в конкретно заданном компоненте системы. Код компонента становится «нечистым» (т.е.с пересекаемыми его аспектами), для которого требуется разработка новых подходов к композиции компонентов, ориентированных на Уро и на выполнение ее функций.

Общие средства композиции объектов ООП или компонентов (вызов процедур, RPC, RMI, IDL и др.) в АОП являются недостаточными, так как аспекты требуют декларативного сцепления между частичными описаниями, а также связывания отдельных обрывков из различных объектов. Одним из механизмов композиции является фильтр композиции, суть которого состоит в обновлении заданных аспектов синхронизации или взаимодействия без изменения функциональных возможностей компонента с помощью входных и выходных параметров сообщений, которые проходят фильтрацию и изменения, связанные с переопределением имен или функций самих объектов.

Фильтры делегируют внутренним компонентам параметры, переадресовывая установленные ссылки, проверяют и размещают в буфере сообщения, локализуют ограничения на синхронизацию и готовят компонент для выполнения.
В ОО–программах могут быть мелкие методы, дополнительно выполняющие расчеты с обращением к другим методам внешнего уровня. Деметер сформулировал закон [40–43], согласно которому длинные последовательности мелких методов не должны выполняться. В результате создается код алгоритма с именами классов, не задействованных в выполнении расчетных операций, а также новый дополнительный класс, который расширяет этот код функциями изменения расчетных программ.

С точки зрения моделирования, аспекты можно рассматривать как каркасы декомпозиции системы, в которых отдельные аспекты синхронизации, взаимодействия и др. пересекают ряд многократно используемых ПИК (рис.5.4).

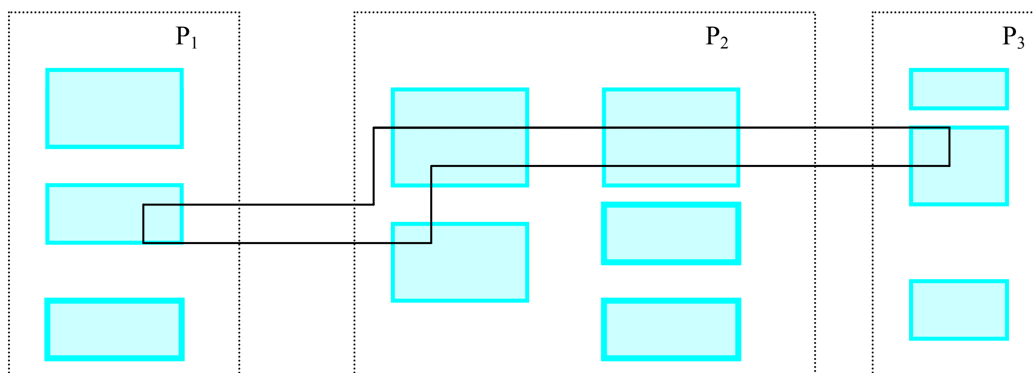

Рис. 5.4. Пример структуры программы из P_1, P_2 и P_3 аспектами защиты

Разным аспектам проектируемой системы могут отвечать и разные парадигмы программирования: объектно–ориентированные, структурные и др. Они по отношению к проектируемой ПрО образуют мультипарадигмную концепцию аспектов, такую как синхронизация, взаимодействие, обработка ошибок и др. и требуют значительных доработок процессов их реализации. Кроме того, можно устанавливать связи с другими предметными областями для описания аспектов приложения в терминах родственных

областей. Появились языки АОП, которые позволяют описывать пересекающиеся аспекты в разных ПрО. В процессе компиляции переплетения объединяются, оптимизируются и генерируются [20] и выполняются в динамике.

Существенной чертой любых аспектов является модель, которая пересекает структуру другой модели, для которой первая модель является аспектом. Так как аспект связан с моделью, то ее можно перестроить так, чтобы аспект стал, например, модулем и выполнял функцию посредника, беря на себя все образцы взаимодействия. Однако решение, таким образом, проблемы пересечения может привести к усложнению и понижению эффективности выполнения созданного модуля или компонента.

Переплетение аспектов с компонентами может проявиться на последующих этапах процесса разработки, поэтому требуется минимизация количества сцеплений между аспектами и компонентами путем реализации ссылок в вариантах использования или сопоставления с образцом. Аспекты реализуются блоками кода с установленными перекрестными ссылками между ними, иными словами в блоках появляются точки связей с сообщениями, обработкой ошибок и т.п.

Связь между характеристиками и аспектами ПС может быть выявлена в ходе анализа ПрО. Тогда создается динамическое связывание или статическое или «жесткое» связывание в период компиляции.

АОП стимулирует разработку новых механизмов композиции, ориентированных на ПрО и на выполнение ее задач. Аспекты с точки зрения моделирования можно рассматривать как каркасы декомпозиции системы с многократным использованием. АОП соответствует мультипарадигмовой концепции, сущность которой состоит в том, что разным аспектам проектируемой ПС, должны отвечать разные парадигмы программирования: объектно–ориентированные или структурные. Каждая из парадигм относительно реализации разных аспектов ПС (синхронизации, внедрения, обработки ошибок и др.) требует усовершенствования и обобщения применительно к каждой новой ПрО.

В АОП используется модель модульных расширений, создаваемая в рамках метамодельного программирования. Эта модель ориентирована на оперативное использование новых механизмов композиции отдельных частей ПС или семейств с учетом предметно–ориентированных возможностей языков (например, SQL) и каркасов, которые поддерживают разного рода аспекты [20].

Технология разработки прикладной системы с использованием АОП базируется на технологии ООП и имеет вид:

1..Декомпозиция функциональных задач с предположением многоразового применения соответствующих модулей и выделение аспектов, т.е свойств их выполнения (параллельно, синхронно, безопасно и т.д.).
2. Анализ языков спецификации аспектов и определение конкретных аспектов для выполнения задач ПрО;
3. Определение в модулях точек соединения аспектов для формирования ссылок на них.
4. Разработка фильтров и описание связей аспектов с функциональными компонентами, выделенными в ПрО. Система фильтров отображается в модели EJB, работающей на стороне сервера и управляющей данными с обеспечением безопасности и защитой доступа;

5. Определение механизмов композиции (вызовов процедур, методов, сцеплений) функциональных модулей многоразового применения и аспектов в точках их соединения, как фрагментов модулей с обеспечением свойств управления выполнением этих модулей, или ссылок из этих точек на другие модули.
6. Создание объектной или компонентной модели, дополнение ее входными и выходными фильтрами сообщений, посылающих объектам с ссылками, задание на выполнение методов или аспектов управления синхронизацией, защитой и т.д.
7. Анализ библиотеки расширений для выбора некоторых функциональных модулей, необходимых для реализации задач домена.
8. Компиляция, отладка модулей и аспектов, а также композиция их в прикладную программу.

Для эффективной реализации аспектов разработаны IP–библиотека расширений, активные библиотеки, Smalltalk и ЯП, расширенные средствами описания аспектов.

В *IP–библиотеке* размещены некоторые функции компиляторов, методов, средства оптимизации, редактирования, отображения. и др. Например, библиотека матриц, с помощью которой вычисляются выражения с массивами, обеспечивается скорость выполнения, предоставления памяти и т.п.[21]. Использование таких библиотек в расширенных средах программирования называют родовым программированием, а решение проблем экономии, перестройки компиляторов под каждое новое языковое расширение, использование шаблонов и результатов предыдущей обработкой относят к области ментального программирования [22].

Библиотека включают отдельные функции компиляторов, средств оптимизации, редактирования, отображения понятий, перестройки отдельных компонентов компиляторов под новое языковое расширения, а также средства программирования на основе шаблонов и т.п. Библиотеки с такими возможностями получили название библиотек генерирующего типа.

Иной вид библиотек АОП – *активные библиотеки*, которые содержат не только базовый код реализации понятий ПрО, но и целевой код обеспечения компиляции, оптимизации, адаптации, визуализацию и редактирование.

Активные библиотеки пополняются средствами и инструментами интеллектуализации агентов, с помощью которых поддерживается разработка специализированных агентов для решения конкретных задач реализуемой ПрО.

5.1.6. Генерирующее (порождающее) программирование

Порождающее программирование (generate programming) основана на генерации и моделировании групп или отдельных элементов ПС из разных продуктов программирования: объектов, компонентов, аспектов, сервисов, ПИК, систем, характеристик, каркасов и т.п. Базисом этого программирования является ООП, дополненное механизмами применения ПИК, а также свойствами изменчивости, взаимодействия, синхронизации и др. [22].

В нем используются другие методы программирования, например, для поддержки инженерии ПрО как дисциплины проектирования семейств ПС из разных ранее указанных продуктов программирования.

В рамках данного программирования построена объединенная технология генерации как отдельных ПС, так и их семейств. В результате в нем сформирован базис будущего программирования, включающий современные методы программирования, новые формализмы и объединяющие модели, посредством которых можно будет создавать более долговременные качественные программные изделия семейств ПС по принципу конвейера.

Главным элементом проектирования ПС является не уникальный программный продукт, созданный из ПИК для конкретных применений, а семейство ПС или конкретные его экземпляры. Элементы семейства не создаются с нуля, а генерируются на основе общей генерирующей модели домена (generative domain model), т.е. модели семейства, включающей средства определения членов семейства, компоненты реализации и ПИК для сборки любого члена семейства и базы конфигурации, специфицирующей членов семейства.

Каждый член семейства отражает максимум знаний о его производстве, а именно, конфигурации, инструментарии измерения и оценки, методах тестирования и планирования, отладки, визуального представления, а также о многократно используемых ПИК из активной библиотеки [21, 22].
Базовый код элементов активной библиотеки содержит целевой код по обеспечению процедур компиляции, отладки, визуализации и др. Фактически компоненты активных библиотек выполняют роль интеллектуальных агентов, в процессе взаимодействия которых создаются новые агенты, ориентированные на предоставление пользователю возможности решать конкретные задачи ПрО. Для выполнения агентами задач генерации, преобразования и взаимодействия должна создаваться инфраструктура, а именно, расширяемая среда программирования.

Эта среда предназначена для конструирования ПС из компонентов библиотек, а также специальных метапрограмм среды, которые осуществляют редактирование, отладку и взаимодействие компонентов непосредственно в расширяемой среде. С другой стороны, имеется возможность пополнять эту среду новыми сгенерированными компонентами в рамках отдельных ПС семейства, которые относятся к числу компонентов многоразового применения.

Целью порождающего программирования является разработка правильных компонентов для целого семейства и автоматическое их предоставление другим членам семейства. Реализации этой цели соответствует два сформировавшихся направления использования ПИК[13–15]:

1) *прикладная инженерия* – процесс производства конкретных ПС из ПИК, созданных ранее в среде самостоятельных ПС, или как отдельных элементов процесса инженерии некоторой ПрО.
2) *инженерия* ПрО – построение семейства ПС путем сбора, классификации и фиксации ПИК, опыта конструирования систем или готовых частей систем конкретной ПрО. При этом создаются системные инструментальные системы поддержки поиска, адаптации ПИК и внедрения их в новый элемент семейства ПС или самого ПС.

Инженерия ПрО включает в себя инженерию приложений как способ создания отдельных одиночных членов семейства, а также метод конструирования семейств приложений и компонентных систем через механизмы разделения задач ПрО на

отдельные члены и многократно используемые решения для сборки отдельных подсистем и членов семейства в общую систему для ПрО.

Основными этапами инженерии ПрО являются:
– анализ ПрО и выявление объектов и отношений между ними;
– определение области действий объектов ПрО;
– определение общих функциональных и изменяемых характеристик, построение модели характеристик, устанавливающей зависимость между различными членами семейства, а также в пределах членов семейства системы;
– создание базиса для производства конкретных программных членов семейства с механизмами изменчивости независимо от средств их реализации;
– подбор и подготовка компонентов многократного применения, описание аспектов выполнения задач ПрО;
– генерация отдельного домена, члена семейства и ПС в целом.

Генерация доменной модели для семейства ПС основывается на модели характеристик, наборе компонентов реализации задач ПрО, конфигурации и спецификации компонентов. Эти элементы генерируются готовую систему или отдельных членов семейства.

Для реализации инженерии ПрО используются следующие вспомогательные процессы:
– *корректировка процессов* для разработки решений на основе ПИК;
– *моделирование изменчивости и зависимостей,* которое начинается с разработки словаря описания различных понятий, фиксации их в модели характеристик и в справочной информации сведений об изменчивости моделей (объектных, Use Case, взаимодействия и др.). Фиксация зависимостей между характеристиками модели избавляет пользователей от некоторых конфигурационных манипуляций, которые выполняются, как правило, вручную;
– *разработка инфраструктуры ПИК* – описание, хранение, поиск, оценивание и объединение готовых ПИК.

При определении членов семейства ПрО используются пространство проблемы и пространство решений.

Пространство проблемы (speace problem) состоит из компонентов семейства системы, в которых используется ПИК, объекты, аспекты и др. Процесс разработки этих членов семейства с ПИК включает в себя и инструменты, созданные в ходе разработки ПрО. В рамках инженерии ПрО разрабатывается модель характеристик, которая объединяет функциональные характеристики системы, характеристики определения свойств выполнения компонентов и изменяемые параметры разных частей семейства, а также решения, связанные с особенностями выполнения групп ПС.

Инженерия ПрО включает разработку моделей групп систем, моделирование понятий ПрО, разработку их моделей характеристик и групп систем для последующего повторного использования. В рамках инженерии ПрО используются горизонтальные и вертикальные типы компонентов, предложенные OMG–комитетом в системе объектного проектирования Corba [23, 24].

К горизонтальным типам компонентов отнесены общие системные средства, а именно, графические пользовательские интерфейсы, СУБД, системные программы, библиотеки расчета матриц, контейнеры, каркасы и т.п. К вертикальным типам компонентов

относятся прикладные системы (медицинские, биологические, научные и т.д.), методы инженерии ПрО, а также компоненты из горизонтального типа по обслуживанию архитектуры многократного применения, интерфейсов и др.

Пространство решений (speace solution) состоит из компонентов, каркасов, образцов проектирования, а также средств их соединения и оценки избыточности. Эти элементы обеспечивают решение задач ПрО. Так, *каркас* оснащен аппаратом обеспечения изменения параметров модели, требующих лишнюю фрагментацию из «множества мелких методов и классов». *Образцы* проектирования обеспечивают создание многократно используемых решений в различных типах ПС. Для задания и реализации таких аспектов, как синхронизация, удаленное взаимодействие, защита данных и т.д. применяются технологии ActiveX и JavaBeans, а также новые механизмы композиции, метапрограммирования и др.

Примером систем поддержки инженерии ПрО и реализации горизонтальных методов является система DEMRAL [22, 16], предназначенная для разработки библиотек: численного анализа, контейнеров, распознавания речи, графовых вычислений и т.д. Основными видами абстракций этих библиотек ПрО являются абстрактные типы данных (abstract data types– ADT) и алгоритмы. DEMRAL позволяет моделировать характеристики ПрО в виде высокоуровневой характеристической модели и предметно–ориентированных языков конфигурирования.

Система конструирования RSEB [22] базируется на вертикальных методах, ПИК и ориентирована на использование Use Case элементов при проектировании крупных ПС. Эффект достигается, когда вертикальные методы инженерии ПрО «вызывают» различные горизонтальные методы, относящиеся к разным прикладным подсистемам. При работе над отдельной частью семейства системы могут быть задействованы такие основные аспекты — взаимодействие, структуры, потоки данных и др. Главную роль, как правило, выполняет один из методов, например, графический пользовательский интерфейс в бизнес–приложениях и метод взаимодействия компонентов в распределенной, открытой среде (например, в CORBA).

5.1.7 Агентное программирование

Понятие интеллектуального и программного агента появилось более 20 лет назад, их роль в программной инженерии все время возрастает [25—29]. Так, в [30] Джекобсон отметил перспективу для использования агентов в качестве разработчиков архитектуры системы из вариантов использования, менеджеров проекта и др. Использование агентов в этих ролях в ближайшем будущем будет способствовать повышению производительности, качества и ускорению разработки ПС.

Основным теоретическим базисом данного программирования являются темпоральная, модальная и мультимодельная логики, дедуктивные методы доказательства правильности свойств агентов и др. Рассмотрим сущность основных прикладных аспектов агентной тематики.

С точки зрения программной инженерии, агент это самодостаточная программа, способная управлять своими действиями в информационной среде функционирования для получения результатов выполнения поставленной задачи и изменения текущего состояния среды.

Агент может обладать такими свойствами:
– автономность – это. способность действовать без внешнего управляющего воздействия;
– реактивность – это способность реагировать на изменения данных и среды, и воспринимать их;
– активность – это способность ставить цели и выполнять заданные действия для достижения этой цели;
– социальность – это способность к взаимодействию с другими агентами (или людьми).

В задачи программного агента входят:
– самостоятельная работа и контроль своих действий;
– взаимодействие с другими агентами;
– изменение поведения в зависимости от состояния внешней среды;
– выдача достоверной информации о выполнении заданной функции и т.п.

С интеллектуальным агентом связываются знания типа: убеждение, намерение, обязательства и т.п. Эти понятия входят в концептуальную модель и связываются между собой операционными планами реализации целей каждого агента. Для достижения целей интеллектуальные агенты взаимодействуют друг с другом, устанавливают связь между собой через сообщения или запросы и выполняют заданные действия или операции в соответствии с имеющимися знаниями.

Агенты могут быть локальными и распределенными. Процессы локальных агентов протекают в клиентских серверах сети, выполняют заданные функции и не влияют на общее состояние среды функционирования. Распределенные агенты, расположенные в разных узлах сети, выполняют автономно (параллельно, синхронно, асинхронно) предназначенные им функции и могут влиять на общее состояние среды. В обоих случаях характер взаимодействия между агентами зависит от таких факторов: совместимость целей, компетентность, не стандартные ситуации т.п. [27] (рис.5.5).

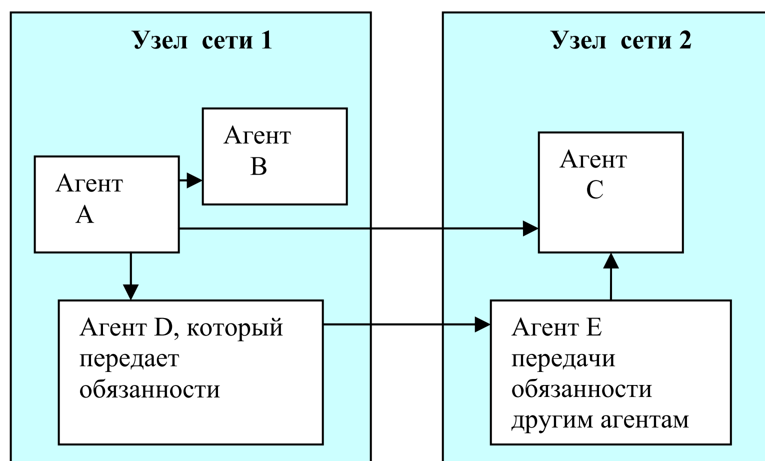

Рис.5.5 Пример взаимодействия агентов в разных средах

Основу агентно–ориентированного программирования составляют:
– формальный язык описания ментального состояния агентов;
– язык спецификации информационных, временных, мотивационных и функциональных действий агента в среде функционирования;
– язык интерпретации спецификаций агента;
– инструменты конвертирования любых программ в соответствующие агентные программы.

Агенты взаимодействуют между собой с помощью разных механизмов взаимодействия, а именно, координация, коммуникация, кооперация или коалиция.
Под *координацией агентов* понимается процесс обеспечения последовательного функционирования при согласованности их поведения и без взаимных конфликтов. Координация агентов определяется:
– взаимозависимостью целей других агентов–членов коалиции, а также от возможного влияния агентов друг на друга;

– ограничениями, которые принимаются для группы агентов коалиции в рамках общего их функционирования;
– компетенцией – знаниями условий среды функционирования и степени их использования.

Главным средством коммуникации агентов является транспортный протокол TCP/IP или протокол агентов ACL (Agent Communication Languages). Управления агентами (Agent Management) выполняется с помощью таких сервисов: передача сообщений между агентами, доступ агента к серверу и т.п.

Коммуникация агентов базируется на общем протоколе, языке HTML и декларативном или процедурном (Java, Telescript, ACL и т.п.) языке описания этого протокола.

Примером активного использования агентов является Интернет. В нем агенты обеспечивают доступ к информации, расположенной в информациооных ресурсах Интернет, анализ, фильтрацию и передачу клиенту результат запроса. В результате выполнения этих функций агенты создают некоторое поведение среды, которое в любой момент времени находится в некотором состоянии, а агент, выполняя заданные действия, изменяет его в целевое состояние и учитывает возможность возникновения нерегулярных состояний (тупиков, отсутствия ресурса и др.).

Одной из систем построения агентов, основанной на обмене сообщениями в ACL, является JATLite. Она включает Java–классы для создания новых агентов, ориентированных на вычисление функций в распределенной среде. Система Agent Builder предназначена для конструирования программных агентов, которые описываются в языке Java и могут взаимодействовать на основе языка KQML (Knowledge Guery and Manipulation Language). Построенные агенты выполняют функции: менеджера проекта и онтологий, визуализации, отладки и др. На реализацию механизмов взаимодействий агентов ориентирована и система JAFMAS. Ряд мультиагентных систем описано в [44].

5.2. Методы теоретического программирования

Теоретическое программирование основывается на функциональных математических дисциплинах (логика, алгебра, комбинаторика) и отражает математический метод анализа ПрО, осмысление постановок задач и разработку программ для получения на компьютере математических результатов. Специалисты с математическим образованием развивают отдельные направления в программировании, объясняя некоторые закономерности в структуре программ и их определении с различных точек зрения: аппарата функций (функциональное программирование, композиционное программирование и др.).

Алгебраисты использовали алгебраический математический аппарат для объяснения действий над объектами программ, выполнения математических операций над их элементами и принципов обработки, исходя из базовых основ алгебры – алгебраическое программирование, алгоритмика и др.

К настоящему времени разработаны теоретические методы с участием украинских ученых для теоретического представления ключевых проблем программирования – алгебраическое программирование (Летичевский А.А. и др.) [31–34];

Экспликативное программирование (Редько В.Н.) определяет теорию дескриптивных и декларативных программных формализмов для адекватного задания моделей структур данных, программ и средств их конструирования. Создана программология – наука о программах, которая объединяет идеи логики, конструктивной математики и информатики и на единой концептуальной основе предоставляет общий формальный аппарат конструирования программ [35–39].

Алгебра алгоритмики (Цейтлин Г.Е.) обеспечивает построение алгоритмов в виде схем, задаваемых графами, элементами которых являются конструкции, производные от структурных конструкций. К операциям алгебры относится суперпозиция, свертка, развертка, а также операции над множествами [40–42].

Далее будем рассматривать эти методы программирования для ознакомления студентов с теоретическими подходами в программировании.

5.2.1. Алгебраическое программирование (АП)

АП обеспечивает описание процессов конструирования программ, алгебраические преобразования, доказательство математических теорем и создание интеллектуальных агентов с помощью математиского аппарата, в качестве которого используется понятие транзитивной системы [31–34].

Данный аппарат позволяет определить поведение систем и их эквивалентность. В качестве транзитивных систем в общем случае могут быть компоненты, программы и их спецификации, объекты, взаимодействующие друг с другом и со средой их существования.

Эволюция такой системы описывается с помощью истории функционирования систем, которая может быть конечной или бесконечной, и включать обзорную часть в виде последовательности действий и скрытую часть в виде последовательности состояний. История функционирования включает в себя успешное завершение вычислений в среде транзитивной системы, тупиковое состояние, когда каждая из параллельно выполняющихся частей системы находятся в состоянии ожидания событий и, наконец, неопределенное состояние, возникающее при выполнении алгоритма, например, с бесконечными циклами.

Расширением понятия транзитивных систем является множество заключительных состояний с успешным завершением функционирования системы и без неопределенных состояний. Главным инвариантом состояния транзитивной системы является поведение системы, которое можно задать выражениями алгебры поведения *F(A)* на множестве операций алгебры А, а именно две операции префиксинга $a\cdot u$, задающие поведение *u* на операции *a*, недетерминированный выбор $u+v$ одной из двух поведений *u* и *v*, который является ассоциативным и коммутативным. Конечное поведение задается константами: Δ, $\perp$, 0, обозначающими соответственно состояние успешного завершения, неопределенного и тупикового состояния.

Алгебра поведения частично задается отношением $\leq$, для которого элемент $\perp$ является наименьшим, а операции алгебры поведения – монотонными. Теоретически алгебра поведения *F(A)* проверена путем доказательства теоремы про наименьшую неподвижную точку.

Транзитивные системы называют бисимуляцийно эквивалентными, если каждое состояние любой из них эквивалентно состоянию другой системы. На множестве

поведений определяются новые операции, которые используются для построения программ агентов. К ним относятся операции: последовательная композиция *(u; v)* и параллельная композиция *u//v*.

Среда E, где находится объект, определяется как агент в алгебре действий А и функции погружения от двух аргументов *Ins(e, u) = e[u]*. Первый аргумент – это поведение среды, второй – поведение агента, который погружается в эту среду в заданном состоянии. Алгебра агента – это параметр среды. Значения функций погружения – это новое состояние одной и той же среды.

Разработанная общая теория выходит за рамки определения вычислительных и распределенных систем, а также механизмов взаимодействия со средой. Базовым

понятием является “действие”, которое трансформирует состояние агентов, поведение которых, в конце концов, изменяется.

Поведение агентов характеризуется состоянием с точностью до бисимиляции и, возможно, слабой эквивалентности. Каждый агент рассматривается как транзитивная система с действиями, определяющими не детерминированный выбор и последовательную композицию (т.е. примитивные и сложные действия).

Взаимодействие агентов может быть двух типов. Первый тип выражается через параллельную композицию агентов над той же самой областью действий и соответствующей комбинацией действий. Другой тип выражается через функцию погружения агента в некоторую среду; результатом трансформации является новая среда.

Язык действий А имеет синтаксис и семантику. Семантика – это функция, определяемая выражениями языка и ставящая в соответствие программным выражениям языка значения в некоторой семантической области. Разные семантические функции дают равные абстракции и свойства программ. Семантика может быть вычислительной и интерактивной. Доказано, что каждая алгебра действий есть гомоморфным образом алгебры примитивных действий, когда все слагаемые разные, а представление однозначно с точностью до ассоциативности и коммутативности в детерминированном

выборе. Установлено, что последовательная композиция – ассоциативная, а параллельная композиция – ассоциативная и коммутативная. Параллельная композиция раскладывается на комбинацию действий компонентов.

Агенты рассматриваются как значения транзитивных систем с точностью до бисимиляционной эквивалентности. Эквивалентность характеризуется в алгебре поведения непрерывной алгеброй с аппроксимацией и двумя операциями: не детерминированным выбором и префиксингом. Среда вводится как агент, куда погружения функция, имеет поведение типа агент и среды. Произвольные непрерывные функции могут быть использованы как функции погружения и эти функции, определены значениями логики переписывания. Трансформации поведения среды, которые определяются функциями погружения, составляют новый тип эквивалентности – эквивалентность погружения.

Создание новых методов программирования с введением агентов и сред позволяет интерпретировать элементы сложных программ как самостоятельно взаимодействующие объекты.

В АП интегрируется процедурное, функциональное и логическое программирование, используются специальные структуры данных – граф термов, который разрешает использовать разные средства представления данных и знаний о ПрО в виде выражений многоосновной алгебры данных.

Наибольшую актуальность имеют системы символьных вычислений, которые дают возможность работать с математическими объектами сложной иерархической структуры. Многие алгебраические структуры (группы, кольца, поля) являются иерархически – модулярными. Теория АП обеспечивает создание математической информационной среды с универсальными математическими конструкциями, вычислительными механизмами, учитывающими особенности разработки ПС и

функционирования. АП является основой формирования нового вида программирования – инсерционного, обеспечивающего программирование систем на основе моделей поведения агентов, транзитивных систем и бисимуляционной эквивалентности [14].

Техника программирования в системе алгебраического программирования АПС использует аппарат переписывания терминов, необходимого при автоматизации доказательства теорем, символьных вычислениях, обработки алгебраических спецификаций. АПС реализован. мониторинг данных, моделирование ситуации, условное прерывание, управление экспериментами и др.

5. 2.2. Экспликативное программирование (ЭП)

ЭП ориентировано на разработку теории дескриптивных и декларативных программных формализмов, адекватных моделям структур данных, программ и средств конструирования из них программ [35–39]. Для этих структур решены проблемы существования, единства и эффективности. Теоретическую основу ЭП составляет логика, конструктивная математика, информатика, композиционное программирование и классическая теории алгоритмов. Для изображения алгоритмов программ используются разные алгоритмические языки и методы программирования: функциональное, логическое, структурное, денотационное и др.

К принципам ЭП относятся.

– *принцип развития* определения понятия программы в абстрактном представлении и постепенной ее конкретизации с помощью экспликаций.
– *принцип прагматичности* или полезности определения понятия программы выполняется с точки зрения понятия "проблема" и ориентирован на решение задач пользователя.
– *принцип адекватности* ориентирован на абстрактное построение программ и решение проблемы с учетом *информационности данных* и *апликативности,* т.е. рассмотрение программы, как функции, вырабатывающей выходные данные на основе входных данных. Функция является объектом, которому сопоставляется *денотат* имени функции с помощью отношения *именования* (*номинации).*
– *принцип дескриптивности* позволяет трактовать программу как сложные дескрипции, построенные из более простых и композиций отображения входных данных в результаты на основе п*ринципа вычислимости.*

Развитие понятия функции осуществляется с помощью п*ринципа композиционности,* т.е. композиция программ (функций) из более простых программ в целях создания новых объектов с более сложными именами (дескрипциями) для функций и включающими номинативные (именные) выражения, языковые выражения, термы и формулы.

Таким образом, процесс развития программы осуществляется в виде цепочки понятий: данные – функция – имя функции – композиция – дескрипция. Понятия "данные – функция – композиция" задают *семантический аспект* программы, а "данные – имя функции – дескрипция" – *синтаксический аспект*. Главным в ЭП является семантический аспект, система композиций и номинативности (КНС), ориентированные на систематическое изучение номинативных отношений при построении данных, функций, композиций и дескрипций [36].

КНС задают специальные языковые системы для описания разнообразных классов функций и называются композиционно–номинативными языками функций. Такие системы тесно связаны с алгебрами функций и данных, построены в семантико–синтаксическом стиле. Они отличается от традиционных систем (моделей программ) теоретико–функциональным подходом, классами однозначных *n*–арных функций. номинативными отображениями и структурами данных.

Для построения математически простых и адекватных моделей программ параметрического типа используется КНС и методы универсальной алгебры, математической логики и теория алгоритмов. Данные в КНС рассматриваются на трех уровнях: абстрактном, булевском и номинативном. Класс номинативных данных обеспечивает построение именных данных, многозначных номинативных данных или мульти–именных данных, задаваемых рекурсивно.

В рамках ЭП разработаны новые средства [36] для определения систем данных, функций и композиций номинативного типа, имена аргументов которых принадлежат некоторому множеству имен *Z,* т.е. композиция определяется на *Z–номинативных* наборах именных функций.

Номинативные данные позволяют задавать структуры данных, которым присущи неоднозначность именования компонентов типа множества, мультимножества, реляции и т.п.

Функции обладают свойством апликативности, их абстракции задают соответственно классы слабых и сильных аппликативных функций. Слабые функции позволяют задавать вычисление значений на множестве входных данных, а сильные – обеспечивают вычисление функций на заданных данных.

Композиции классифицируются уровнями данных и функций, а также типами аргументов. Экспликация композиций соответствует абстрактному рассмотрению функций как слабо аппликативных функций, и их уточнение строится на основе понятия детерминанта композиции, как отображения специального типа. Класс аппликативных композиций предназначен для конструирования широкого класса программ. Доказана теорема о сходимости класса таких композиций на классе монотонных композиций.

Практическая проверка теоретического аппарата формализации дедуктивных и ОО БД прошла в ряде экспериментальных проектов в *классе манипуляционных* данных БД заданных в SQL–подобных языках [39].

5.2.3. Алгоритмика программ

На протяжении многих лет Цейтлин Г.Е. занимается разработкой теоретических аспектов алгоритмов, представляемых блок–схемами, как способа детализации и

реализации алгоритмов. Аппарат блок–схем пополнен математическими формулами, свойственными математике, которые используются при аналитических преобразованиях и обеспечивают улучшение качества алгоритмов [40–42].

Построение и исследование алгебры алгоритмов впервые осуществил В.М. Глушков в рамках проектирования логических структур ЭВМ. В результате была построена теория систем алгоритмических алгебр (САА), которая затем была положена в основу

обобщенной теории структурированных схем алгоритмов и программ, называемой теперь алгоритмикой [41].

Объектами алгоритмики являются модели алгоритмов и программ, представляемые в виде схем. Метод алгоритмики базируется на компьютерной алгебре и логике и используется для проектирования алгоритмов прикладных задач. Построенные алгоритмы описываются с помощью ЯП и реализуются соответствующими системами программирования в машинное представление.

В рамках алгоритмики разработаны специальные инструментальные средства реализации алгоритмов, которые базируются на современных объектно–ориентированных средствах и методе моделирования UML. Тем самым обеспечивается полный цикл работ по практическому применению разработанной теории алгоритмики при реализации прикладных задач, начиная с постановки задачи, формирования требований и разработки алгоритмов до получения программ решения этих задач.

Алгебра алгоритмов. Под алгеброй алгоритмов АА= {А, Ω} понимается основа А и сигнатура Ω операций над элементами основы алгебры. С помощью операции сигнатуры может быть получен произвольный элемент $q \in$ АА, который называется системой образующих алгебры. Если из этой системы не может быть исключен ни один элемент без нарушения ее свойств, то такая система образующих называется базисом алгебры.

Операции алгебры удовлетворяют следующим аксиоматическим законам: ассоциативности, коммутативности, идемпотентности, закону исключения третьего, противоречия и др. Алгебра, которой удовлетворяют перечисленные операции, называется булевой.

В алгебре алгоритмов используется алгебра множеств, элементами которой являются множества и операции над множествами (объединение, пересечение, дополнение, универсум и др.).

Основным объектами алгебры алгоритимки являются схемы алгоритмов и их суперпозиции, т.е. подстановки одних схем в другие. С подстановкой связана развертка, которая соответствует нисходящему процессу проектирования алгоритмов, и свертка, т.е. переход к более высокому уровню спецификации алгоритма. Схемы алгоритмов соответствуют конструкциям структурного программирования:

Последовательное выполнение операторов А и В записывается в виде композиции А*В; альтернативное выполнение операторов А и В (fu (А, В)) означает, если u истинно, то выполняется А иначе В; цикл (u (А, В)) выполняется, пока не станет истинным условие u (u – логическая переменная). С помощью этих элементарных конструкций строиться более сложная схема П алгоритма:

$$\Pi ::= ((u_1)\ A_1\}\},$$
$$A_1 ::= \{\ (u_2)\ A_2 * D\}$$
$$A_2 ::= A_3 * C,$$
$$A_3 ::= \{\ (u)\ A, B\},\ u ::= u_2 \wedge u_1.$$

Проведя суперпозицию путем свертки приведенной схемы алгоритма П, получается формула:

$$\Pi ::= (u_1)\ (u_2)\ (\ (u_2 \wedge u_1)\ A, B\ ) * C * D).$$

Важным результатом является сопоставительный анализ аппарата алгебры алгоритмики и следующих известных алгебр.

Алгебра Дейкстры АД = {АСС, L(2), СИГН} – двухосновная алгебра, основными элементами которой являются множество АСС операторов, представленных структурными блок–схемами, множество L(2) булевых функций в сигнатуре СИГН, в которую входят операции дизъюнкции, конъюнкции и отрицания, принимающие значения из L(2). С помощью специально разработанных механизмов преобразования АД в алгебру алгоритмики установлена связь между альтернативой и циклом, т.е. ((u) A} = ((u) E, A * ((u) A}}, произвольные операторы представлены суперпозицией основных операций и констант. Операция фильтрации Φ (u) = ((u) E, N} в АД представлена суперпозицией тождественного Е и неопределенного N операторов и альтернативы алгебры алгоритмики, где N – фильтр разрешения выполнения операций вычислений. Оператор цикла **while do** также представлен суперпозицией операций композиции и цикла в алгебре алгоритмики.

Алгебра схем Янова АЯ включает в себя операции построения неструктурных логических схем программ. Схема Янова состоит из предикатных символов множества P{p_1, p_2,…), операторных символов множества A{a_1, a_2,…} и графа переходов. Оператором в данной алгебре есть пара A{p}, состоящая из символов множества A и множества предикатных символов. Граф перехода представляет собой ориентированный граф, в вершинах которого располагаются преобразователи, распознаватели и один оператор останова. Дуги графа задаются стрелками и помечаются они знаками + и –.

Преобразователь имеет один преемник, а распознаватель – два. Каждый распознаватель включает в себя условие выполнения схемы, а преобразователь представляет операторы, включающие логические переменные, принадлежащие множеству {p_1, p_2,…).

Каждая созданная схема АЯ отличается большой сложностью, требует серьезного преобразования при переходе к представлению программы в виде соответствующей последовательности действий, условий перехода и безусловного перехода. В работе [86] разработана теория интерпретации схем Янова и доказательство эквивалентности двух операторных схем исходя из особенностей алгебры алгоритмики.

Для представления схемы Янова аппаратом алгебры алгоритмики сигнатура операций АЯ вводятся композиции A* B и операции условного перехода, который в зависимости от условия u выполняет переход к следующим операторам или к оператору помеченному меткой (типа goto). Условный переход трактуется как бинарная операция П (u, F), которая зависит от условия u и разметок схемы F. Кроме того, производится замена альтернативы и цикла типа while do. В результате выполнения бинарных операций получается новая схема F', в которой установлена П (u) вместо метки и булевы операции конъюнкции и отрицания. Эквивалентность выполненных операций преобразования обеспечивает правильность неструктурного представления.

Система алгебр Глушкова АГ = {ОП, УС, СИГН}, где ОП и УС – множество операторов суперпозиции, входящих в сигнатуру СИГН, и логических условий, определенных на информационном множестве ИМ, СИГН = {СИГНад ∪ Прогн.}, где

СИГНад – сигнатура операций Дейкстры, а Прогн. – операция прогнозирования. Сигнатура САА включает в себя операции алгебры АД, операции обобщенной

трехзначной булевой операции и операцию прогнозирования (левое умножение условия на оператор u = (A* u'), порождающая предикат u = УС такой, что u(m) = u'(m'), m' = A (m), A ∈ОП. ИМ – множество обрабатываемых данных и определения операций из множеств ОП и УС. Сущность операция прогнозирования состоит в проверке условия u в состоянии m оператора A и определения условия u', вычисленного в состоянии m' после выполнения оператора A.Данная алгебра ориентирована на аналитическую форму представления алгоритмов и оптимизацию алгоритмов по выбранным критериям.

Алгебра булевых функций и связанные с ней теоремы о функциональной полноте и проблемы минимизации булевых функций также сведены до алгебры алгоритмики. Этот специальный процесс отличается громоздкостью, и рассматриваться не будет, можно только отослать к главному источнику [86].

Алгебра алгоритмики и прикладные подалгебры. Алгебра алгоритимки пополнена двухуровневой алгебраической системой и механизмами абстрактного описания данных (классами алгоритмов). Под многоосновной алгоритмической системой (МАС) понимается система $S = \{\{D_i \mid i \in I\}; СИГН_0, СИГН_n\}$, где D_i –основы или сорта, $СИГН_0$, $СИГН_n$ – совокупности операций и предикатов, определенных на D_i .Если они пусты, то определяются многоосновные модели – алгебры. Если сорта интерпретируются как множество обрабатываемых данных, то МАС представляет собой концепцию АТД, в виде подалгебры, широко используемую в объектно–ориентированном программировании. Тем самым устанавливается связь с современными тенденциями развития современного программирования.

Практическим результатом исследований алгебры алгоритмики является построение оригинальных инструментальных систем проектирования алгоритмов и программ на основе современных средств поддержки ООП.

Читателям данной темы предоставляется возможность познакомиться более подробно с приведенными источниками.

5.2.4. Формальные методы

Понятие формальные методы более всего связано с математическими техниками спецификаций, верификации та доказательства правильности создаваемых программ. Эти методы содержат математическую символику, формальную нотацию, аппарат вывода и потому они трудно используются рядовыми программистами. Кроме того, постоянно развиваемые эти средства не вкладываются в стандартизованные процессы ЖЦ, в котором регламентированы все основные и дополнительные процессы (управление качеством, проектом, ресурсами и др.).

За многие годы своего существования (более 20–лет) такие известные формальные методы, как VDM, Z, RAISE [43–46] используются редко, эпизодически в реальных проектах, более всего в университетских и академических организациях и до промышленного производства фактически не дошли.
Это связано с тем, что формальные техники трудно использовать практически особенно в системах реального времени, где особенно важно применение формальных методов для создания более надежных программ, стоимость разработки которых возрастает, так как тратиться много времени на спецификацию и верификацию.

Наука формального проектирования ПС получила значительный прогресс при создании обобщенного UML, который доведен до стандарта, отодвинув на второй план такие средства спецификации, как RAISE, VDM и др.

Далее рассматриваются некоторые техники спецификации моделей программ и методы формального доказательства.

Графова модель VDM создается с помощью композиционной теоремы [5], которая определяется в виде ациклического графа, узлы которого – модули, а дуги – интерфейс между модулями.

Каждому модулю модели соответствует сервис как provider, так и потребителя (consumer) услуг. Дуга графу от модуля M к N определяет интерфейс, который может предоставлять модуль N для модуля M.

Базисом формального метода представления модели есть спецификация интерфейсов модулей для двух выше указанных пользователей сервисов. Интерфейс удовлетворяет следующим свойствам:
– разделенность (separable) модулей с точки зрения проектирования и спецификации, интерфейс которых не требует описания среды модуля;
– композиционность (composition), когда модули соединяются через композиционную теорию в систему с гарантированными связями между ними.

Данная теория базируется на модели интерфейса для взаимодействия модулей системы между собою через сетевые протоколы (TCP, UDP и др.). Каждый протокол передает разным модулям этой модели параметры для нахождения нужного сервиса.

Модель интерфейса задает связь модуля со средой в виде дискретного события (event), которое возникает только тогда, когда модуль и среда действуют вместе, и создают событие, рассматриваемое со стороны интерфейса. Иными словами, интерфейс специфицируется в виде множества последовательностей событий для наблюдения за поведением модулей модели.

Предположим, что S определяет спецификацию модуля M, которая включает в себя все возможные случаи его поведения и задает разные ситуации, которыми могут быть:
– событие интерфейса или состояние наблюдателя системы;
– анализ является конечным или представлен неопределенной последовательностью;
– формализм определения этой последовательности;
– условия выполнения события.

Эти предположения разрешают обеспечить разноуровневую систему взаимодействия модулей модели через механизм протоколов и систему наблюдения за событиями.

Формальный метод и спецификация RAISE. RAISE– метод и RSL–спецификация (RAISE Spesification Language) [45, 46] были разработаны в 1985–1998гг. как результат предварительного исследования формальных методов. Метод содержит нотации, техники и инструменты для использования формальных методов при конструировании ПС и доказательстве их правильности. Метод имеет программную поддержку в виде набора инструментов и методик, которые постоянно развиваются и используются для конструирования и доказательства правильности программ. описанных в RSL и ЯП (С++ и Паскаль).

Язык RSL содержит абстрактные параметрические типы данных (алгебраические спецификации) и конкретные типы данных (модельно–ориентированные), подтипы, операции для задания последовательных и параллельных программ, предоставляет аппликативный и императивный стиль спецификации абстрактных программ, а также формальное конструирование программ в других ЯП и доказательства их правильности. Синтаксис этого языка близок к синтаксису языков C++ и Паскаль.

В RSL–языке имеются предопределенные абстрактные типы данных и конструкторы сложных типов данных, такие как произведение (product), множества (sets), списки (list), отображения (map), записи (record) и т.п. Далее рассмотрим, для примера, некоторые конструкторы сложных типов данных.

1. *Произведение* типов – это упорядоченная конечная последовательность типов T_1, T_2 , …, T_n произведения (**product**) $T_1 \times T_2 \times, \ldots, \times T_n$. Представитель типа имеет вид (v_1, v_2 , …, v_n), где каждое v_i есть значением типа T_i. Компонент произведения можно получить get и переслать set, т.е.

get component (i, d) = getvalue (i, d),

set component (d, i, val) = d $\Rightarrow\nabla$ (I $\rightarrow$ val).

Количество компонентов произведения d находится таким образом:

size (d) = id ∇ (null (couter inc(counter))).

Конструктор произведения d_1 и d_2 строит произведение $d_1 \times d_2$ вида

product (d_1 , d_2) = id ∇ (size (d_1) $\Rightarrow$ couter 1) ∇ (null (couter 2) inc couter 2))).

Для каждого конкретного типа product ($T_1 \times T_2 \times, \ldots, \times T_n$) можно построить конструктор значения этого типа из отдельных компонентов произведения таким образом:

make product (value $_1$, …, value $_n$) = (value $_i \Rightarrow$ 1) $\nabla \ldots \nabla$ (value $_n \Rightarrow$ n),

где каждое значение value $_i$ имеет тип T_i , а результирующее значение – тип произведения $T_1 \times T_2 \times, \ldots, \times T_n$.

2. *Списки* типов – это последовательность значений одного типа списка **list** T, могут быть конечным списком типов T^k и неконечными списком типов $T^{\,n}$. в качестве структур данных типа списку может быть бинарное дерево, в котором есть голова (head) и сын (tail), следует за ним в списке и хвост. Операциями для списка является операция hd взятия первого элемента списка, т.е. головы и операция tl – хвоста остальные элементы.

Функция Caddr (I) = L $\Rightarrow$ tail $\Rightarrow$tail $\Rightarrow$ Head выбирает из списка I –элемент. Индекс элемента помогает выбрать нужный элемент списка Index(I, idx) = L (idx) = while ($\neg$ is null(idx))do((L $\Rightarrow$ tail $\Rightarrow$L) ∇dec(idx)) L $\Rightarrow$ Head. Для определения количества элементов в списке выполняется функция

len (L) = (ld ∇ null (result))
while (L $\Rightarrow$) do ((L $\Rightarrow$ tail $\Rightarrow$ tail $\Rightarrow$L) ∇ inc (result))
result $\Rightarrow$

Элемент списка находится так:

elem (L) = (ld ∇ empty (result))
while (L $\Rightarrow$) do ((L $\Rightarrow$ tail $\Rightarrow$L) ∇
(result $\uparrow$ (L$\Rightarrow$ head $\Rightarrow$) $\Rightarrow$ elem) $\Rightarrow$ result)
result $\Rightarrow$.

Аналогично можно представить функции конкатенации, преобразование типов данных, добавления элемента в голову и хвост списка и др.

3. *Отображение* – это структура (**map**), которая ставит в соответствие значениям одного типа значение другого типа. С другой стороны, отображение является бинарным отношением декартового произведения двух множеств, как совокупности двухкомпонентных пар, в которой первый компонент arg содержит элементы аргументов отображения, а другой компонент res соответствующие элементы значений этого отображения.

В языке представлены разные допустимые операции над отображениями: наложение, объединение, композиция, срез и др. Среди этих видов отношений рассмотрим только композицию отображений.(m_1, m_2).

(ld ∇ (compose (m_1, m_2) ⇒ m)) apply (m, elem)
apply to composition (m_1, m_2, elem) =
(ld ∇ (image (elem, m_1) ⇒ s) restrict (m_2, s) ⇒ map
(ld ∇ (map getname elem ⇒ name)) getvalue (name, map).

При этом используются функции:

Apply (m, elem) = image (elem, m) elem ⇒,
Apply (m, elem) = getvalue (elem, m) elem ⇒.

4. *Запись* – это совокупность именованных полей. Этот тип соответствует типу **record** в языке Паскаль и struct в языке С++. В языке RAISE определено два конструктора типа, record, shurt record, которые описываются в виде – type record id =

type mk_id (short _record id) ::=
destr_id_1 : type_$expr_1$ ↔ recon_id
. . .
destr_id_n : type_$expr_n$ ↔ recon_id.

Идентификатор mk_id является конструктором типа record, для которого даны деструкторы destr_id_n , как функции получения значения компонентов записи.

5. *Объединение* – это конструктор **Union,** обеспечивающий объединение типов
type id = id_1 , id_2 ,…, id_n
при котором тип id получает одно из значений в списке элементов.

Конструктор этого типа имеет вид:
type id = id_from_ id_1 (id_to_ id_1: id_1) │… │ id_from_ id_n (id_to_ idn: id_n)/
Операции над самим типом не определены в языке RAISE.

Рассмотренные формальные структуры данных языка RAISE позволяет математически описывать и конструировать новые структуры данных в проектируемых программах. Их проще проверять на правильность методами верификации.

Контрольные вопросы и задания

1. Дайте характеристику структурного метода.
2. Приведите основные особенности и возможности объектно–ориентированного программирования.
3. Какие структуры имеются в языке UML для наглядного проектирования?
4. Приведите пути развития компонентного программирования.

5. Приведите базовые определения в компонентном программировании.
6. Определите основные понятия и этапы жизненного цикла компонентного программирования.
7. Определите основные элементы аспектно–ориентированного программирования.
8. Дайте характеристику инженерии ПрО.
9. Объекты генерирующего программирования и краткая их характеристика.
10. Представьте главные теоретические методы программирования.
11. Дайте определение формальных методов программирования.

Литература к теме 5.

1. *Demark D.A., McGowan R.L.* SADT: Structured Analysis and Design Technique. New York: McCray Hill, 1988.– 378 c.

2. Skidmore.S, Mills.G, Farmer R. SSADM: Models and Mehtods. –Prentice–Hall, Englewood Cliffs, 1994.

3. *Марка Д.А., МакГруэн К.* Методология структурного анализа и проектирования.–М.: МетаТехнология, 1997.– 346с.

4. *Буч Г.* Объектно–ориентированный анализ и проектирование с примерами приложений на С++, 2–е изд. – М.: "Изд–во Бином", 1998. – 560 с.

5 *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно–ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.

6. *The Unified* Modeling Language (UML) Specification. – V. 1.3. UML Specification, revised by the OMG. – July 1999. – 620 p.

7. *Рамбо Дж., Джекобсон А , Буч Г. UML:* специальный справочник.– СПб.: Питер.– 2002.– 656с.

8. *Crnkovik I, Larsson S., Stafford J.* Component–Based Software Engineering: building systems from Components at 9th Conference and Workshops on Engineering of Computer – Based Systems.– Software Engineering Notes.–2002.– vol.27.–N 3.–c.47–50.

9. *Gamma E., Helm R., Johnson R., and Vlissides J.* Design Patterns, Elements of Reusable Object–oriented Software,– N.– Y.: Addison–Wesley, 1995. – 345p.

10. *Component* Object Model. – www.microsoft.com/tech/COM. asp

11. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования//Кибернетика и системный анализ, 2003.– №1.– с.39–55.

12. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии //Проблемы программирования. – 2000. – №1–2.– С.351–360.

13. *Batory D., O'Malley S.* The Design and Implementation of Hierarchical Software Systems with Reusable Components/ ACM Transactions on Software Engineering and Methodology. – N 4,– 1, October 1992. – P.355–398.

14. *Weide B., Ogden W., Sweden S.* Reusable Software Components/ Advances in Computers, vol. 33. – Academic Press, 1991. – pp.1–65.

15. *Jacobson I., Griss M., Johnson P.* Software Reuse: Architecture, Process and organization for Business Success – Addison Wesley, Reading , MA, May 1997.–501p.

16. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.

17. *Рихтер Дж.* Программирование на платформе Microsoft .NET FRAMEWORK. – М.: Издательско– торговый дом "Русская Редакция", 2002. – 512 с.

18. *Kiselev. I.* Aspect–Oriented Programming with AspectJ. Indianapolis, IN, USA: SAMS Publishing, 2002.–164p.

19. *Homepage* of the Aspect–Oriented Programming, Xerox Palo Alto Research Center (Xerox Parc) Palo Alto, CA, www.parc.xerox.com/aop

20.*Павлов В.* Аспектно–ориентированное программирование//Технология клиент–сервер, № 3–4.– С.3–45.

21. *Golub G., C.van Loan*. Matrix Computation. Third editor –The John HopkinsUniversity Press, Baltimore and London, 1996.
22. *К.Чернецки, У.Айзенекер.* Порождающее программирование. Методы, инструменты, применение.– Издательский дом «Питер».– Москва– СП… Харьков, Минск.– 2005.– 730с.
23. *Сигел Дж.* CORBA 3. – Москва: Малип, – 2002. – 412 с.
24. *Роджерсон Д.* Основы COM. – Руск.пер.– Microsoft Press.– 363с.
25. *Ndumu D.T., Nwana.* Research and Development challenges for Agent–based system..– IEEProccedings Software Engineering– 1997.– 144.–№ 01.–p.26–37.
26. Shoham. Y. Agent–oriented programming.–Artif.Intell. 1993, 60, №1.–p.51–92.
27. *Трахтенгерц Э.А.* Взаимодействие агентов в многоагентных средах //Автоматика и телемеханика.– М.: Наука .–1998.–№8.– с.3–52.
28. *Nwana H.S., Lee J., Jennings N.R*. Coordination Software agent systems /British Telecommunication Technology Journal, 1996.–14(4).
29. *Дрейган Р*. Будущее программных агентов.– PC Magazine March 25,– 1997.–190с.
30. *Яковсон А*. Мечты о будующем программирования. Открытые системы.– М.: 2005.– №12.– с.59–63.
31. *Летичевский А.А, Маринченко В.Г.* Объекты в системе алгебраического программирования //Кибернетика и системный анализ. –1997.– N2. – С.160–180.
32. *Летичевский А.А., Капитонова Ю.В., Волков В.А., Вышемирский В.В., Летичевский А.А. (*мол.). Инсерционное программирование// Там же.– 2003.– №1.– 12–32.
33. *Летичевский А.А., Капитонова Ю.В.* Доказательство теорем в математической информационной среде //Там же. – 1998. – №.4. – С. 3 – 12.
34. *Letichevsky A.A., Gilbert D.R.* A model for interaction of agents and environments //Recent trends in algebra's development technique Language.–2000. – P.311 – 329.
35. *Редько В.Н.* Экспликативное программирование: ретроспективы и перспективы//Проблемы программирования.– 1998. – №2.–С. 22 – 41.
36. *Никитченко Н.С.* Композиционно–номинативный подход к уточнению понятия программы // Там же.– 1999.– №1.– С. 16–31.
37. *Редько В.Н.* Композиционная структура программологии //Кибернетика и системный анализ. – 1998. – № 4. – С. 47–66.
38. *Редько В.Н.* Основания программологии // Там же. – 2000.– № 1.– С. 35–57.
39. *Редько В.Н., Брона Ю.Й., Буй Д.Б.* Реляційні бази даних: табличні алгебри та SQL–подібні мови //Видав. дім "Академперіодика", Київ, 2001.–195с.
40. *Цейтлин Г.Е.* Введение в алгоритмику.– Изд.–во Фара, 1999.–310с.
41. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра. Языки. Программирование. – Наукова думка.– 1974.–317с. (перераб. и переизд. 1979г., 1989г.).
42. *Ющенко Е.Л , Сужко С,В,, Цейтлин Г.Е., Шевченко А.И.* Алгоритмические алгебры.– Учебник (укр.).– ИЗММ.– 1997.–480с.
43. *Abrial I.R., Meyer B.* Spesification Language Z. –Boston: Massachusetts Computer Associates Inc.. 1979.–378 p.
44. *Biorner D.Jones C.B*. The Vienna Development Methods (VDM): The Meta – Language.– Vol. 61 of Lecture Notes in Computer Science .– Springer Verlag, Heiderberg, Germany, 1978.–215p.
45. *The RAISE* Language Group. The RAISE Spesification Language. BCS Practitioner Series .– Prentice Hall, 1982.–397 p.
46. *The RAISE* Methods Group. The RAISE Development Methods. BCS Practitioner Series .– Prentice Hall, 1985.–493p.

Тема 6.
ИНЖЕНЕРИЯ ПРИЛОЖЕНИЙ И ИНЖЕНЕРИЯ ПРЕДМЕТНОЙ ОБЛАСТИ

Введение

В последние годы быстро развиваются такие направления программной инженерии, как построение ПС из ПИК; инженерные методы проектирования, которые характеризуются проверкой достижения показателей качества компонентов на этапах ЖЦ и оценкой затраченных ресурсов и стоимости. Главное место среди этих методов занимает компонентный подход к построению ПС, так как принцип использования готовых компонентов является основой этого подхода и стратегическим направлением повышения производительности разработчиков и обеспечения качества ПС.

Компонентный подход расширяет существующие методы программирования ПС этим принципом, а также проверкой правильности ПИК для надежного приспособления к новым условиям ПС, формализацией и приведением в порядок процесса разработки из них ПС соответственно требованиям заказчика. В результате получается значительное упрощение процесс разработки самих компонентов и ПС из них и соответственно уменьшение сроков и затрат на разработку ПС.

Исследования и разработки в области инженерии программирования, основанного на использовании готовых ранее разработанных ПИК привели к тому, что сформировалось и используются два инженерных направления применения разных видов готовых ПИК при создании новых ПС [4]:

1) инженерия построения новых одиночных ПС уникального типа из ПИК. Это направление получило название *прикладная инженерия* (application engineering), которой соответствует процесс производства конкретных ПС как совокупности компонентов, подсистем и ПИК одного класса, созданных раньше как самостоятельные программные продукты или как элементы процесса инженерии некоторой предметной области;

2) инженерия построения готовых частей систем в конкретной ПрО. Это направление получило название *инженерия проблемной области* (domain engineering), которой соответствует процесс классификации и фиксации ПИК многоразового пользования в рамках конкретной ПрО в виде готовых частей системы, самой системы или семейства систем. Данный процесс поддерживается системными инструментами обеспечения сбора, поиска, адаптации ПИК к новым условиям создаваемой системы семейства. Этим обеспечивается повторное использование не только элементов ПИК, а и инструментов поиска.

Обе разновидности инженерии базируются на объектно-ориентированных (ОО) методах анализа ПрО, методах проектирования классов объектов и образцов многоразового использования, которые создаются в разных системах и фиксируются как готовые решения для применения в других областях. Инженерность приложения и предметной области заключается в использовании в процессе проектирования методов поиска и оценки применимости ПИК, стоимости их приобретения и планирования работ по изготовлению из ПИК новых систем в заданные сроки. Инженерия ПИК предполагает проведение классификации и каталогизации готовых компонентов в

специальных хранилищах для организации поиска, выбора с требуемыми функциями и характеристиками для последующего использования в новых проектах.

Анализ современных систем поддержки *инженерии приложения* (OMT, RUP, OOA/D и др.) показывает, что они ориентированы на разработку одиночных ПС и имеют такие недостатки:

1) отсутствует различие между разработкой и областью разработки для повторного использования и разработкой с повторным использованием. Областью разработки для повторного использования являются некоторые совокупности компонентов и подсистем. Процесс разработки с повторным использованием основывается на инструментальных средствах поиска и выбора готовых компонентов, которые создавались в процессе разработки одиночной системы для повторного использования;
2) повторное использование не базируется на модели ПрО, а ПИК создаются с ориентацией на создание отдельных одиночных программ;
3) отсутствует моделирование изменяемости компонентов в рамках одного приложения или нескольких приложений, которая может быть обеспечена использованием диаграмм классов UML с представлением изменяемых параметров и операций наследования, агрегации или параметризации.

Данные недостатки устраняются в рамках инженерии ПрО, для домена которой создается характеристическая модель, которая учитывает изменчивость параметров и разных характеристик для группы программных подсистем. Инженерия ПрО использует методы инженерии приложения по использованию ПИК и включает процессы корректировки и моделирования характеристик ПрО и их изменяемости; использует другие модели (Use Case, модели взаимодействия, переходов и т.п.) и механизмы повторного использования ПИК (хранение, поиск, оценивание, объединение).

Инженерия приложений фактически предназначена для создания целевых одиночных ПС из соответствующего набора ПИК, а инженерия предметной области ставит задачу создания множества программных систем для ПрО (домена) и совокупностей ПрО с выделением в этой ПрО отдельных функций или группы функций для проектирования подсистем, обладающих общими свойствами и характеристиками для многоразового использования в других доменах этой совокупности ПрО.

6. 1. Инженерия ПИК

Большой объем накопленных программных продуктов мало используется при изготовлении новых систем, поэтому на тысячах предприятий продолжается дублирование разработки программ, которые массово используются. Инженерия ПИК является инструментом решения проблемы дублирования и сокращения времени и стоимости разработки новых ПС с одновременным снижением затрат и сложности

Инженерия ПИК – это систематическая и целенаправленная деятельность по подбору реализованных программных артефактов, и представленных в виде ПИК, анализу их функций для добавления в качестве готовых в проектируемую систему и их интеграция с другими компонентами. Согласно стандарту ISO/IEC 12207 эта деятельность классифицируется как организационная и планируемая инженерная деятельность, которая заключается в выявлении общих и специфических черт компонентов для принятия решений об их использовании в разработке новых ПС [1, 6-10].

ПИК нуждается во вложении капитала для их разработки, сохранении и приобретении. Системное применение ПИК оказывает содействие получению прибыли за счет экономии затрат на разработку, а менеджмент помогает принять решение по эффективному вкладу средств в ПИК, оценки рисков его использования и объемов отдачи от применения. Оценка риска выполняется по каждому атрибуту ПИК: сначала вычисляются ранги риска атрибутов, исходя из количества вопросов к ним, потом вычисляется уровень риска каждого атрибуту и всех атрибутов в целом.

Повторное использование базируется на любых порциях формализованных знаний, полученных от реализации завершенных разработок и зафиксированных в разнообразных формах: от конкретных параметризованных программных модулей к каркасам, архитектурам и средам.

Методология разработки системы из готовых компонентов основывается на изучении спектра задач во вновь разрабатываемой системе, выявлении в ней задач с общими подходами к их решению и нахождения подходящих ПИК с определением их возможностей и принятия решений о целесообразности применения в будущей системе.

Разработчику системы необходимо иметь большой опыт в представлении и не одной, а нескольких однотипных задач, чтобы суметь обнаружить их общность, определить приемы настройки на характерные для каждой задачи особенности и принять ответственное решение.

Базисом ПИК являются хранилища с информационным каталогом этих ПИК, соответствующих типичным функциям, которые настраиваются на новые условия с меньшими усилиями, чем новая разработка. Информационной основой подбора ПИК в специальных хранилищах является классификация и каталогизация.

Сущность классификации состоит в предоставлении информации о функциях, среде и требуемых ресурсов для ПИК, которые размещаются в информационных репозитариях программной продукции.

Каталогизация - это средство физического размещения готовых компонентов в каталоге репозитария после завершения их реализации в структуре некоторой ПС для организация поиска формализованных ПИК, оценки их пригодности для последующего использования.

Действия по классификации и каталогизации компонентов и ПИК являются систематическими и целенаправленными, необходимыми многим специалистам, которые заинтересованы в готовой программной продукции для решения своих задач. При этом с помощью каталога можно узнать о ПИК как о готовых деталях и правилах их соединения в программную конструкцию. Именно эта сторона и характеризует повторное использование, как систематическую деятельность по созданию и использованию каталога ПИК.

Систематическое повторное использование – это капиталоемкий подход, который предусматривает наличие двух процессов в ЖЦ разработки ПС.

<u>Первый процесс –</u> создание ПИК путем:
– изучения спектра решаемых задач предметной области, выявление среди них общих подходов к реализации;

– реализации задач и функций в виде компонентов, которые будут повторно используемыми компонентами;
– построение каталога, предназначенного для обеспечения поиска необходимых компонентов.

Для успешной реализации данного процесса необходимо иметь определенный опыт в решении нескольких подобных между собой задач, позволившими выявить как их общие черты, так и различия, чтобы найти общее решение для их реализации, а также приемы настройки на характерные для каждой задачи особенности.

Второй процесс – конструирование новых систем из готовых компонентов путем:
– понимания сущности новой разработки, цели ее создания и предъявляемых к ней требований;
– поиска в каталоге готовых компонентов, которые кажутся подходящими для их использования;
– сопоставление цели новой разработки с возможностями найденных ПИК и принятия решений о целесообразности их использования;
– применение отобранных ПИК и интеграция их в новую разработку с обеспечением необходимых соединений друг с другом.

Первый процесс требует вложения капитала, второй – получения прибыли за счет экономии трудозатрат. Инвестиции в повторное использование требуют оценки эффективности вложений капитала, прогнозирования сроков и объемов возврата вложений, оценки рисков и др. Бизнес повторного использования, как любой бизнес, требует специальных условий по менеджменту всей инженерной деятельности. Критерии успеха такого бизнеса определяются выполнением следующих условий:

– повторное использование должно обеспечиваться меньшими трудозатратами, чем разработка ПС как разовых продуктов;
– поиск пригодных для использования компонентов требует меньших трудозатрат, чем новая разработка их функций для нужд проектируемой системы;
– настройка компонентов на новые условия среды применения должна обеспечиваться меньшими трудозатратами, чем новая разработка.

Основная парадигма ПИК – "писать – один раз, выполнять – много раз, где угодно". Архитектура, в которую встраивается готовый ПИК, поддерживает стандартные механизмы для работы с компонентами как со строительными блоками. Чтобы обеспечить высокий уровень использования ПИК они должны обладать такими основными свойствами: функциональность, удобство использования и качество реализации.

Разновидности ПИК. В качестве ПИК могут использоваться формализованные артефакты деятельности разработчиков ПС, которые отражают некоторую функциональность для применения в новых разработках. Под *артефактом* понимается реальная порция информации, которая может создаваться, изменяться и использоваться при выполнении деятельности, связанной с разработкой ПС различного назначения. Артефактами могут быть:

– промежуточные продукты процесса разработки ПС (требования, постановки задач, архитектура и др.);
– описания, полученные в процессе разработки ПС (спецификации, модели, каркас (framework и т.п.)

– готовые компоненты ПС или отдельные ее части;
– продукции, фреймы, диаграммы, паттерны и т.п.

К компонентам ПИК выдвигаются такие требования, как независимость от конкретной платформы, наличие стандартного интерфейса со средой, возможность взаимодействия с системами без внесения в них изменений.

Разработке ПС с помощью ПИК соответствует модель ЖЦ со следующими этапами:
– анализ объектов и отношений реализуемой ПрО для выявления ПИК, обладающих общими свойствами, присущими группам объектов этой области;
– адаптация имеющихся в базе ПИК и разработка новых компонентов, не представленных в этой базе и доведение их до уровня ПИК;
– разработка интерфейсов компонентов и их размещение в базе ПИК;
– интеграция ПИК для получения конфигурации создаваемой системы.

Повторные компоненты могут быть готовыми прикладными и общесистемными компонентами.

Прикладные компоненты выполняют отдельные задачи и функции прикладной области деятельности – ПрО (бизнес–домены, коммерция, экономика и т.п.), которые могут использоваться как готовые при разработке новых прикладных систем с их использованием.

К *общесистемным компонентам* относятся системные компоненты общего назначения и универсальные общесистемные сервисные средства, которые обеспечивают системное обслуживание и предоставление универсального сервиса для всех создаваемых программных систем разного назначения. К компонентами общего назначения относятся: трансляторы, редакторы тестов, системы генерации, интеграции, загрузчики и др. Они используются всеми прикладными системами в процессе их проектирования и выполнения. Универсальные системные компоненты обеспечивают функционирование любых (в том числе и прикладных) компонентов, обмен данными и передачей сообщений между всеми видами компонентов. К ним относятся ОС, СУБД, сетевое обеспечение, электронная почта и др.

Связь между прикладными и общесистемными средствами осуществляется через стандартные интерфейсы, обеспечивающие взаимодействие разных типов компонентов через механизмы передачи данных и сообщений.

На современном рынке программных продуктов циркулируют следующие виды готовых компонентов:

– структуры данных,
– процедуры и функции на ЯП высокого уровня,
– алгоритмы, программы,
– классы объектов и абстрактные классы,
– API – модули,
– Веб–ресурсы,
– сервисы и среда развертывания (например, компоненты типа Java Beans, CORBA, COM, .NET)
– готовые абстрактные решения – паттерны и фреймы.

Все многообразие видов и типов готовых компонентов требует от разработчиков их поиска и изучения для использования в новых ПС. Процесс разработки новых ПС с помощью разных видов ПИК является капиталоемким, в нем в качестве капитала выступают готовые ПИК, на применение которых затрачивается меньше средств, чем на повторную их разработку.

6.2. Спецификация ПИК

В качестве ПИК могут быть объекты, созданные в рамках объектно–ориентированного программирования с наследованием их реализации, а также компоненты в компонентном программировании, для них наследуется не реализация, а интерфейсы. При этом компонент обладает такими свойствами:

– связывания компонентов на последних этапах разработки ПС,
– инкапсуляции (компонент, как “черный ящик ” без вмешательств в код),
– наследования интерфейсов,
– повторное использование кода.

Для компонентов повторного использования сложилось несколько разных определений.

Определение 1. Компонент ПИК – это некоторая функция с определенными атрибутами, обеспечивающая функциональность, взаимодействие со средой и поведение.

Определения 2. Готовый к повторному использованию компонент представляет собой совокупность методов определенной сигнатуры и типов данных, которые передаются и возвращаются после выполнения метода.

Определение 3. ПИК – это самостоятельный программный элемент, который удовлетворяет определенным функциональным требованиям, требованиям архитектуры, структуры и организации взаимодействия в заданной среде, имеет спецификацию, помогающую пользователю объединять его с другими компонентами в интегрированную ПС.

Нас более всего интересует последнее определение компонента, модель спецификации которого имеет вид:

$$ПИК = (T, I, F, R, S),$$

где T – тип компоненты, I – множество интерфейсов компонента; F – функциональность компонента; R – реализация (скрытая часть) – программный код; S – сервис для взаимодействия со средой или шаблон развертывания.

Каждый из элементов спецификации компонента представляет собою видимую или скрытую от пользователя часть его абстракции.

В зависимости от сложности ПИК их можно разделить на следующие группы:

– простые компоненты (функция, модуль, класс и пр.);
– объекты–компоненты, имеющие интерфейс, функциональность и реализацию на любом ЯП, а также возможность дополнять спецификации шаблоном развертывания и интеграции;

– готовые к использованию ПИК (например, beans компоненты в Java, AWT компоненты, классы и др.);
– сложные ПИК типа каркасов, паттернов с элементами группирования из нескольких простых ПИК и взаимодействия между ними при решении одной общей задачи ПС.
Большое количество готовых компонентов требует от разработчиков и пользователей задания их категории, т.е. метаинформации о том, какие классы совместимы с заведомо определенными семантическими ограничениями описания ПИК, и состоят из:

– интерфейсов, которые реализуют компоненты,
– механизмов повторного использования,
– среды развертывания компонента
– сервиса, поддерживаемого компонентом,
– ролей, которые выполняют компоненты в ПС,
– формализованные языки описания ПИК.

Современная технология применения ПИК базируются на таких особенностях:

– отображение, как способность ПС анализировать самого себя и описывать свои возможности динамично во время выполнения, а не во время компиляции, что обеспечивает управление большинством свойств, событий и методов компонента;
– анализа компонента для определения его возможностей.
– отсутствие средств поддержки реинженерии программного компонента и необходимости задания параметров его разработки;
– способности компонента к рефакторингу т.е. к трансформации компонента с сохранением функциональности, но с возможным изменением структуры и исходного кода для повторного использования.
– сохранение параметров конфигурации (шаблонов отладки) в постоянной памяти для использования в нужное время;
– регистрация сообщений о событиях, полученных от других объектов либо через ссылки (например, beans компоненты и инструментарий архива в технологии JAVA), а также группирование компонентов в JAR файле для дальнейшего повторного использования;
– использование компонента в разных языковых средах;
– адаптация ПИК к разным контекстам их использования и выделение свойств, которые мешают повторному использованию и модификации для применения в конкретных целях.

Компоненты в отличие от объектов могут изменяться и пополняться новыми функциями и интерфейсами. Они конструируются в виде некоторой программной абстракции, состоящей из трех частей: информационной, внешней и внутренней.

Информационная часть содержит описание: назначение, даты изготовления, условия применения (ОС, платформа и т.п.), возможностей ПИК, среды окружения и др.

Внешняя часть – это интерфейс, который определяет взаимодействие компонента с внешней средой и с платформой, на которой он будет выполняться и включает следующие общие характеристики:
– интероперабельность – способность взаимодействовать с компонентами других сред;
– переносимость – способность компонента выполняться на разных платформах компьютеров;
– интеграционость – объединение компонентов на основе интерфейсов в более сложные ПС;

– нефункциональность – требование на обеспечение безопасности, надежности и защиты компонентов и данных.

Внутренняя часть компонента – это программный фрагмент кода, системная или абстрактная структура, представленные в виде проекта компонента, спецификации и выходного кода.
Данная часть компонента состоит из полей (рис.6.1):
– интерфейса (interfaces),
– реализации (implementation),
– схемы развертки (deployment).

Структурные части компонента		
Интерфейс	**Реализация**	**Схема развертывания**
♦ Один или несколько; ♦ Уникальность именования в пределах системы; ♦ Клиентский или серверный (входной или выходной); ♦ Определенная сигнатура; ♦ Описание методов взаимодействия	♦ Одна или несколько; ♦ Ориентация на конкретную платформу и операционное окружение ♦ Выбор конкретной реализации; ♦ Поддержка интерфейсов компонента	♦ Типовость процедуры развертывания; ♦ Управляемость; ♦ Настраиваемость на операционную среду; ♦ Модифицируемость

Рис.6.1. Структурные части компонента

Интерфейс компонента содержит обращения к другим компонентам через описание параметров средствами языков IDL или APL. В нем указываются типы данных и операции передачи параметров для взаимодействия компонентов друг с другом. Каждый компонент может реализовать совокупность интерфейсов. Интерфейс компонента – это видимая часть спецификации компонента, которая является неизменной и обязательной в описании компонента. Например, система Inspector Components предназначена для изменения необходимых параметров интерфейса компонента без вмешательства в его код.

Параметры интерфейса могут определяться типом ПИК и включать в себя инварианту спецификации с указанием: типа и названия компонента, входных и выходных параметров, методов компонента и др. В языке JAVA, например, можно определять такие типы компонентов: проекты, формы (AWT компоненты), beans компоненты, COBRA компоненты, RMI компоненты, стандартные классы-оболочки, БД, JSP компоненты, сервелети, XML документы, DTD документы и т.п.

Реализация – это код, который будет использоваться при обращении к операциям, определенным в интерфейсах компонента. Компонент может иметь несколько реализаций, например, в зависимости от операционной среды или от модели данных и соответствующей системы управления базами данных, которая необходима для функционирования компонента.

Развертка - это физический файл или архив, готовый к выполнению, который передается пользователю и содержит все необходимые инструкции для создания, настройки и функционирования компонента.

Компонент описывается в ЯП, не зависит от операционной среды (например, от среды виртуальной машины JAVA) и от реальной платформы (например, от платформ в системе CORBA), где он будет функционировать.

Расширением понятия компонента есть паттерн – абстракция, которая содержит описание взаимодействия совокупности объектов в общей кооперативной деятельности, для которой определены роли участников и их ответственность. Паттерн является повторяемой частью программного компонента, как схемы или взаимосвязи контекста описания решения проблемы.

6.3. Репозитарий компонентов

ПИК и производные от них объекты размещаются в разных хранилищах (библиотеках, репозитариях ПС, в репозитариях Интернет) и используются многократно при построении ПС [30–31]. Например, каждый репозитарий (например, библиотека GreenStone) ориентирован на одну или несколько предметных областей.

В общем случае репозитарий представляет собой систему средств для хранения, пополнения наработанных ПИК, включает в себя инфраструктуру разработки ПС из компонентов, организацию доступа к содержащимся в нем ПИК для последующего их использования в новых проектах ПС.

С функциональной точки зрения репозитарий работает по принципу информационно-поисковой системы, объектами хранения которой являются разные типы документов, тексты и др. Им ставится в соответствие информация, содержащая формализованные спецификации экземпляров коллекции документов, которые отображают понятия ПрО, ключевые слова, правила доступа и др.

В отличие от них компоненты ПИК записываются в репозитарий с поисковым образом, создаваемый путем аннотирования ПИК на основе описания информационной части ПИК. В соответствии терминологии UML, лица, которые обеспечивают функционирование репозитария, называются актерами, а сами работы с ПИК – сценариями.

Репозитарий компонентов ПС упрощает и сокращает сроки разработки ПС за счет:
– отображения в них базовых функций и понятий ПС;
– скрытия представления данных, операций обновления и получения доступа к этим данным;
– обработки исключительных ситуаций, возникающих в процессе выполнения и др.

При представлении поискового образа ПИК используются также информационные модели, которые обеспечивают систему хранения, поиска и сопоставления ПИК, принадлежащих репозитарию, который виртуально разделен на разделы, соответствующие представленном в нем ПрО, перечень которых составляет классификатор первого уровня. Классификаторами следующих уровней могут служить отдельные понятия, определенные для ПрО.

Каждому понятию соответствует информационная модель, в состав которой входят паспортные данные (имя и адрес разработчика, способ приобретения, цена, и т.п.), сведения о среде реализации (ОС, ЯП, СУБД и т.п.), описание аппаратных ресурсах, имени ПрО, к которому относится ПИК в системе классификации и категорий ПИК, а также описание нефункциональных требований к создаваемой системе (безопасность, конфиденциальность, показатели качества системы и прочее).

Для отображения ПИК в репозитарии проводится их классификация и каталогизация, аннотирование и размещение.

Классификация компонентов обеспечивает унификацию представления информации о компонентах для последующего их поиска и отбора из среды репозитария. Она проводится с учетом следующих их групп:
– компонент типа модуль, класс и т.п.;
– компоненты, которые имеют интерфейс (входные и выходные параметры, пред-, пост- условия функционирования), функциональность и реализацию на ЯП, из которых создается ПИК со спецификацией шаблона развертывания;
– готовые к употреблению ПИК;
– сложные ПИК типа каркасы и паттерны, которые обеспечивают взаимодействие простых ПИК.

Каталогизация направлена на физическое размещение кодов ПИК в репозитарии для извлечения их при необходимости встраивания новый программный проект ПС. Для выбранных компонентов осуществляется их настройка на условия среды функционирования.

Инженерия ПИК и других компонентов ($КОМ_1$, …, $КОМ_k$) в разработку новых ПС осуществляется примерно по технологии, представленной на рис.6.1. Если компоненты написаны на разных ЯП, создаются интефейсные модули (**Int**$_1$,…, **Int**$_k$**)**, в которых подготавливаются и преобразуются типы передаваемых данных.

1. Разработка компонентов (КОМ) на ЯП интеграции
2. Выбор ПИК
3. Разработка интерфейсов (Int) для КОМ и ПИК
4. Генерация интерфейсов пары КОМ на ЯП
5. Разработка среды и репозитария КОМ
6. Типизация и классификация компонентов
7. Тестирование КОМ, интерфейсов, ПС
8. Интеграция разных видов компонентов
9. Загрузка ПС в среде выполнения
10. Сопровождение компонентной ПС
11. Эволюция компонентной ПС

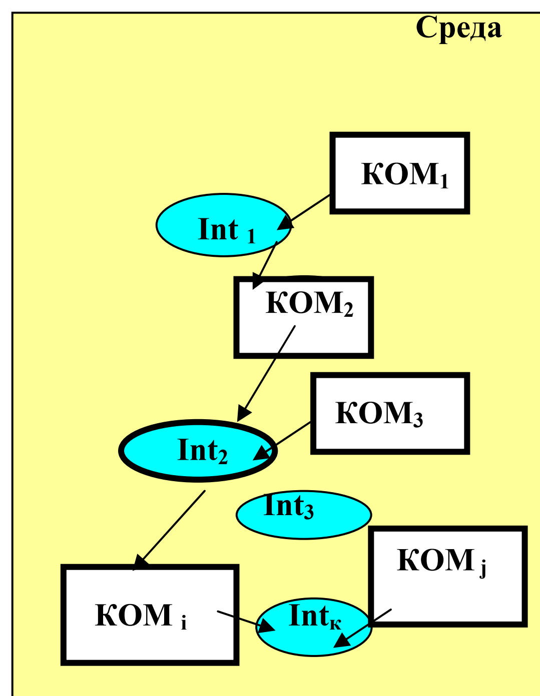

Рис 6.1 Технология инженерии компонентов в разработке ПС

6.4. Описание интерфейса объектов-компонентов в распределенной среде

Для объединения компонентов в ПС необходимым условием является наличие для них формально определенных интерфейсов в языках IDL и APL, а также механизмов динамического контроля связей между компонентами.

Спецификация интерфейса в API и IDL включает описание функциональных свойств компонентов, их типов и порядка задания операций передачи аргументов и результатов для взаимодействия компонентов. Описание интерфейса представляет собой интерфейсный посредник между двумя объектами.

ПС построенная из компонентов и предназначенная для функционирования в распределенной среде имеет некоторые особенности в структуре, а именно она состоит из двух частей, каждая из которых выполняется на разных процессах и взаимодействуют друг с другом через вызов интерфейсных функций. Первая часть – серверная программа, а вторая — клиентская (далее просто сервер и клиент).

В функции интерфейсного модуля клиента входят:
– подготовка внешних данных клиента (параметров),
– набор вызовов этих процедур или обращение к сервису сервера,
– обработка разных ошибок, возврат данных от сервера к клиенту.

Общие функции интерфейсного модуля сервера содержат:
– ожидание сообщений клиента и их обработка; запуск удаленной процедуры и передача ей параметров клиента;
– возврат результатов процедуры клиенту, уничтожения удаленной процедуры и др.

Структура интерфейсного модуля не зависит от ЯП взаимодействующих объектов и в целом одинакова для всех. Это связано со стандартизированной его структурой и общим языком спецификации интерфейса, синтаксис которого представлен ниже в форме Бекуса–Наура:

<интерфейс объекта> ::= **Object** *<имя_Объекта>* ***:{***<множество исходных интерфейсов>*}; {*<множество входных интерфейсов>*}* **end**

<множество входных интерфейсов> ::= <множество интерфейсов>
<множество выходных интерфейсов> ::= <множество интерфейсов>
<множество интерфейсов> ::= ∅ | <интерфейс>; <множество интерфейсов>*;*
<интерфейс> :: = **Interface** ***<****имя_интерфейса*>***: {***<множество функций>*}*
end
<множество функций> :: = ∅ | <функция>; <множество функций>*;*
<функция>:::= **function** **<**имя*_функции*>***:*** <множество параметров> **end**
<множество параметров> ::= <параметр> | <параметр>, <множество параметров>
<параметр> ::= <тип> (<вид параметра>)
<вид параметра> :: = **in | out | inout**

Тип описывается средствами языков программирования (C++, Pascal и т.п.) и обеспечивает взаимодействие между процессами, а в качестве <вида параметра> могут быть:

in — входной параметр,
out — выходной параметр,
inout — совместный параметр.

Интерфейсные объекты в распределенной среде являются посредниками между клиентом и сервером (*stub* для клиента и *skeleton* для сервера). Их описания отображаются в те языки программирования, в которых описаны соответствующие им объекты или компоненты.

Описание интерфейса в IDL-языке начинается с ключевого слова **interface**, за которым следует: идентификатор интерфейсного посредника, описание типов параметров и операций вызова объектов.

Общая структура описания интерфейса имеет вид

interface A { ... }
interface B { ... }
interface C: B,A { ... }.

Параметрами операций (op_dcl) в задании интерфейсов могут быть такими:

– тип данных (type_dcl);
– константа (const_dcl);
– исключительная ситуация (except_dcl),возникающая в процессе выполнения метода объекта;
– атрибуты параметров (attr_dcl).

Описание типов данных начинается ключевым словом *typedef*, за которым следует базовый или конструируемый тип и его идентификатор. В качестве константы может быть некоторое значение типа данного или выражение, составленное из констант. Типы данных и констант могут быть: integer, boolean, string, float, char и др.

Описание операций op_dcl включает в себя:

– наименование операции интерфейса;
– список параметров (от нуля и более);
– типы аргументов и результатов, если они имеются (иначе - *void*);
– управляющий параметр и задание исключительной ситуации и др.

Атрибуты параметров могут начинаться служебными словами:

in – при отсылке параметра от клиента к серверу;
out – при отправке параметров-результатов от сервера к клиенту;
inout – при передаче параметров в оба направления (от клиента к серверу и обратно).

Описание интерфейса может наследоваться другим объектом, тогда оно становится базовым. Пример описания базового интерфейса приведен ниже:

const long l=2
interface A {
void f (in float s [l]); }
interface B {
const long l=3 }
interface C: B, A { }.

Интерфейс С использует интерфейс В и А. Это означает, что интерфейс С наследует описание их типов данных, которые по отношению к С становятся глобальными. Но при этом синтаксис и семантика остаются неизменными. Из приведенного примера видно, что операция f в интерфейсе С наследуется из А.

Механизм наследования интерфейса подразумевает, что все имена сохраняются без их переопределения. Особенно это касается описания операций, которые должны иметь

уникальные обозначения. Имена операций могут использоваться во время выполнения интерфейса с помощью *skeleton* при его динамическом вызове.

Общая структура описания модуля в языке IDL с интерфейсом приведена ниже:

```
Regust Operations
module CORBA {
interface Reguest {
Status add-arg (
in Identifier name,
in Flags arg_flags
  );
Status invoke (
in Flags invoke_flags // invocation flags
  );
Status send(
Status get_respouse (
 out Flags response_flags // response flags
      );
    };
 };
```

Предложенный язык описания интерфейса объектов содержит средства общие с языком описания интефейсов IDL.

6.5. Инженерия приложений и предметной области

Базисом инженерии программирования, основанного на использовании ПИК, является, как было сказано выше, прикладная инженерия и инженерия ПрО, которые базируются на методах накопления, поиска и использования готовых ПИК, программ, а также отдельных частей ПС многоразового применения.

Прикладная инженерия – это инженерия ПИК и процесс создания ПС из готовых компонентов и ПИК.

Инженерия ПрО ориентирована на создание архитектуры ПрО - каркаса (фреймворка), представленной ПИК, компонентами многоразового применения из семейства программ ПрО и их интерфейсов.

Основными этапами инженерии ПрО являются:
– анализ ПрО и выявление объектов и отношений между ними;
– определение области действий объектов ПрО;
– определение общих функциональных и изменяемых характеристик, построение модели характеристик, устанавливающей зависимость между различными членами семейства, а также в пределах членов семейства системы;
– создание базиса для производства конкретных программных членов семейства с механизмами изменчивости независимо от средств их реализации;
– подбор и подготовка компонентов многократного применения, описание аспектов выполнения задач ПрО;
– генерация отдельного домена, члена семейства и ПС в целом.

В основе генерации модели ПрО для семейства ПС лежит модель характеристик и набор компонентов реализации задач ПрО. Используя данную модель, знания о

конфигурациях и спецификации компонентов участвующих в этом процессе, можно автоматизировано сгенерировать отдельный член семейства, а также ПО для всей ПрО.

Инженерия ПрО включает в себя следующие вспомогательные процессы:
– корректировка процессов для разработки решений на основе ПИК;
– моделирование изменчивости и зависимостей компонентов многоразового использования, фиксации их в модели характеристик и в справочнике информации об изменении моделей (объектных, Use Case и др.). Фиксация зависимостей между характеристиками модели избавляет разработчиков от некоторых конфигурационных операций, выполняемых, как правило, вручную;
– разработка инфраструктуры ПИК – описание, хранение, поиск, оценивание и объединение готовых ПИК;
– создание репозитария ПИК и компонентов многоразового использования в классе задач ПрО (рис.6.2);
– обеспечение безопасности, защиты данных, изменений;
– обеспечение синхронизации и взаимодействия компонентов и ПИК.

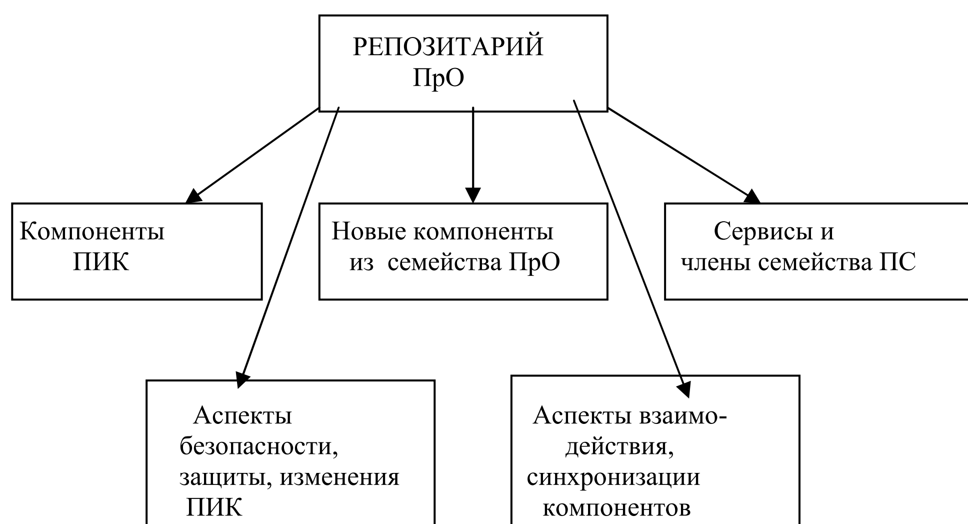

Рис.6.2. Структура репозитария в интегрированной среде ПрО

Архитектурное проектирование домена (Domain design) – это определение архитектуры домена на основе программных компонентов – специфичных активов/ ресурсов.

Архитектура домена – каркас для ПИК, активов и формально определенных интерфейсов должна согласовываться с моделью домена, стандартами организации и оцениваться на соответствие выбранной методологии архитектурного проектирования.

Технология доменной инженерии базируется на **новом процессе** в модели ЖЦ (ISO/IEC 12207) и включает в себя стандартизированные подпроцессы:
– ф*ормированння ресурсров* (Asset provision) – разработка или приобретение ресурсов (активов), которые могут использоваться при компоновки нових программных систем или подсистем.
– разработка базы ресурсов (asset–based development), в основе которой лежит концепция повторного использования (software reuse) – ПИК, обеспечивающая компоновку программных продуктов домена;

– сопровождение ресурсов (Asset maintenance) – модификация и эволюция модели, архитектуры и продуктов домена за счет готовых ресурсов типа ПИК.

Данная технология нуждается в разработке методик и инструментов для эффективного ее выполнения, а также для генерации системы из ПИК и компонентов многоразового применения на основе спецификаций требований к системе.

В результате применения технологии доменной инженерии в софтверной организации будет создаваться, поддерживаться и развиваться архитектурный базис из множества ПИК, хранящийся в репозитарии и учитывающий общие и специфические особенности разных сторон деятельности в доменах.

Основным требованием к инженерии ПрО является обеспечение многоразового применения используемых решений для семейства ПС, а в инженерии приложений – производство (линейка) одиночной системы из ПИК по требованиям к ней.

6.6. Инженерия оценивания стоимости реализации ПрО из компонентов

Инженерия программирования ПС для ПрО создаваемой из компонентов, которые вновь разрабатываются из-за отсутствия готовых, а также компонентов многоразового использования и ПИК, включает в себя оценку стоимости разработки ПС в целях получения сделанных затрат на разработку продукта, составленного из совокупности взаимосвязанных компонентов, реализующих функции ПрО.

Общую стоимость создания компонентной системы будем считать, состоящую из таких составных элементов:

$$C = C_1 + C_2 + C_3 + C_4 ,$$

где C_1 – стоимость анализа функций ПрО, C_2 – стоимость подбора ПИК из репозитария или библиотеки методов с учетом вновь разработанных компонентов, C_3 – стоимость интеграции всех компонентов в систему, C_4 – стоимость определения и обработки данных ПС.

Рассмотрим отдельно каждую составную единицу стоимости ПС.

Стоимость анализа функций ПрО имеет вид

$$C_1 = \sum_{I}^{M} b^1_i \, C_{1i} \, F_i \, (D_i),$$

где D_i – данные i–функции в ПС, M – количество функций F в системе,

$$b^1_i = \begin{cases} 1, \text{ когда функция реализована в компонентах ПС,} \\ 0, \text{ в противном случае.} \end{cases}$$

Стоимость поиска и исследования возможностей применения ПИК, полученного с репозитария, для реализации некоторой определенной функции ПрО, которая вычисляется с помощью выражения:

$$C_2 = \sum_{j}^{N} \sum_{I}^{M} a^2_{ji} \, C_2 \, (F_{ji}) + C_2 \, (PF_{ji}),$$

где $C_2 \, (F_{ji})$ – стоимость поиска ПИК для функции F_i, сформулированной на этапе анализа ПрО, N – количество новых компонентов и ПИК, $C_2(PF_{ji})$ – стоимость разработки некоторых типичных программных компонентов,

$$a^2_{ji} = \begin{cases} 1, \text{ когда } j\text{– компонент используется функцией } F_i , \\ 0, \text{ в противном случае.} \end{cases}$$

Стоимость композиции компонентов определяется следующим образом:

$$C_3 = \sum_{j}^{N} \sum_{I}^{M} \sum_{r}^{K} d^2_{jik}\, C_3(I_{jr}),$$

где $C_3(I_{jr})$ – стоимость создания интерфейсных модулей пары компонентов K_i и K_r,

$$d^2_{jik} = \begin{cases} 1, \text{ когда } r \text{ – параметр из набора } X=(X_1, \dots, X_r) \text{ есть входным} \\ \quad \text{для } J \text{ –компонента, } r \text{– функции } (r=1,\dots, K), \\ 0, \text{ в противном случае.} \end{cases}$$

Таким образом, конечный результат оценивания стоимости ПС получается путем суммирования $C = C_1 + C_2 + C_3 + C_4$ (расчет C_4 громоздкий – не приводится) и имеет вид:

$$C = \begin{cases} \sum^{M} b^1_u\, C_{1u}\, F_i(D_i) + \sum^{N} \sum^{M} a^2_{ji}\, C_2(F_{ji}) + C_2(PF_{ji}) \quad + \\ \quad\quad\quad\quad\quad\quad\quad\quad J \quad I \\ \sum_{j}^{N} \sum_{I}^{M} \sum_{r}^{K} d^2_{jik}\, C_3(I_{jr}) + C_4\,. \end{cases}$$

Основными ограничениями данного выражения является необходимость реализации заданных функций в ПС, наличие средств интеграции пар компонентов K_i и K_r, которые могут быть заданы в любых современных ЯП в заданной среде функционирования, количество компонентов К соответствует заданным функциям, которые обеспечивают решение задач ПрО.

Расчет стоимости C для компонентных систем является трудоемким процессом. Общая стоимость уменьшается, если описание компонентов выполнено на одном из ЯП, за счет отсутствия интерфейсных модулей преобразования данных в системе.

Таким образом, инженерия программирования компонентных систем характеризуется степенью использования в них накопленной программной продукции в виде ПИК и компонентов ПрО многоразового использования. Она требует не только их подбора для применения в новых разработках ПС, но соответствующих инженерных оценок качества, стоимости, риска от приобретения, трудозатрат на разработку с учетом полученных выгод (а также потерь при изменениях и адаптации ПИК) от использования уже произведенного программного изделия и т.п.

Литература к теме 6.

1. *Бабенко Л.П., Лаврищева Е.М.* Основы програмной инженерии (укр.).– Киев, Знання. –2001.–269с.
2. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного пограммирования//Кибернетика и системный анализ, 2003.– №1.– с.39–55.
3. *Jacobson I., Griss M., Jonsson P.* Software Reuse.–N.Y.:Adison Wesley, 1997.
4. *Crnkovik I, Larsson S., Stafford J.* Component-Based Software Engineering: building systems from Components at 9th Conference and Workshops on Engineering of Computer – Based Systems.- Software Engineering Notes.-2002.- vol.27.-N 3.-c.47-50.
5. *К.Чернецки, У.Айзенекер*. Порождающее программирование. Методы, инструменты, применение.– Издательский дом «Питер».– Москва– Санкт-Петербург… Харьков, Минск.– 2005.–730с.
6. *Meyer B*. On to Components. Computer. – vol. 32, N 1, January 1999. – pp.139–140.
7. *Lowy J.* COM and .NET Component Services. - O'Reilly, 2001. – 384 p.
8. *Batory D., O'Malley S.* The Design and Implementation of Hierarchical Software Systems with Reusable Components/ ACM Transactions on Software Engineering and Methodology. – N 4, vol. 1, October 1992. – pp.355–398.

9. *Weide B., Ogden W., Sweden S.* Reusable Software Components/ Advances in Computers, vol. 33. – Academic Press, 1991. – pp.1–65.
10. *Jacobson I., Griss M., Johnson P.* Software Reuse: Architecture, Process and organization for Business Success – Addison Wesley, Reading , MA, May 1997.–501p.

Тема 7

МЕТОДЫ ВЕРИФИКАЦИИ И ТЕСТИРОВАНИЯ ПРОГРАММ И СИСТЕМ

В фундаментальную концепцию проектирования ПС входят базовые положения, стратегии, методы, которые применяются на процессах ЖЦ и обеспечивают верификацию, проверку правильности путем доказательства и тестирование на множестве тестовых наборов данных. К методам проектирования ПС относятся структурные, объектно–ориентированные и др. Их основу составляют теоретические, инструментальные и прикладные средства, применяемые на процессах тестирования.

Теоретические средства определяют процесс программирования и тестирования программного продукта. Это методы верификации и доказательства правильности составленной спецификации программ, метрики (Холстеда, цикломатичная сложность Маккейба и др.) измерения отдельных характеристик, и выступают они в роли формализованных элементов теории определения правильности и гарантии свойств конечного ПО. Например, концепция «*чистая комната*» базируется на некоторых формализмах доказательства и изучения свойств процессов кодирования и тестирования программ. Что касается тестирования, то это проверка спецификации нотаций программ, используемых при описании тестов и покрытия соответствующих критериев программ [1–7].

Инструментальные средства – это такие способы поддержки кодирования и тестирования (компиляторы, генераторы программ, отладчики и др.), а также организационные средства планирования и отбора тестов для программ, которые обеспечивают обработку текста на ЯП и подготовку для них соответствующих тестов.

Для проверки правильности программ и систем используются следующие основные направления обеспечения правильности ПС.

1. *Формальное доказательство* корректности программ осуществляется с помощью теоретических методов, основанные на задании формальных систем правил и утверждений, используемых при доказательстве правильности операторов программы и результатов их выполнения в режиме интерпретации [5, 8–11]. К средствам формальной проверки правильности относятся верификация и валидация ПС, которые вошли в состав регламентированных процессов ЖЦ стандарта ISO/IEC 12207.
2. *Тестирование* – это системный метод обнаружения ошибок в ПО путем исполнения выходного кода ПС на тестовых данных, сбор рабочих характеристик в динамике выполнения ПС в конкретной операционной среде [1–7]. Методы тестирования позволяют выявить в процессе выполнения ПС различные ошибки, дефекты и изъяны, вызванные аномальными ситуациями, сбоями оборудования и аварийным прекращением работы ПО.
3. Организационные аспекты проверки правильности.

Далее излагаются последовательно эти направления.

7. 1. Методы доказательства программ

Формальное математическое доказательство основывается на аксиомах, а сам процесс основывается на спецификации описания алгоритма доказываемой программы.

Доказательство корректности начинается с предположения о том, что в начале работы программы удовлетворяются некоторые условия, называемые предварительными условиями или предусловиями. Для проведения доказательства разрабатываются *утверждения* о правильности выполнения операторов программы в различных точках программы. Создается набор утверждений, каждое из которых является следствием предусловий и последовательности инструкций, приводящих к соответствующей отмеченной точке программы, для которой сформулировано данное утверждение. Если утверждение соответствует конечному оператору программы, то выполняется заключительное утверждение и пост условие, позволяющее сделать вывод (заключение) о правильности работы программы [5, 8–11].

К методам проверки правильности программ относятся:
1) методы доказательство правильности программ;
2) верификация и аттестация программ.

7.1.1. Методы доказательства правильности программ

Эти методы появились в 80–е годы и разделяются на два класса:
1. Точные методы доказательства правильности программ.
2. Методы доказательства частичной правильности программ.

Наиболее известными точными методами доказательства программ являются метод рекурсивной индукции или индуктивных утверждений Флойда и Наура и метод структурной индукции Хоара и др. Эти методы основываются на утверждениях и пред и пост–условиях.

7.1.1.1. Общая характеристика формальных методов доказательства

Над этим направлением работали многие теоретики–программисты, некоторые из них, которые являются базовыми методами доказательства программ, кратко излагаются ниже.

Метод Флойда основан на определении условий для входных и выходных данных и в выборе контрольных точек в доказываемой программе так, чтобы путь прохождения по программе проходил хотя бы через одну контрольную точку. Для этих точек формулируются утверждения о состоянии переменных в этих точках (для циклов эти утверждения должны быть истинными при каждом прохождении цикла – инварианты цикла).

Каждая точка рассматривается, как индуктивное утверждение, т.е формула, которая остается истинной при возвращении в эту точку программы и зависит не только от входных и выходных данных, но от значений промежуточных переменных. На основе индуктивных утверждений и условий на аргументы программы строятся условия проверки правильности этой программы. Для каждого пути программы между двумя точками устанавливается соответствие условий правильности и определяется истинность этих условий при успешном завершении программы на данных, которые удовлетворяют входным условиям.

Формирование таких утверждений оказалось очень сложной задачей, особенно для программ с высокой степенью параллельности и взаимодействия с пользователем. Как оказалось на практике, достаточность таких утверждений трудно проверить.

Доказательство корректности применялось для уже написанных программ и тех, что разрабатываются путем последовательной декомпозиции задачи на подзадачи, для каждого из них формулировалось утверждение с учетом условий ввода и вывода и соответствующих входных и выходных утверждений. Доказать истинность этих условий – основа метода доказательства полноты, однозначности и непротиворечивости спецификаций.

Метод Хоара – это усовершенствованный метод Флойда, основанный на аксиоматическом описании семантики ЯП исходных программ. Каждая аксиома описывает изменение значений переменных с помощью операторов этого языка. Операторы перехода, рассматриваемый как выход из циклов и аварийных ситуаций, и вызовов процедур определяются правилами вывода, каждое из которых задает индуктивное высказывание для каждой метки и функции исходной программы Система правил вывода дополняется механизмом переименования глобальных переменных, условиями на аргументы и результаты.

Метод рекурсивных индукций Дж. Маккарти состоит в структурной проверке функций, работающих над структурными типами данных, изменяет структуры данных и диаграммы перехода во время символьного выполнения программ. Тестовая программа получает детерминированное входное состояние при вводе данных и условий, которое обеспечивает фиксацию переменных и изменение состояния.

Выполняемая программа рассматривается как серия изменений состояний, самое последнее состояние считается выходным состоянием и если оно получено, то программа считается правильной. Моделирование выполнения кода является дорогим процессом, обеспечивающим высокое качество исходного кода.

Метод Дейкстры включает два подхода к доказательству правильности программ. Первый – основан на модели вычислений, оперирующих с историями результатов вычислений при работе программ и анализом выполнения модели вычислений.
Второй подход базируется на формальном исследовании текста программы с помощью предикатов первого порядка применительно к классу асинхронных программ, в которых возникают состояния при выполнении операторов. В конце выполнения программы уничтожаются накопленные отработанные состояния программы. Основу этого метода составляет – перевычисление, математическая индукция и абстракция.

Перевычисление базируется на инвариантах отношений, которые проверяют границы вычислений в проверяемой программе. *Математическая индукция* применяется для проверки циклов и рекурсивных процедур с помощью необходимых и достаточных условий повторения вычислений. *Абстракция* задает количественные ограничения, которые накладываются методом перевычислений.

Доказательство программ по данному методу можно рассматривать как доказательство теорем в математике, используя аппарат математической индукции при неформальном доказательстве правильности, который может привести к ошибкам, как в самом аппарате, так и в программе.

В общем метод математической индукции разрешает доказать истинность некоторого предположения $P(n)$, в зависимости от параметра n, для всех $n \geq n_0$, и тем самым доказать случай $P(n_0)$. Истинность $P(n)$ для любого значения n, доказывает $P(n+1)$ и тем самым доказательство истинности $P(n)$ для всех $n \geq n_0$.

Этот путь доказательства используется для утверждения *A* относительно программы, которая при своем выполнении достигает определенной точки. Проходя через эту точку *n* раз, можно получить справедливость утверждения *A(n)*, если докажем:

1) что справедливо *A(1)* при первом проходе через заданную точку,
2) если справедливо A(*n*) (при *n*–проходах через заданную точку), то справедливо и A (*n+1*) при прохождении через заданную точку *n+1* раз.

Чтобы доказать, что некоторая программа правильная, надо правильно описать, что эта программа делает, т.е ее логику. Такое описание называется *правильным высказыванием* или просто утверждением. Исходя из предположения, при котором работающая программа в конце концов успешно завершится, утверждение о правильности будет справедливо.

7.1.1.2 Модель формального доказательства конкретности программы

Сущность формального доказательства заключается в преобразовании кода программы к логической структуре [3, 6] Составляется описание утверждений, которые задают вход–выход программ с помощью логических операторов, а также комбинациями логических переменных (true/false), логическими операциями (конъюнкция, дизъюнкция и др.) и кванторами всеобщности и существования (табл.7.1.).

Таблица 7.1

Логические операции

Название	Примеры	Значение
Конъюнкция	$x \& y$	x и y
Дизъюнкция	$x * y$	x или y
Отрицание	$\neg x$	не x
Импликация	$x \rightarrow y$	если x то у
Эквивалентность	$x = y$	x равнозначно y
Квантор всеобщности	$\forall x\, P(x)$	для всех x, условие истинно
Квантор существования	$\exists x\, P(x)$	существует x, для которого $P(x)$ истина

1. Для примера метода доказательства предлагается к рассмотрению задача сортировки одномерного массива целых чисел *T* длины *N* (*T [1:N]*) для получения из него эквивалентного массива *T'* той же длины *N,* что и T, элементы которого располагались бы в порядке возрастания их значений.

Входные условия запишем в виде начального утверждения:
A_{beg}: *(T [1:N]* – массив целых) *& (T' [1:N]* массив целых).

Выходное утверждение A_{end} запишем как конъюнкции таких условий:
(а) (*T*– массив целых) *&* (*T'* – массив целых)
(б) $(\ \forall i$ если $i \le N$ то $\exists j\ (T'(i) \le T'(j)))$
(в) $(\ \forall i$ если $i < N$ то $(T'(i) \le T'(i+1))$),

то есть A_{end} – это *(T –* массив целых) *& (T' –* массив целых) *&* $\forall i$ если $i \le N$ *то* $\exists j$ $(T'(i) \le T'(j))$ *&* $\forall i$ если $i < N$ *то* $(T'(i) \le T'(i+1))$.

Для расположения элементов массива Т в порядке возрастания их величин в массиве Т' в используется алгоритм пузырьковой сортировки, суть которого заключается в предварительном копировании массив Т в массив Т', а затем проводится сортировка элементов согласно условия их возрастания. Алгоритм сортировки представлен на блок–схеме (рис.7.1).

Операторы алгоритма размещены в прямоугольниках. Условия, с помощью которых происходит выбор альтернативных путей в алгоритме, представлен в параллелограммах. Кружками отмечены точки с начальным A_{beg} конечным условиями A_{end} и состояниями алгоритма: кружок с нулем обозначает начальное состояние, кружок с одной звездочкой – состояние после обмена местами двух соседних элементов в массиве Т', кружок с двумя звездочками обозначает состояние после обмена местами всех пар за один проход массива *T'*.

Кроме уже известных переменных *T, T'* и *N*, в алгоритме использованы еще два переменные: *i* – типа целая и *M* – булева, значением которой являются логические константы true и false.

2. Для доказательства того, что алгоритм действительно обеспечивает выполнение исходных условий, рассмотрим динамику выполнения этих условий последовательно в определенных точках алгоритма. Заметим, что указанные точки делят алгоритм на соответствующие части, правильность любой из которых обосновывается в отдельности.

Так оператор присваивания означает, что для всех i $(i \le N \ \& \ i>0)$ выполняется $(T'[i] := T[i])$. Результат выполнения алгоритма в точке с нулем может быть выражен утверждением:

(*T[1: N]* – массив целых) *&* *(T '[1: N]* – массив целых)
& ($\forall i$ *если* $i \le N$ $(T[i] = T[i])$*).*

Доказательство очевидно, поскольку за семантикой оператора присваивания, обеспечивающая поэлементную пересылку чисел из *T* в *T'* , сами элементы при этом не изменяются и порядок их в *T* і *T'* одинаковый. Получили, что условие (б) исходного утверждения выполнено. Первая строка доказанного утверждения совпадает с условием (а) исходного утверждения A_{end} и остается справедливой до конца работы алгоритма. В точке алгоритма с одной звездочкой выполняется оператор

$(i < N)\ (T'(i)) > T'(i+1) \rightarrow (T'(i)$ и $T'(i+1)$ для изменения местами элементов.
В результате работы оператора будет справедливым такое утверждение:

$\exists i$ если $i < N$ *то* $(T'(i) < T'(i+1))$, которое является частью условия (в) утверждения A_{end} (для одной конкретной пары смежных элементов массива *T'*).
Очевидно, что семантика оператора обмена местами не нарушает условия (б) выходного утверждения A_{end}.

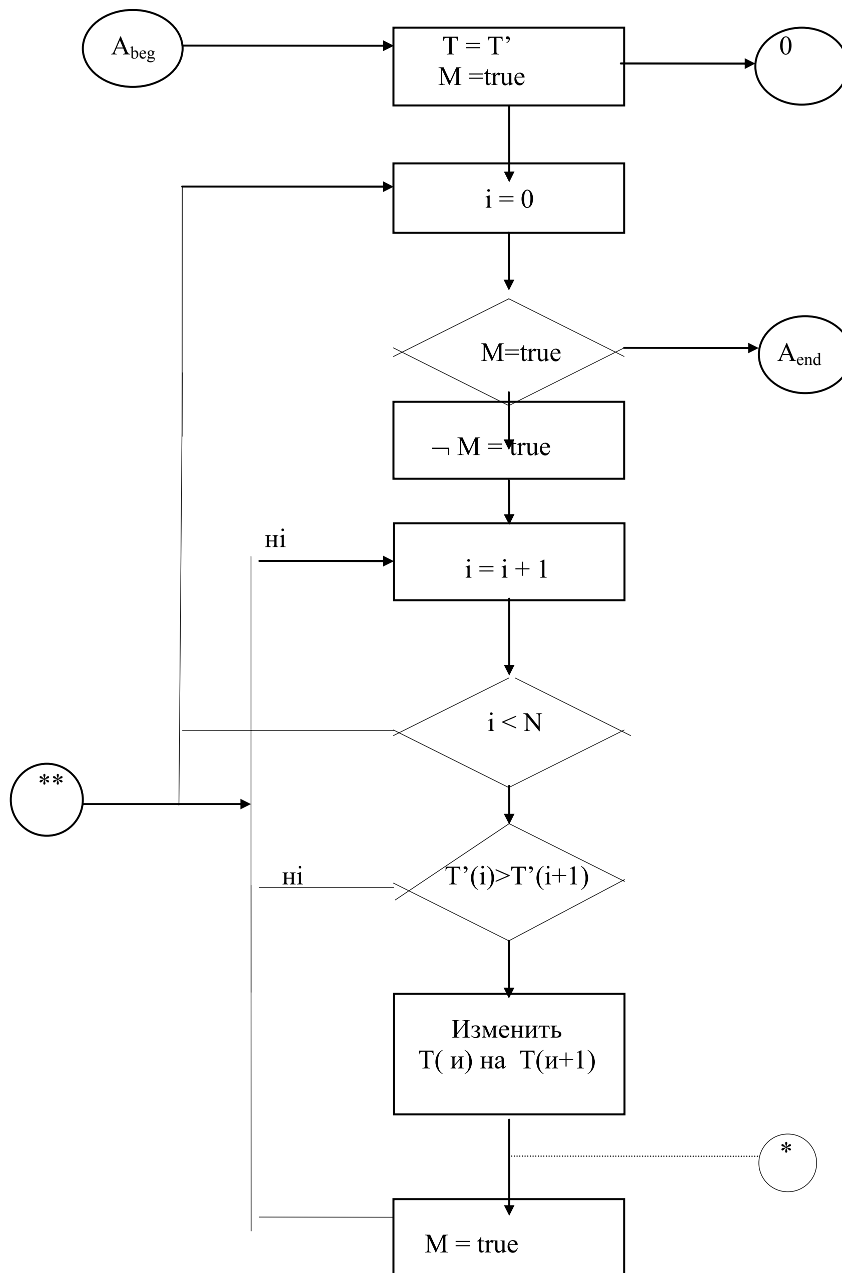

Рис.7.1. Схема сортировки элементов массива T'

В точке с двумя звездочками выполнены все возможные операции обмена местами пар смежных элементов массива *T′* за один проход через *T′*, то есть оператор обмена работал один или больше раз. Однако пузырьковая сортировка не дает гарантии, что достигнуто упорядочение за один проход по массиву *T′*, поскольку после очередного обмена индекс i увеличивается на 1 независимо от того, как соотносится новый элемент $T'(i)$ с предшествующим элементом $T'(i-1)$.

В этой точке также справедливое утверждение:
$\exists i$, если $i < N$ *то* $T'(i) < T'(i+1)$.

Часть алгоритма, обозначенная точкой с двумя звездочками выполняется до тех пор, пока не будет упорядочен весь массив, то есть не будет выполняться условие (в) утверждения A_{end} для всех элементов массива T':
$\forall i$, если $i < N$ *то* $T'(i) < T'(i+1)$.

Иными словами, выполнение исходных условий обеспечена соответствующим преобразованием массива и порядком их выполнения. Доказано, что выполнение программы завершено успешно.

Из этого утверждения генерируется серия теорем, которые доказываются. Начиная с первого утверждения и переходя от одного преобразования к другому, вырабатывается путь вывода, который состоит в том, что если одно утверждение есть истинное, то есть истинно и другое.

Другими словами, если первое утверждение – A_1 и первая точка преобразования – A_2 , то первой теоремой есть: $A_1 \rightarrow A_2$..
Если A_3 есть такой точкой преобразования, то второй теоремой будет: $A_2 \rightarrow A_3$.

Таким образом, формулируется общая теорема:
$A_i \rightarrow A_j$, где A_i и A_j – смежные точки преобразования.

Последняя теорема формулируется так, что условие «истинное» в последней точке и отвечает истинности выходного утверждения: $A_k \rightarrow A_{end}$.

Соответственно, можно возвратиться потом к точке преобразования A_{end} и к предшествующей точке преобразования. Доказали, что $A_k \rightarrow A_{end}$, а значит и $Aj \rightarrow Aj_{+1}$ и так далее, пока не получим, что $A_1 \rightarrow A_0$.
Результат любого другого подхода к доказательству правильности будет аналогичный.

4. Чтобы доказать, что программа корректная, необходимо последовательно расположить все части, которые начинаются с A_1 и заканчиваются A_{end}. Последовательность этих частей определяет, что истинность входного условия обеспечивает истинность выходного условия.

5. После идентификации всех частей проверяется истинность каждой части программы с утверждением, что входные утверждения являются следствием выходного утверждения, которые отвечают преобразованиям ее частей. Доказательство программы завершено.

7.1.2. Техника символьного выполнения

Техника логического доказательства игнорирует структуру и синтаксис ЯП, в которых тестовые программы выполняются. В случае, когда техника доказывает, что спроектированные компоненты являются правильными, они необязательно выполняются. Одна из таких техник – *символьное выполнение* – включает моделирование выполнения кода, используя символы вместо переменных данных. Тестовая программа рассматривается как имеющая детерминированное входное состояние при вводе данных и условий. Благодаря этому каждая строка кода выполняется, а данная техника проверяет, изменилось ли состояние. Каждое измененное состояние сохраняется, и выполняемая программа рассматривается как серия изменений состояний. Последнее состояние каждой части будет выходным состоянием и программа будет правильной.

Символьная проверка приводится на примере нескольких строк фрагмента программы:

```
a = b + c ;
```

```
if (a > d) task x ( ) ;    //  выполнить задание x
else task y ( );           //  выполнить задание y
```

В нем проверяется условие a>d на истинность или нет. Поскольку выполнение условного кода включает значения переменных a и d, то рассматривается два случая, определенные на двух отдельных эквивалентных классах, когда условие верно и получен соответствующий результат. И второй случай, когда условие не выполняется.

Описание доказательства может оказаться длиннее написанного фрагмента программы и нет уверенности, что это описание не содержит ошибок. Данная стратегия имеет преимущество перед логическим доказательством теорем, поскольку полагается на трассирование изменяющихся условий операторов программв. Главными элементами доказательства корректности программы являются фрагмент программы, входные и выходные значения переменных, а также процесс слежение за тем, как этот фрагмент программа будет выполняться.

7.1.3. Методы просмотра структуры программы

Метод просмотра применяется к инспекции созданных программ независимыми экспертами и с участием самих разработчиков. На начальном этапе проектирования инспекция предполагает проверку полноты, целостности, однозначности, непротиворечивости и совместимости документов с исходными требованиями к ПС.

На этапе реализации системы инспекция состоит в анализе текста программы и проверку соблюдения требований к ней и стандартных руководящих документов технологии программирования. Эффективность инспекции заключается в том, что эксперты пытаются взглянуть на проблему "со стороны", подвергнуть ее всестороннему критическому анализу и просмотру словесных объяснений способов проектирования. Непосредственный просмотр исходного кода позволяют обнаружить ошибки в логике и в описании алгоритма.

Эти приемы позволяют на более ранних этапах проектирования обнаружить ошибки путем многократного просмотра исходных кодов. Методы просмотра не формализованы и определяются степенью квалификации экспертов группы. Основные методы рассматриваются ниже.

Метод простого структурного анализа. Он ориентирован на анализ структуры программы и потоков данных и базируется на использовании теория графов для представления структуры программы в виде графа, каждая вершина которого – это оператор, а дуга – передача управления между операторами. Согласно графу определяется достижимость вершин программы и выход потоков управления для ее завершения и обнаружение логических ошибок [5,12].

Для проведения анализа потоков данных данный граф пополняется указателями переменных, их значениями и ссылками на операторы программы. При тестировании потоков данных определяются значения предикатов в логических операторах, по которым формируются пути выполнения программы. Для прослеживания путей устанавливаются точки, в которых имеется ссылка на переменную до присвоения ей значения, либо переменной присваивается значение без ее описания, либо делается повторное описание переменной или переменной, к которой нет обращения.

Метод анализа дерева отказов. Этот метод пришел в программную инженерию из техники, в которой он широко применяется для анализа неисправностей аппаратуры. Суть метода состоит в выборе "ситуации отказа» в определенной компоненте системы, прослеживании событийных цепочек, которые могли бы привести к ее возникновению, и в построении дерева отказов, использующего предикаты *и, или.* С другой стороны, просматривается влияние отказа в одной компоненте на программное обеспечение в целом. При данном способе строиться дерева отказов. Данный метод применяется как на модульном уровне, так и на уровне анализа функционирования комплекса. Известен опыт его использования при разработке систем реального времени.

Метод проверки непротиворечивости. Применяется при анализе логики программы для выявления операторов, которые никогда не используются, а также для обнаружения противоречий в логике программы. Этот метод часто называют методом *анализа потоков управления* на входных данных, часть из которых представляется в символьном виде. Результатом выполнения являются значения переменных, выраженные формулами над входными данными. В этом методе выделяются два вида задач:

1. Построения тестового набора данных для заданного пути, определяющего этот путь при символьном выполнении. В случае отсутствия такого набора делается заключение о нереализуемости (непротиворечивости) данного пути.

2. Определение пути, который будет пройден при заданных ограничениях на входные данные в виде некоторых областей значений входных объектов, и задание символьных значений переменных в конце пути, т.е. построение функции, которая реализует отображение входных данных в выходные.

Рассмотрим эти виды задач.
1). *Символическое выполнение* программа P (X, Y) на некоторых наборах данных $D = (d_1, d_2, ..., d_n)$, $D \subset X$, X – множество входных данных программы P.

Пронумеруем операторы программы P. Состояние выполнения программы – это тройка
$< N, pc, Z >$,
где N – номер текущего оператора программы P, pc (part condition) – условие выбора пути в программе (вначале это true), что является логическим выражением над D; Z – множество пар $\{ < z_i, e_i > | \; z_i \in X \cap Y$, в которых z_i – переменная программы, а e_i – ее значение; Y – множество промежуточных и выходных данных.

Семантика символического выполнения задается правилами оперирования символьными значениями, согласно которым арифметические вычисления заменяются алгебраическими.

Зададим семантику символического выполнения операторов через базовые конструкции языков программирования. Для императивных языков базовыми конструкциями есть операторы присваивания, перехода и условные операторы.

В операторе присваивания $Z = e(x, y)$, $x \in X$, $y \in Y$, в выражение $e(x, y)$ производится подстановка символьных значений переменных x и y, в результате чего получается выражение $e(D)$, которое становится значением переменной z $(z \in X \cup У)$. Вхождение в полученное выражение $e(D)$ переменных из Y означает, что их значения, а также значение z не определены.

По оператору перехода управление передается оператору, помеченному соответствующей меткой. В условном операторе « <u>если</u> α *(x, y)* <u>то</u> B1 **иначе** B2 » вычисляется выражение α *(x, y)*. Если оно определено и равно α' (D), то формируются логические формулы:

$$pc \rightarrow \alpha'(D) \quad (1)$$
$$pc \rightarrow \neg\, \alpha'(D) \quad (2)$$

Если *pc* – ложно (*false*), то только одна из последних формул может быть выполнимой, тогда :
– если выполнима формула (1) , то управление передается на B1;
– если выполнима формула (2), то управление передается на B2;
– если (1), (2) не выполнимы (то есть из *pc* не следует ни $\alpha'(D)$, ни $\neg\alpha'$ (D)), тогда по крайней мере один набор данных, который удовлетворяет pc и соответствует части «<u>то</u>» условного оператора, а также есть набор данных, соответствующий <u>иначе</u> этого условного оператора.

Таким образом, образуются два пути символьного выполнения, которым соответствуют свои *pc*:

$$pc_1 = pc \wedge \alpha'(D)$$
$$pc_2 = pc \wedge \neg\, \alpha'(D)$$

Символическое выполнение, нацеленное на построение тестового набора данных, реализуется следующим алгоритмом. Пусть *pc* = *true*, тогда
– для заданного пути формируется *pc* согласно семантики операторов, для условного оператора семантику трактуем как преобразование *pc* вида (1) или (2);
– решаем обе системы уравнений (1) и (2).

Решение дает тест проверки путей программы, если такого решения нет, то это означает невыполнимость пути.

2). *Определение пути* при заданных ограничениях на входные данные проводится таким шагами:

– полагаем $pc = \beta(D)$, где $\beta(D)$ – входная спецификация, когда ее нет, то *pc* = *true*;
– производим символьное выполнение операторов, если встречается ветвление, то запоминается состояние в данной точке или выбирается одна из ветвей; выбирается новая спецификация и выполняется условный оператор, при котором формируется состояние программы с условием *pc*;
– в конце пути совокупность состояний выходных переменных определяет функцию программы и набор данных, который удовлетворяет *pc* и является тестом, который покрывает данный путь;
– для всех промежуточных $\delta(x,y)$ с выходной спецификацией $\gamma(x,y)$ делается попытка доказать выполнимость следующих логических формул:

$$pc \rightarrow \delta(x,y) \quad \text{и} \quad pc \rightarrow \gamma(x,y),$$

где *pc* является значением текущего условия в данной точке программы.

Для доказательства этих формул требует провести верификацию программы на данном участке пути. Для установления пути выражения декомпозируются на множество неравенств. Если это множество содержит некоторые несовместимости, то путь нельзя установить. В случае совместимости множеств создается множество данных, которые будут использоваться во время символьного выполнения.

Символьное тестирование применяться при определении тестовых данных для проверки отдельных путей с использованием символьных значений для реальных. При этом входные значения обозначаются символами, а операторы выполняются с использованием элементарной алгебры. Выполнение заданного пути обычно включает условные переходы, символьные выражения, до которых входят как алгебраические, так и булевские конструкции.

С целью проведения статического анализа используется различные инструменты, которые позволяют исследовать структуру программы и определить виды ошибок в программе: невыполнимые коды, неинициализированные переменные, которыми хотят воспользоваться, инициализированные, но не использованные переменные и др.

Многие отказываются от формального доказательства. Это связано с тем, например, алгоритм пузырьковой сортировки намного более простой, чем его логическое описание и доказательство, а также программы обработки нечисловых данных могут быть более трудными для понимания логики, чем числовые. Техника доказательства, которая базируется на входных утверждениях для их трансформации в выходные логические правила еще не означает, что в программе нет ошибок, поскольку нельзя распознать ошибки в программах, в интерфейсах, спецификациях, в синтаксисе и семантике ЯП или в документации.

7.1.2. Верификация и аттестация программ

Верификация и аттестация (валидация) – это методы, которые обеспечивают соответственно проверку и анализ правильности выполнения заданных функций и соответствия ПО требованиям заказчика, а также заданным спецификациям ПО [12–17]. Эти методы обозначены в стандарте ISO/IEC 12207 [18] как самостоятельные процессы ЖЦ и используются, начиная от этапа анализа требований и кончая проверкой правильности функционирования программного кода на заключительном этапе – тестировании.

Верификация – это проверка того, правильно ли система работает в соответствии с ее спецификацией и заданными требованиями заказчика, Этот процесс ЖЦ стандарта ISO/IEC 12207 позволяет сделать заключение о корректности сделанной системы.

Валидация является методом проверки соответствия спроектированного ПО требованиям и потребностям заказчика и предполагает выполнение на этапах ЖЦ разного рода действий для получения корректных программ и систем:

– планирование процедур проверки и контроля проектных решений с помощью методик и просмотра хода разработки;
– повышения уровня автоматизации проектирования программ с использованием – CASE–систем [19];
– проверка правильности функционирования программ с помощью методик тестирования на наборах целевых тестов;
– структурирование системы на модули, их спецификации, реализация и использование их как повторных компонентов (reuse) [10, 20];
– адаптация продукта к условиям использования;
– управления проектом.

Валидация опирается на просмотры и инспекции промежуточных результатов на каждом этапе ЖЦ с целью анализа на соответствие их требованиям и тем самым

позволяет подтвердить, что ПО имеет корректную реализацию начальных требований и условий к системе

Таким образом, основными особенностями методов верификации и валидации является проверка полноты, непротиворечивости и однозначности спецификаций требований к созданному ПО.

Верификация и валидация предполагают планирование этих процессов в целях распределения ресурсов и сосредоточения проверки на наиболее критичных элементах проекта, а именно:
– компонентов системы;
– интерфейса компонентов системы (программные, технические и информационные) и взаимодействий объектов (протоколов и сообщений) для функционирования в современных распределенных средах;
– средств доступа к БД и файлам, которые обеспечивают защиту от несанкционированного доступа к ним разных пользователей;
– документация на ПО;
– тестов и тестовых процедур;
– специальных средств защиты информации в системе.

По окончании проектирования приведенных элементов соответственно проводится:
– проверка правильности перевода отдельных компонентов в выходной код, а также описаний их интерфейсов, трассировки этих компонентов в соответствии с требованиями заказчика к функциям системы;
– анализ способов доступа к файлам или БД соответственно требований, принципов передачи данных и процедур манипулирования данными;
– проверка средств защиты на удовлетворение требованиям заказчика и правильности реализации.

По завершению процессов верификации и валидации создается комплект материалов отображающий правильность формирования требований, спецификация элементов системы, результатов проведения инспекций и тестирования программ.

7.1. 3. Методы верификации объектно–ориентированных программ

Верификация таких программ имеет свою специфику и ее можно рассматривать исходя из композиционного метода, применяя к ним известные методы доказательства правильности композиционных программ [14, 16, 17].

Выделим несколько этапов для верификации исходного проектирования объектных моделей и программ:

1. Верификация базовых (простых) объектов сводится к верификации структуры, где атрибуты объектов являются данными структуры, а внутренние операции объекта — функциями над этими данными.

2. Верификация объектов, построенных с помощью наследования, агрегации или инкапсуляции, осуществляется исходя из следующих предположений:
– базовые объекты считаются проверенными, если их операции (функции) приняты за теоремы;
– доказательство объектов сводится к этим теоремам;

– доказывается, что все операции, которые применяются над подобъєктами, не выводят их из множества состояний, для которых они верифицированы.

3. Верификация интерфейсов объектов сводится к доказательству:
– правильности данного интерфейса;
– достаточности параметров интерфейса.

Метод верификации ООП на основе композиционного подхода можно использовать как “вниз” так и “вверх”. Сначала доказывается правильность построенной ОМ для некоторой ПрО. Верификация ОМ требует реализации на этапах ЖЦ следующих вопросы:

– удаление лишних атрибутов объектов и их интерфейсов в ОМ, внесение необходимых изменений и повторное доказательство правильности объекта;
– выбор типа множества для атрибутов объекта и проверка реализации операций и множество значений этого типа.

Правильность спецификаций любого объекта ПС доказывается независимо от правильности смежных объектов и верификации их интерфейсов. Это является заключительной проверкой ОМ.

7.2. Методы тестирования программ

Тестирование – это способ семантической отладки (проверки) программы, заключающийся в обработке последовательности различных контрольных наборов тестов, для которых известен результат. Тестирование основывается на выполнении программы и получении результатов выполнения тестов [1–7, 21, 22].

Тесты подбираются так, чтобы они охватили как можно больше различных типов ситуаций обработки элементов программы. Более слабое требование – выполнение хотя бы один раз каждого разветвления программы в определенном направлении.

Исторически первой разновидностью тестирования была отладка.
Отладка означает проверку описания программного объекта в ЯП на наличие в нем ошибок и последующее их устранение. Ошибки обнаруживаются компиляторами при проверки синтаксической правильности. После этого выполняется верификация для установлению правильности кода и валидация на проверку соответствия продукта заданным требованиям.

Следующим шагом является функциональное тестирование для проверки реализованных функций в соответствии с их спецификацией. Создаются функциональные тесты на основе внешних спецификаций функций и проектной информации на этапов ЖЦ. Тестирование по внешним спецификациям проводится с учетом требований, сформулированных на этапе анализа предметной области. Методы функционального тестирования подразделяются на статические и динамические.

7. 2. 1. Статические методы тестирования

Статические методы используются при проведении инспекций и рассмотрении спецификаций компонентов без их выполнения.

Техника статического анализа заключается в методическом просмотре (или обзоре) и анализе структуры программ, а также в доказательстве правильности. Статический анализ направлен на анализ документов, разрабатываемых на всех этапах ЖЦ и заключается в инспекции исходного кода и сквозного контроля программы.

Инспектирование ПО – это статическая проверка соответствия программы заданным спецификациями, проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, схем или исходного кода программ) на процессах ЖЦ. Просмотры и инспекции результатов проектирования и соответствия их требованиям заказчика обеспечивают более высокое качество создаваемых ПС.

При инспекция программ рассматриваются документы рабочего проектирования на этапах ЖЦ совместно независимыми экспертами и участниками разработки ПС.. На начальном этапе проектирования инспектирование предполагает проверку полноты, целостности, однозначности, непротиворечивости и совместимости документов с исходными требованиями к программной системе. На этапе реализации системы под *инспекцией* понимается анализ текстов программ на соблюдение требований стандартов и принятых руководящих документов технологии программирования.

Эффективность такой проверки заключается в том, что привлекаемые эксперты пытаются взглянуть на проблему "со стороны" и подвергают ее всестороннему критическому анализу.

Эти приемы позволяют на более ранних этапах проектирования обнаружить ошибки или дефекты путем многократного просмотра исходных кодов. Символьное тестирование применяться для проверки отдельных участков программы на входных значения – символах.

Кроме того, разрабатывается множество новых способов автоматизации символьного выполнения программ. Например, автоматизированное средство статического контроля для языково–ориентированной разработки, ряд инструментов автоматизации доказательства корректности. С целью проверки спецификаций параллельных вычислений систем используется автоматизированный аппарат сетей Петри.

7.2.2. Динамические методы тестирования

Динамические методы используются в процессе выполнения программ. Они базируются на графе, который связывает причины ошибок с ожидаемыми реакциями на эти ошибки. В процессе тестирования накапливается информация об ошибках, которая используется при оценке надежности и качества ПС.

Динамическое тестирование ориентировано на проверку корректности ПС на множестве тестов, прогоняемых по ПС, в целях проверки и сбора данных на этапах ЖЦ и проведения измерения отдельных элементов тестирования для оценки характеристик качества, указанных в требованиях посредством выполнения системы на ЭВМ. Оно основывается на систематических, статистических, (вероятностных) и имитационных методах.

Дадим краткую характеристику этим методам.

Систематические методы тестирования делятся на методы, в которых программы рассматриваются как "черный ящик" (используется информация о решаемой задаче), и методы, в которых программа рассматривается как " белый ящик" (используется информация о структуре программы). Этот вид называют тестированием с управлением по данным или управлением по входу–выходу. Цель – выяснение обстоятельств, при которых поведение программы не соответствует ее спецификации. При этом количество обнаруженных ошибок в программе является критерием качества входного тестирования.

Целью динамического тестирования программ по принципу «черного ящика» является выявление одним тестом максимального числа ошибок с использованием небольшого подмножества возможных входных данных.

Методы «черного ящика» обеспечивают:
– эквивалентное разбиение;
– анализ граничных значений;
– применение функциональных диаграмм, которые в объединении с реверсивным анализом дают достаточно полную информацию о функционировании тестируемой программы.

Эквивалентное разбиение состоит в разбиении входной области данных программы на конечное число классов эквивалентности так, чтобы каждый тест, являющийся представителем некоторого класса, был эквивалентен любому другому тесту этого класса.

Классы эквивалентности выделяются путем перебора входных условий и разбиения их на две или более групп. При этом различают два типа классов эквивалентности: правильные, задающие входные данные для программы, и неправильные, основанные на задании ошибочных входных значений.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа: выделение классов эквивалентности и построение тестов. При построении тестов, основанных на выборе входных данных, проводится символическое выполнение программы.

Итак, методы тестирования по принципу «черного ящика» используются для тестирования функций, реализованных в программе, путем проверки несоответствия между реальным поведением функций и ожидаемым поведением с учетом спецификаций требований. Во время подготовки к этому тестированию строятся: таблицы условий, причинно–следственные графы и области разбивки. Кроме того, подготавливаются тестовые наборы, учитывающие параметры и условия среды, которые влияют на поведение функций. Для каждого условия определяется множество значений и ограничений предикатов, которые тестируются.

Метод «белого ящика» позволяет исследовать внутреннюю структуру программы, причем обнаружение всех ошибок в программе является критерием исчерпывающего тестирования маршрутов ее потоков (графа) передач управления, среди которых рассматриваются:

(а) критерий покрытия операторов – набор тестов в совокупности должен обеспечить прохождение каждого оператора не менее одного раза;

(б) критерий тестирования ветвей (известный как покрытие решений или покрытие переходов) – набор тестов в совокупности должен обеспечить прохождение каждой ветви (каждого выхода оператора) по крайней мере один раз.

Критерий (б) соответствует простому структурному тесту и наиболее распространен на практике. Для удовлетворения этого критерия необходимо построить систему путей, содержащую все ветви программы. Нахождение такого оптимального покрытия в некоторых случаях осуществляется просто, а в других является более сложной задачей.

Тестирование по принципу «белого ящика» ориентировано на проверку прохождения всех путей программ посредством применения путевого и имитационного тестирования

.

Путевое тестирование применяется на уровне модулей и графовой модели программы путем выбора тестовых ситуаций, подготовки данных и включает тестирование:
– операторов, которые должны быть выполнены хотя бы один раз, без учета ошибок, которые могут остаться в программе из–за большого количества логических путей и необходимости прохождения подмножеств этих путей;
– путей по заданному графу потоков управления для выявления разных маршрутов передачи управления с помощью путевых предикатов, для вычисления которого создается набор тестовых данных, гарантирующих прохождение всех путей. Однако, все пути выполнить невозможно, поэтому остаются и невыявленные ошибки, которые могут проявиться в процессе эксплуатации;
– блоков, разделяющих программы на отдельные части–блоки, которые выполняются один раз или многократно при нахождении путей в программе, включающих совокупность блоков реализации одной функции либо нахождения входного множества данных, которое будет использоваться для выполнения указанного пути.

«Белый ящик» базируется на структуре программы, а «черный ящик», когда о структуре ничего неизвестно. Для выполнения тестирования с помощью этих «ящиков» известными считаются выполняемые функции, входы (входные данные) и выходы (выходные данные), а также логика обработки, представленные в документации.

7.2.3. Функциональное тестирование

Целью функционального тестирования является обнаружение несоответствий между реальным поведением реализованных функций и ожидаемым поведением в соответствии со спецификацией и исходными требованиями. Функциональные тесты должны охватывать все реализованные функции с учетом наиболее вероятных типов ошибок. Тестовые сценарии, объединяющие отдельные тесты, ориентированы на проверку качества решения функциональных задач.

Функциональные тесты создаются по внешним спецификациям функций, проектной информации и по тексту на ЯП, относятся к функциональным его характеристикам и применяются на этапе комплексного тестирования и испытаний для определения полноты реализации функциональных задач и их соответствия исходным требованиям.

В задачи функционального тестирования входят:
– идентификация множества функциональных требований;

– идентификация внешних функций и построение последовательностей функций в соответствии с их использованием в ПС;
– идентификация множества входных данных каждой функции и определение областей их изменения;
– построение тестовых наборов и сценариев тестирования функций;
– выявление и представление всех функциональных требований с помощью тестовых наборов и проведение тестирования ошибок в программе и при взаимодействии со средой.

Тесты, создаваемые по проектной информации, связаны со структурами данных, алгоритмами, интерфейсами между отдельными компонентами и применяются для тестирования отдельных компонентов и интерфейсов между ними. Основная цель – обеспечение полноты и согласованности реализованных х функций и интерфейсов между ними.

Комбинированный метод "черного ящика" и "прозрачного ящика" основан на разбиении входной области функции на подобласти обнаружения ошибок. Подобласть содержит однородные элементы, которые все обрабатываются корректно либо некорректно. Для тестирования подобласти производится выполнение программы на одном из элементов этой области.

Предпосылками функционального тестирования являются:
– корректное формирование требований и ограничений к качеству ПО;
– корректное описание модели функционирования ПО в среде его эксплуатации заказчиком;
– адекватность модели ПО заданному классу.

7. 3. Организационные аспекты процесса тестирования

Под организацией проведения тестирования понимается::

– выделение объектов тестирования,
– проведение классификации ошибок для рассматриваемого класса тестируемых программ,
– подготовка тестов, их выполнение и поиск разного рода ошибок и дтказов в компонентах и в системе в целом;
– служба проведения и управление процессом тестирования.

Объектами тестирования могут быть компоненты, групп компонентов, подсистема и система. Для каждого из них формируется стратегия проведения тестирования. Если объект тестирования относится к белому или черному «ящикам», состав компонентов которого неизвестный, то тестирование проводится посредством вводом в него входных тестовых данных для получения выходных данных. Стратегическая цель тестирования состоит в том, чтобы убедиться, что каждый рассматриваемый вводной набор данных соответствует ожидаемым выходным выходных данным. При таком подходе к тестированию не требуется знания внутренней структуры и логики объекта тестирования.

Проектировщик тестов должен заглянуть внутрь «черного ящика» и исследовать детали процессов обработки данных, вопросы обеспечения защиты и восстановления данных, а также интерфейсы с другими программами и системами. Это способствует подготовке тестовых данных для проведения тестирования.

Для некоторых типов объектов группа тестирования не может сгенерировать представительное множество тестовых наборов, которые демонстрировали бы функциональную правильность работы компоненты при всех их возможных наборах тестах.

Поэтому предпочтительным является метод «белого ящика», при котором можно использовать структуру объекта для организации тестирования по различным ветвям. Например, можно выполнить тестовые наборы, которые проходят через все операторы или все контрольные точки компоненты для того, чтобы убедиться в правильности их работы.

Классификация ошибок. Международный стандарт ANSI/IEEE–729–83 разделяет все ошибки в разработке программ на следующие

Ошибка (error) – состояние программы, при котором выдается неправильные результаты, причиной которых являются изъяны (flaw) в операторах программы или в технологическом процессе ее разработки, что приводит к неправильной интерпретации исходной информации, а следовательно и к неверному решению.

Дефект (fault) в программе является следствием ошибок разработчика на любом из этапов разработки и может содержаться в исходных или проектных спецификациях, текстах кодов программ, эксплуатационной документация и т.п. Дефект обнаруживается в процессе выполнения программы.

Отказ (failure)– это отклонение программы от функционирования или невозможность программы выполнять функции, определенные требованиями и ограничениями и рассматривается как событие, способствующее переходу программы в неработоспособное состояние из–за ошибок, скрытых в ней дефектов или сбоев в среде функционирования.

Отказ может быть результатом следующих причин:
– ошибочная спецификация или пропущенное требование, т.е. спецификация точно не отражает того, что предполагал пользователь;
– спецификация может содержать требование, которое невозможно выполнить на данной аппаратуре и программном обеспечении;
– проект программы может содержать ошибки (например, база данных спроектирована без защиты от несанкционированного доступа пользователя, а требуется защита);
– программа может быть неправильной, т.е. она выполняет несвойственный алгоритм или он сделан не полностью.

Таким образом, отказы как правило, являются результатами одной или более ошибок в программе, а также наличия разного рода дефектов.

Ошибки на этапах ЖЦ тестирования. Приведенные типы ошибок распределяются я по этапам ЖЦ и им соответствуют такие источники их возникновения:

– непреднамеренное отклонение разработчиков от рабочих стандартов или планов реализации;
– спецификации функциональных и интерфейсных требований выполнены без соблюдения стандартов разработки и т.п., что приводит к нарушению функционирования программ;

– организации процесса разработки – несовершенна или недостаточное управление руководителем проекта ресурсами (человеческими, техническими, программными и т.д.) и вопросами тестирования и интеграции элементов проекта.

Рассмотрим этапы тестирования, определенные в соотвествии с рекомендациями стандарта ISO/IEC 12207, и приведем типы ошибок, которые обнаруживаются на каждом из них.

Этап разработки требований. При определении исходной концепции системы и определении исходных требований заказчика к системе возникают ошибки аналитиков при спецификации верхнего уровня системы и построении концептуальной модели предметной области.

Характерными ошибками этого этапа являются:
– неадекватность описания спецификациями требований конечных пользователей;
– некорректность спецификации взаимодействия ПО со средой функционирования или с пользователями;
– несоответствие требований заказчика к отдельным и общим свойствам ПО;
– некорректность описания функциональных характеристик;
– необеспеченность инструментальными средствами поддержки всех аспектов реализации требований заказчика и др.

Этап проектирования. Ошибки при проектировании компонентов могут возникать при описании алгоритмов, логики управления, структур данных, интерфейсов, логики моделирования потоков данных, форматов ввода–вывода и др. В основе этих ошибок лежат дефекты спецификаций аналитиков и ошибок проектировщиков. К ним относятся ошибки, связанные с :
– определением интерфейса пользователя со средой;
– описанием функций (неадекватность целей и задач компонентов, которые обнаруживаются при проверке комплекса компонентов);
– определением процесса обработки информации и взаимодействия между процессами (результат некорректного определения взаимосвязей компонентов и процессов);
– некорректным заданием данных и их структур при описании отдельных компонентов и ПС в целом;
– некорректным описанием алгоритмов модулей;;
– определением условий возникновения возможных ошибок в программе;
– нарушением принятых для проекта стандартов и технологий.

Этап кодирования. На данном этапе возникают ошибки, которые являются результатом дефектов проектирования, ошибок программистов и менеджеров процесса разработки и отладки. Причиной ошибок являются:

– безконтрольность в допустимости значений входных параметров, индексов массивов, параметров циклов, выходных результатов, деления на 0 и др.;
– неправильная обработка нерегулярных ситуаций при анализе кодов возврата от вызываемых подпрограмм, функций и др.;
– нарушение стандартов кодирования (плохие комментарии, нерациональное выделение модулей и компонент и др.);
– использование одного имени для обозначения разных объектов или разных имен для обозначения одного объекта, плохая мнемоника имен;
– несогласованное внесение изменений в программу разными разработчиками и др.

Этап тестирования. На этом этапе ошибки допускаются тестировщиками, а также программистами при выполнении технологии сборки и тестирования, выбора тестовых наборов и сценариев тестирования и др. Отказы в программном обеспечении, вызванные такого рода ошибками, должны выявляться, устраняться и не отражаться на статистике ошибок компонент и программного обеспечения в целом.

Этап сопровождения. На этапе сопровождения причиной ошибок являются недоработки и дефекты эксплуатационной документации, малые показатели модифицируемости и удобочитаемости, а также некомпетентность лиц, ответственных за сопровождение и/или усовершенствование ПО. В зависимости от сущности вносимых изменений на этом этапе могут возникать практически любые ошибки, аналогичные ранее перечисленным ошибкам на предыдущих этапах.

Все ошибки, которые возникают в программах, принято подразделять на следующие классы [12, 23]:
– логические и функциональные ошибки;
– ошибки вычислений и времени выполнения;
– ошибки ввода–вывода и манипулирования данными;
– ошибки интерфейсов;
– ошибки объема данных и др.

Логические ошибки являются причиной нарушения логики алгоритма, внутренней несогласованности переменных и операторов, а также правил программирования. Функциональные ошибки являются следствием неправильно определенных функций, нарушения порядка их применения или отсутствия полноты их реализации и т.д.

Ошибки вычислений возникают по причине неточности исходных данных и реализованных формул, погрешностей методов, неправильного применения операций вычислений или операндов. Ошибки времени выполнения связаны с не обеспечением требуемой скорости обработки запросов или времени восстановления программы.

Ошибки ввода–вывода и манипулирования данными являются следствием некачественной подготовки данных для выполнения программы, сбоев при занесении их в базах данных или при выборке из нее.

Ошибки интерфейса относятся к ошибкам взаимосвязи отдельных элементов друг с другом, что проявляется при передаче данных между ними, а также при взаимодействии со средой функционирования.

Ошибки объема относятся к данным и являются следствием того, что реализованные методы доступа и размеры баз данных не удовлетворяют объемам информации системы или интенсивности ее обработки.

Приведенные основные классы ошибок свойственны разным типам компонентов ПО и проявляются они в программах по–разному. Так, при работе с БД возникают ошибки представления и манипулирования данными, логические ошибки в задании прикладных процедур обработки данных и др. В программах вычислительного характера преобладают ошибки вычислений, а в программах управления и обработки – логические и функциональные ошибки. В ПО, состоящем из множества разноплановых программ реализации разных функций, могут содержаться ошибки разных типов и т.д. Ошибки интерфейсов и нарушение объема характерны для любого ПО.

Анализ типов ошибок в программах является необходимым условием создания планов тестирования и методов тестирования для обеспечения правильности ПО.

На современном этапе развития средств поддержки разработки ПО (CASE–технологии, объектно–ориентированные методы и средства проектирования моделей и программ) проводится такое проектирование, при котором ПО защищается от наиболее типичных ошибок и тем самым предотвращается появление программных дефектов.

Связь ошибки с отказом. Наличие ошибки в программе, как правило, приводит к отказу ПО при его функционировании. Для анализа причинно–следственных связей "ошибка–отказ" существуют следующие действия:

– идентификация изъянов в технологиях проектирования и программирования;
– взаимосвязь изъянов процесса проектирования и допускаемых человеком ошибок;
– классификация отказов, изъянов и возможных ошибок, а также дефектов на каждом этапе разработки;
– сопоставление ошибок человека, допускаемых на определенном этапе разработки, и дефектов в объекте, как следствий ошибок спецификации проекта, моделей программ и т.д.);
– проверка и защита от ошибок на всех этапах ЖЦ, а также обнаружение дефектов на каждом этапе разработки;
– сопоставление дефектов и отказов в ПО для разработки системы взаимосвязей и методики локализации, сбора и анализа информации об отказах и дефектах;
– разработка подходов к документированию процессов и испытания ПО.

Конечная цель причинно–следственных связей "ошибка–отказ" заключается в определении методов и средств тестирования и обнаружения ошибок определенных классов, а также критериев завершения тестирования на множестве наборов данных; в определении путей совершенствования организации процесса разработки, тестирования и сопровождения ПО.

Приведем следующую классификацию типов отказов:

– аппаратный, при котором общесистемное ПО не работоспособно;
– информационный, вызванный ошибками во входных данных и передаче данных по каналам связи, сбое устройств ввода (следствие аппаратных отказов);
– эр готический, вызванный ошибками оператора при его взаимодействии с машиной (этот отказ является вторичным отказам и может привести к информационному или функциональному отказам);
– программный при наличии ошибок в компонентах и др.

Некоторые ошибки могут быть следствием недоработок при определении требований, проекта, генерации выходного кода или документации, С другой стороны, они порождаются в процессе разработки программы или при разработке интерфейсов отдельных элементов программы (нарушение порядка параметров, меньше или больше параметров и т.п.). На рис.7.2 показаны подобные случаи ошибочных ситуаций и причин их появления в каждой разработке.

Источники ошибок. Ошибки могут быть порождены в процессе разработки проекта, компонент, кода и документации. Как правило, они обнаруживаются при выполнении или сопровождении программного обеспечения в самых неожиданных и разных ее точках.

Некоторые ошибки в программе могут быть следствием недоработок при определении требований, проекта, генерации кода или документации. С другой стороны, ошибки порождаются в процессе разработки программы или интерфейсов ее элементов (например, при нарушении порядка задания параметров связи – меньше или больше, чем требуется и т.п.).

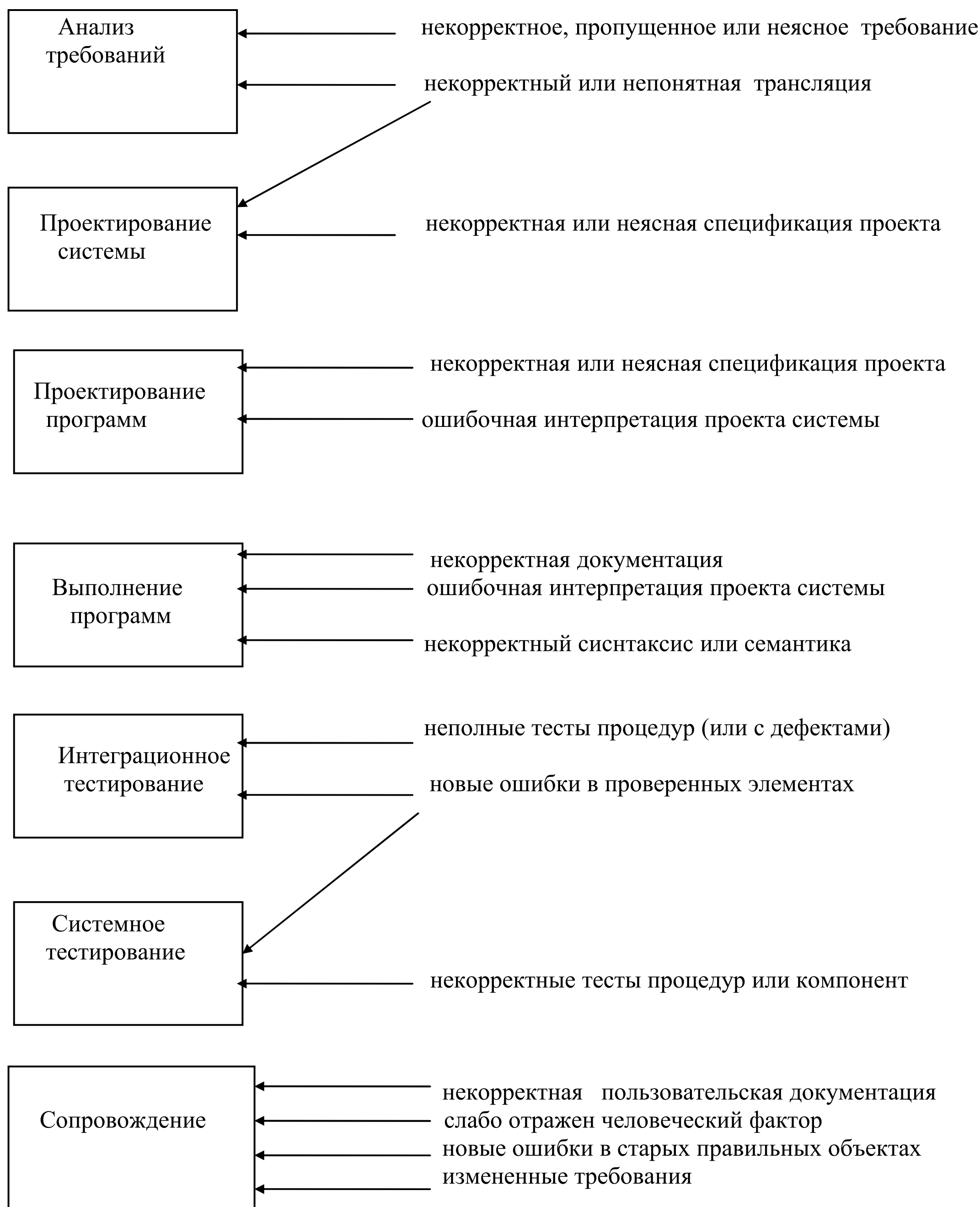

Рис. 7.2. Виды ошибок на этапах ЖЦ

Причиной появления ошибок зачастую является: непонимание требований заказчика; неточная спецификация требований в документах проекта и др. Это приводит к тому, что реализуются некоторые функции системы, которые будут работать не так, как предлагает заказчик. В связи с этим проводится совместное обсуждение непонимания некоторых деталей требований для их уточнения заказчика и разработчика.

Команда разработчиков системы может также изменить синтаксис и семантику описания системы. Однако некоторые ошибки могут быть не обнаружены (например, неправильно заданы индексы или значения переменных этих операторов).

Исходя из того, что каждая организация по разработке ПО (особенно общесистемного назначения), сталкивается с проблемами нахождения ошибок, ей приходится классифицировать типы обнаруживаемых ошибок и определять свое отношение к этим ошибкам.

На основе многолетней деятельности в области создания ПО разные фирмы создали свою классификацию ошибок, основанную выявлении причин их появления в процессе разработки, функциях, в областях появлении ошибок в ПО. Известно много различных подходов к классификации ошибок, рассмотрим некоторые из них.

Фирма IBM разработала подход к классификации ошибок, называемый ортогональной классификацией дефектов [23]. Подход предусматривает разбиение ошибок по категориям с ответственностью разработчиков за них.

Схема классификации является продукто– и организационно–независимой и может применятся ко всем стадиям разработки ПО разного назначения. Табл. 7.1 дает список ошибок согласно данной классификации. В ней разработчику предоставляется возможность идентифицировать не только типы ошибок, но и места, где пропущены или совершенны ошибки, а также неинициализированная переменная или инициализированной переменной присвоено неправильное значение.

Ортогональная классификация дефектов IBM   Таблица 7.1

Контекст ошибки	Классификация дефектов
Функция	Ошибки интерфейсов конечных пользователей ПО, вызванные аппаратурой или связаны с глобальными структурами данных
Интерфейс	Ошибки во взаимодействии с другими компонентами, в вызовах, макросах, управляющих блоках или в списке параметров
Логика	Ошибки в программной логике, неохваченной валидацией, а также в использовании значений переменных
Присваивание	Ошибки в структуре данных или в инициализации переменных отдельных частей программы
Зацикливание	Ошибки, вызванные ресурсом времени, реальным временем или разделением времени
Среда	Ошибки в репозитории, в управлении изменениями или в контролируемых версиях проекта
Алгоритм	Ошибки, связанные с обеспечением эффективности, корректности алгоритмов или структур данных системы
Документация	Ошибки в записях документов сопровождения или в публикациях

Ортогональность схемы классификации заключается в том, что любой ее термин принадлежит точно одной категории. Другими словами, прослеживаемая ошибка в системе должна находиться в одном из классов, это дает возможность двум разработчикам классифицировать ошибки одинаковым способом.

Фирма Hewlett–Packard использовала классификацию Буча, установив процентное соотношение ошибок, обнаруживаемых в ПО на разных стадиях разработки (рис. 7.3) [14].

Рис.7.3 Процентное соотношение ошибок при разработке ПО

7.3.1. Организация подготовки тестов

Для проверки правильности программ специально разрабатываются тесты и тестовые данные. Под *тестом* понимается некоторая программа, предназначенная для проверки работоспособности другой программы и обнаружения в них ошибочных ситуаций. Тестовую проверку можно провести также путем введения в проверяемую программу отладочных операторов, которые будут сигнализировать о ходе ее выполнения и получения результатов.

Тестовые данные служат для проверки работы системы и готовятся разным способом: генератором тестовых данных, проектной группой на основе внемашинных документов или имеющихся файлов, пользователем по спецификациям требований и т.д. Очень часто разрабатываются специальные формы входных документов, в которых отображается процесс выполнения программы с помощью тестовых данных.

Создаются тесты, проверяющие:
– полноту функций;
– согласованность интерфейсов;
– корректность выполнения функций и правильность функционирования системы в заданных условиях;
– надежность выполнения системы;
– защиту от сбоев аппаратуры и не выявленных ошибок и др.

Тестовые данные готовятся как для проверки отдельных программных элементов, так и для групп программ или комплексов на разных стадиях процесса разработки. На рис. 7.5 приведена классификация тестов проверки по объектам тестирования на основных стадиях разработки.

Многие типы тестов готовятся заказчиком для проверки работы программной системы. Структура и содержание тестов зависят от вида тестируемого элемента, которым может быть: модуль, компонента, группа компонент, подсистема или система. Некоторые тесты зависят от цели и необходимости знать: работает ли система в соответствии с ее проектом, удовлетворены ли требования и участвует ли заказчик в проверке работы тестов и т.п.

Рис.7.5 Классификация тестов проверки

В зависимости от задач, которые ставятся перед тестированием программ, эта команда составляет тесты проверки промежуточных результатов проектирования элементов системы на стадиях ЖЦ, а также создает тесты испытаний окончательного кода системы.

Тесты интегрированной системы. Тесты для проверки отдельных элементов системы и тесты интегрированной системы имеют общие и отличительные черты. Так, на рис.7.6 в качестве примера приведены шаги стадии интеграции системы из готовых оттестированных элементов, заданы связи между разными шагами тестирования интегрируемой программной системы.

Для каждого шага команда тестировщиков готовит тесты и наборы данных, используемые для проверки разных состояний интегрированной системы и соответствий с принятыми ограничениями на каждом шаге и требованиями к элементу тестирования.

Первый шаг этого процесса – интеграция элементов, которая основывается на интеграционных тестах, создаваемых по спецификациям проекта. Рассмотрим этот шаг более подробно на примере схемы (рис. 7.4) интеграции отдельных элементов.

Каждая компонента этой схемы тестируется отдельно от других компонент с помощью тестов, включающих наборы данных и сценарии, составленные в соответствии с их типами и функциями, специфицированными в проекте системы. Тестирование проводится в контрольной среде на предопределенном множестве тестовых данных и операциях, производимыми над ними.

Тесты обеспечивают проверку внутренней структуры, логики и граничных условий выполнения каждой компоненты.

Согласно приведенной схеме, вначале тестируются компоненты A, B, D независимо друг от друга и каждая с отдельным тестом. После их проверки выполняется следующий шаг – проверка интерфейсов для последующей их интеграции, суть которой заключается в анализе выполнения операторов вызова A → E, B → C, D → G, на нижних уровнях графа: компоненты – E, C, G. При этом предполагается, что указанные вызываемые компоненты так же должны быть

отлажены отдельно. Аналогично проверяются все обращения к компоненте F, являющейся связывающим звеном с вышележащими элементами.

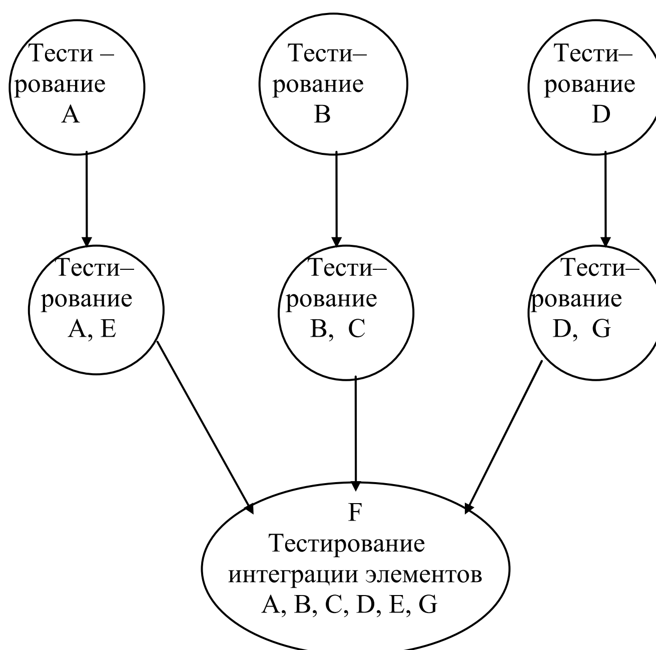

Рис.7.4. Интеграционное тестирование компонент

При этом могут возникать ошибки, в случае неправильного задания параметров в операторах вызова или при вычислениях процедур или функций. Возникающие ошибки в связях устраняются, а затем повторно проверяется связь с компонентой F в виде троек: компонента – интерфейс – компонента.

Следующим шагом тестирования системы является проверка функционирования системы с помощью тестов проверки функций и требований к ним. После проверки системы на функциональных тестах идет проверка системы на исполнительных и испытательных тестах, подготовленных согласно требований к ПО, аппаратуре и выполняемым функциям. Испытательному тесту предшествует верификация и валидация ПО.

Тест испытаний системы в соответствии с требованиями заказчика проверяется в реальной среде, в которой система будет в дальнейшем использоваться.

7.3.2. Команда тестировщиков

За функциональные и исполнительные тесты несут ответственность разработчики, а заказчик больше влияет на составление тестов испытаний и инсталляции системы.

Для этих целей, как правило, создается служба проверяющих ПС в виде команды тестировщиков, которая не зависит от штата разработчиков ПС. Ннекоторые члены этой команды являются опытными или даже профессионалами в этой области. К ним относятся аналитики, программисты, инженеры–тестировщики, которые посвящают все время проблемам тестирования систем. Они имеют дело не только со спецификациями, но и с методами и средствами тестирования, организуют создание и выполнение тестов на машине. С самого начала создания проекта тестировщики включают в процесс планы составления тестовых наборов и сценариев, а также графиков выполнения тестов.

Профессиональные тестировщики работают совместно с группой управления конфигурацией, чтобы обеспечить их документацией и другими механизмами для связи между собой тестов с требованиями проекта, конфигурацией и кодом. Они разрабатывают методы и процедуры тестирования. В эту команду включаются дополнительные люди, которые знакомы с требованиями системы или с подходами к их разработке. Аналитиков включают в качестве членов команды, так как они понимают проблемы определения спецификаций заказчиков.

Многие специалисты сравнивают тестирование системы с созданием новой системы, в которой аналитики отражают потребности и цели заказчика, работая совместно с проектировщиками и добиваясь реализации идей и принципов работы системы.
Проектировщики системы сообщают команде тестировщиков проектные цели, чтобы они знали декомпозицию системы на подсистемы и функции, а также принципы их работы. После проектирования тестов и тестовых покрытий команда тестировщиков подключает проектантов для анализа возможностей системы.

Так как тесты и тестовые сценарии являются прямым отражением требований и проекта в целом, перспективы управления конфигурацией системы определяются именно этой командой. Встречаемые ошибки в программе и изменения в системе отражаются в документации, требованиях, проекте, а также в описаниях входных и выходных данных или в других разрабатываемых артефактах. Вносимые изменения в процессе разработки приводят к модификации тестовых сценариев или в большей части к изменению планов тестирования. Специалисты по управлению конфигурацией учитывают эти изменения и координируют составление тестов.

В команду тестировщиков входят также пользователи. Они оценивают получаемые результаты, удобство использования, а также высказывают свое мнение о принципах работы системы.

Уполномоченные заказчика планируют работы для тех, кто будет использовать и сопровождать систему. При этом они могут привнести некоторые изменения в проект из–за неполноты заданных требований и сформулировать системные требования для проведения верификации системы и принятия решений о ее готовности и полезности.

План тестирования. Для проведения тестирования создается план (Test Plan), в котором описываются стратегии, ресурсы и график тестирования отдельных компонент и системы в целом. В плане отмечаются работы для разных членов команды, которые выполняют определенные роли в этом процессе. План включает также места теста в каждом процессе, степень покрытия программы тестами и процент тестов, которые выполняются со специальными результатами.

Тестовые инженеры создают множество тестовых сценариев (Test Cases), каждый из которых проверяет результат взаимодействия между актером и системой на основе определенных пред– и пост–условий использования таких сценариев. Сценарии в основном относятся к тестированию по типу белого «ящика» и ориентированы на проверку структуры и операций интеграции компонентов системы.

Для проведения тестирования тестовые инженеры предлагают процедуры тестирования (Test Procedures), включающие валидацию объектов и верификацию тестовых сценариев в соответствии с планом графикам. Оценка тестов (Test Evaluation) заключается в оценке результатов тестирования и степени покрытия программ

сценариями и статуса полученных ошибок. На рис. 7.5. приведен круг обязанностей тестового инженера.

Тестировщик интегрированной системы проводит тестирование интерфейсов и оценку результатов выполнения соответствующих тестов. И наконец, тестовик системы является ответственным за выполнение системных тестов и проведенного тестирования предыдущими членами команды. При выполнении системных тестов, как правило, находятся дефекты, являющиеся результатом глубоко скрытых погрешностей в программах, обнаруживаемых при длительной прогонке системы на тестовых данных и сценариях.

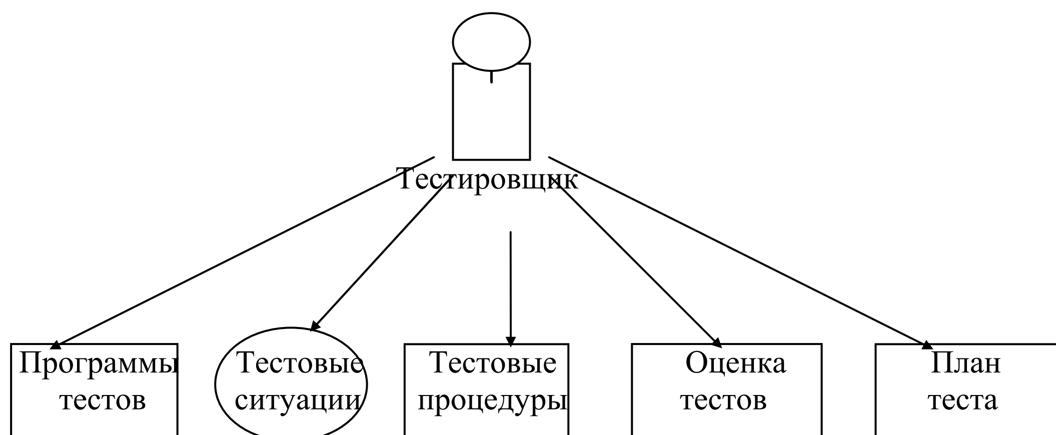

Рис.7.5. Ответственности инженера– тестировщика

7.3.3. Организация процесса тестирования

Все способы тестирования ПС объединяются базой данных, где помещаются результаты тестирования системы. В ней содержатся все компоненты, тестовые контрольные данные, результаты тестирования и информация о документировании процесса тестирования.

База проекта поддерживается специальными инструментальными средствами типа CASE, которые обеспечивают ведение анализа ПрО, сборку данных об их объектах, потоках данных и тому подобное. База проекта хранит также начальные и эталонные данные, которые используются для сопоставления данных, накопленных в базе с данными, которые получены при тестировании.

При тестировании выполняются разные виды расчета характеристик этого процесса и способы планирования и управления:

1. Расчет продолжительности выполнения функций путем сбора средних значений о скорости выполнения операторов без выполнения программы на машине. Выявляются компоненты, которые требуют большого времени выполнения в реальной среде.

2. Управления выполнением состоит в организации подбора тестов проверки, их выполнении, селекции результатов тестирования и проведении сопоставления с эталонными значениями. Результаты процесса отображаются на дисплеи, например, в графической форме (пути прохождения по графу программы), в виде

последовательности диаграмм UML, а также в виде информации об отказах и ошибках или конкретных значениях исходных параметров программы. Эти данные анализируются разработчиками для формулирования выводов о направлениях дальнейшей проверки правильности программы или их завершение.

3. Планирование тестирования предназначено для распределения сроков работ по тестированию, распределения тестировщиков по отдельным видам работ и составления ими тестов проверки работы системы. Практически определяется стратегия и пути тестирования. В диалоге запрашиваются реальные значения процесса выполнения или выдачи структуры о разветвления вершин графа и параметров циклов. Проверенные циклы, как правило, изымаются из путей выполнения программы. При планировании путей выполнения создаются соответствующие тесты, критерии и входные значения.

4. Документирование результатов тестирования в соответствии с действующим стандартом ANSI/IEEE 829, включает описание:
– задач, назначение и содержание ПС, а также описание функций соответственно требованиям заказчика;
– технологии разработки системы;
– планов тестирования различных объектов, необходимых ресурсов, соответствующих специалистов для проведения тестирования и технологических способов;
– тестов, контрольных примеров, критериев и ограничений оценки результатов программного продукта, а также процесса тестирования;
– учета процесса тестирования, составление отчетов об аномальных событиях, отказах и дефектах в итоговом документе системы.

Контрольные вопросы и задания

1. Назовите формальные методы проверки правильности программ.
2. Какие процессы проверки зафиксированы в стандарте?
3. Какие объекты входят в доказательство правильности программ?
4. Назовите основные методы доказательства корректности программ и базис этих методов.
5. Определите типы логических операций, используемых при логическом доказательстве корректности программ.
6. В чем состоит отличие техники формального доказательства от символьного выполнения программ?
7. Сформулируйте основные задачи верификации и валидации программ.
8. В чем отличие верификации и валидации?
9. Определите процесс тестирования.
10. Назовите методы тестирования.
11. Объясните значения терминов «черный ящик», «белый ящик».
12. Назовите объекты тестирования и подходы к их тестированию.
13. Какая существует классификация типов ошибок в программах?
14. Определите основные этапы ЖЦ тестирования ПО.
15. Наведите классификацию тестов для проверки ПО.
16. Какие задачи выполняет группа тестировщиков?
17. Какая организация работ проводится для проведения тестирования.

Литература к теме 7

1. *Майерс Г.* Искусство тестирования программ. – Пер.с англ.М: Финансы и статистика. – 1982. – 176 с.
2. *Липаев В.В.* Отладка сложных программ.–М.: Энергоатомиздат, 1993.–296с.

3. *Липаев В.В.* Тестирование программ.–М: Радио и связь,–1986.–295с.
4. *Канер С., Фолк Д., Нгуен Е.К.* Тестирование программного обеспечения: Пер с англ. – К.: DiaSoft. – 2000. – 544 с.
5. *Weyuker E.J., Ostrand T.J.* Theories of program testing and the application of revealing subdomains // IEEE Trans.Soft.Eng. – 1980, –V.6, –№. 3, – P. 236–246.
6. *Software unit* test coverage and adequacy. / Zhu H., Hall P. A. // ACM Computing Surveys, 29, –№ 4, Dec. 1997. –P. 336–427.
7. *Коул Дж., Горем Т. и др.* Принципы тестирования ПО //Открытые системы. – 1998.–№2. www.osp.ru/os/1998/02/60.htm
8. *Burstall R.M.* Program proving as hand simulation with a little induction. – Proc. IFIP Congress 74, North–Holland, 1974. –P.80 – 89.
9. *Dijkstra T.W.* Finding the Correctness proof of a concurrent program. – Proc.Konf. Nederland Acad.Wetenach, 1978. – 81. – N2. – p.207– 215.
10. *Clint M., Hoare C.A.R.* Program proving: jumps and functions. — Acta Informatikee, 1972. — 1. — N3. — P.214—224.
11. *Pfleeger S.L.* Software Engineering. Theory and Practice. – Prentice Hall, 1998. – 576p.
12. *Grossman D., McCobe C.* Perfomance Testing a Large Finance Application. – IEEE Software. – 1996. – Sept. – P.50 –60.
13. *Y.Wang, J.King, J.Kourt, M.Ross, S.Staples*. On testable odject–oriented programming// Software Engineering Notes, volume 22, N4. –1997.– pp.84–90
14. *Perry D.E. and Kaiser C.E.* Adequate testing and object–oriented programming // Journal of Object–Oriented Programming, January /Febrary. –1990. – p.13–19.
15. ANSI / IEEE Std. 10122–1986. Standard for Software Verification and Validation Plans // IEEE . – New York . – 1986. – 61p.
16. *Dolores R. Wallase M. Ippolito, b. Cuthill.* Reference Information for the Software Verification and Validation Process // NIST Special Publication . – 1996 . – 500–234. – 80p.
17. *Herhart S.L.* Program Verification in the 90's.// Proc.Conf. on Computing in the 1980's, 1978.– P.80–89.
18. ISO/IEC 12207: 1995.– Information technology - Software life cycle processes) Информационные технологии - Процессы жизненного цикла программного обеспечения..
19. *CASE–93.* Proceeding Sixth Intern.//Workshopon Computer Aided Software Engineering.– Singapure. –1993. – July 19–23.–418p.
20. *Jacobson J.* Object–oriented Software Engineering. – Revised Printing. – Addison–Wesley. – 1995. – 528p.
21. *Коротун Т.М.* Совершенствование процессов тестирования программного обеспечения // Проблемы программирования.–1998.–№3.–С.59–64.
22. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии.– Киев, 2001.– Знання.– 269с.
23. *Koomen, T., and M. Pol.* 1998. Improvement of the test process using TPI. http://www.iquip.nl
24. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных систем. –К.: Наукова думка, 1997. –229с.
25. *Липаев В.В.* Методы обеспечения качества крупномасштабных программных средств. – М.: СИНТЕГ.– 2003.–520 с.
26. *Drabick R.* Growth of maturity in the testing process. International Software Testing Institute 1999. http://www.softtest.org/articles/rdrabick3.htm.
27. *Software Engineering* Body of Knowledge (SWEBOK). // ISO/IEC JTC1/SC7 N2517. Software & System Engineering Secretariat, Canada, 2001. – 220 p.

Тема 8
МЕТОДЫ ИНТЕГРАЦИИ, ПРЕОБРАЗОВАНИЯ И ИЗМЕНЕНИЯ КОМПОНЕНТОВ И ДАННЫХ

По данной теме рассматриваются отдельно четыре метода, связанные с решением проблемы интеграции компонентов и данных, обнаружения ошибок и внесения исправлений в программы и данные:

1. Методы интеграции разработанных компонентов для функций системы и готовых ПИК и системы поддержки процесса интеграции и обеспечения взаимодействия компонентов в разных средах;
2. Методы преобразования типов данных программ, записанных на разных ЯП, для обеспечения их взаимосвязи, а также рекомендации стандарта ISO/IEC 11404-1996 по обеспечению независимости типов данных от современных ЯП;
3. Преобразование данных, хранящихся в разных типах БД в целях замены устаревших БД на новые.
4. Методы внесения изменений в компоненты и в ПС (реинженерия, рефакторинг и реверсная инженерия).

8.1. Методы интеграции (композиции) компонентов

Термин интеграция олицетворяет собой действия по обеспечению способа взаимодействия разных компонент (аппаратных и программных) и представления данных в рамках одной системы или среды. К интеграционным операциям можно отнести: сборку, комплексирование, композицию, взаимодействие и др. Некоторые из этих операций (например, комплексирование) означают объединение компонент по в единое целое (в комплекс, систему, агрегат и др.). Такой подход к решению проблемы интеграции соответствовал периоду бурного развития и существования больших машин (mainframes), которые позволяли объединять программные компоненты в монолитные системы и комплексы больших размеров (до 100–200тыс. команд). Одновременно с этим интеграция коснулась и данных (файлы, БД, интегрированные БД и др.).

Однако с приходом на смену больших машин разных видов малых компьютеров, объединяемых в локальные и глобальные сети проблема интеграции приобрела другой смысл. Понятие монолитной системы заменилось интегрированной, распределенной системой или средой, включающей все необходимые средства для обеспечения единообразного взаимодействия с ними разных пользователей. Появились и в настоящее время функционируют крупные международные распределенные системы (RPC Sun, OSF DCE, COM, SOM, CORBA и др.), которые предоставляют средства интеграции программных компонент в каркасы и конфигурации, основанные на стандартной модели взаимодействия компонент в открытых системах (Open Systems Interconnection – OSI) [1].

Эталонная модель OSI имеет семь уровней, на каждом из них обеспечивается взаимосвязь компонентов. На верхнем уровне модели обеспечивается доступ к служебным программам компьютерной сети (передача данных, почтовая служба, управление сетью). Приложение передает запросы служебным программам и процессам сети через уровень представления данных, которые осуществляют кодирование (перекодирование) данных и представление их в соответствующую для заданной машины форму. Связывающим звеном являются прикладные элементы обслуживания типа ASF (Application Service Elements), а также множество других служебных функций.

Эта модель описывает функции и назначение семи ее уровней, определяет взаимодействие между службами и компонентами на каждом уровне сети. Непосредственной связи между службами не существует, а взаимодействие осуществляется через уровни. Эта модель фактически задает две модели взаимодействия компонентов:
– горизонтальная модель для связи программных процессов на разных компьютерах одного уровня;
– вертикальная модель для взаимодействия компонентов между уровнями.

С прикладной точки зрения основная задача разработчиков заключается в выборе необходимой современной среды функционирования программных компонентов, способов их обращения друг к другу и возможностей преобразования передаваемых данных.

Распределенные системы общего назначения построены согласно модели OSI и предоставляют прикладным приложениям готовый набор ([20]): сервисных операций (управление и хранение объектов, обслуживание очередей и запросов к БД, др.); общих средств обслуживания приложений (администрирование, управление интерфейсами, др.); средств описания и поддержки взаимодействий объектов приложений, инструментариев (автоматизация приложений и БД, генераторы интерфейсов взаимодействия и др.); типовых функциональных компонентов для предметных областей: медицины, финансов, страхования и др.

К распределенным системам, обеспечивающим взаимодействие компонентов относятся:
– ONC SUN, OSF DSE [2], основанные на механизме вызова удаленных процедур;
– DCOM [3] с возможностью связи распределенных объектов и документов;
OMA (Object Managment Architecture) [4] с широким набором средств взаимодействия объектов с помощью брокера с помощью брокера объектных запросов;
– система JAVA [5] , основанная на методе вызова RMI и др.

К основным механизмам взаимодействия компонентов и объектов сетевой среды на уровне внешних языков относятся:
– RPC–язык (Remote Procedure Call) вызова удаленных процедур;
– язык описания интерфейсов (Interface Definition Langugе – IDL),
– механизм посылки запросов RMI в ЯП Java [5–7].

RPC–механизм включает языки высокого и низкого уровня для описания интерфейса взаимодействующих удаленных компонентов. На языке высокого уровня в интерфейсе компонента описывается оператор *RPC* из библиотеки удаленных процедур системы, который инициирует прохождения сообщения по сети и выполнение. На языке низкого уровня можно дать подробное описание в параметрах оператора вызова (тип протокола, размер буфера данных, контрольные функции и др.) всех особенностей прохождения запроса по сети.

Данный механизм обеспечивает взаимосвязь одного процесса с удаленно расположенным от него другим процессом (например, сервером) на другой машине с помощью протоколов UDP и TCP/IP. Связывающим звеном между вызываемым и вызывающим процессом является интерфейс объекта (stub) или посредник клиента при его взаимодействии с сервером сети. Вызывающий объект клиенте обращается к stub–клиента для посылки сообщения stub–сервера в целях выполнения удаленной процедуры.

Механизм посылки запроса в системе CORBA включает оператор вызова удаленного метода/функции объекта, системные средства его поддержки с помощью протоколов IIOP, GIOP, которые выполняет брокер ORB. Оператор запроса и его параметры формально описываются на IDL – языке и размещаются в интерфейсе объекта (stub–клиента) для обращения к серверу через интерфейс stub / skeleton в целях выполнения указанного в сообщении удаленного метода.

Интерфейсы (stub и skeleton) отображаются в ЯП объектов с помощью IDL–генератора. Функции посылки запроса выполняет брокер ORB системы CORBA, которая содержит:
– язык IDL и генератор трансформации описания интерфейса в соответствующий ЯП;
– общий объектный сервис (Common Object Services) для управления событиями, транзакциями, интерфейсами, запросами и др.;
– общие средства (Common Facilities) – электронная почта, телекоммуникация, управление информацией, эмулятор программ и др. для использования любыми группами компонент и приложений.

Брокер ORB является главной парадигмой взаимодействия компонентов в разных приложениях и средах. Он реализован многими фирмами и содержится в системах: COM, SOM, Nextstep и др. Этим обеспечена совместимость программных компонентов друг с другом, сделанных для разных сред, а также взаимодействие между самими брокерами в объединенных сетях.

Вызов удаленного метода RMI реализован в системе JAVA и предназначен для проектирование приложений со стандартным вызовом удаленных компонентов. По своим функциям этот метод аналогичен функциям брокера ORB, обеспечивает передачу сообщений, их синхронизацию и освобождение памяти после выполнения компонентов. Виртуальная машина (virtual machine) работает с byte–кодами компонентов на других ЯП и интерпретирует коды той машины, для которой компонент был скомпилирован. Иными словами, разработан новый подход к проектированию связей и управлений объектами в распределенных системах, которые описываются в языках JAVA и C++.

Таким образом, рассмотренные средства современных распределенных систем – это средства высокого уровня для прикладного программирования компьютерных систем.

8.2. Методы преобразования программ и данных

Программы, расположенные на разных типах компьютеров, взаимодействуют друг с другом через передачу данных, которая предполагает преобразование форматов данных для одной платформы в представление другой платформы, а также преобразование отличающихся типов и сложных структур данных, записанных в разных ЯП.

Преобразования типов данных проводится по трем направлениям.

<u>Первое направление</u> связано с разными форматами представления данных в программах, которые расположены в разных средах или на разных платформах. Процедура преобразования данных из одного формата в другой получило название маршаллинга (marshalling) данных и включает линеризацию сложных структур данных с учетом порядка расположения байтов и стратегии их выравнивания до границ на каждой платформе. Например, в системе CORBA для этих целей используется

стандарт общего формата представления данных – CDR (Common Representation Data) [4].

Второе направление связано с наличием отличий в описании типов данных разных ЯП и необходимостью эквивалентного их преобразования с помощью таких механизмов: удаленный вызов процедур RPC [1, 2] и RMI [4], языка описания интерфейсов Stub в IDL и стандарта, определяющего независимые от языков типы данных» (ISO/IEC 11404–96) [8].

Третье направление связано с заменой одной БД на другую, имеющие отличие в моделях данных (иерархические, сетевые, реляционные) и функционируют в разных средах СУБД [9, 10].

В основе рассмотрения этих трех направлений лежат методы и подходы решения проблемы преобразования данных, а также результаты исследований и разработок, освещенные в работах [1–14].

8.2.1. Парадигма преобразования данных

Под *парадигмой преобразования* данных будем понимать формализмы описания базовых типов и структур данных в современных ЯП, методы преобразования форматов данных (кодирование и декодирование) одного компьютера к соответствующему представлению другого компьютера, методы устранения отличий в представлении типов данных в разных ЯП и методы преобразования данных при замене БД.

Средствам представления данных и их форматов являются:
– стандарты кодировки данных (XDR – eXternal Data Representation, CDR, NDR – Data Representation) и их трансформации;
– ЯП программных компонентов и механизмы обращения друг к другу;
– языки описания интерфейсов компонентов – RPC, IDL и RMI для обеспечения передачи данных между разными компонентами.

К методам трансформации форматов данных относится формальный набор правил кодирования и декодирования (маршаллинг) данных, линеризация сложных структур и расположения данных в передающей и соответственно в принимающей платформе компьютеров.

Механизмами передачи данных являются:
– протоколы передачи данных (TCP/IP, UDP, GIOP и др.) [11];
– классы функций преобразования отличающихся типов и структур данных ЯП и генерации соответствующих новых типов данных [12, 13];
– системные процедуры по обеспечению маршаллинга данных между разными объектами распределенной среды неоднородных компьютеров [7, 12] .

При передаче данных от компонента в одном ЯП компоненту на другом языке может потребоваться устранить отличия в представлении типов данных в этих ЯП с помощью эквивалентного их преобразования. Соответствие типов данных устанавливается с помощью специальных функций, общесистемных средств, либо рекомендаций стандарта, регламентирующего независимые от языков типы данных (ISO/IEC 11404–96).

8.2.2. Формальное описание данных в ЯП и их преобразование

Основными ЯП, используемыми для описания компонентов для распределенной сети, являются С++, Паскаль, JAVA и др.

При обращении разноязыковых компонентов устанавливается взаимно однозначное соответствие между фактическими параметрами $V = \{ v^1, v^2, \ldots v^к \}$ вызывающего компонента и формальными параметрами $F = \{ f^1, f^2, \ldots, f^{к1} \}$ вызываемого компонента, а также строится отображение (mapping) типов данных одного ЯП в соответствующие типы другого ЯП.

В общем случае задача отображения $A: П \rightarrow Ф$ для множеств параметров V и F состоит в построении:

$$П = \{ V^1, V^2, \ldots, V^m \}, \quad Ф = \{ F^1, F^2, \ldots, F^m \}$$

$$\bigcup_{t=1}^{m} V^t = V, \qquad V^t \cap V^{t'} = \varnothing \quad при \ t \le t'$$

$$\bigcup_{t=1}^{m} F^t = F, \qquad F^t \cap F^{t'} = \varnothing \quad при \ t = t'$$

Построение отображения выполняется за два этапа.

1) Построение операций преобразования типов данных $T_\alpha = \{T_\alpha^t\}$ для множества языков $L = \{l_\alpha\}_{\alpha=1,n}$.

2) Построение отображения типов данных для каждой пары взаимодействующих компонентов в ЯП $l_{\alpha 1}$ и $l_{\alpha 2}$ с применением операций селектора S и конструктора C.

Для проведения формализованного преобразования типов данных используется алгебраический подход, при котором каждому типу данных T_α^t ставиться в соответствие алгебраическая система

$$G_\alpha^t = \langle X_\alpha^t, \Omega_\alpha^t \rangle,$$

где t – тип данных, X_α^t – множество значений, которые могут принимать переменные этого типа данных, Ω_α^t – множество операций над этими типами данных.

Для простых типов данных ЯП $t = b$ *(bool), c (char), i (int), r (real)* и сложных типов данных $t =$ *a (array), z (record), u (union), e (enum),* как комбинация простых типов данных, построены две алгебраические системы:

$$\Sigma_1 = \{ G_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r \}$$

$$\Sigma_2 = \{ G_\alpha^a, G_\alpha^z, G_\alpha^u, G_\alpha^e \ldots \} \qquad (1)$$

Каждая из систем определяется на множестве значений типов данных и операций над ними:

$$G_\alpha^t = \langle X_\alpha^t, \Omega_\alpha^t \rangle, \quad где \ t = b, c, i, r, a, z, u, e.$$

Операциям преобразования каждого t типа данных соответствует изоморфное отображение двух алгебраических систем, построенных для совместимых типов данных двух ЯП.

В классе систем (1) преобразование типов данных $t \rightarrow q$ для пары языков l_α и l_b обладает такими свойствами отображений:

1) Системы G_{α}^{t} и G_{β}^{q} являются изоморфными (типы *q, t* определены на том же множестве).

2) Между значениями X_{α}^{t} и X_{β}^{q} существует изоморфизм, если множества операций Ω_{α}^{t} и Ω_{β}^{q} различны. Если множество операций $\Omega = \Omega_{\alpha}^{t} \cap \Omega_{\beta}^{q}$ не пусто, то имеем изоморфизм двух систем $G_{\alpha}^{t'} = < X_{\alpha}^{t}, \Omega >$ и $G_{\beta}^{q'} = < X_{\beta}^{q}, \Omega >$. Если тип t – строка, а тип q – вещественный, то между множествами X_{α}^{t} и X_{β}^{q} не существует изоморфного соответствия.

3) Мощности алгебраических систем должны быть равны $/G_{\alpha}^{t}/ = /G_{\beta}^{q}/$. Алгебраические системы линейно упорядочены и поэтому любое отображение 1), 2) сохраняет линейный порядок.

8.2.3. Средства стандарта ISO/IEC 11404–1996 для преобразования данных

Цель этого стандарта состоит в том, чтобы обеспечить не только описание типов данных в языке LI (Language Independent), их генерацию, но и преобразование типов данных ЯП в LI–язык и наоборот. Стандарт предлагает специальные правила и характеристические операций генерации примитивных типов данных и их объединений LI–языка в более простые структуры данных ЯП, а также при определении параметров интерфейса, задаваемых в языках IDL, RPC и API.

Независимые от ЯП типы данных LI–языка разделены на примитивные, агрегатные, сгенерированные типы данных (рис.1), семейство типов данных и генератор типов данных.

Рис.8.1. Независимые от ЯП типы данных стандарта ISO/IEC 11404–1996

Типы данных в стандарте должны описываться в LI–языке, который в отличие от средств описания типов данных в ЯП является более общим языком, содержащим все существующие типы в ЯП и типы данных, ориентированные на генерацию других типов данных. Средствами LI описываются параметры вызова, как элементы

интерфейса, необходимые при обращении к стандартным сервисам и готовым программным компонентам.

Стандарт имеет раздел объявления типов данных (рис.8.2), переименования существующих; объявление новых генераторов, значений и результатов. Каждый тип данных объявляется по шаблону, включающему описание и спецификатор типа данных, значение в пространстве значений, синтаксическое описание и операции над типами данных.

LI–язык предлагает следующие виды преобразования данных:
– внешнее преобразование из внутренних типов данных ЯП в LI–типы данных;
– внутреннее преобразование из LI–типы данных, в тип данных ЯП;
– обратное внутреннее преобразование.

Рис.8.2. Объявление типов данных в стандарте ISO/IEC 11404–1996

Суть *внешнего преобразования* типов данных и генераторов типов данных состоит в следующем:
а) для каждого примитивного типа для сгенерированного внешнего типа данных преобразование связывается с одним LI–типом данных;
в) для каждого внутреннего типа данных преобразование определяет связь между допустимым значением внутреннего типа данных и эквивалентным значением соответствующего LI–типа данных;
с) для каждого значения LI–типа данных, участвующего в преобразовании, определяется существование значения любого внутреннего типа данных, преобразуемого в LI–тип данных со взятием этого значения.

Внешнее преобразование документирует аномалии при идентификации внутренних типов и дает гарантию того, что интерфейс между программными компонентами адекватно задается сервисным средством и игнорирует среду ЯП.

Внутреннее преобразование связывает примитивный тип данных или сгенерированный в LI–тип данных с конкретным внутренним типом данных ЯП. Представители отдельного семейства LI–типа данных могут преобразовываться в различные

внутренние типы данных ЯП. Данное преобразование обладает следующими свойствами:

а) для каждого LI–типа данных (примитивного или сгенерированного) преобразование определяет наличие этого типа данных в ЯП;
в) для каждого LI–типа данных преобразование определяет отношение между допустимым значением этого типа и эквивалентным значением соответствующего внутреннего типа ЯП;
с) для каждого значения внутреннего типа данных преобразование определяет является ли это значение образом (после преобразования) какого–то значения LI–типа данных и способ преобразования.

Обратное внутреннее преобразование для LI–типа данных состоит в преобразовании значений внутреннего типа данных в соответствующее значение LI–типа при наличии соответствия и отсутствия двусмысленности. Это преобразование для ЯП является коллекцией обратных внутренних преобразований LI–типа данных.

В стандарте приведен набор приложений. В приложении А приведен перечень действующих стандартов (около 40), определяющих наборы символов. Для обеспечения совместимости используемых и реализуемых типов данных в приложении В содержатся рекомендации по идентификации типов данных и описанию аннотаций для атрибутов, параметров и др. В приложении С даны рекомендации по соответствующим внутренним типы данных, которые должны преобразовываться LI–типы данных. В приложении D показано, что синтаксис LI–языка является подмножеством стандарта IDM (Interface Definition Notation), предназначеного для описания интерфейса в LI–языке, Приведен вариант внутреннего преобразования LI–типов данных в типы данных ЯП Паскаль (ISO/IEC 7185–90). В нем рассмотрены примеры преобразования примитивных типов данных LI–языка (логический, перечислимый, символьный, целый рациональный и др.) в типы данных языка Паскаль.

Предложенные в стандарте рекомендации, средства описания типов данных и методы их преобразования является универсальными и в настоящий момент не имеют общей программной поддержки.

8.3. Преобразование данных БД и замена БД

При переносе данных с одной БД на другую возникают проблемы, связанные с различием логических структур данных, справочников и классификаторов, используемых СУБД, а также изменениями, внесенными в БД [9,10]. Эти проблемы объясняются поясняются следующим:

1. Многомодельность представления данных в различных БД (иерархические, сетевые, реляционные модели) и СУБД;
2. Различия в логических структурах данных, в справочниках и классификаторах, в системах кодирования информации;
3. Поддержка различных языков для представления текстовой информации;
4. Разные типы СУБД и постоянное развитие их БД в процессе эксплуатации.

Проблема 1 решается путем перехода к реляционной модели данных и СУБД, поскольку они обладают более мощным математическим аппаратом, опирающимся на теорию множеств и математическую логику. Реляционная модель данных состоит из структурной, манипуляционной и целостной частей. В них соответственно фиксируется структура данных, описание программ в SQL–языке и требования к целостности.

Целостность не поддерживается в иерархических или сетевых моделей, поэтому при переходе к реляционным БД целостности данных нарушается.

Проблемы 2 вызваны тем, что логическая структура данных представляет собой концептуальную схему БД, в которой описаны основные объекты БД и связи между ними. Поэтому при изменении предметной области, переход на новую СУБД предполагает проектирование новой структуры БД, проведение сопоставления на соответствие данных в старой и новой БД, а также изменения справочной информация и классификаторов.

Проблема 3 определяется разноязычными текстовыми представлениями информации в БД. В старых БД используется, как правило, один язык, а в новых может быть несколько, поэтому необходимо организовать хранение данных с простым доступ к текстовым данным и установлении соответствия текстовых данных, записанных на разных языках.

Проблему 4 можно сформулировать как метод хранения и обработки разных данных, вызванных спецификой СУБД иерархического, сетевого и реляционного типов. Наличие явной несовместимости типов и структур моделей данных, различные языки манипулирования данными приводят к тому, что нельзя сгенерировать на языке старой СУБД скрипты по переносу данных с последующим запусков этих скриптов в среде другой СУБД. Каждая СУБД обеспечивает внесение изменений в БД, которые в некоторой степени меняют и концептуальную модель данных, если в нее вносятся новые объекты. Внесенные изменения должны отображаться в справочниках и классификаторах, которые обеспечивают перенос данных из старой БД с учетом внесенных текущих изменений.

8.3.1. Основные этапы преобразования данных в БД

Учитывая приведенные проблемы, рассмотрим пути их решения. Отметим, что промышленная эксплуатация систем, работающих с БД, может продолжаться достаточно долго. При этом изменяются прикладные программы, работающие с БД, повторно преобразуются данные, если в систему введена новая БД, а часть ранее определенных данных уже перенесены в новую БД.. Это влечет за собой доработку прикладных программ доступа к данным, чтобы приспособить их к измененной структуре новой БД или к старой БД. Для переноса данных из старой БД в новую разрабатываются скрипты с приведенной логической структурой БД или DBF–файлы, которые вначале размещаются в транзитной БД, а затем с учетом особенностей новой основной БД переносятся в нею. Может оказаться, что процесс приведения структур транзитной БД к новой окажется нецелесообразным и разработку новой БД проводить "с нуля". При этом заполненные справочники и классификаторы потребуется дополнить появившимися новыми данными.

Проблемы преобразования данных при использовании разных СУБД возникают из–за того, что данные имеют различные способы хранения, среди которых могут быть несовместимые типы данных, а также доступ к данным осуществляется разными языками манипулирования данных, используемых СУБД.

Преобразование данных может проводиться несколько раз путем создания специальных скриптов и файлов с учетом ранее введенных данных, снятия дублирования данных и корректного приведения несовместимых типов данных При этом могут возникнуть

ошибки, связанные с изменением форматов данных, дополнением старых справочников новыми данными и т.п.

Этапы преобразования данных. Процесс преобразования данных состоит из трех главных частей:

1. Перенос данных между СУБД (перенос данных из старой БД в транзитные файлы и затем занесение данных из этих файлов в транзитную БД;
2. Обработка данных в транзитной базе в случае изменения кодировки данных, приведение в соответствие структур старой и новой баз данных, а также кодов справочников и классификаторов;
3. Перенос данных из транзитной базы в основную базу данных и проверка преобразования данных.

Первый метод замены представляет собой наиболее безболезненный для пользователей и разработчиков.

Второй метод замены представляет собой создание нового проекта системы на основе имеющейся модели данных. При третьем варианте создается система заново и в новую БД заносятся унаследованные данные из старой БД. Поскольку структуры БД различны, то создаются, как правило, временные приложения, осуществляющие нужные преобразования данных в процессе их переноса в новую БД.

При применении первого и второго метода структура старой БД сохраняется и никакого преобразования данных и соответствия справочников и классификаторов не требуется – они используют единый формат хранения данных.

8.3.2. Унифицированные файлы для передачи данных между разными БД

Проблема преобразования и переноса данных между различными СУБД решается в основном двумя методами, основанными соответственно на использовании:

1) специального драйвера (две СУБД соединяются друг с другом и напрямую передают данные, используя интерфейс);
2) транзитных файлов, в которые копируются данные из старой БД и переносятся в новую БД.

Процесс преобразования и переноса данных из разных БД в новую БД приведен на рис.8.3.

При **первом методе** две СУБД соединены напрямую и передают данные, используя определенный интерфейс или драйвер. Иными словами,. они используют специальные программы взаимодействия двух СУБД, при которых вторая СУБД понимает результаты выполнения запросов на языке манипулирования данными первой СУБД, и наоборот. Данные на выходе первой СУБД являются данными на входе второй СУБД в языке манипулирования данными второй СУБД, такие данные могут быть внесены в транзитную БД.

Метод сложный в реализации и требует поставки с СУБД программ переноса данных из других СУБД, которые привязаны к старой и новой СУБД. Поэтому второй метод переноса данных между различными СУБД является более предпочтительным..

Второй метод заключается в том, что из старой БД данные переносятся в транзитные файлы, SGL–скрипты, DBF–файлы с заранее заданными форматами данных, которые пересылаются через сеть в новую транзитную БД.
Данные из транзитных файлов с помощью специальных утилит или средств новой СУБД затем переносятся в транзитную БД для дальнейшей обработки. Если вторая СУБД реляционного типа, то данные в транзитных файлах должны быть представлены к табличному виду. Если первая СУБД не реляционна, то данные должны быть приведены к табличному виду и первой нормальной форме.

Дальнейшая нормализация данных и приведение их к структуре новой БД осуществляется в транзитной БД. Существуют пять нормальных форм задания структур данных, чаще всего используется 3–я или 4–я нормальная форма. Каждая высокая форма нормализации содержит в качестве подмножества более низкую форму первой нормальной формы, содержащей скалярные значения.

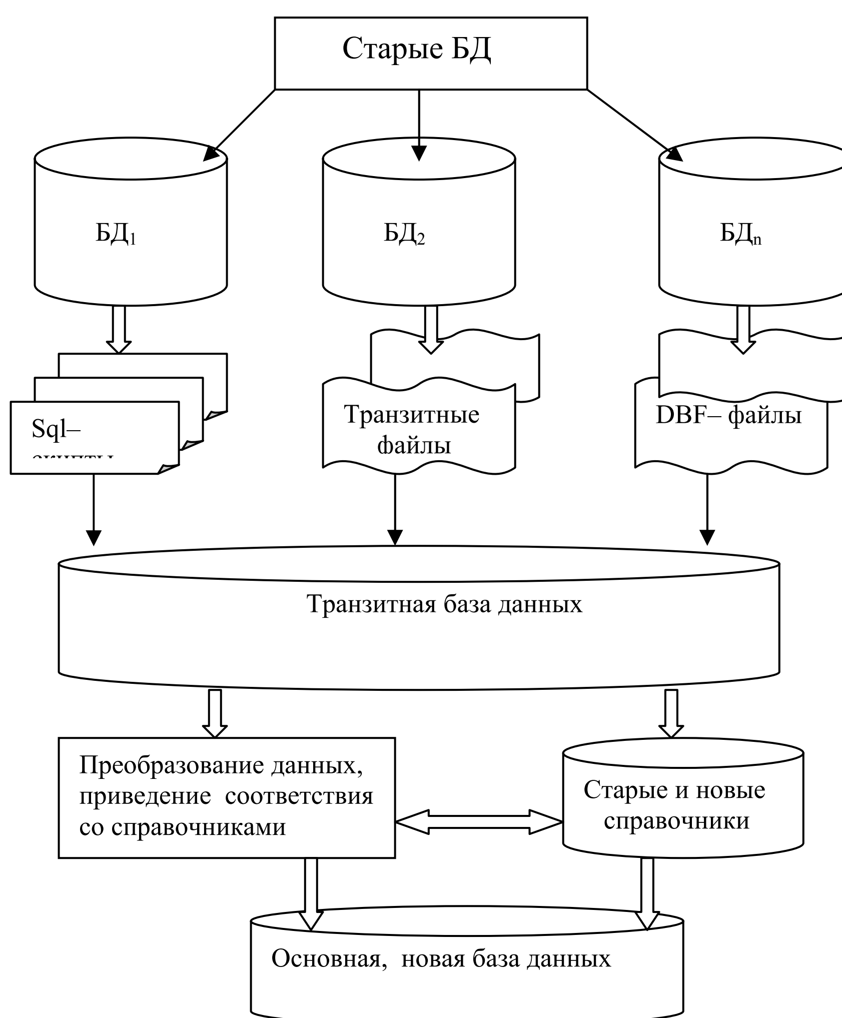

Рис. 8.3. Процесс преобразования и формирования новой БД из старых БД

Иными словами, отношение находятся в первой нормальной форме, если они хранятся в табличном виде (все ячейки в строке таблицы расположены в строго определенной последовательности) и каждая ячейка таблицы содержит только атомарные значения (каждый элемент не является множеством).

Отношение находится в *третьей нормальной форме* тогда и только тогда, когда каждый кортеж состоит из значения первичного ключа, которое идентифицирует некоторую сущность, и набора пустых значений или значений независимых атрибутов, описывающих эту сущность. Т.е. отношение находится в третьей нормальной форме, когда неключевые атрибуты являются взаимно независимыми и зависят от первичного ключа. Неключевой атрибут – это атрибут, который не задействован первичным ключом рассматриваемого отношения.

Два или несколько атрибутов являются взаимно независимыми, если ни один из них не зависит функционально от какой–либо комбинации остальных атрибутов. Подобная независимость подразумевает, что каждый атрибут может быть обновлен независимо от остальных.

Процесс нормализации отношений позволяет избавиться от проблем, которые могут возникнуть при обновлении, внесении или удалении данных, а также при обеспечении целостности данных.

Структуры старых баз данных, не всегда приводятся к третьей нормальной форме, поэтому требуется, чтобы данные, находящиеся в транзитных файлах, существовали хотя бы в первой нормальной форме и относились к реляционной модели.

В качестве унифицированного формата транзитных файлов используется формат DBF–файлов, поскольку многие СУБД, такие как DB2, FoxPro и некоторые другие хранят данные в таких файлах, тем самым не требуется начальный перенос данных из старой СУБД в транзитные файлы. Большинство СУБД, формат хранения данных которых отличается от формата DBF–файлов, снабжены утилитами или драйверами, которые позволяют перенести данные в такой формат.

8. 4. Методы внесения изменений в компоненты и в ПС

Активное использование созданных ПС проводится на процессе сопровождения. При этом возникают разного рода ошибки, которые требуют внесения изменений после того, как ошибка обнаружена или возникла необходимость в улучшении и некоторых характеристик системы.

В отличие от технического обеспечения, которое с течением времени требует ремонта, программное обеспечение не "снашивается" и поэтому процесс сопровождения нацелен более всего на эволюцию системы, то есть не только на исправление ошибок, а и на замену ее отдельных функций и возможностей.

Типичными причинами внесения изменений являются:

- выявление дефектов в системе во время эксплуатации, которые не были обнаружены на этапе тестирования;
- выяснение несоответствия или невыполнения некоторых требований заказчика, благодаря чему система не выполняет отдельные функции;
- изменение условий заказчиком, которые предполагают корректировку ранее поставленных им требований.

Как утверждают эксперты, процесс внесения изменений в эксплуатируемую систему достаточно дорогой, оценки его стоимости достигают от 60 % до 80 % от общей стоимости разработки системы.

К видам сопровождения относятся**:**

- корректировка – внесение изменений в ПС для устранения ошибок, которые были найдены после передачи системы в эксплуатацию;
- адаптация продукта к измененным условиям (аппаратуре нового типа) использования системы после ее передачи в эксплуатацию;
- *предупредительное* сопровождение – деятельность, ориентированная на обеспечение адаптации системы к новым техническим возможностям.

Одна из проблем, влияющая на процесс внесения изменений, – это степень подготовки персонала, способного вносить необходимые изменения при возникновении определенных условий.

В связи с тем, что почти каждые 8–10 лет происходит смена архитектур компьютеров, ЯП и операционных сред, возникают проблемы сопровождения готовых ПС и их компонентов в новой среде или архитектуре, решение которых приводит к изменению либо обновлению отдельных элементов системы или системы полностью.

В общем, процесс изменения ПС проводятся путем:
– анализа исходного кода для внесения в него изменений;
– настройки компонентов и системы на новые платформы;
– кодирования и декодирование данных при переходе с одной платформы на другую;
– изменения функций системы или добавления новых;
– расширения возможностей (сервиса, мобильности и др.) компонентов;
– преобразования структуры системы или отдельных ее компонентов.

Сущность всех видов изменений в один компонент и совокупности компонентов ПС направлена на придание старой ПС нового назначения и определения условий применения.

Методы изменения ПС служат способом продления жизни наследуемых и стареющих программ. С теоретической точки зрения эти методы изучены недостаточно, а с практической точки зрения многие программисты решают задачи внесения изменений в ПС постоянно. Например, широкий круг специалистов охватила проблема 2000 года и ее решение мировым сообществом, а также создание современных оффшорных зон (Индия, Россия, Украина и др.) для систематической переделки функционирующих программ к новым возможностям ОС, языков и платформ современных компьютеров и т.п.

Процесс обновления компонента включает в себя [14–20]:

– реинженерию, целью которой является перепрограммирование отдельных компонентов с учетом новых ЯП и условий новых платформ и сред, а также расширение возможностей ПС;
– рефакторинг, в задачу которого входит изменение существующего компонента или его адаптация к новым условиям вместе с изменением его интерфейсов (добавление, расширение и т.д.) или изменения механизмов управления экземплярами компонентов (добавление новых функций) или системных сервисов;
– реверсная инженерия, цель которой – перестройки или полная переделки компонентов, а иногда и перепрограммирование системы в новый ЯП.

8.4.1. Реинженерия программных систем

Реинженерия (reengineering) – это повторная реализация программы (системы) в целях повышения удобства ее эксплуатации, сопровождения или изменения ее функций. Она включает такие процессы как повторное документирование системы, реорганизация и реструктуризация, перевод системы на один из более современных ЯП, модификация и модернизация структуры и системы данных. При этом архитектура системы может оставаться неизменной.

Метод реинженерии – целевое средство получения нового компонента благодаря последовательности операций внесения изменений, модернизации или модификации, а также перепрограммирования или адаптации компонентов.

Реинженерия компонентов – это совокупность моделей, методов и процессов изменения структуры и возможностей компонентов с целью получения компонента с новыми возможностями.

Новые компоненты необходимо идентифицировать по определенным действующим правилам именования компонентов и методов их применения в системах из компонентов. Решение проблемы идентификации связано с необходимостью создания компонентных конфигураций и каркасов системы из компонентов.

С технической точки зрения реинженерия – это решение проблемы эволюции системы. Если предположить, что архитектура системы не изменяется, то преобразовать централизованную систему в распределенную, компоненты которой размещаются в операционной среде на разных компьютерах, является довольно сложным процессом. (Например, изменить язык программирования старой системы (Fortran, Cobol и др.) на современные объектно–ориентированные языки Java или C++.

Однако с коммерческой точки зрения реинженерию принимают часто за единственный способ сохранения наследуемых систем в эксплуатации. Подходы к эволюции системы, при которых изменяется все, являются дорогостоящими либо рискованными.

Так как ПС много, то полная замена или радикальная реструктуризация их в большинстве затруднена. Методами реинженерии проще совершенствовать структуру, документацию или отдельные компоненты системы. Сопровождение старых систем действительно стоит дорого, однако реинженерия может продлить время их существования.

По сравнению с более радикальными подходами к совершенствованию систем реинженерия имеет следующие преимущества.

1. Снижение рисков. При повторной разработке ПО существует риск получения неудовлетворительного результата, который определяется внесением ошибок в системные спецификации, снижением надежности или изменением функционирования некоторых программ. Снизить возникающие риски можно за счет удаления ошибок и улучшения качества работы измененных программ.

2. Снижение затрат. Себестоимость реинженерии значительно ниже, чем разработка нового ПО. Согласно данным различных коммерческих структур она в четыре раза дешевле, чем повторная разработка системы.

Реинженерия применяется при изменении деловых процессов, снижении количества излишних видов деятельности в них и повышением эффективности отдельных деловых процессов. Это можно сделать путем внедрения новых программ или модификации существующих программ, выполняющихся на процессе. Если бизнес–процесс зависит наследуемых систем, используемых в процессе, то это надо учитывать при планировании каких–либо изменений.

Основное различие между реинженерией и новой разработкой системы состоит в том, что написание системной спецификации начинается не с «нуля», а с учета возможностей старой наследуемой системы при разработке спецификации новой системы. К основным этапам этого процесса относятся:

- перевод исходного кода путем конвертирования программы в старом языке программирования на современную версию этого языка либо на другой язык программирования;
– анализ программ для документирования структуры и функциональных ее возможностей;
модификация структуры программ для ее упрощения, понятности и наращивания новых свойств;
– разбиение на модули для группирования и устранения избыточности, что приводит к изменению структуры системы;
– изменение системных данных, с которыми работает программа для согласования с проведенным изменениями в программе.

Преобразование исходного кода программ – наиболее простой способ реинженерии программ. Оно может быть выполнено автоматически (автоматизировано). Причинами перевода на другой язык могут быть:

1. Обновление платформы аппаратных средств, на которой может не выполняться компилятор исходного языка программ.
2. Недостаток квалифицированного персонала для программ, написанных в специфических ЯП, вышедших из употребления.
3. Изменение структуры организации программы в связи с переходом на общий стандартный ЯП для снижения затрат на сопровождение программных систем.

К операциям реинженерии относятся:
– именование компонентов и их идентификацию;
– расширение функций существующей реализации компонента;
– перевод языка компонента на новый современный язык программирования;
– реструктуризация структуры компонента;
– модификация описания компонента и его данных.

8.4.2. Рефакторинг компонентов

Рефакторинг получил развитие в объектно–ориентированном программировании [2] в связи с применением шаблонов проектирования, методов улучшения кода и структуры объектно–ориентированных программ. Были разработаны целые библиотеки типичных трансформаций искомых объектов (классов), которые улучшают те или другие характеристики. Он вошел в компонентное программирование и базируется на типичной структуре, модели компонента и ориентирован на изменении не только компонентов, а и их интерфейсов.

Метод рефакторинга компонента – это целевой способ получения нового компонента на базе существующего, включает операции модификации (изменение, замещение, расширение) компонентов и интерфейсов, состоящие в преобразовании состава компонентов ПС или в изменения отдельного компонента системы для придания ему новых функциональных и структурных характеристик, которые наиболее полно удовлетворяют требования компонентной конфигурации.. Фактически такие изменения компонентов соответствуют процессу проектирования компонентных ПС.

Рефакторинг – это совокупность моделей, методов, процессов, применяемых к определенным классам объектов и компонентов для получения новых или изменения существующих объектов– компонентов с целью повышения качественных характеристик. ПС или добавление новых возможностей к существующей компонентной системе.

Процессы рефакторинга могут быть разными, их главная цель – получение новых компонентов, однозначно идентифицированных, и включающих в себя следующие операции над компонентами и интерфейсами по организации проведения изменений:

– добавление новой реализации для существующего и нового интерфейса;
– замена существующей реализации новой с эквивалентной функциональностью;
– добавление нового интерфейса (при условии наличия соответствующей реализации);
– расширение существующего интерфейса**;**
– расширение интерфейса для новых системных сервисов в компонентной среде.

Каждая операция рефакторинга является базовой, атомарной функцией преобразования, которая сохраняет целостность компонента. *Под целостностью компонента* понимается совокупность правил, ограничений и зависимостей между составными элементами компонента, которые разрешают рассматривать компонент как единую и цельную структуру со своими свойствами и характеристиками

После выполнения операции рефакторинга измененные компоненты должны быть идентичными исходным и сохранять функции. В случае коренного изменения группы компонентов системы в связи с введением новых функций, система приобретает новую функциональность.

Операции над компонентами должны удовлетворять следующим условиям:
– объект, полученный в результате рефакторинга является компонентом с соответствующими свойствами, характеристиками и типичной структурой;
– операция не изменяют функциональность, то есть новый компонент может примениться в раннее построенных компонентных системах;
– перестройки или переделки компонентов, а иногда и перепрограммирования в случае реверсной инженерии [4, 5].

8.4.3. Реверсная инжеиерия

Методы реверсной инженерии разработаны в среде объектно–ориентированного программирования [1]. Они базируется на выполнении базовых операций визуализации (visual) и измерения метрик (metric) программных систем в рамках модели, которая предлагает следующие цели:

– обеспечение высокого качества системы и переосвидетельствование ее размера, сложности и структуры;

– поиск иерархии классов и атритутов программных объектов с целью наследования их в ядре системы;
– идентификация классов объектов с определением размера и /или сложности всех классов системы;
– поиск патернов, их идентификация, а также фиксация их места и роли в структуре системы.

Этот подход ориентирован на индустриальные системы в миллион срок кода с использованием метрических оценок характеристик системы. Он разрешает генерацию тестов для проверки кодов, а также проведение метрического анализа системы для получения фактических значений внутренних и внешних характеристик системы.

В результате анализа системы строится модель, которая содержит список классов и патернов системы, которые могут модифицироваться и перепроектироваться, а также с помощью которых можно организовать процесс эволюции системы. Если некоторый класс плохо спроектирован (например, много методов и имеются пустые коды) или система не выполняет нужную работу, то проводится сбор информации для модели системы.

В данном подходе действия по визуализации системы освещаются на двухмерном экране в виде иерархического дерева. В нем узлы отображают объекты и их свойства, а отношение между ними задаются контурами команд фрагментов программ. При этом применяется таблица метрик, в которой находятся сведения о метриках классов объектов (число классов, методов, атрибутов, подклассов и строк кода), метрик методов объектов (количество параметров, вызовов, сообщений и т.п.), метрик атрибутов объектов (время доступа, количество доступов в классе и подклассе и т.п.).

В процессе визуализации ведется сбор метрических данных о системе. Если реально определены все данные в разных фактических метриках программной системы, выполняются оценка качества и разработка плана перестройки устаревшей системы на новую с получением тех же возможностей или новых дополнительных.

Вывод. Преобразование исходного кода с целью внесения разного рода изменений (реинженерии, рефакторинга, реверсной инженерии) является эффективным, если основные изменения выполняются автоматически с помощью специально созданных систем, либо с помощью программ конвертирования кода с одного языка на другой с учетом наличия разных типов данных в этих ЯП, либо с помощью системы сопоставления с образцом, представленным в виде списка команд для описания перевода с одного языкового представления на другое, которое реализуется сравнением и сопоставлением с такими же образцами в новом языке.

Автоматизированный перевод не возможен, если структурные компоненты исходного кода не имеют соответствия в новом языке. Это бывает в том случае, когда исходный язык содержит встроенные условные команды компиляции, которые не поддерживаются в новом языке. В этом случае совершенствование создаваемой системы выполняется вручную со значительными затратами человеческих и финансовых ресурсов.

Контрольные вопросы и задания

1. Определите цели и задачи метода интеграции в программной инженерии.
2. Назовите системы, которые поддерживают процессы интеграции и преобразования данных.
3. Охарактеризуйте кратко современные системы взаимосвязи объектов – COM, CORBA, JAVA и др.
4. Назовите методы вызова компонентов в распределенных средах.
5. Какую роль выполняет брокер объектных запросов
6. Определите проблему преобразования данных в ЯП.
7. Какие требуется провести преобразования передаваемых по сети данных от объекта JAVA в к объекту в С++ и обратною
8. Определите проблемы преобразования данных, связанные с заменой одной БД на другую.
9. Какие методы переноса данных существуют?
10. Определите цели и задачи изменения ПС при проведении сопровождения.
11. Какие выполняются работы при сопровождении, когда вносятся изменения?
12. Дайте краткую характеристику проблем, возникающих при сопровождении системы.
13. Определите основные задачи реинженерии ПО.
14. Определите основные операции рефакторинга компонентов.
15. Определите основные операции реинженерии программных систем.

Литература к теме 8.

1. *Open* Software Foundation. Inroduce to Open Software Foundation. Disributed Computed Environments.– Englewood Cliffs: Prentice Hall, 1993.– 437p.
2. *Corbin J.* The art of Distributed Application. Programming Techn. For Remote Procedure Calls.– Berlin: Springer Verlag, 1992.– 305p.
3. *Роджерсон Д.* Основы COM. Руск..пер.– Microsoft Press.– 361c.
4. CORBA. The Common Object Request Broker: Architecture and Specification. Revision 2.0. Copyright 1991, 1992, 1995 by Sun Microsystems, Inc.–1995.–621 p.
5. *Монсон–Хейфел Р.* Enterprise JavaBeans. – СПб: Символ–Плюс, 2002. – 672 с.
6. *Барлет Н., Лесли А., Симкин С.* Программирование на JAVA. Путеводитель.– Киев.– 1996.– 736c.
7. *Иванников В.П., Дышлевый К.В., Мажелей С.Г., Содовская Д.Б., Шебуняев А.Б.*Распределенные объектно–ориентированные среды.– Москва // РАН.ИСП. Труды ИСП,.–2000.– с.84–100.
8. *Гост 30664* (ИСО/МЭК 11404:1996). Информационные технологии. Языки программирования, их среда и системный интерфейс. Зависимые от языков типы данных/ Межгосударственный стандарт.–Межгосударственный совет по стандартизации, метрологии и сертификации, 2000. – 112 с.
9. *Джордан Д.* Обработка объектных бах данных в С++. Программирование по стандарту ODMG: Пер.с англ. – М.: Издательский дом “Вильямс”, 2001. – 384 с.
10. *Дунаев С. Б.* Доступ к базам данных и техника работы в сети. – М.: Диалог–Мифи, 1999. – 416 с.
11. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.
12. *Лаврищева Е.М., Грищенко В.М.* Сборочное программирование .–Киев.– Наукова думка.– 1991. –213с.

13. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии// Программирование, 2000.– №1–2 (Труды второй междун.науч..конф. УкПрог–2000, 23–26мая 2000г. –Киев).– с.351–360.
14. *Lanza M., Ducasse S.* Polimetric Views –A lightweight Visual Approach to Reverse Engineering //IEEET Transaction on Software Engineering.– 2003.– Sept., №3 (ISSN 0098–5589).– P.782–796.
15. *Фаулер М.* Рефакторинг: улучшение соответствующего кода. – СПб.: Символ–Плюс, 2003. – 432 с.
16. *Пантелеймонов А.А.* Аспекты реинженерии приложений с графическим интерфейсом пользователя//Проблемы программирования .–2001.–№1–2.– С.53–62.
17. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии «Знание».–2001. – 269с.
18. *Соммервилл И.* Инженерия программного обеспечения.– Изд. Дом «Вильямс», Москва
19. *Игнатенко П.П., Неумоин В.Н., Бистров В.М.* Об обеспеченни эффективного реинжинеринга прикладних програмних систем // Пробл. программирования. – 2001. – №1–2. – с. 42–52.
20. *Гласс Г., Нуазо Р.* Сопровождение программного обеспечения. Пер.с анг. // Под ред. Ю.А.Чернышова.– М.:.Мир.– 1983.–256 с.

Тема 9
МОДЕЛИ КАЧЕСТВА И НАДЕЖНОСТИ В ПРОГРАММНОЙ ИНЖЕНЕРИИ

Разработка ПС достигла такого уровня развития, что начали развиваться и использоваться инженерные методы, способствующие обеспечению высокого качества, в частности надежности, компонентов и системы в целом. Повышение качества – основная цель инженерных методов в программировании и задача разработчиков и заказчиков. Для достижения этих целей сформировались методы определения требований к качеству, подходы к выбору и усовершенствованию моделей метрического анализа показателей качества, методы количественного измерения показателей качества на этапах ЖЦ.

Главной составляющей качества является надежность, которой уделяется большое внимание со стороны многих специалистов в области надежности технических средств и тех критических систем (реального времени, радарные системы, системы безопасности и др.), для которых надежность является главной целевой функцией их реализации. Как следствие, разработано более сотни математических моделей надежности, являющихся функциями от ошибок, оставшихся в ПС, от интенсивности отказов или частоты появления дефектов в ПС. По ним производится оценка показателя – надежность ПС.

Качество ПО было предметом стандартизации, создан стандарт ГОСТ 2844–94, в котором дано определение качества ПО, как совокупность свойств (показателей качества) ПО, которые обеспечивают его способность удовлетворять потребности заказчика, в соответствии с назначением. Этот стандарт регламентирует базовую модель качества и его показатели, главным среди них является надежность. Стандарт ISO/IEC 12207 определил не только основные процессы ЖЦ разработки ПС, но и организационные и дополнительные процессы, которые регламентирую инженерию, планирование и управление качеством ПС.

На этапах ЖЦ проводится анализ качества ПО, ориентированные на:
– достижение качества ПО в соответствии с требованиями и критериями;
– верификацию и аттестацию (валидацию) промежуточных результатов ПО на этапах ЖЦ и измерение степени достижения отдельных его показателей;
– тестирование готовой ПС, сбор данных об отказах, дефектах и др. ошибках в системе и оценивание надежности по соответствующим моделям надежности.

Изложение данной темы будем проведено по представлению моделей качества и надежности, способы их применения в создаваемых ПС.

9.1. Модель качества ПО

Качество ПО является относительным понятием, которое имеет смысл только при учете реальных условий его применения, поэтому требования, предъявляемые к качеству, ставятся в соответствии с условиями и конкретной областью их применения.

Качество ПО характеризуется тремя главными аспектами: качество программного продукта, качество процессов ЖЦ и качество сопровождения или внедрения (рис. 9.1).

Аспект, связанный с процессами ЖЦ, характеризуется степенью формализации, достоверностью и качеством самих процессов ведения разработки ПО, а также верификацией и валидацией полученных промежуточных результатов на процессах ЖЦ, уменьшающих количество ошибок в ПО и тем самым способствующих повышению качества готового продукта.

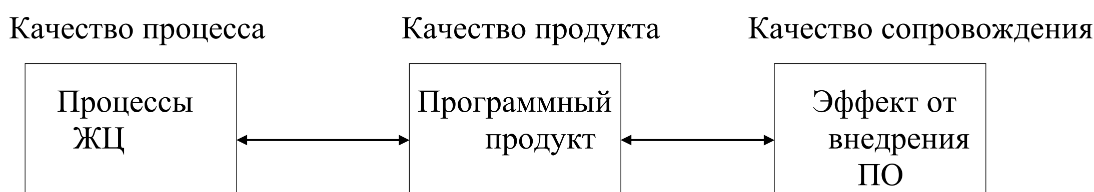

Рис. 9.1 Основные аспекты качества ПО

Качество продукта целиком и полностью определяется процессами ЖЦ. Эффект от внедрения полученного программного продукта в значительной степени зависит от качества сопровождения и знаний обслуживающего персонала.

Модель качества ПО имеет следующие четыре уровня детализации.

Первый уровень соответствует определению характеристик (показателей) качества для ПО, каждая из них отражает отдельную точку зрения пользователя на качество. Согласно стандартов [1–4] определено шесть характеристик или шесть показателей качества в стандартной модели качества:

1. функциональность (functionality),
2. надежность (realibility),
3. удобство (usability),
4. эффективность (efficiency),
5. сопровождаемость (maitainnability),
6. переносимость (portability).

Второму уровню соответствуют атрибуты качества для каждой характеристики, которые детализируют разные аспекты конкретной характеристики. Набор атрибутов характеристик качества используется при оценки качества.
Третий уровень предназначен измерения качества с помощью метрик, каждая из них согласно стандарта [1] определяется как комбинация метода измерения атрибута и шкалы измерения значений атрибутов. Для оценки атрибутов качества на этапах ЖЦ (при просмотре документации, программ и результатов тестирования программ) используются метрики с заданным оценочным весом для нивелирования результатов метрического анализа совокупных атрибутов конкретного показателя и качества в целом. Атрибут качества определяется с помощью одной или нескольких методик оценки на этапах ЖЦ и на завершающем этапе разработки ПО.

Четвертый уровень задает оценочный элемент метрики для оценки количественного или качественного значения отдельного атрибута показателя ПО с учетом его веса.

В зависимости от назначения, особенностей и условий сопровождения ПО выбираются наиболее важные характеристики качества и их приоритеты. Выбранные для каждой характеристики атрибуты и их приоритеты отражаются в требованиях на разработку систем. Для программных систем, при разработки которых в требованиях не указан приоритет характеристик качества, используется приоритет эталона – класса ПО, к которому оно относится.

Модель качества приведена на рис.9.2, а короткое описание семантики каждой из шести характеристик качества и ее атрибутов приводится ниже.

1). Функциональность – совокупность свойств, определяющих способность ПО выполнять в заданной среде перечень функций в соответствии с требованиями к обработке и общесистемным средствам.

Под *функцией* понимается некоторая упорядоченная последовательность действий для удовлетворения некоторых потребительских свойств. Функции бывают целевые (основные и вспомогательные).
К атрибутам функциональности относятся:
– функциональная полнота – свойство компонента, которое показывает степень достаточности основных функций для решения задач в соответствии с назначением ПО;
– правильность (точность) – атрибут, который показывают степень достижения правильных результатов;
– интероперабельность – атрибут, который показывают на возможность взаимодействовать ПО со специальными системами и средами (ОС, сеть);
– защищенность – атрибут, который показывают на необходимость предотвратить несанкционированный доступ (случайный или умышленный) к программам и данным.

2). Надежность – совокупность атрибутов, которые определяют способность ПО преобразовывать исходные данные в результаты при условиях, зависящих от периода времени жизни (износ и его старение не учитывается). Снижение надежности ПО происходит из–за ошибок в требованиях, проектировании и выполнении. Отказы и ошибки в программах появляются на заданном промежутке времени [8-13].

К подхарактеристикам надежности ПО относятся:
– безотказность – атрибут, который определяет функционирование системы без отказов (программы или оборудования);
– устойчивость к ошибкам – атрибут, который показывают на способность ПО выполнять функции при аномальных условиях (сбоях аппаратуры, ошибках в данных и интерфейсах, нарушениях в действиях оператора и др.);
– восстанавливаемость – атрибут, который показывают на способность программы к перезапуску для повторного выполнения и восстановления данных после отказов.

К некоторым типам систем (реального времени, радарных, систем безопасности, коммуникация и др.) предъявляются требования для обеспечения высокой надежности (недопустимость ошибок, точность, достоверность, удобство применения и др.). Таким образом, надежность ПО в значительной степени зависит от числа оставшихся и не устраненных ошибок в процессе разработки на этапах ЖЦ. В ходе эксплуатации ошибки обнаруживаются и устраняются.

Если при исправлении ошибок не вносятся новые, или, по крайней мере, новых ошибок вносится меньше, чем устраняется, то в ходе эксплуатации надежность ПО непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность ПО.

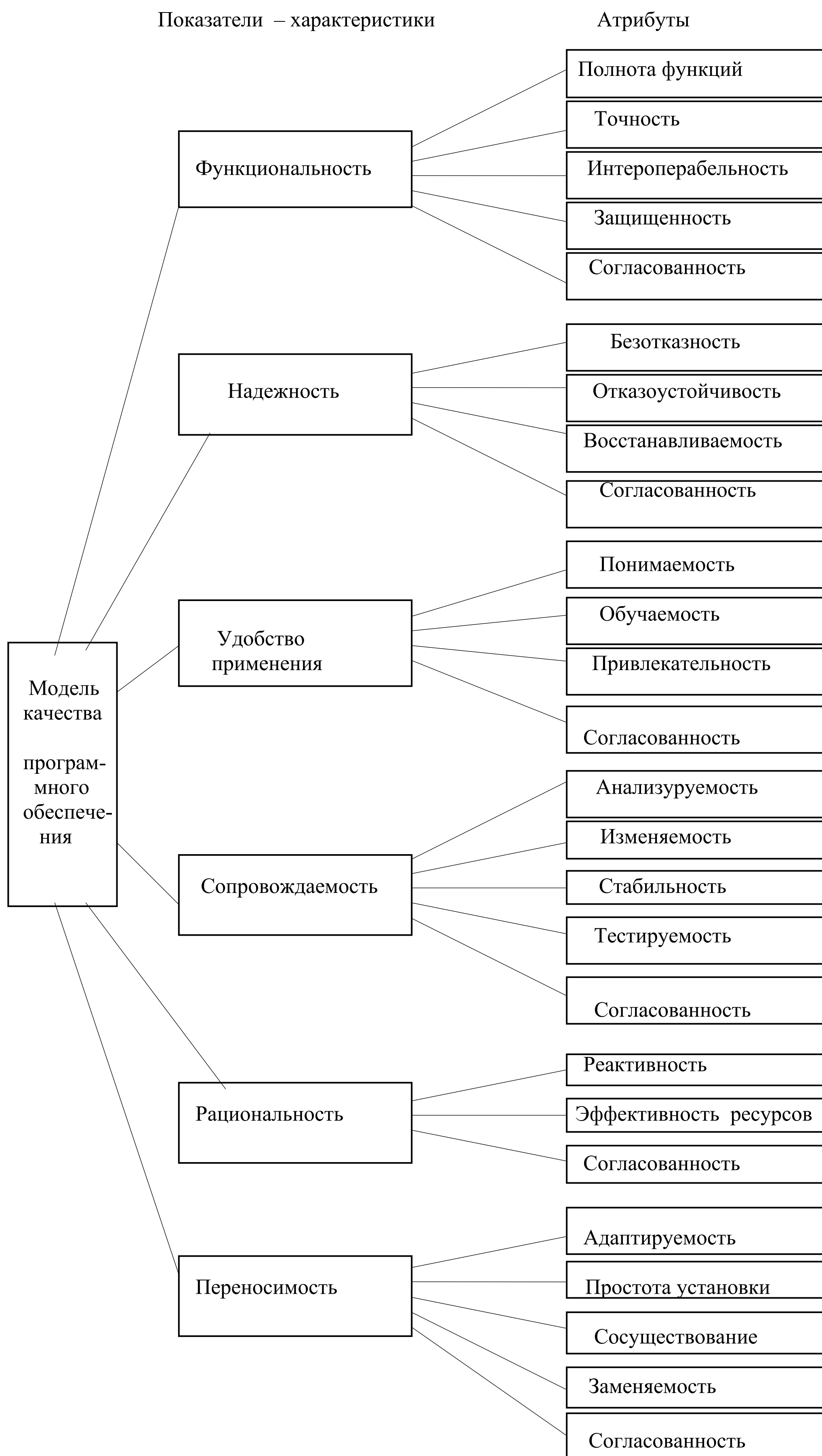

Рис. 9.2. Модель характеристик качества

К факторам, влияющим на надежность ПО, относятся:
– риск как совокупность угроз, приводящих к неблагоприятным последствиям и ущербу системы или среды ее функционирования;
– угроза как проявление нарушения безопасности системы;
– целостность как способность системы сохранять устойчивость работы и не иметь риска.

Обнаруженные ошибки могут быть результатом угрозы извне или отказов, они повышают риск и уменьшают некоторые свойства надежности системы.

3). Удобство применения характеризуется множеством атрибутов, которые показывают на необходимые и пригодные условия использования (диалоговое или недиалоговое) ПО определенным кругом пользователей для получения соответствующих результатов. В стандарте [3] удобство применения определено как специфическое множество атрибутов программного продукта, характеризующих его эргономичность.

Подхарактеристиками удобства применения являются:
– понимаемость – атрибут, который определяет усилия, затрачиваемые на распознавание логических концепций и условий применения ПО;
– изучаемость (легкость изучения) – атрибут, который определяет усилия пользователей при определении применимости ПО, используя, например, операционный контроль, ввод, вывод, диагностику, а также процедуры, правила и документацию;
– оперативность – атрибут, который показывает на реакцию системы при выполнении операций и операционного контроля;
– согласованность – атрибут, который показывает соответствие разработки требованиям стандартов, соглашений, правил, законов и предписаний.

4). Эффективность – множество атрибутов, которые определяют взаимосвязь между уровнем выполнения ПО, количеством используемых ресурсов (средства, аппаратура, другие используемые материалы – бумага для печатающего устройства и др.) и услуг выполняемых штатным обслуживающим персоналом и др.

К подхарактеристикам эффективности ПО относятся:
– реактивность – атрибут, который показывает время отклика, обработки и выполнения функций;
– эффективность ресурсов – атрибут, показывающий количество и продолжительность используемых ресурсов при выполнении функций ПО;
– согласованность: атрибут, который показывает соответствие данного атрибута с заданными стандартами, правилами и предписаниями.

5). Сопровождаемость

Сопровождаемость – множество свойств, которые показывают на усилия, которые надо затратить на проведение модификаций, включающих корректировку, усовершенствование и адаптацию ПО при изменении среды, требований или функциональных спецификаций.

Сопровождаемость включает подхарактеристики:
– анализируемость – атрибут, определяющий необходимые усилия для диагностики в отказов или идентификации частей, которые будут модифицироваться;

– изменяемость – атрибут, определяющий усилия, которые затрачиваются на модификацию, удаление ошибок или внесение изменений для устранения ошибок или введения новых возможностей в ПО или в среду функционирования;
– стабильность – атрибут, указывающие на риск модификации;
– тестируемость – атрибут, показывающий на усилия при проведении валидации, верификации с целью обнаружения ошибок и несоответствий требованиям, а также на необходимость проведения модификации ПО и сертификации;
– согласованность – атрибут, который показывает соответствие данного атрибута с определенными в стандартах, соглашениях, правилах и предписаниях.

6). Переносимость – множество показателей, указывающих на способность ПО приспосабливаться к работе в новых условиях среды выполнения. Среда может быть организационной, аппаратной и программной. Поэтому перенос ПО в новую среду выполнения может быть связан с совокупностью действий, направленных на обеспечение его функционирования в среде, отличной от той среды, в которой оно создавалось с учетом новых программных, организационных и технических возможностей.

Переносимость включает подхарактеристики:
– адаптивность – атрибут, определяющий усилия, затрачиваемые на адаптацию к различным средам;
– настраиваемость (простота инсталлирования) – атрибут, который определяет на необходимые усилия для запуска или инсталляции данного ПО в специальной среде;
– сосуществование – атрибут, который определяет возможность использования специального ПО в среде действующей системы;
– заменяемость – атрибут, который обеспечивают возможность интероперабельности при совместной работе с другими программами с необходимой инсталляцией или адаптацией ПО;
– согласованность – атрибут, который показывают на соответствие стандартам или соглашениями по обеспечению переноса ПО.

9.1.1. Метрики качества программного обеспечения

В настоящее время в программной инженерии еще не сформировалась окончательно система метрик. Действуют разные подходы и методы определения их набора и методов измерения [6–8, 14, 15].

Система измерения ПО включает метрики и модели измерений, которые используются для количественной оценки его качества.

При определении требований к ПО задаются соответствующие им внешние характеристики и их подхарактеристики (атрибуты), определяющие разные стороны функционирования и управления продуктом в заданной среде. Для набора характеристик качества ПО, заданных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

Согласно стандарта [1] метрики определяются по модели измерения атрибутов ПО на всех этапах ЖЦ (промежуточная, внутренняя метрика) и особенно на этапе тестирования или функционирования (внешние метрики) продукта.

Остановимся на классификации метрик ПО, правилах для проведения метрического анализа и процесса их измерения.
Типы метрик. Существует три типа метрик:
– метрики программного продукта, которые используются при измерении его характеристик – свойств;
– метрики процесса, которые используются при измерении свойства процесса, используемого для создания продукта.
– метрики использования.

Метрики программного продукта включают:
– внешние метрики, обозначающие свойства продукта, видимые пользователю;
– внутренние метрики, обозначающие свойства, видимые только команде разработчиков.

Внешние метрики продукта включают такие метрики:
– надежности продукта, которые служат для определения числа дефектов;
– функциональности, с помощью которых устанавливается наличие и правильность реализации функций в продукте;
– сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда);
– применимости продукта, которые способствуют определению степени доступности для изучения и использования;
– стоимости, которыми определяется стоимость созданного продукта.

Внутренние метрики продукта включают метрики:
– размера, необходимые для измерения продукта с помощью его внутренних характеристик;
– сложности, необходимые для определения сложности продукта;
– стиля, которые служат для определения подходов и технологий создания отдельных компонент продукта и его документов.

Внутренние метрики позволяют определить производительность продукта и они являются релевантными по отношению к внешним метрикам.

Внешние и внутренние метрики задаются на этапе формирования требований к ПО и являются предметом планирования способов достижения качества конечного программного продукта.

Метрики продукта часто описываются комплексом моделей для установки различных свойств и значений модели качества или для прогнозирования. Измерения проводятся, как правило, после калибровки метрик на ранних этапах проекта. Общей мерой является степень трассируемости, которая определяется числом трасс, прослеживаемых с помощью моделей сценариев (например, UML) и которыми могут быть количество:
– требований;
– сценариев и действующих лиц;
– объектов, включенных в сценарий, и локализация требований к каждому сценарию;
– параметров и операций объекта и др.

Стандарт ISO/IEC 9126–2 определяет следующие типы мер:
– мера размера ПО в разных единицах измерения (число функций, строк в программе, размер дисковой памяти и др.);

– мера времени (функционирования системы, выполнения компонента и др.);
– мера усилий (производительность труда, трудоемкость и др.);
– меры учета (количество ошибок, число отказов, ответов системы и др.).

Специальной мерой может выступать уровень использования повторных компонентов и измеряется как отношение размера продукта, изготовленного из готовых компонентов, к размеру системы в целом. Данная мера используется при определении стоимости и качества ПО. Примерами метрик являются:

– общее число объектов и число повторно используемых;
– общее число операций, повторно используемых и новых операций;
– число классов, наследующих специфические операции;
– число классов, от которых зависит данный класс;
– число пользователей класса или операций и др.

При оценки общего количества некоторых величин часто используются средне статистические метрики (например, среднее число операций в классе, среднее число наследников класса или операций класса и др.).

Как правило, меры в значительной степени являются субъективными и зависят от знаний экспертов, производящих количественные оценки атрибутов компонентов программного продукта.

Примером широко используемых внешних метрик программ являются метрики Холстеда – это характеристики программ, выявляемые на основе статической структуры программы на конкретном языке программирования: число вхождений наиболее часто встречающихся операндов и операторов; длина описания программы как сумма числа вхождений всех операндов и операторов и др.

На основе этих атрибутов можно вычислить время программирования, уровень программы (структурированность и качество) и языка программирования (абстракция средств языка и ориентации на данную проблему) и др.

Метрики процессов включают метрики:
– стоимости, определяющие затраты на создание продукта или на архитектуру проекта с учетом оригинальности, поддержки, документации разработки;
– оценки стоимости работ специалистов за человека–дни либо месяцы;
– ненадежности процесса – число не обнаруженных дефектов при проектировании;
– повторяемости, которые устанавливают степень использования повторных компонентов.

В качестве метрик процесса могут быть время разработки, число ошибок, найденных на этапе тестирования и др. Практически используются следующие метрики процесса:
– общее время разработки и отдельно время для каждой стадии;
– время модификации моделей;
– время выполнения работ на процессе;
– число найденных ошибок при инспектировании;
– стоимость проверки качества;
– стоимость процесса разработки.

Метрики использования служат для измерения степени удовлетворения потребностей пользователя при решении его задач. Они помогают оценить не свойства

самой программы, а результаты ее эксплуатации – эксплуатационное качество. Примером может служить точность и полнота реализации задач пользователя, а также ресурсы (трудозатраты, производительность и др.), потраченные на эффективное решение задач пользователя. Оценка требований пользователя проводится в основном с помощью внешних метрик.

9.1.2. Стандартный метод оценки значений показателей качества

Оценка качества ПО согласно четырех уровневой модели качества начинается с нижнего уровня иерархии, т.е. с самого элементарного свойства оцениваемого атрибута показателя качества согласно установленных мер. На этапе проектирования устанавливают значения оценочных элементов для каждого атрибута показателя анализируемого ПО, включенного в требования.

По определению стандарта ISO/IES 9126–2 метрика качества ПО представляет собой "модель измерения атрибута, связываемого с показателем его качества". Для пользования метриками при измерения показателей качества данный стандарт позволяет определять следующие типы мер:
– меры размера в разных единицах измерения (количество функций, размер программы, объем ресурсов и др.);
– меры времени – периоды реального, процессорного или календарного времени (время функционирования системы, время выполнения компонента, время использования и др.);
– меры усилий – продуктивное время, затраченное на реализацию проекта (производительность труда отдельных участников проекта, коллективная трудоемкость и др.);
– меры интервалов между событиями, например, время между последовательными отказами;
– счетные меры – счетчики для определения количества обнаруженных ошибок, структурной сложности программы, числа несовместимых элементов, числа изменений (например, число обнаруженных отказов и др.).

Метрики качества используются при оценки степени тестируемости после проведения испытаний ПО на множестве тестов (безотказная работа, выполнимость функций, удобство применения интерфейсов пользователей, БД и т.п.).

Наработка на отказ, как атрибут надежности определяет среднее время между появлением угроз, нарушающих безопасность, и обеспечивает трудно измеримую оценку ущерба, которая наносится соответствующими угрозами.

Очень часто оценка программы проводится по числу строк. При сопоставлении двух программ, реализующих одну прикладную задачу предпочтение отдается короткой программе, так как её создает более квалифицированный персонал и в ней меньше скрытых ошибок и легче модифицировать. По стоимости она дороже, хотя времени на отладку и модификацию уходит больше. Т.е. длину программы можно использовать в качестве вспомогательного свойства при сравнении программ с учетом одинаковой квалификации разработчиков, единого стиля разработки и общей среды.

Если в требованиях к ПО было указано получить несколько показателей, то просчитанный после сбора данных при выполнении показатель умножается на

соответствующий весовой коэффициент, а затем суммируются все показатели для получения комплексной оценки уровня качества ПО.

На основе измерения количественных характеристик и проведения экспертизы качественных показателей с применением весовых коэффициентов, нивелирующих разные показатели, вычисляется итоговая оценка качества продукта путем суммирования результатов по отдельным показателям и сравнения их с эталонными показателями ПО (стоимость, время, ресурсы и др.).

Т.е. при проведении оценки отдельного показателя с помощью оценочных элементов просчитывается весомый коэффициент k – метрика, j – показатель, i – атрибут. Например, в качестве j – показателя возьмем переносимость. Этот показатель будет вычисляться по пяти атрибутам ($i = 1, ..., 5$), причем каждый из них будет умножаться на соответствующий коэффициент k_i.

Все метрики j – атрибута суммируются и образуют i – показатель качества. Когда все атрибуты оценены по каждому из показателей качества, производится суммарная оценка отдельного показателя, а потом и интегральная оценка качества с учетом весовых коэффициентов всех показателей ПО.

В конечном итоге результат оценки качества является критерием эффективности и целесообразности применения методов проектирования, инструментальных средств и методик оценивания результатов создания программного продукта на стадиях ЖЦ.

Для изложения оценки значений показателей качества используется стандарт [4] в котором представлены следующие методы: измерительный, регистрационный, расчетный и экспертный (а также комбинации этих методов).

Измерительный метод базируется на использовании измерительных и специальных программных средств для получения информации о характеристиках ПО, например, определение объема, числа строк кода, операторов, количества ветвей в программе, число точек входа (выхода), реактивность и др.

Регистрационный метод используется при подсчете времени, числа сбоев или отказов, начала и конца работы ПО в процессе его выполнения.

Расчетный метод базируется на статистических данных, собранных при проведении испытаний, эксплуатации и сопровождении ПО. Расчетными методами оцениваются показатели надежности, точности, устойчивости, реактивности и др.

Экспертный метод осуществляется группой экспертов – специалистов, компетентных в решении данной задачи или типа ПО. Их оценка базируются на опыте и интуиции, а не на непосредственных результатах расчетов или экспериментов. Этот метод проводится путем просмотра программ, кодов, сопроводительных документов и способствует качественной оценки созданного продукта. Для этого устанавливаются контролируемые признаки, коррелируемые с одним или несколькими показателями качества и включаемые в опросные карты экспертов. Метод применяется при оценке таких показателей как, анализируемость, документируемость, структурированность ПО и др.

Для оценки значений показателей качества в зависимости от особенностей используемых ими свойств, назначения, способов их определения используются шкалы:

– метрическая (1.1 – абсолютная, 1.2 – относительная, 1.3 – интегральная);
– порядковая (ранговая), позволяющая ранжировать характеристики путем сравнения с опорными;
– классификационная, характеризующая только наличие или отсутствие рассматриваемого свойства у оцениваемого программного обеспечения.

Показатели, вычисляемые с помощью метрических шкал, называются количественными, а с помощью порядковых и классификационных – качественными.

Атрибуты программной системы, характеризующие ее качество, измеряются с использованием метрик качества. Метрика определяет меру атрибута, т.е. переменную, которой присваивается значение в результате измерения. Для правильного использования результатов измерений каждая мера идентифицируется шкалой измерений.

Стандарт ISO/IES 9126–2 рекомендует применять 5 видов шкал измерения значений, которые упорядочены от менее строгой к более строгой:
– номинальная шкала отражает категории свойств оцениваемого объекта без их упорядочения;
– порядковая шкала служит для упорядочивания характеристики по возрастанию или убыванию путем сравнения их с базовыми значениями;
– интервальная шкала задает существенные свойства объекта (например, календарная дата);
– относительная шкала задает некоторое значение относительно выбранной единицы;
– абсолютная шкала указывает на фактическое значение величины (например, число ошибок в программе равно 10).

9.1.3. Управление качеством ПС

Под *управлением качества* понимается совокупность организационной структуры и ответственных лиц, а также процедур, процессов и ресурсов для планирования и управления достижением качества ПС. Управление качеством – SQM (Software Quality Management) базируется на применении стандартных положений по гарантии качества – SQA (Software Quality Assurance) [4, 15].

Цель процесса SQA состоит в гарантировании того, что продукты и процессы согласуются с требованиями, соответствуют планам и включает следующие виды деятельности:
– внедрение стандартов и соответствующих процедур разработки ПС на этапах ЖЦ;
– оценка соблюдения положений этих стандартов и процедур.

Гарантия качества состоит в следующем:
– проверка непротиворечивости и выполнимости планов;
– согласование промежуточных рабочих продуктов с плановыми показателями;
– проверка изготовленных продуктов заданным требованиям;
– анализ применяемых процессов на соответствие договору и планам;
– среда и методы разработки согласуются с заказом на разработку;
– проверка принятых метрик продуктов, процессов и приемов их измерения в соответствии с утвержденным стандартом и процедурами измерения.

Цель процесса управления SQM состоит в том, чтобы провести мониторинг (систематический контроль) качества для гарантии, что продукт будет удовлетворять потребителю и предполагает выполнение следующих видов деятельности:

– определение количественных свойств качества, основанных на выявленных и предусмотренных потребностях пользователей;
– управление реализацией поставленных целей для достижения качества.

SQM основывается на гарантии того, что:
– цели достижения требуемого качества установлены для всех рабочих продуктов в контрольных точках продукта;
– определена стратегия достижения качества, метрики, критерии, приемы, требования к процессу измерения и др.;
– определены и выполняются действия, связанные с предоставлением продуктам свойств качества;
– проводится контроль качества (SQA, верификация и валидация) и целей, если они не достигнуты, то проводится регулирование процессов;
– выполняются процессы измерения и оценивании конечного продукта на достижение требуемого качества.

Основные стандартные положения [1–4, 15] по созданию качественного продукта и оценки уровня достигнутого выделяют два процесса обеспечения качества на этапах ЖЦ ПС:

– гарантия (подтверждение) качества ПС, как результат определенной деятельности на каждом этапе ЖЦ с проверкой соответствия системы стандартам и процедурам, ориентированным на достижении качества;
– инженерия качества, как процесс предоставления продуктам ПО свойств функциональности, надежности, сопровождения и других характеристик качества.

Процессы достижения качества предназначены для:
а) управления, разработки и обеспечения гарантий в соответствии с указанными стандартами и процедурами;
б) управления конфигурацией (идентификация, учет состояния и действий по аутентификации), риском и проектом в соответствии со стандартами и процедурами;
в) контроль базовой версии ПС и реализованных в ней характеристик качества.

Выполнение указанных процессов включает такие действия:
– оценка стандартов и процедур, которые выполняются при разработке программ;
– ревизия управления, разработки и обеспечение гарантии качества ПО, а также проектной документации (отчеты, графики разработки, сообщения и др.);
– контроль проведения формальных инспекций и просмотров;
– анализ и контроль проведения приемочного тестирования (испытания) ПС.

Для организации, которая занимается разработкой ПС в том числе из компонентов, инженерия качества ПС должна поддерживаться системой качества, управлением качеством (планирование, учет и контроль).

Инженерия качества включает набор методов и мероприятий, с помощью которых программные продукты проверяются на выполнение требований к качеству и снабжаются характеристиками, предусмотренными в требованиях на ПО.

Система качества (Quality systems – QS) [15] - это набор организационных структур, методик, мероприятий, процессов и ресурсов для осуществления управления качеством. Для обеспечения требуемого уровня качества ПО применяются два подхода. Один из них ориентирован на конечный программный продукт, а второй - на процесс создания продукта.

При подходе, ориентированном на продукт, оценка качества проводится после испытания ПС. Этот подход базируется на предположении, что чем больше обнаружено и устранено ошибок в продукте при испытаниях, тем выше его качество.

При втором подходе предусматриваются и принимаются меры по предотвращению, оперативному выявлению и устранению ошибок, начиная с начальных этапов ЖЦ в соответствии с планом и процедурами обеспечения качества разрабатываемой ПС. Этот подход представлен в серии стандартов ISO 9000 и 9000-1,2,3. Цель стандарта 9000–3 состоит в выдаче рекомендаций организациям-разработчикам создать систему качества по схеме, приведенной на рис.9.3.

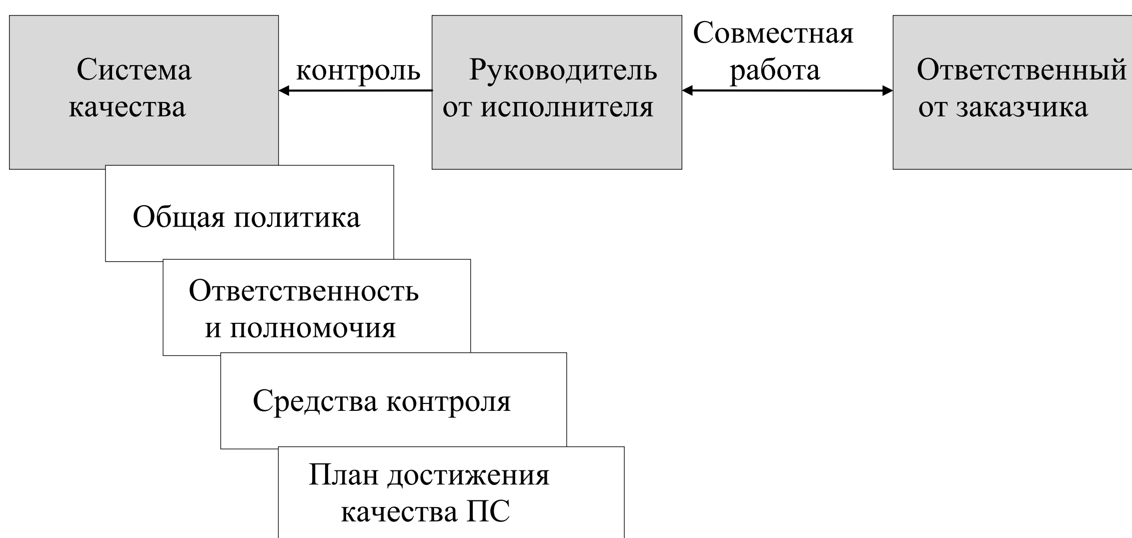

Рис.9.3. Требования стандарта к организации системы качества

Важное место в инженерии качества отводится процессу измерения характеристик процессов ЖЦ, его ресурсов и создаваемых на них рабочих продуктов. Этот процесс реализуются группой качества, верификации и тестирования. В функции этой группы входит: планирование, оперативное управление и обеспечение качества.

Планирование качества представляет собою деятельность, направленную на определение целей и требований к качеству. Оно охватывает идентификацию, установление целей, требований к качеству, классификацию и оценку качества. Составляется календарный план–график для проведения анализа состояния разработки и последовательного измерения спланированных показателей и критериев на этапах ЖЦ.

Оперативное управление включает методы и виды деятельности оперативного характера для текущего управления процессом проектирования, устранения причин неудовлетворительного функционирования ПС.

Обеспечение качества заключается в выполнении и проверки того, что объект разработки выполняет указанные требования к качеству. Цели обеспечения качества могут быть внутренние и внешние. Внутренние цели - создание уверенности у руководителя проекта, что качество обеспечивается. Внешние цели - это создание уверенности у пользователя, что требуемое качество достигнуто и результатом является качественное программное обеспечение.

Как показывает опыт, ряд фирм, выпускающие программную продукцию, имеют системы качества, что обеспечивает им производить конкурентоспособную продукцию. Система качества включает мониторинг спроса выпускаемого нового вида продукции, контроль всех звеньев производства ПС, включая подбор и поставку готовых компонентов системы.

При отсутствии соответствующих служб качества разработчики ПО должны применять собственные нормативные и методические документы, регламентирующим процесс управления качеством ПО для всех категорий разработчиков и пользователей программной продукции.

9.2. Модели оценки надежности

Из всех областей программной инженерии надежность ПС является самой исследованной областью. Ей предшествовала разработка теории надежности технических средств, оказавшая влияние на развитие надежности ПС. Вопросами надежности ПС занимались разработчики ПС, пытаясь разными системными средствами обеспечить надежность, удовлетворяющую заказчика, а также теоретики, которые, изучая природу функционирования ПС, создали математические модели надежности, учитывающие разные аспекты работы ПС (возникновение ошибок, сбоев, отказов и др.) и оценить реальную надежность. В результате надежность ПС сформировалась как самостоятельная теоретическая и прикладная наука [9–13, 17-25].

Надежность сложных ПС существенным образом отличается от надежности аппаратуры. Носители данных (файлы, сервер и т.п.) обладают высокой надежностью, записи на них могут храниться длительное время без разрушения, поскольку разрушению и старению они не подвергаются.

С точки зрения прикладной науки *надежность* – это способность ПС сохранять свои свойства (безотказность, устойчивость и др.) преобразовывать исходные данные в результаты в течение определенного промежутка времени при определенных условиях эксплуатации. Снижение надежности ПС происходит из–за ошибок в требованиях, проектировании и выполнении. Отказы и ошибки зависят от способа производства продукта и появляются в программах при их исполнении на некотором промежутке времени.

Для многих систем (программ и данных) надежность является главной целевой функцией реализации. К некоторым типам систем (реального времени, радарные системы, системы безопасности, медицинское оборудование со встроенными программами и др.) предъявляются высокие требования к надежности, такие как недопустимость ошибок, достоверность, безопасность, защищенность и др.

Таким образом, оценка надежности ПС зависит от числа оставшихся и не устраненных ошибок в программах на этапах ЖЦ В ходе эксплуатации ПС ошибки обнаруживаются и устраняются. Если при исправлении ошибок не вносятся новые, или, по крайней мере, новых ошибок вносится меньше, чем устраняется, то в ходе

эксплуатации надежность ПС непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность системы и соответственно ее качество.

Надежность является функцией от ошибок, оставшихся в ПС после ввода его в эксплуатацию. ПС без ошибок является абсолютно надежным. Но для больших программ абсолютная надежность практически недостижима. Оставшиеся необнаруженные ошибки проявляют себя время от времени при определенных условиях (например, при некоторой совокупности исходных данных) сопровождения и эксплуатации системы.

Для оценки надежности ПС используются такие статистические показатели как вероятность и время безотказной работы, возможность отказа и частота (интенсивность) отказов. Поскольку в качестве причин отказов рассматриваются только ошибки в программе, которые не могут самоустраниться, то ПС следует относить к классу невосстанавливаемых систем.

При каждом проявлении новой ошибки, как правило, проводится ее локализация и исправление. Строго говоря, набранная до этого статистика об отказах теряет свое значение, так как после внесения изменений программа, по существу, является новой программой в отличие от той, которая до этого испытывалась.

В связи с исправлением ошибок надежность, т.е. отдельные ее атрибуты, будут все время изменяться, как правило, в сторону улучшения. Следовательно, их оценка будет носить временный и приближенный характер. Поэтому возникает необходимость в использовании новых свойств, адекватных реальному процессу измерения надежности, таких, как зависимость интенсивности обнаруженных ошибок от числа прогонов программы и зависимость отказов от времени функционирования ПС и т.п.

К факторам гарантии надежности относятся:
– риск, как совокупность угроз, приводящих к неблагоприятным последствиям и ущербу системы или среды;
– угроза, как проявление неустойчивости, нарушающей безопасность системы;
– анализ риска – изучение угрозы или риска, их частота и последствия;
– целостность – способность системы сохранять устойчивость работы и не иметь риска;

Риск преобразует и уменьшает свойства надежности, так как обнаруженные ошибки могут привести к угрозе, если отказы носят частотный характер.

9.1.1. Основные понятия в проблематике надежности ПС

Формально модели оценки надежности ПС базируются на теории надежности и математическом аппарате с допущением некоторых ограничений, влияющих на эту оценку. Главным источником информации, используемой в моделях надежности, является процесс тестирования, эксплуатации ПС и разного вида ситуации, возникающие в них. Ситуации порождаются возникновением ошибок в ПС, требуют их устранения для продолжения тестирования.

Базовые понятия, которые используются в моделях надежности ПС, являются следующие [9–13].

Отказ ПС (failure) – это переход ПС из работающего состояния в нерабочее или когда получаются результаты, которые не соответствуют заданным допустимым значениям. Отказ может быть вызван внешними факторами (изменениями элементов среды эксплуатации) и внутренними – дефектами в самой ПС.

Дефект (fault) в ПС – это последствие использования элемента программы, который может привести к некоторому событию, например, в результате неверной интерпретации этого элемента компьютером (как ошибка (fault) в программе) или человеком (ошибке (error) исполнителя). Дефект является следствием ошибок разработчика на любом из процессов разработки – в описании спецификаций требований, начальных или проектных спецификациях, эксплуатационной документации и т.п. Дефекты в программе, не выявленные в результате проверок, является источником потенциальных ошибок и отказов ПС. Проявление дефекта в виде отказа зависит от того, какой путь будет выполнять специалист, чтобы найти ошибку в коде или во входных данных. Однако не каждый дефект ПС может вызвать отказ или может быть связан с дефектом в ПС или среды. Любой отказ может вызвать аномалию от проявления внешних ошибок и дефектов.
Ошибка (error) может быть следствием недостатка в одном из процессов разработки ПС, который приводит к неправильной интерпретации промежуточной информации, заданной разработчиком или при принятии им неверных решений.

Интенсивность отказов это частота появления отказов или дефектов в ПС при ее тестировании или эксплуатации.

При выявлении отклонения результатов выполнения от ожидаемых во время тестирования или сопровождения, осуществляется поиск, выяснение причин этих отклонений и исправление связанных с этим ошибок.

Модели оценки надежности ПС в качестве входных параметров используют сведения об ошибках, отказах, их интенсивности, собранных в процессе тестирования и эксплуатации.

9.2.2. Классификация моделей надежности

Как известно, что на данный момент времени разработано большое количество моделей надежности ПС и их модификаций. Каждая из этих моделей определяет функцию надежности, которую можно вычислить при задании ей соответствующих данных, собранных во время функционирования ПС. Основными данными являются отказы и время. Другие дополнительные параметры связаны с типом ПС, условиями среды и данных.

В виду большого разнообразия моделей надежности, разработано несколько подходов к классификации этих моделей. Эти подходы в целом основываются на истории ошибок в проверяемой и тестируемой ПС на этапах ЖЦ. Одной из классификаций моделей надежности ПО является классификация Хетча [5, 16]. В ней предлагается разделение моделей на: прогнозирующие, измерительные и оценочные (рис 9.1).

Прогнозирующие модели надежности основаны на измерении технических характеристик создаваемой программы: длина, сложность, число циклов и степень их вложенности, количество ошибок на страницу операторов программы и др. Например, модель Мотли–Брукса основываются на длине и сложности структуры программы

(количество ветвей, циклов, вложенность циклов), количестве и типах переменных, а также интерфейсов. В этих моделях длина программы служит для прогнозирования количества ошибок, например, для 100 операторов программы можно смоделировать интенсивность отказов.

Модель Холстеда дает прогнозирование количества ошибок в программе в зависимости от ее объема и таких данных, как число операций (n_1) и операндов (n_2), а также их общее число (N_1, N_2).

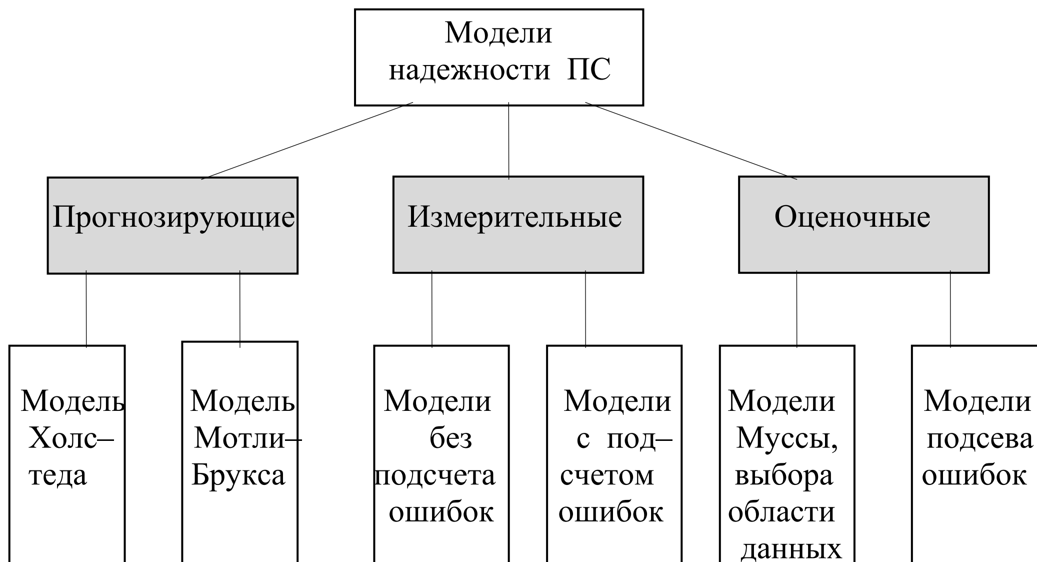

Рис.9.1. Классификация моделей надежности

Время программирования программы им предлагается вычислять по следующей формуле:

$$T = (n_1 N_2 \ (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n_1 / 2 n_2 S,$$

где S – число Страуда (Холстед принял равным 18 – число умственных операций в единицу времени).

Объем вычисляется по формуле:

$V = (2 + n_2^*) \log_2 ((2 + n_2^*)$, где n_2^* – мах число различных операций.

Измерительные модели предназначены для измерения надежности программного обеспечения, работающего с заданной внешней средой и имеющие следующие ограничения:

– программного обеспечения не модифицируется во время периода измерений свойств надежности;

– обнаруженные ошибки не исправляются;

– измерение надежности проводится для зафиксированной конфигурации программного обеспечения.

Типичным примером таких моделей является модель Нельсона и Рамамурти–Бастани и др.

Модель оценки надежности Нельсона основывается на выполнении k–прогонов программы при тестировании и позволяет определить надежность

$R(k) = exp\,[-\sum \nabla t_j\, \lambda(t)]$,

где t_j – время выполнения j–прогона, $\lambda(t) = -[ln\,(1-q_i)\ \nabla j]$ и при $q_i \le 1$ она интерпретируется как интенсивность отказов.

В процессе испытаний программы на тестовых n_l прогонах оценка надежности вычисляется по формуле

$R(l) = 1 - n_l / k$, где k – число прогонов программы.

Таким образом, данная модель рассматривает полученные количественные данные о проведенных прогонах.

Оценочные модели основываются на серии тестовых прогонов и проводятся на этапах тестирования ПС. В тестовой среде определяется вероятность отказа программы при ее выполнении или тестировании.

Эти типы моделей могут применяться на этапах ЖЦ. Кроме того, результаты прогнозирующих моделей могут использоваться как входные данные для оценочной модели. Имеются модели (например, модель Мусы), которые можно рассматривать как оценочную и в тоже время, как измерительную модель [19, 20].

Другой вид классификации моделей предложил Гоэл [21, 22], согласно которой модели надежности базируются на отказах и разбиваются на четыре класса моделей:
– без подсчета ошибок,
– с подсчетом отказов,
– с подсевом ошибок,
– модели с выбором областей входных значений.

Модели без подсчета ошибок основаны на измерении интервала времени между отказами и позволяют спрогнозировать количество ошибок, оставшихся в программе. После каждого отказа оценивается надежность и определяется среднее время до следующего отказа. К такой модели относится модель Джелински и Моранды, Шика Вулвертона и Литвуда–Вералла [23, 24].

Модели с подсчетом отказов базируются на количестве ошибок, обнаруженных на заданных интервалах времени. Возникновение отказов в зависимости от времени является стохастическим процессом с непрерывной интенсивностью, а количество отказов является случайной величиной. Обнаруженные ошибки устраняются и поэтому количество ошибок в единицу времени уменьшается. К этому классу моделей относится модель Шумана, Шика–Вулвертона, Пуассоновская модель и др.[24–,27]

Модели с подсевом ошибок основаны на количестве устраненных ошибок и подсеве внесенном в программу искусственных ошибок, тип и количество которых заранее известны. Затем строиться соотношение числа оставшихся прогнозируемых ошибок к числу искусственных ошибок, которое сравнивается с соотношением числа обнаруженных действительных ошибок к числу обнаруженных искусственных ошибок. Результат сравнения используется для оценки надежности и качества программы. При внесении изменений в программу проводится повторное тестирование и оценка надежности. Этот подход к организации тестирования отличается громоздкостью и редко используется из–за дополнительного объема работ, связанных с подбором, выполнением и устранением искусственных ошибок.

Модели с выбором области входных значений основываются на генерации множества тестовых выборок из входного распределения и оценка надежности, проводится по полученным отказам на основе тестовых выборок из входной области. К этому типу моделей относится модель Нельсона и др.

Таким образом, классификация моделей роста надежности относительно процесса выявления отказов, фактически разделена на две группы:
модели, которые рассматривают количество отказов как Марковский процесс;
модели, которые рассматривают интенсивность отказов как Пуассоновский процесс.

Фактор распределения интенсивности отказов разделяет модели на экспоненциальные, логарифмические, геометрические, байесовские и др.

9.2.3. Модели надежности Марковского и Пуассоновского типов

Марковский процесс характеризуется дискретным временем и конечным множеством состояний. Временной параметр пробегает неотрицательные числовые значения, а процесс (цепочка) определяется набором вероятностей перехода p_{ij} (n), т.е. вероятностью на n– шаге перейти из состояния i в состояние j. Процесс называется однородным, если он не зависит от n.

В моделях, базирующихся на процессе Маркова, предполагается, что количество дефектов, обнаруженных в ПС, в любой момент времени зависит от поведения системы и представляется в виде стационарной цепи Маркова [13, 17, 18]. При этом количество дефектов конечное, но является неизвестной величиной, которая задается в для модели виде константы. Интенсивность отказов в ПС или скорость прохода по цепи зависит *лишь от количества дефектов,* которые остались в ПС.

К этой группе моделей относятся: Джелинського–Моранды [23], Шика–Вулвертона [24], Шантикумера [24] и др.

Ниже рассматриваются некоторые модели надежности, которые обеспечивают рост надежности ПО (называются моделями роста надежности [17]), находят широкое применение на этапе тестирования и описывают процесс обнаружения отказов при следующих основных предположениях:

– все ошибки в ПС не зависят друг от друга с точки зрения локализации отказов;
– интенсивность отказов пропорциональна текущему числу ошибок в ПС (убывает при тестировании программного обеспечения);
– вероятность локализации отказов остается постоянной;
– локализованные ошибки устраняются до того, как тестирование будет продолжено;
– при устранении ошибок новые ошибки не вносятся.
Приведем основные обозначения используемых величин при описании моделей роста надежности:

m – число обнаруженных отказов ПО за время тестирования;
X_i – интервалы времени между отказами $i–1$ и i, при $i =1,..., m;$
S_i – моменты времени отказов (длительность тестирования до i– отказа), $S_i = X_k$ при $i = 1,..., m$;
T – продолжительность тестирования программного обеспечения (время, для которого определяется надежность);
N – оценка числа ошибок в ПО в начале тестирования;

M – оценка числа прогнозированных ошибок;
MT – оценка среднего времени до следующего отказа;
$E\ (T_p)$ – оценка среднего времени до завершения тестирования;
$Var\ (T_p)$ – оценка дисперсии;
$R\ (t)$ – функция надежности ПО;
$Zi\ (t)$ – функция риска в момент времени t между i–1 и i отказами;
c – коэффициент пропорциональности;
b – частота обнаружения ошибок.

Далее рассматриваются несколько моделей роста надежности, основанные на этих предположениях и использовании результатов тестирования программ в части отказов, времени между ними и др.

Модель Джелински–Моранды. В этой модели используются исходные данные, приведенные выше, а также:
m – число обнаруженных отказов за время тестирования,
X_i – интервалы времени между отказами,
T – продолжительность тестирования.
Функция риска $Z_i\ (t)$ в момент времени *t* расположена между *i–1* и *i* имеет вид:

$$Z_i\ (t) = c\ (N - n_{i-1}),$$

где $i = 1, .., m; \quad T_{i-1} < t < T_i$.

Эта функция считается ступенчатой кусочно–постоянной функцией с постоянным коэффициентом пропорциональности и величиной ступени – с.

Оценка параметров с и N производится с помощью системы уравнений:

$$\sum_{i=1}^{m} 1 / (N - n_{i-1}) - \sum_{i=1}^{m} c\ X_i = 0$$

$$n/c - NT - \sum_{i=1}^{m} X_i\ n_{i-1} = 0$$

При этом суммарное время тестирования вычисляется так: $T = \sum_{i=1}^{m} X_i$

Выходные показатели для оценки надежности относительно указанного времени *T* включают:
– число оставшихся ошибок $M_m = N–m;$
– среднее время до текущего отказа $MT_m = 1/(N–m)\ c$;
– среднее время до завершения тестирования и его дисперсия

$$E\ (T_p) = \sum_{i=1}^{N-n} (1/ ic),$$

$$Var\ (T_p) = \sum_{i=1}^{N-n} 1/(ic)^2$$

При этом функция надежности вычисляется по формуле:

$$Rm\ (t) = exp\ (-(N- m)\ ct),$$

при $t>0$ и числе ошибок, найденных и исправленных на каждом интервале тестирования, равным единице.

Модель Шика–Волвертона. Модель используется тогда, когда интенсивность отказов пропорциональна не только текущему числу ошибок, но и времени, прошедшему с момента последнего отказа. Исходные данные для этой модели аналогичны выше рассмотренной модели Джелински–Моранды:

m – число обнаруженных отказов за время тестирования,

X_i – интервалы времени между отказами,

T – продолжительность тестирования.

Функции риска $Z_i(t)$ в момент времени между $i–1$ и $i–m$ отказами определяются следующим образом:

$$Z_i(t) = c\,(N - n_{i-1}),\ где\quad i = 1,\dots, m;\quad T_{i-1} < t < T_i,$$

$$T = \sum_{i=1}^{m} X_i\ .$$

Эта функция является линейной внутри каждого интервала времени между отказами, возрастает с меньшим углом наклона.

Оценка c и N вычисляется из системы уравнений:

$$\sum_{i=1}^{m} 1/(N - n_{i-1}) - \sum_{i=1}^{m} X_i^{\,2}/2 = 0$$

$$n/c\ - \sum_{i=1}^{m} (N - n_{i-1})\, X_i^{\,2}/2\ = 0$$

К выходным показателям надежности относительно продолжительности T относятся:

– число оставшихся ошибок $Mm = N–m$;

– среднее время до следующего отказа $MTm = (\pi / (2\,(N–m)\,c))^{1/2}$;

– среднее время до завершения тестирования и его дисперсия

$$E(T_p) = \sum_{i=1}^{N-m} (\pi / (2\,i\,c))^{1/2}\ ,$$

$$Var(Tp) = \sum_{i=1}^{N-m} ((2 - \pi/2) /\ i\,c\,);$$

Функция надежности вычисляется по формуле:

$$R_T(t) = exp\,(-(N–m)\,ct^2/2)\ ,\ t \ge 0.$$

Модель Пуассоновского типа базируется на выявлении отказов и моделируется неоднородным процессом, который задает $\{M(t),\ t\ge 0\}$ – неоднородный пуассоновский процесс с функцией интенсивности $\lambda(t)$, что соответствует общему количеству отказов ПС за время его использования t.

Модель Гоело–Окумото. В основе этой модели лежит описание процесса обнаружения ошибок с помощью неоднородного Пуассоновского процесса, ее можно рассматривать как модель экспоненциального роста. В этой модели интенсивность отказов также зависит от времени. Кроме того, в ней количество выявленных ошибок трактуется как случайная величина, значение которой зависит от теста и других условных факторов.

Исходные данные этой модели:

m – число обнаруженных отказов за время тестирования,

X_i – интервалы времени между отказами,

T – продолжительность тестирования.

Функция среднего числа отказов, обнаруженных к моменту t имеет вид:

$$m(t) = N(1 - e^{-bt}),$$

где b – интенсивность обнаружения отказов и показатель роста надежности $q(t) = b$.

Функция интенсивности $\lambda(t)$ в зависимости от времени работы до отказа равна

$$\lambda(t) = Nb^{-bt}, \quad t \geq 0.$$

Оценка b и N получаются из решения уравнений:

$$m/N - 1 + exp(-bT) = 0$$

$$m/b - \sum_{i=1}^{m} t_i - N_m \, exp(-bT) = 0$$

Выходные показатели надежности относительно времени T определяют:

1) среднее число ошибок, которые были обнаружены в интервале [0, T]

$$E(N_T) = N\, exp(-bT),$$

2) функцию надежности

$$R_T(t) = exp(-N(e^{-bt} - e^{-bt(t+m)})), \quad t \geq 0.$$

В этой модели обнаружение ошибки, трактуется как случайная величина, значение которой зависит от теста и операционной среды. В других моделях количество обнаруженных ошибок рассматривается как константа.

В моделях роста надежности исходной информацией для расчета надежности являются интервалы времени между отказами тестируемой программы, число отказов и время, для которого определяется надежность программы при отказе. На основании этой информации по моделям определяются следующие показатели надежности:

– вероятность безотказной работы;
– среднее время до следующего отказа;
– число необнаруженных отказов (ошибок);
– среднее время для дополнительного тестирования программы.

Модель анализа результатов прогона тестов использует в своих расчетах общее число экспериментов тестирования и число отказов. Эта модель определяет только вероятность безотказной работы программы и выбрана для случаев, когда предыдущие модели нельзя использовать (мало данных, некорректность вычислений). Формула определения вероятности безотказной работы по числу проведенных экспериментов имеет вид:

$$P = 1 - Nex/N,$$

где Nex – число ошибочных экспериментов, N – число проведенных экспериментов для проверки работы ПС.

Таким образом, можно сделать вывод о том, что модели надежности ПС основаны на времени функционирования и/или количестве отказов (ошибок), полученных в

программах в процессе их тестирования или эксплуатации. Модели надежности учитывает случайный Марковский и пуассоновский характер соответственно процессов обнаружения ошибок в программах, а также характер и интенсивность отказов.

Контрольные вопросы и задания

1. Определите понятие – качество ПО.
2. Назовите основные аспекты и уровни модели качества ПО.
3. Определите характеристики качества ПО и их назначение.
4. Какие методы используются при определении показателей качества?
5. Определите метрики программного продукта и их составляющие.
6. Какие стандарты в области качества ПО существуют?
7. Назовите основные цели и задачи системы управления качеством.
8. В чем суть инженерии качества?
9. Назовите содержание классификации моделей надежности.
10. Определите типы моделей надежности и их базис.
11. Какие данные необходимы для оценивания надежности ПО?

Литература к теме 9.

1. *ISO/IEC* 9126. Infofmation Technology. – Software Quality Characteristics and metrics. – 1997.
2. *ДСТУ* 2844–1994. Программные средства ЭВМ. Обеспечение качества. Термины и определения..
3. *ДСТУ* 2850–1994. Программные средства ЭВМ. Обеспечение качества. Показатели и методы оценки качества программного обеспечения.
4. *ДСТУ* 3230–1995. Управление качества и обеспечение качества. Термины и определения.
5. Haag S., Raja H.K., Sekade L.L. Quality Function Deployment. Usage in Software Development// Comm. of ACM.– 1998. –39. –N1.
6. *Кулаков А.Ю*. Оценка качества программ ЭВМ .–Киев: Техніка.–1984.–167с.
7. *Липаев В.В.* Методы обеспечения качества крупномасштабных программных систем. – М.: СИНТЕГ.– 2003.–510 с.
8. *Андон Ф.И., Суслов В.Ю., Коваль Г.И., Коротун Т.М.* Основы качества программных систем.–Киев, Академпериодика.– 2002.–502с.
9. *Липаев В.В.* Надежность программного обеспечения АСУ.–М.: Сов.радио, 1977.–400с.
10. *Майерс Г.* Надежность программного обеспечения .– М.: Мир, 1980.–360с.
11. *Гласс Г.* Руководство по надежному программированию.–М.: Финансы и Статистика, 1982.–256с.
12. *Тейер Т., Липов Р., Нельсон Э.* Надежность программного обеспечения.–М.: Мир, 1981.– 325с.
13. *Барлоу Р., Прошан Ф.* Математическая теория надежности. Пер.с анг. М.: 1969.–483с.
14. *Meyer B*. The role of Object–Oriented Metrics.– Computer, 1998. –№11.–p.23–125.
15. *NASA* –STD–2201/ Software Assurance Standart, 1993.
16. *ISO* 14598. Information Technology – Software product evolution– Part1: General overview.–1996.
17. *Мороз Г.Б., Лаврищева Е.М.* Модели роста надежности программного обеспечения.– Киев.–Препринт 92–38, 1992.– 23с.

18. *Коваль Г.И.* Подход к прогнозированию надежности ПО при управлении проектом // Проблемы программирования. –2002. – № 1 – 2. – С. 282 – 290.
19. *John D. Musa,* Anthony Iannino, and Kazuhira Okumoto. Software Reliability: Measurement, Prediction, Application. Whippany, NJ: McGraw–Hill, 1987.
20. *Musa J.D. Okumoto K.A*. Logarithmic Poisson Time Model for Software Reliability Measurement //Proc. Sevent International Conference on Software Engineering. – Orlando, Florida. – 1984. –P. 230–238.
21.*Goel A.L.* Software reliability models& Assumptions, Limitations and Applicability// IEEE Trans.– N2.–p.1411–1423.
22. *Sukert A.N., Goel A.L.* A guidebook for software reliability assessment /Proc. Annual Reliability and Maintainability Symp. – Tokio (Japan). – 1980. –P. 186 – 190.
23. *Jelinski Z., Moranda P.* Software reliability research /Statistical computer performance evaluation W.Freiberger, Ed. Academic Press. –1972. – P. 465–484.20.
24. *Shick G.J., Wolverton R.W.* An analysis of computing software reliability models /IEEE Tras. Software Eng. – V. SE–4. – № 2. – 1978. P. 104–120.
25. *Yamada S., Ohba M., Osaki S.* S–shaped software reliability grows modeling for software error detection / // IEEE Trans. Reliability. – 1983. – R–32. – № 5. – P. 475–478.
26. *Schneidewind N.F.* Software Reliability Model with Optimal Selection of Failure Data //IEEE Trans. on Software Eng. – 1993. – № 11. –P. 1095–1104.
27. *Shanthikumar J.G.* Software reliability models: A Review //Microelectron. Reliab. – 1983. –V. 23. –№ 5 – P. 903–943.

Тема 10
МЕТОДЫ УПРАВЛЕНИЯ ПРОЕКТОМ, РИСКОМ И КОНФИГУРАЦИЕЙ

Инженерное программирование охватывает процессы планирования и управления проектом, а также связанными с ними процессы оценивания заданных сроков и стоимости на предмет распределения всех ресурсов таким образом, чтобы успешно выполнить проект, а также выявлять и управлять возникающие риски, как в подборе исполнителей так и в использовании инструментальных средств для постепенного изготовления версий системы.

Материал данной темы состоит из трех разделов:
1. Методы управления программным проектом.
2. Методы обнаружения и устранения рисков в проекте.
3.Управление конфигурацией системы.

10.1. Методы управления проектами

Согласно мировой статистики, не все реализуемые программные проекты завершаются успешно, 33% из них являются провальными по следующим причинам:
– требования заказчика не выполняются ,
– проект не вложился в стоимость,
– проект не вложился в заданные сроки,
– этапы работ оказались нескординированными друг с другом,
– менеджер не ориентирует разработчиков на применение новейших методов и средств программирования, планирования и соблюдение стандартных соглашений на применение модели ЖЦ.

Основные положения управления проектом, задачи и методы которого отрабатывались на технических проектах (например, первый проект разработки лайнера для перевозки пассажиров из Европы в Америку), привели к тому, что Генри Гант впервые предложил диаграммную схему учета времени выполнения проекта. На сегодня эти задачи сформулированы следующим образом:
– планирование проекта и составление графиков работ выполнения проекта,
– управление проектными работами и командой исполнителей,
– управление рисками,
– оценивание продукта и используемых процессов в целях усовершенствования и др..

Управление проектом – это руководство работами команды исполнителей проекта для реализации проекта с использованием общих методов управления, планирования и контроля работ (видение будущего продукта, стартовые операции, планирование итераций, мониторинг и отчетность), планирование и управление рисками, эффективной организацией работы команды и коммуникационными потоками в команде исполнителей.

Основными составляющими любого проекта являются следующие: **ресурсы** (людские, финансовые и технические) **время и стоимость** выполнения проекта**.** Ответственность за координацию и реализацию этих трех составляющих несет менеджер проекта, а ответственность за идейную, функциональную сторону проекта несет главный специалист (в программном проекте – главный программист).

Успешное выполнение проекта зависит от уровня знаний методов управления менеджером проекта. Если он прошел школу управления техническим проектом и

взялся за реализацию программного проекта, как правило, то ему необходимо учитывать такие особенности программного продукта:

– этот продукт не материален, его нельзя увидеть в процессе конструирования (как это имеет место, например, при строительстве здания) и повлиять на его реализацию более оперативно;

– стандарты ЖЦ прямо не ориентированы на нужный вид продукта, как это имеет место в технических дисциплинах (автомобильной, авиационной и др.), их необходимо адаптировать к виду и типу проекта и разработать методики их выполнения исполнителями;

– программные продукты создают длительное время на компьютерной технике, которая быстро устаревает в виду постоянного обновления архитектуры.
– элементной базы и языков программирования.

Эта концепция обеспечивает концентрацию внимания менеджера на критических роботах. Однако, основным преимуществом метода критического пути есть возможность управления сроками выполнение задач, которые не лежат на критическом пути. Этот метод разрешает рассчитать возможные календарные графики выполнения комплекса работ на основе описанной логической структуры сети и оценок времени выполнения выполнение каждой работы [1–4].

Накопленный опыт в создании технических проектов был систематизирован (в институте управления проектами в США) и разработано ядро знаний – PMBOK (Project Management Body of Knowledge [2]. В нем малыми проектами считаются проекты, содержащие 100 работ и 15 исполнителей, средними – 500 работ и 50 исполнителей и большими – 1000 работ и 100 исполнителей.

В ядре PMBOK определены основные аспекты разработки проектов:

– методы управления, планирования и контроля работ;
– эффективная организация проектной группы (команды);
– инструментарий менеджера проекта (например, система Project Management фирмы Microsoft).

10.1.1. Методы управления программным проектом

Сложилось несколько методов, эффективно применяемых в практике реализации программных проектов. Рассмотрим их.

10.1.1.1. Метод критического пути CPM

Основополагающим моментом создания этого метода является исследование возможности эффективного использования вычислительной машины Univac на фирме "Dupon" при планировании и создании планов-графиков больших комплексов работ по модернизации заводов этой фирмы. В результате был создан рациональный и простой метод (Уолкера – Келли) управления проектом с использованием ЭВМ, который был назван CPM (Critical Path Method) методом критического пути.

Критический путь - наиболее длинный полный путь в сети; работы, которые лежат на этом пути, также называются критическими. Именно продолжительность критического пути определяет наименьшую общую продолжительность работ из проекта в целом. Время выполнения всего проекта в целому может быть сокращен за счет сокращения времени выполнения задач, которые лежат на критическом пути. Соответственно, любая задержка выполнения задач критического пути приводит к увеличению времени

выполнения проекта. Эта концепция обеспечивает концентрацию внимания менеджера на критических роботах. Однако, основным преимуществом метода критического пути есть возможность управления сроками выполнение задач, которые не лежат на критическом пути. Этот метод разрешает рассчитать возможные календарные графики выполнения комплекса работ на основе описанной логической структуры сети и оценок времени выполнения выполнение каждой работы [1–6].

Метод основан на графическом представлении задач (работ) и видов действий на проекте и заданием ориентировочного времени их выполнения. Это представление задается в виде графа (рис 10.1), в вершинах которого располагаются работы и время выполнения каждой работы под вершинами либо на дугах графа. Граф целесообразно строить тогда, когда работы и время их выполнения являются заданными (определенными).

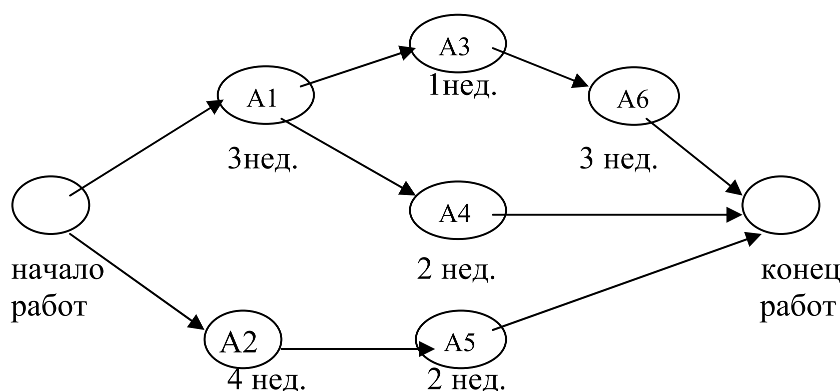

Рис.10.1. Граф задания сроков выполнения работ

Критический путь в графе указывает максимальную продолжительность работ на графе (от начальной работы до последней). При выполнении проекта выбираются и выполняются работы, которые не влияют на время выполнения других (независимых) работ проекта или на их продолжительность. Работы на критическом пути могут сокращаться за счет изменения времени выполнения.

Представление в таком виде работ называется *сетевая диаграммой* и служит для графическое отображение работ проекта, их взаимосвязей, последовательностей и времени выполнения. В графе вершины отображают работы, а линии взаимные связи между работами. Этот граф является наиболее распространенным представлением сети на сегодняшний день.

10.1.1.2. Метод анализа и оценки PERT

Параллельно с разработкой CPM, в военно-морских силах США был создан (фирма "Буз, Аллен & Гамильтон") метод анализа и оценки программ PERT (Program Evaluation and Review Technique) для реализации проекта разработки ракетной системы "Polaris", объединяющей около 3800 подрядчиков с числом операций более 60 тыс. [6].

Применение метода PERT позволяло точно знать руководству данной программы, что нужно делать в каждый момент времени и какой исполнитель это делает, а также определять вероятность своевременного завершения отдельных операций. Руководство программой создания ракетной системы оказалось настолько успешным, что проект удалось завершить на два года раньше запланированного срока.

Метод PERT представляется сетевыми диаграммами с вершинами-событиями, а работа – в виде линии между двумя событиями, отображающими начало и конец данной работы. В целом расхождения между этими двумя методами сетевого представления графа работ являются незначительными. Однако этот метод, в отличие от CMP, учитывает возникающие неопределенности во времени выполнения каждой операции.

Представление более сложных связей между работами для задания узлов графа в виде вершина-событие является более сложным и потому этот метод реже используется на практике.

В этом методе возможное время выполнения операций оценивается с помощью трех оценок:

– оптимистичной (О),

– пессимистической (Р),

– вероятностной (В).

А далее вычисляется по формуле: (О+4В+П)/6, с указанием его на сетевом графике.

Для уменьшения провалов проектов используются и другие аналитические методы и усовершенствованные языки разработки систем. Как показывает опыт, "продвинутые" технологии решают вопросы успешного создания проекта с учетом факторов, которые уменьшают его провал [3]:
1. Проект начинать с правильного шага.
2. Поддержка темпа работы.
3. Обеспечение прогресса и правильных решений.
4. Посмертный анализ завершенного проекта.

1). Проект начинать с правильного шага. К правильному шагу относится формирование команды разработчиков из хороших специалистов, в том числе не более 20% звезд, наиболее приспособленных для хорошей работы над проектом (много звезд - создание конфликтов). В команду входят надежные разработчики с совместимыми характерами и рабочими привычками. Звезды решают сложные вопросы, разрабатывают более ответственные алгоритмы и проводят техническое обучение остальных членов команды. Для уточнения всех особенностей проекта разработчики садятся за стол переговоров с заказчиком и составляют взаимно приемлемый документ.

Следующий шаг - это создание среды для эффективной работы и уменьшения вероятности возникновения конфликтов. Продуктивной среда для работы команды являются: белые стены мест встреч для формальных и неформальных переговоров, личные рабочие места и современные лицензионные компьютерные средства.
2). Поддержка темпа работы состоит в:
– уменьшении текучести кадров;
– контроле качества выполняемых работ;
– управлении процессом разработки, а не людьми.

Текучесть кадров – проблема программной индустрии, так как перемещение или уход отдельного специалиста, вынуждает других членов группы к трудоемкой работе,

связанной с изучением незаконченных и не полностью документированных программ. Большой промежуток времени между уходом специалиста и нахождением замены может привести к краху отдельных частей проекта.

Контроль качества необходимое условие создания качественного проекта. Им занимаются с самого начала разработки проекта, устанавливая процедуры проверки качества, а также используя разработчиков, которые могут создавать высококачественный продукт.

Управление процессом разработки. Исполнители должны пользоваться свободой для прихода на работу в нефиксированное время и в нестрогой одежде. Требуется управлять разработкой, подвергая критике результаты труда, а не режим работы.

3. Поддержка прогресса и правильных решений. Программа отличается от других продуктов тем, что ее нельзя пощупать. Разработка программы начинается с концептуальной модели, а результаты ее применения влияют на получение продукта. Для поддержки прогресса, как правило, выбирается технологии с необходимым экономическим и технологическим анализом, который отвечает требованиям рынка и важности компании.

4. Посмертный анализ. Это изучение своих ошибок при реализации предыдущего проекта, чтобы не повторить их в новом проекте. Так как каждая команда и фирма имеет свои особенности, которые влияют на процесс разработки, то при анализе можно выделить закономерности , которые влияют на получение результатов.

10.1.2. Планирование проекта

Планирование представляет собой процесс распределения и назначения ресурсов (материальных и людских) с учетом стоимости и времени выполнения проекта. Планирования являются необходимой предпосылкой выполнения любой, даже самой простой задачи. Неадекватное планирование может привести к срыву проекта или к получению в среде проекта неадекватных результатов.

Планирование и перепланирование представляет собой наиболее емкостную во времени часть управления проектом, в особенности на ранних стадиях проекта. В прошлом, проекты в области программных систем не имели планов и оценок их следования и выполнения. Современные методики предлагают средства для исправления этой ситуации, предоставляя в распоряжение менеджеров инструменты и методы, которые разрешают работать в направлении достижения реальных и достижимых оценок работ и планов.

Планирование в том или ином виде выполняется в течение всего срока реализации проекта. В начале ЖЦ проекта обычно разрабатывается неофициальный первоначальный план - грубое представление о том, что потребуется выполнить в случае реализации проекта. Решение о выборе проекта в значительной мере базируется на оценках этого первоначального плана. Формальное и детальное планирование проекта начинается после принятия решения о его реализации.

Планирование заключается в составлении следующих планов:
– работ со сроками их выполнения по методу критического пути CPM или PERT;
– достижения требуемого качества методами проверки промежуточных результатов процессов ЖЦ;

– управление рисками,
– аттестации результатов проектирования и деятельности исполнителей проекта,
– управление конфигурацией и др.

Составляется график работ по следующей схеме (рис.10.2):

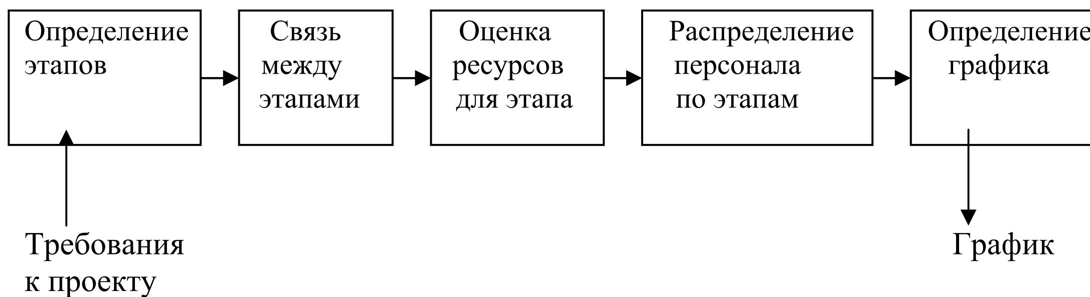

Рис.10.2. Шаги составления графика работ на проекте

При планировании по методу PERT событие или дата в плане является некоторой вехой осуществления отдельных работ проекта. Веха используется для отображения (отметки) состояния завершения тех или иных работ. В контексте проекта менеджеры используют вехи для того, чтобы обозначить важные промежуточные результаты, которые должны быть достигнуты в процессе реализации проекта. Последовательность вех, определенных менеджером, часто называется *планом по вехам или по событиям*. Определение плана достижения соответствующих вех образуют календарный план на основе вех.

На этапе планирования могут использоваться также сетевая разбивка работ (СРР) и диаграммы Ганта [7].

СРР представляет собой иерархическая структура последовательной декомпозиции задач проекта на подзадачи. На нижнем уровне располагаются работы, которые являются детализированным элементами деятельности, они отображаются в сетевой модели СРР, как в иерархической структуре и помогают провести:

– структуризацию работ на основные компоненты и подкомпоненты,
– определить направления деятельности для достижения комплекса целей,
– распределить ответственных за выполнение отдельных работ на проектеа,
– получить отчетность и обобщение информации по проекту.

План содержит описание цикла разработки ПС по фазам, состояниям проекта и представлением каждой из них в терминах отдельных задач или деятельностей. В плане отражаются связи между процессами и определяется интервал времени для выполнения каждой деятельности, а также время ее начала и завершения. В план работ включается также описание разных видов демонстраций заказчику: функций, подсистем, надежности, защиты и др. К документам плана относятся: комплект руководства пользователя для выполнения заданного набора операций, коммуникации системы с другими подсистемами и др.

Диаграмма Ганта - горизонтальная линейная диаграмма, на которой задачи проекта представляются сроками в виде отрезков времени и имеют даты начала и окончания, возможно с задержками и другими временными параметрами.

План в виде графа СРР имеет фазы, шаги и деятельности, а также начало и конечную деятельность на процессе (рис.10.3).

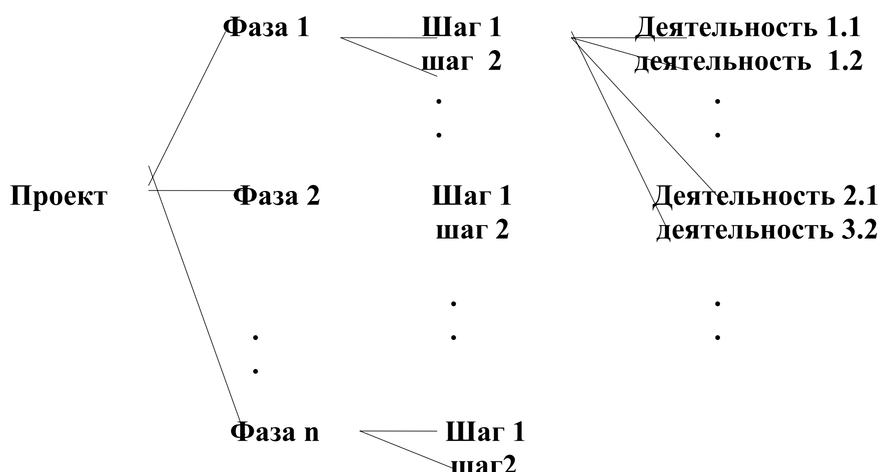

Рис 10.3. Пошаговый граф плана проекта

Каждая фаза описывается с помощью параметров:

– начальная точка выполнения процесса,
– продолжительность,
– срок,
– результат (конечная точка).

При построении сетевого графика работ создается граф (рис.10.4), в котором указываются:
начальная точка – событие или набор событий, которые произошли до начала выполнения данной фазы процесса, и для которой описывается набор условий начала процесса;
продолжительность – интервал времени, за которое процесс должен успешно завершить свое выполнение;
срок – дата, до которой процесс полностью или частично завершает свое выполнение;
конечная точка процесса – контрольная точка, в которой заказчик проверяет качество полученных результатов процесса.

Дуге, выходящей из начальной вершины и входящей в заключительную вершину, соответствует временная пометка 0. С помощью этих меток задается время выполнения процесса.

В графе могут присутствовать циклические пути. По графу проводят анализ критических путей, т.е. определяют данные о продолжительности каждого процесса. План проекта проводится в терминах этапов: планирование, проектирование, кодирование, тестирование и сопровождение. Для первых двух методов планирование затрагивает: определение спецификаций, бюджета и расписания, а также развития плана проекта в целом.
В графе могут присутствовать циклические пути. По графу проводят анализ критических путей, т.е. определяют данные о продолжительности каждого процесса. План проекта проводится в терминах этапов: планирование, проектирование, кодирование, тестирование и сопровождение. Для первых двух методов планирование

затрагивает: определение спецификаций, бюджета и расписания, а также развития плана проекта в целом.

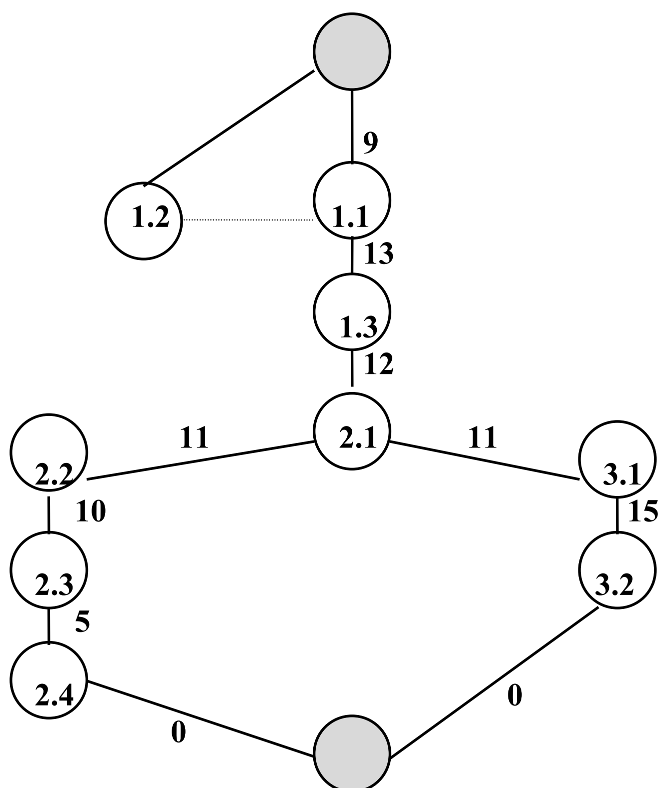

Рис.10.4. Граф работ и сроков (на дугах) для проекта

В настоящее время значительная часть современных средств проектирования нацелена на визуализацию структуры проекта и процессов деятельности, которые на иконке могут быть выделены цветом или иконкой, это позволяет увидеть, какие виды деятельности должны выполняться параллельно, и какие находятся на критическом пути (наприер в системе Project Management).

10.1.3. Организационные аспекты управления в проекте

Распределение работ по ролям. Наиболее часто определение ролей исполнителей проекта соответствует этапам разработки. В графе могут присутствовать циклические пути. По графу проводят анализ критических путей, т.е. определяют данные о продолжительности каждого процесса. План проекта проводится в терминах этапов: планирование, проектирование, кодирование, тестирование и сопровождение. Для первых двух методов планирование затрагивает: определение спецификаций, бюджета и расписания, а также развития плана проекта в целом.

Состав и количество сотрудников, входящих в группу проекта, зависит от масштаба работ и опыта сотрудников. Сотрудники должны быть настолько квалифицированными, что могут выявить ошибки и неточности в проекте на самых ранних стадиях ведения разработки. Разделение труда сотрудников по этапам имеет свои определенные преимущества, но требует специальной техники общения между группами сотрудников для эффективной работы (проверки, просмотры, откаты назад, сквозной контроль).

Специалисты, которые наиболее подходят к выполнению каждой из перечисленных ролей, различаются между собой:

- способностью выполнять работу;
- интересом к работе;
- опытом работы с подобным проектом, инструментами, языками, технологиями и ОС;
- способностью к обучению;
- коммуникабельностью с другими сотрудниками;
- способностью разделить ответственность с другими;
- профессионализмом и знанием методов управления.

Менеджер проекта должен знать способности того или иного сотрудника выполнить определенную работу по проектированию или по тестированию системы в целом. Работающие в одной группе должны разделять одни и те же взгляды по проведению порученной им работы и пользоваться одним стилем программирования. Разделение большого участка работы на меньшие части должно соответствовать фрагментам работы, определению ролей и ответственности каждого сотрудника в проекте.

Стиль работы. Разным людям свойственны разные стили выполнения работы и коммуникации с другими сотрудниками [3]. Отличаются сотрудники тем, что одни сначала взвешивают все детали и собирают всю информацию, а потом принимают решения по всем вопросам. Другие разбивают работу на фрагменты и принимают решение постепенно для каждого фрагмента. Некоторые сотрудники предпочитают формировать рабочее решение путем высказывания своего мнения другим сотрудниками и принимать окончательное решение сообща (стиль *экстраверта*), другие предпочитают и интересоваться мнением других по тому или другому вопросу, а потом самостоятельно принимать решение *(интроверты*). Некоторые сотрудники полагаются на свою интуиции и профессиональный опыт при принятии решения (*интуитивисты)*, другие – руководствуются только рациональными и логическими доводами (*рационалисты*). В реальной рабочей среде более часто встречаются смешанные типы сотрудников.

Рациональный экстраверт считается хорошим руководителем. Он стремится обсудить проблему, но не позволяет влиять на принятие окончательного решения. Стиль его работы – спрашивать у своих подчиненных то, что касается главной линии ведения проекта (их не интересуют подробности, частности, детали документации и т. д.).

Рациональный интроверт избегает эмоциональных обсуждений, ему необходимо время, чтобы обдумать все возможные пути решения проблемы и просчитать все шаги. Он тщательно взвешивают все «за и против», собирая все факты. Его репутация хорошего работника для него очень важна, он считает, что работа должна занимать большую часть времени и требует этого от других. Он аккуратен и точен.

Интуитивный экстроверт часто принимает решение на эмоциональной почве. Стремится больше рассказать о себе и своих планах, чем выслушать других. Часто базируется на предыдущем опыте работы, по натуре он испытатель. Ему важно, чтобы другие признали его идеи. Ему удобнее работать в коллективе, где устоялись хорошо организованные связи между сотрудниками.

Интуитивный интроверт - это творец, он начинает творить, только после того, как собрал подходящую для себя информацию. Уинстон Черчель принадлежал к этому типу. Перед тем, как принять решение, он слушал и читал по этому вопросу все

материалы. И часто принимал решение, базируясь на своих впечатлениях об услышаном, был хорошим слушателем, собирал полную информацию для принятия правильного решения, принимая во внимание не столько факты и объекты исследования, сколько связи и отношения между ними.

При работе над проектом необходимо учитывать стиль работы не только своего подчиненного, но и заказчика. Если заказчик интроверт, то ему нужно предоставлять больше информации и времени на обдумывание для принятия решения, В случае экстроверта необходимо больше общаться с ним, позволять высказывать свои требования и идеи. Интуитивисту необходимо подкидывать больше новых идей, поощрять творчество и, если он рационален, то надо для него проводить больше демонстраций, базирующихся на фактах и схемах.

Организация проекта. Для хорошей организации ведения проекта подбирается подходящая структура проекта на основании следующих данных:
– рабочие стили членов группы;
– число людей в группе;
– стиль работы с заказчиками и разработчиками.

Один из популярных стилей ведения проекта впервые использовался в IBM (рис.10.5). В нем главным ответственным за проектирование системы и ведение разработки является руководитель группы программистов. Ему непосредственно подчиняются программисты, которые имеют право последнего слова при принятии решений – главные программисты. Главный программист руководит своей подгруппой программистов и непосредственно посвящен в детали проекта и разработки программы.

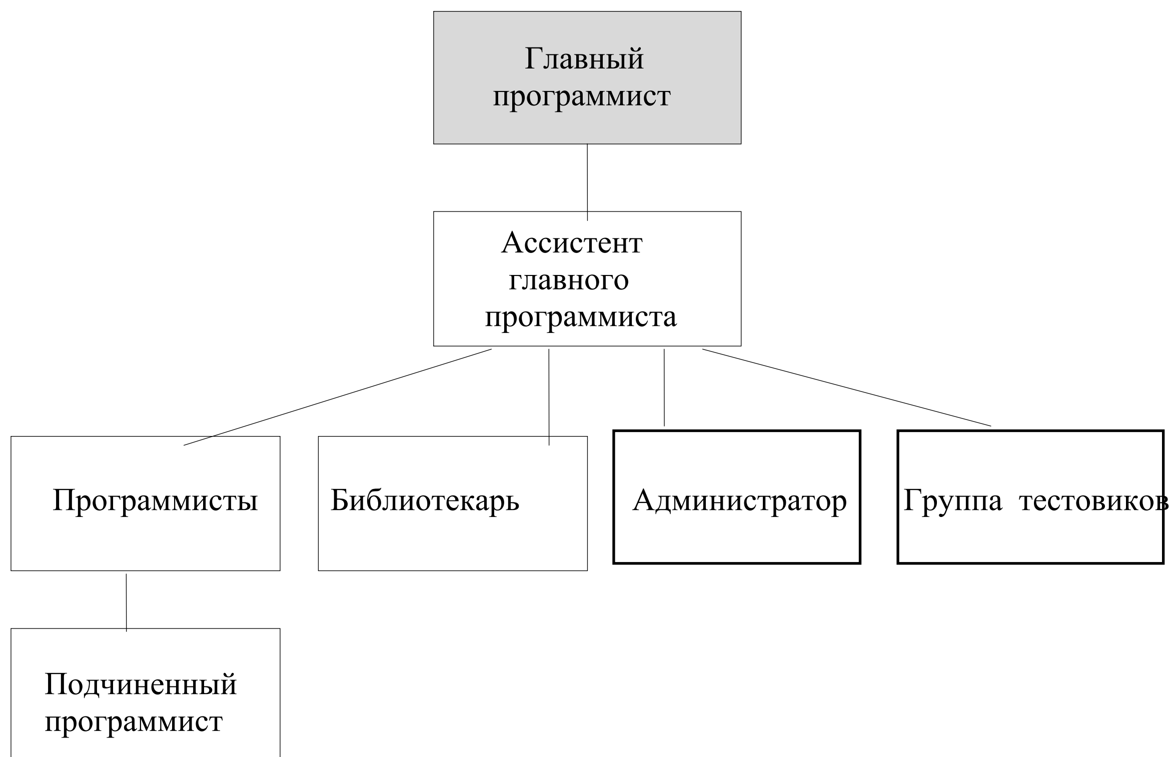

Рис.10.5. Структура организации группы главного программиста

Ассистент главного программиста дублирует, замещает главного программиста, когда это необходимо. Библиотекарь – ответственный за всю документацию проекта: компилирование и тестирование всех модулей библиотеки. Введение этой должности позволяет сконцентрироваться программистам на их непосредственной работе, а не на поиске ошибок и создании необходимых материалов.

В группу входит администратор и группа тестировщиков. Старшие программисты и младшие непосредственно подчиняются старшим. Хотя структура такой рабочей группы иерархическая, каждый член группы может общаться непосредственно с главным программистом или с другими сотрудниками. Главный программист должен сам просматривать части основного проекта и программ.

Альтернативная структура ведения проекта описана Вейнбергом (Weinberg) [3], так называемое обезличенное программирование, при котором все несут одинаковую ответственность за качество продукта. В проекте не концентрируются на персоналиях, критике подвергается программный продукт, а не члены группы. Такая структура подходит для маленьких групп программистов.

Ответственность за моделирование работ в проекте. В [3] в рамках военного ведомства разработана общая структура команды для создания интегрированного продукта (Integrated Product Development Team). Модель ответственности команды приведена на рис.10.6.

Рис. 10.6 Модель ответственности лиц в интегрированном проекте

Модульное программирование системы осуществлялось с помощью компилятора Ада при моделировании этапов: проектирования, анализа и построения программного обеспечения, а также при управлении конфигурацией, тестировании, объединении элементов проекта через интерфейс или при переходе от основных фреймов к рабочим станциям. Менеджер проекта управляет распределением обязанностей и устанавливает сроки для выполнения трех одинаковых по размеру задач проекта.

Участники проекта работали по матричной организации, при которой, каждый инженер входил в состав определенного типа работ (проектирование или тестирование) в одном или более проектах. Суть организации рабочей группы интегрированного создания продукта состоит в возможности работать сообща в соответствии с общими законами дисциплины для всех типов групп и отдельными средствами каждой группы.

В приведенной модели, группа – это комбинация разных сотрудников, ответственная за результат своей работы. Заказчик влияет на результат или на выбор пути для достижения результата. При разработке обязанности и роли сотрудников постоянно меняются. Это удобно для проекта, в котором требуется проведение оперативных процедур и часто меняются планы. Для планов устанавливаются сроки в пределах недели или даже часов. Для организации работ большого коллектива используются карты обязанностей, схемы, отражающие сроки выполнения работ для каждой части проекта. Показателем того, насколько выполнено задание является диаграмма планируемых и реально выполненных работ. Часто эта модель обязанностей объединяется с моделью «из рук в руки» (hand-off), которая предполагает использование сценариев и образцов взаимодействия между сотрудниками, при которых результат работы одной группы передается как исходное данное для работы другой группе.

10.1.4. Оценивание проекта

Одной из наиболее важных работ является оценка стоимости проекта. Общая стоимость проекта определяется исходя из стоимости отдельных частей, условий выполнения работ, наличного штата исполнителей, используемых методов и инструментов. В стоимость проекта входит все создает стиль ведения проекта: компьютеры, программное обеспечение, площади, мебель, телефоны, модемы, канцелярские товары и многое другое. Иногда должны быть созданы дополнительные условия (например, безопасность).

К дополнительным расходам относятся системы тестирования, кодирования или другие CASE системы. Центральной оценкой в проекте является оценка затрат на ведение проекта, выражаемая, например, в человеко-днях исполнителей работ в проекте. Эти оценки проводятся на ранней стадии ведения проекта и составления плана. Специалисты с опытом могут первоначально оценить стоимость проекта с погрешностью меньше 10%.

Правильность оценки зависит от компетентности, опыта, объективности и восприятия эксперта. Метод построения оценки может быть «сверху-вниз» или «снизу-вверх».Стоимость старой системы чаще всего экстраполируется на новую с некоторыми корректировками.

Когда такой возможности нет, эксперты проводят пессимистическую (x), оптимистическую, и реальную. Разные методы приближенного оценивания по отношению к реалистическим рассмотрены в [3]. Эксперт проводит опрос всех членов рабочей группы, и в дальнейшем проводит коррекцию каждой оценки, выводя на их основе наиболее правдоподобную.

Во всех приведенных организационных методах оценки работ есть свои недостатки, связанные с трудностью определения степени отличия каждого модуля старой система от новой. Иногда система оценок, которая успешно работает в одной компании, не

работает в другой. В некоторых рабочих группа пессимистическая и оптимистическая оценки могут сильно отличаться.

Алгоритмические методы оценки. К ним относятся модель, в которой отображаются связи между затратами в проекте и факторами, которые на них влияют. Модель – это уравнение, в котором затраты – зависимая переменная, а влияющие факторы – независимые переменные.

Например, стоимость проекта определяется по формуле: $E = (a+bS^c)\ m\ (X)$, где S - оценка размера системы, *а, в, с* – эмпирические константы, X – вектор факторов стоимости размерностью n, m – регулирующий множитель, основанный на затратных факторах. В [3] предлагается модель в виде соотношения, полученного экспериментальным путем: $E = 5.25S^{0.91}$.

Эта модель применялась при оценке проекта, в котором программные системы имели размер от 4000 до 467000 строк кода, написанных на 28 различных языках программирования высокого уровня для 66 компьютеров и на которые затрачено от 12 до 11758 человека-месяцев.

В [4] предлагается техника моделирования, использующихся в уравнении затрат организации-разработчика:

$$E = 5.5+0.73S^{1.16}.$$

В большинстве моделей оценка зависит от размера системы в строках кода. Модель COCOMO Боєма [ ] собрала в себе три техники измерений по проекту. В первых моделях применялись показатели цены, учитывался персонал и свойства проекта, продукта и среды. Модель включает оценку трех стадий ведения проекта. На первой стадии строится прототип для задач повышенного риска (интерфейс пользователя, ПО, система взаимодействия, реализации и др.) и проводится оценка затрат (например, число таблиц в БД, экраны и отчетные формы др.).

На второй стадии ведется оценка затрат на проектирование и реализацию функциональных точек проекта, отраженных в требованиях к проекту.

На третьей стадии оценка относится к завершенному проектированию, когда размер системы может быть определен в терминах готовых строк программы и других факторов.

Базовой моделью оценки служит следующее уравнение: $E=bS^c\ m(X)$, где первичная оценка $b\ S^c$ корректируется с помощью вектора стоимости m (X). Эта модель развивается с учетом анализа объектов (число старых и новых объектов). Параметр *c* в уравнении изменяется от 0 до 1.0 для первой стадии и от 1.01 до 1.26 для остальных.
Таким образом, можно сформулировать основные вехи для эффективного и успешного управления программным проектом:
1.Определение границ системы и точек выполнения разработки.
2. Формирование временного плана выполнения работ в точках проекта.
3. Определение структуру рабочей группы, периода, видов работ и ресурсов.
4. Техническое описание планируемой системы (аппаратное и ПО проекта, компиляторы, интерфейсы, оборудование и др.) и ограничений на время, безопасность и т.п.
5. Использование стандартов, процедур, техник и инструментов ведения проекта.

6. Разработка планов достижения качества, управления конфигурацией, подготовки документации.
7. Разработка плана управления данными и источниками информации.
8. Разработка плана тестирования, измерения и оценивания результатов работ.
9. Составления плана обучения пользователей системы.
10. Определения плана безопасности (конфиденциальность, пароли и др.)
11. Составление плана управления рисками.
12. План сопровождения с указанием ответственных за изменение кода, ремонт оборудования, использование документации и др.
13. Описание алгоритмов, инструментов, техники просмотра или инспекции кода, языков ведения проекта, языков кодирования и тестирования.

10.2. Методы управление рисками

Причиной возникновения рисков являются неопределенности, существующие в каждом проекте. Риски могут быть “известные”, которые определены, оценены и которые можно планировать. Риски “неизвестные”, которые не идентифицированы и не могут быть спрогнозированы [9–12].

Риск - это нежелательное событие, которое может иметь непредвиденные негативные последствия. Если в проекте идентифицировано множество возможных событий риска, которые могут повлечь за собой негативные последствия, то такой проект является склонным к риску.

Многие компании уделяют внимание разработке и применению корпоративных методов управления рисками, которые учитывают специфику проектов и корпоративные методы управления.

Американский Институт управления проектами (PMI) разрабатал стандарты в области управления проектами и в последнее время переработал разделы, регламентирующие процедуры управления рисками. В новой версии PMBOK шесть процедур управления рисками:
1. Планирование управления рисками – выбор подходов и планирование деятельности по управлению рисками проекта.
2. Идентификация рисков – определение рисков, способных повлиять на проект, и документирование их характеристик.
3. Качественная оценка рисков – качественный анализ рисков и условий их возникновения с целью определения их влияния на успех проекта.
4. Количественная оценка – количественный анализ вероятности возникновения и влияния последствий рисков на проект.
5. Планирование реагирования на риски– определение процедур и методов по ослаблению отрицательных последствий рисковых событий и использованию возможных преимуществ.
6. Мониторинг и контроль рисков для определения остающихся рисков, выполнение плана управления рисками проекта и оценка действий по минимизации рисков.

Все эти процедуры взаимодействуют друг с другом, а также с другими процедурами. Каждая процедура выполняется, по крайней мере, один раз в каждом проекте. Несмотря на то, что эти процедуры рассматриваются как дискретные элементы с четко определенными характеристиками, на практике они могут частично совпадать и взаимодействовать.

Планирование управления рисками – это процесс принятия решений по применению и планированию управления рисками для конкретного проекта. Этот процесс может включать в себя решения по организации, кадровому обеспечению процедур управления рисками проекта, выбор предпочтительной методологии, источников данных для идентификации риска, временной интервал для анализа ситуации. Важно спланировать управление рисками, адекватное как уровню и типу риска, так и важности проекта для организации.

Идентификация рисков выполняется для определения рисков, которые способны повлиять на проект. Идентификация рисков не будет эффективной, если она не будет проводиться регулярно на протяжении реализации проекта. К этой идентификации должны привлекаться менеджеры проекта, заказчики, пользователи и независимые специалисты. Эта процедура выполняется как итерационный процесс. Вначале идентификация рисков может быть выполнена частью менеджеров проекта или группой аналитиков рисков. Далее ей может заниматься основная группа исполнителей из менеджеров проекта. Для формирования объективной оценки в завершающей стадии могут участвовать независимые специалисты.

Качественная оценка рисков – это процесс представления качественного анализа идентификации рисков и определения рисков, требующих быстрого реагирования. Такая оценка рисков определяет степень важности риска и выбирает способ реагирования. Доступность сопровождающей информации помогает легче расставить приоритеты для разных категорий рисков. Качественная оценка рисков включает в себя оценку условий возникновения рисков и определения их воздействия на проект с помощью стандартных методов и средств. Применение этих средств помогает частично избежать неопределенности, которые часто встречаются в проекте. В течение ЖЦ проекта должна происходить постоянная переоценка рисков.

Количественная оценка рисков определяет вероятность возникновения рисков и влияние последствий рисков на проект, что помогает группе управления проектами верно принимать решения и избегать неопределенностей. Количественная оценка рисков позволяет определять:

– вероятность достижения конечной цели проекта

– степень воздействия риска на проект и объемы непредвиденных затрат и материалов, которые могут понадобиться;

– риски, требующие скорейшего реагирования и большего внимания, а также влияние их последствий на проект;

– фактические затраты и предполагаемые сроки окончания работ на проекте.

Количественная оценка рисков включает и качественную оценку, а также требует дентификации рисков. Количественная и качественная оценки рисков могут применяться в отдельности или вместе, в зависимости от времени и бюджета.

Планирование реагирования на риски – это разработка методов и технологий снижения отрицательного воздействия рисков на проект. Берет на себя ответственность за эффективность защиты проекта от воздействия на него рисков. Планирование включает в себя идентификацию и распределение каждого риска по категориям. Эффективность разработки реагирования определяется последствиями воздействия риска на проект (положительным или отрицательным).

Стратегия планирования реагирования должна соответствовать типам рисков, рентабельности ресурсов и временным параметрам. Риски обсуждаются во время встреч, должны быть адекватны задачам на каждой стадии проекта и согласованы со всеми членами группы по управлению проектом. Может быть несколько вариантов стратегий реагирования на риски.

Мониторинг и контроль – это процедуры слежения за идентификацией рисков, обеспечению выполнения плана рисков и оценки его эффективности с учетом понижения риска. Показатели рисков, связанные с осуществлением условий выполнения плана фиксируются. Мониторинг и контроль сопровождает процесс внедрения проекта в жизнь.
Качественный контроль выполнения проекта предоставляет информацию, помогающую принимать эффективные решения для предотвращения возникновения рисков. Для предоставления полной информации о выполнении проекта необходимо взаимодействие между всеми менеджерами проекта.

Цель мониторинга и контроля состоит в выяснении таких ситуаций:
– реакция на риски внедрена в соответствии с планом и необходимы изменения,
– изменение рисков по сравнению с предыдущими значениями,
– определение влияния рисков и принятие необходимых мер,
– реакция на риски является запланированной или является случайным процессом.

Рис.10.7. Шаги по управлению риском

Контроль может повлечь за собой выбор альтернативных стратегий, принятие корректив, перепланировку проекта для достижения базового плана. Между менеджерами проекта и группой риска должно быть постоянное взаимодействие и фиксация всех изменений и явлений. Отчеты по выполнению проекта и системе рисков регулярно формируются.

Управление рисками основывается на рассмотрении двух основных типа риска: общий риск для всех типов проектов и специфический тип риска.

К первому типу риска относится риск, возникающий при недопонимании требований, при нехватке профессионалов или недостатке времени на тестирование. Риск второго типа выражается недостатками проекта (незавершенность проекта к обещанному сроку и др.). Управление риском включает в себя шаги управления риском (см. рис 10.7.).

Для каждого возможного риска определяется показатель степени его вероятности и показатель потерь, связанных с риском. Во время проведения регрессионного тестирования отыскиваются критические ошибки. В зависимости от того, насколько эта ошибка критична и от того, какие показатели риска действуют, вычисляется ущерб риска. Деятельность по управлению риском включает выполнение задач: уменьшения риска, планирование риска, резолюцию на обнаруженный риск. Уменьшение риска можно достигнуть, если избегать риск при изменении требований,перераспределять риск, отслеживать риск и управлять им.

Систему управления риском можно представить в виде отношения:

$$\frac{\textit{Ущерб до минимизации - ущерб после минимизации}}{\textit{Цена минимизации риска.}}$$

Минимизацию риска можно получить прототипированием. Боэм [2] идентифицировал 10 наиболее часто возникающих причин риска в проекте:

1. Сокращение штата или набор неквалифицированных сотрудников.
2. Нереалистические в проекте планы и бюджеты.
3. Разработка функционально неправильных программных элементов.
4. Разработка неудачного пользовательского интерфейса.
5. Неудачная постановка требований.
6. Постоянное изменение требований.
7. Недостатки во внутренней организации работ.
8. Недостатки взаимосвязи с заказчиком.
9. Неумение работать в реальном времени.
10. Ограниченные компьютерные ресурсы.

10.3. Управление конфигурацией программной системы

Под *конфигурацией системы* понимается конкретный состав (версия) системы, включающий технические, программные и программно-технические средства, объединенные между собой специальными процедурами сборки, для достижения конкретных целей обработки данных [9-12].

Элементами конфигурации являются:

– единица конфигурации – ЕК (Configuration Item) – элемент, выделенный для целей управления и обработки на процессорах компьютера системы;

– базис конфигурации – БК (Configuration Baseline) – набор формально рассмотренной и утвержденной основы системы из состава ЕК и документации, устанавливающей возможность дальнейшего развития системы;
– программные компоненты системы.

Конфигурация состоит из одной и более ЕК, базис конфигурации определяет входящие в конфигурацию ЕК, к которым относятся технические решения, перечень главных ЕК и специализированные процедуры их объединения в единое целое для функционирования и выполнения компонентов в заданной последовательности. Чем больше в системе компонентов, тем больше вероятность того, что отдельные из них могут изменяться в связи с обнаруженными ошибками, уточнениями или дополнениями, как новых функций, так и оборудования.
Идентификация конфигурации – это именование всех элементов системы с учетом , которое базируется структуризации и построение схемы классификации и кодирования этих элементов, а также на методах представления и ведения версий конфигурации с использованием входящих элементов.

К элементам управления конфигурацией также относятся физические и функциональные характеристики, схема и версия конфигурации. Целью УК является обеспечение целостности системы путем наблюдения за производимыми изменениями за структурой и элементами конфигурации.

Под *целостностью конфигурации* понимается способность системы воспроизводить и выполнять заданные функции после физических и функциональных изменений отдельных элементов и повторного их объединения в новую версию системы.

10.3.1. Управление конфигурацией

Управление конфигурацией (УК) – дисциплина, обеспечивающая идентификацию элементов конфигурации системы при ее создании для проведения систематического контроля, учета и аудита внесенных изменений, а также поддержки целостности и работоспособности системы.

Согласно действующего стандарта IEEE Std.610-90 УК включает следующие основные задачи:
1. Идентификация конфигурации (Configuration Identification).
2. Контроль конфигурации (Configuration Control).
3. Учет статуса конфигурации (Configuration Status Accounting).
4. Аудит конфигурации аудит (Configuration Audit).

УК для больших систем создается с помощью методов и средств, обеспечивающих идентификацию элементов этой системы, контроль вносимых изменений и возможность определения фактического состояния системы при разработке и эксплуатации в любой момент времени. УК базируется на точной и достоверной информации о состоянии системы и планах проведения изменений.

С формальной точки зрения УК состоит в дисциплинированном применении технического, административного управления и методов наблюдения за определенными и документированными функциональными и физическими характеристиками отдельных пунктов конфигурации и элементов системы, а также методов управления изменениями, подготовки отчетов по выполненным изменениям и процедурам их проверки на соответствие поставленным требованиям.

Работы по УК, как правило, выполняет специальная служба, которая определяет возможные ограничения на функционирование системы в заданных условиях операционной среды, планирование внесения изменений, проверку разных частей системы, сбор данных и учет внесенных изменений в систему и конфигурацию. К деятельности этой службы относится также управление проектом, контроль качества и целостности конфигурации системы и ее сопровождение.

Структура службы зависит от сложности системы, этапов развития проекта и от специалистов организации-разработчика системы и заказчика. От хорошей организации работы службы зависит эффективность УК. Взаимосвязь видов деятельности по УК представлена рис.10.8.

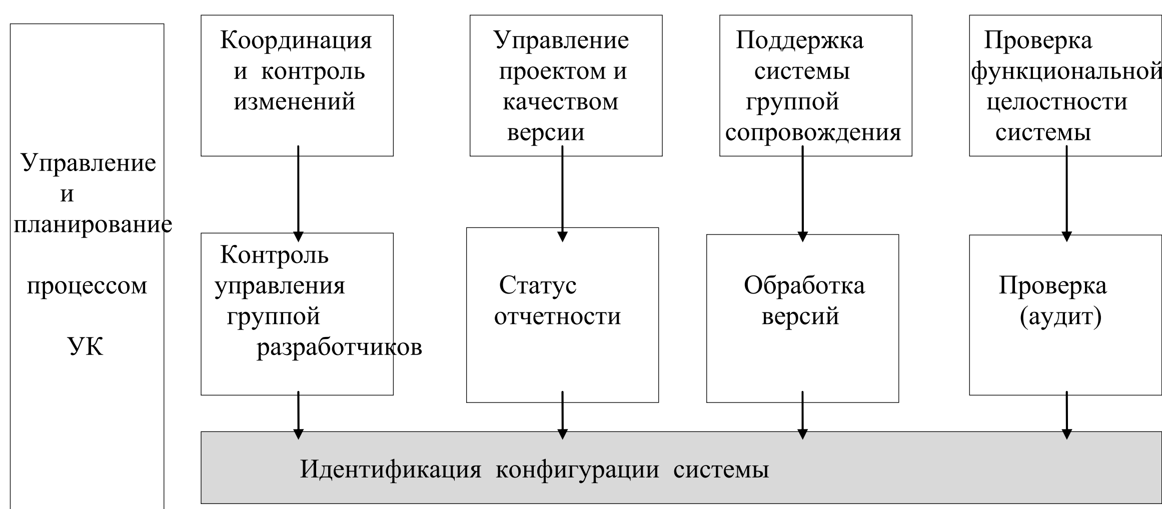

Рис.10.8. Виды деятельности УК

Результатом УК является отчет о проведенных изменениях версии системы и документации, а также документ о передаче измененной версии пользователю.

Для достижения целей УК должно планироваться и выполняться с учетом возникающих ограничений ОС и оборудования у заказчика. Процессом планирования занимаются менеджеры службы управления проектом. Предложения на изменение компонентов системы подаются в эту службу, для проведения анализа и определения целесообразности внесения изменений в версию системы и ее конфигурацию, оценку стоимости этих работ, разработку предложений в виде утвержденного перечня изменений для их реализации.

Процесс изменений включает в себя определение типов изменений, организацию их проведения и формирование концепции допуска отклонений и отказов по отношению к требованиям проекта системы. Результатом внесения изменений является новая версия системы, документация по проведению на ней испытаний и пользовательская документация на систему.

Заказчик оценивает предложения на внесение изменений и дает разрешение на проведение наиболее важных изменений, влияющих на ее технические характеристики или стоимость. Анализ и контроль проведения изменений конфигурации системы проводит специальная группа службы управления. Она выполняет систематический учёт и контроль внесения изменений на всех этапах ЖЦ.

План изменений в конфигурацию системы утверждается формальными процедурами, расчетами оценок влияния изменений на стоимость, принятие решений об изменениях или отказ от них. Запросы на внесение изменений выполняются в соответствии с процедурами разработки системы на этапах ЖЦ или на этапе сопровождения системы. Поскольку требуемые изменения могут проводиться одновременно с разработкой, предусматривается трассирование изменений при построении новых версий. Каждое проведенное изменение подвергаются детальному аудиту.

За внесением изменений проводится контроль текущей версии системы с использованием выходных кодов в репозитории, проверки исходного кода и полученной версий. Инструменты контроля имеются в фирмах Rational's ClearCase и SourceSafe of Microsoft системы Unix.
После завершения изменений и испытания системы проводится тиражирование системы и документации для передачи системы и ее конфигурации заказчику.
В конфигурацию системы входят сведения об аппаратных и программных элементах системы. При этом на систему могут задаваться ограничения с учетом контактов с заказчиком, аудиторами, информации из разных источников (спецификации требований, описаний, отчетов и др.), а также состава инструментальных средств и рекомендаций государственных или межведомственных стандартов.

10.3.2. Планирование УК

Зависит от типа проекта, организационных мероприятий, ограничений и общих рекомендаций по руководству конфигурацией. К видам планирования УК системы относятся: идентификация, определение статуса и аудита конфигурации, управление изменениями конфигурации.

При планировании составляются планы, выбираются инструменты, анализируются требования проекта, интерфейсы компонентов и т.п. К средствам обеспечения планирования относятся:
– система управления кодами компонент, их переводом и объединением в конфигурацию системы;
– базовые библиотеки и ресурсы;
– специальные группы контроля системы и ее конфигурации;
– СУБД для ведения проекта и хранения изменений в систему.

К основным задачам планирования конфигурации относятся:
– фиксация разных заданий на изменения и выбор инструментария для их выполнения;
– определение человеко-часов и инструментальных ресурсов, стандартов, затрат на внесение изменений и др.;
– установление связей с заказчиком для проведения контроля системы и конфигурации, а также проведение оценки системы;
– определение последовательности работ УК.

Результаты планирования отмечаются в плане УК проекта, а также в документе внесения изменений в версию, конфигурацию или в систему.

10.3.3. Идентификация элементов конфигурации

Выполняется с помощью методов структуризации, классификации и именования элементов в составе системы и ее версий. При проведении идентификации проводится:
– определение стратегии идентификации для получения учтенной версии системы;
– именование составных элементов частей и всей конфигурации системы;
– установление соотношения между количеством выполняемых задач и количеством пунктов конфигурации;
– ведение версии системы (или ее частей) и документирование;
– выбор элементов базиса конфигурации и его формальное обозначение.

При идентификации используется библиотека элементов, версий и изменений системы. Основу идентификации составляет конфигурационный базис – набор формально рассмотренной и утвержденной конфигурационной документации, как основы для дальнейшего развития или разработки системы.
Выделение в продукте контролируемых единиц конфигурации – сложная задача, как правило, является составной частью процесса высокоуровневого или архитектурного проектирования и выполняется системными архитекторами. Построение адекватной схемы классификации и идентификации объектов конфигурационного управления. выполняется одновременно со структуризацией продукта и заключается в определении правил уникальной идентификации (кодирования, маркирования):
– конфигурации продукта и ее версий;
– контролируемых единиц конфигурации и их версий;
– всех составляющих конфигурационного базиса и их редакций.

Результатом создания и применения схемы идентификации дает возможность быстро и гарантированно отличать разные продукты друг от друга, версии одного продукта между собой, единицы конфигурации продукта и их версии.

10.3.4. Управление версиями

Версия системы включает элементы конфигурации и версию системы для передачи получателю [6, 7]. Управление версиями состоит в выполнении действий:
– *интеграции или* композиции корректной и окончательной версии системы из элементов конфигурации, которые реализованы на этапах ЖЦ. Функционирование кода системы зависит от аппаратных средств и инструментов, с помощью которых строилась система;
– *выбора инструментария* построения версии, оценки возможностей среды и средств автоматизации процесса построения отдельных версий с корректной конфигурацией программного обеспечения и данных;
– *управления вариантами версий* из совокупности готовых идентифицированных элементов системы, удовлетворяющие заданным требованиям заказчика на систему.

При формировании версий системы учитываются ограничения на разработку системы во время этапов ЖЦ, которые, как правило, порождают ряд отклонений от требований на разработку элементов конфигурации системы. Например, принимается решение об изменении конфигурации способом, который не совпадает с предложенным и согласованным с заказчиком. Когда новая версия системы получена, заказчику передаются версия, конфигурация, документация и инструменты управления версиями для самостоятельного внесения изменений в эти элементы в процессе сопровождения системы.

Ярким примером формирования 21 версии на ОС 360 (1965-1980гг.) является фирма IBM. В ОС постоянно и поэтапно добавлялись новые функциональные возможности и вносились изменения в предыдущую версию при ее эксплуатации [13]. Над развитием дополнительных возможностей данной ОС и внесением изменений в предыдущую версию постоянно работал коллектив фирмы. Трудоемкость разработки очередной версии ОС считалась пропорциональной интервалу времени между регистрациями очередных версий и принималась за единицу измерения сложности создания новой версии [7].

В качестве меры трудоемкости сопровождения и создания очередной версии использовалось число модулей (ограниченных размеров со стандартизованным описанием), подвергающихся изменениям и дополнениям. Кроме того, оценивалась интенсивность работ по созданию версии, которая измерялась числом измененных модулей в единицу времени. После 12 лет постоянных изменений в ОС, 21 версия работала более стабильно, в нее почти не вносились изменения, так как претензий со стороны пользователей в основном не поступало.

Метрический анализ процесса развития ОС 360 позволил установить, что объем среднего прироста системы на каждую версию соответствовал примерно 200 модулям. При этом общий объем увеличился от 1 тыс. модулей в первых версиях до 5 тыс. модулей в последних версиях. Когда уровень прироста сложности был большим, для устранения ошибок или дополнительных корректировок иногда создавались промежуточные версии с меньшим числом изменений.

В результате появилось понятие «критической массы» или критической сложности модифицируемой системы. Если при модернизации и выпуске очередной версии системы объем доработок превышает «критический», то возрастает вероятность ухудшения характеристик системы или необходимость введения промежуточной версии с внесением некоторых изменений. «Критический» объем доработок ОС–360 около 200 модулей оставался постоянным, несмотря на рост квалификации коллектива, совершенствование технических и программных средств и др. В первых версиях объем доработок составлял 20% модулей, а в последних версиях снизился до 5%.

10.3.5. Конфигурационный контроль

Под конфигурационным контролем принято понимать управление изменениями в ходе ЖЦ или эксплуатации системы. Процесс создания продукта включает непрерывные корректировки, которые имеют отношение к уже согласованному и /или утвержденному конфигурационному базису (КБ). В этом плане предметом конфигурационного контроля являются:

– изменения в утвержденном КБ и связанные с ними корректировки в конфигурации и /или ЕК;
– дефекты и отклонения в конфигурации продукта относительно утвержденного КБ.

Имеются в виду формальные процедуры инициализации, анализа, принятия и контроля исполнения управленческих решений по предложенным изменениям, обнаруженным дефектам и отклонениям в конфигурации и /или ЕК продукта.

Формальная обработка запросов на изменение КБ. После того, когда заинтересованные участники проекта достигли взаимопонимания по требованиям, архитектуре и другим техническим решениям, соответствующие проектные документы

считаются утвержденными и не могут произвольно модифицироваться. Т.е. любая потребность в изменении, исходящая от любого участника проекта, должна пройти формальную процедуру, включающую такие шаги:

1. Регистрация предложения /запроса на изменение.
2. Анализ влияния предложенного изменения на имеющийся задел, объем, трудоемкость, график и стоимость работ по проекту.
3. Принятие решения по запросу на изменение (удовлетворить, отказать или отложить).
4. Реализация утвержденного изменения и его верификация.

Управление дефектами и отклонениями от утвержденного КБ. Второй важнейшей составляющей конфигурационного контроля является управление несоответствиями между конфигурацией или ЕК продукта и конфигурационным базисом. С точки зрения управления все несоответствия принято делить на дефекты и отклонения. К дефектам относят те несоответствия, которые имеют непосредственное отношение к целевому использованию продукта по его назначению. Все остальное относится к отклонениям. Если дефекты в продукте носят негативный характер, то они подлежат устранению.

Для устранения дефектов и выявленных отклонений проводится:

– регистрация информации о полученном дефекте /отклонении;
– анализ и: диагностика места и причины дефекта /отклонения, оценка объема, трудоемкости, сроков и стоимости переделок;
– принятие решения по устранению дефекта /отклонения, реализация и верификация этих недостатков.

Подобного рода решения являются управленческими, их принимают руководители соответствующего уровня или их полномочные представители. Как правило, уровень принятия решения по изменению программного продукта должен быть принят на проекте на уровне согласования или утверждения документов соответствующего конфигурационного базиса.

Наиболее удобной формой реализации такого управленческого решения являются руководящий совет по конфигурационному контролю CCB (Configuration Control Board), как родоначальник теории и практики конфигурационного управления.

10.3.6. Учет статуса конфигурации

Суть этого учета состоит в регистрации и предоставлении информации для эффективного КУ. Предметом учета является информация о текущем статусе идентифицированных объектов конфигурационного управления, предложенных изменениях, а также о выявленных дефектах и отклонениях от утвержденного конфигурационного базиса.

Отчетность по статусу конфигурации является ключевым фактором принятия управленческих решений по проекту информационной системы или ПО. Более того, оперативно регистрируемые и регулярно обновляемые данные учета статуса конфигурации являются исходным материалом для формирования количественных оценок или метрик производительности и качества работ на проекте. Применение этих метрик позволяет принимать не только правильные, но и эффективные управленческие решения по созданию программного проекта.

В системе учета статуса конфигурации накапливаются сводные отчеты о количестве обнаруженных и исправленных дефектов, поступивших и реализованных запросов на

изменения, динамике внесения изменений в конфигурацию продукта во времени и др. Данной отчетностью практически пользуются все участники проекта: заказчики, аналитики, разработчики, тестировщики, внедренцы, служба качества и руководство проекта. На их основе проводится количественная оценка производительности и качества работ по проекту.

10.3.7. Конфигурационный аудит

Аудит – это ревизия или проверка перед выпуском очередной версии ПО или перед сдачей системы заказчику. Кроме того, в обоих случаях аудиторская работа большей частью связана с рассмотрением и оценкой документации – данных, сводок, отчетов.

Конфигурационный аудит производится непосредственно перед выходом новой версии продукта, его части, т.е. практически всегда исходит из ответственности момента по тем или иным обязательствам перед заказчиком. Конфигурационный аудит включает работу по двум взаимосвязанным направлениям:
– функциональный аудит конфигурации для подтверждения соответствия фактических характеристик конфигурации/единиц продукта предъявляемым требованиям;
– физический аудит конфигурации, как подтверждение взаимного соответствия документации и фактической конфигурации продукта.

Функциональный аудит - это не верификация или валидация продукта путем тестирования, а проверка того, что тестирование проведено в установленном объеме, результаты документированы и подтверждают соответствие характеристик продукта предъявляемым требованиям. При этом все изменения реализованы, критичные дефекты устранены, а по всем выявленным отклонениям от конфигурационного базиса приняты адекватные решения. Он обеспечивает сверку выпущенного продукта согласно документов конфигурационного базиса, а также проверка того, что данная конфигурация построена в соответствии с установленными процедурами и из корректных версий соответствующих компонентов. Конфигурационный аудит проводится независимыми экспертами, например - представителями службы качества.

Контрольные вопросы и задания

1. Как решаются задачи менеджмента программного проекта?
2. Определите процесс планирования менеджмента проекта.
3. Определите понятие управления риском.
4. Объясните стратегию оценки стоимости продукта по Боэму.
5. Как решаются задачи менеджмента программного проекта?
6. Определите процесс планирования менеджмента проекта.
7. Определите понятие управления риском.
8. Объясните стратегию оценки стоимости продукта по Боэму.
9.Что понимается под процессом управления конфигурациею ПО?
10. Приведите основные задачи управления конфигурациею.
11. Дайте общую характеристику понятий идентифкации, учета статуса.
12. Какие действия выполняются в процесси управления версиями ПО?
13. Сформулируйте основные задачи учета и аудита.

Литература к теме 10.

1. *Reiter D.J.* Software managment, IEEE Computer Society Press, Los Alomos.- 1993.
2. *B. Duncan.* A Guide to the Project Management Body of Knowledge // PMBOK GUIDE.– 2000.– Edition / www.pmi.org/publication/download/2000welcome.html.
3. *Боэм Б.У.* Инженерное проектирование программного обеспечения.- М:, Радио и связь.-1985.- 511с.
4. *Pfleeger S.L.* Software Engineering. Theory and Practice.- Prentice Hall, 1998.-576p.
5. *Андон Ф.И., Суслов В.Ю., Коваль Г.И., Коротун Т.М.*Основы инженерии качества программных систем.– Киев, Академпериодика.–2002.–502с..
6. *R.H. Thayer, ed.*, Software Engineering Project Managment, 2nd.ed., IEEE CS Press, Los Alamitos, Calif. 1997.–391p.
7. *Бабенко Л.П., Лаврищева Е.М.* Основы програмной инженерии. Учебник (укр.язык). – Киев: Знання, 2001. –269 с.
8. *ISO/IEC* TR 16326:1999. Guide for the application of ISO/IEC 12207 to project management.
9. *IEEE Std 1058-1998.* IEEE Standard for Software Project Management Plans.
10. *Glib T.* Principles of software engineering management. – Wokingham, England: Addison-Wesley, 1998. – 396 p.
11. *Гультяев А.К.* MS PROJECT 2002. Управление проектами. Русская версия; Практическое пособие. – Спб.КОРОНА, 2003. –592 с.
12. *Джалота П.* Управление программными проектами на практике, Лори, 2005.-
13. *Брукс П.* Мифический человеко – месяц. – Г.: Мир, 1972. – 234 с.
14. *Черников А.* Теория и практика управления проектами // Компьютерное обозрение. – 2003.– №10 – С. 24-39.
15. *Первое* знакомство с Microsoft Office project Professional 2003. – Microsoft, 2003. – 34 с.

Тема 11

СРЕДСТВА И ИНСТРУМЕНТЫ В ПРОГРАММНОЙ ИНЖЕНЕРИИ

В работе в качестве основного объекта всех методов проектирования новых ПС на процессах ЖЦ применяется компонент. К средствам описания компонентов относятся языки программирования (JAVA, C++), язык описания интерфейсов IDL, язык моделирования UML, средства описания взаимодействия компонентов (вызовы, протоколы) в распределенной среде, а также разные виды инструментальной поддержки этих описаний (система JAVA, CORBA, RUP, Rational Rose и др.) со средствами обслуживания и использования готовых сервисов и программ.

Данный раздел посвящен рассмотрению двух основных направлений программной инженерии – средствам и инструментам, включая:
1. Языковые средства описания компонентов, их интерфейсов и моделей ПС (JAVA, CORBA, IDL, UML).
2. Инструментальные средства обеспечения процесса построения ПС из объектов и компонентов.
3. Методы и средства разработки производственной архитектуры MSF

11.1. Языковые средства описания компонентов и методов интеграции

Компонент – это единица интеграции, специфицированная так, чтобы можно было ее объединять с другими компонентами в ПС. Важнейшее свойство компонента – отделение его интерфейса от реализации, в отличие от объектов в объектно–ориентированных ЯП, в которых реализация класса отделена от определения класса [1]..

Интеграция компонентов и развертывания – независима от ЖЦ разработки ПС и замена в ней компонента не требует перекомпиляции всей ПС или переналадки всех связей между компонентами. Доступ к компоненту проводится через его интерфейс. Компонент включает спецификацию функциональных и нефункциональных свойств (атрибутов качества – точность, надежность, секретность и др.) требований, сценариев, тестов и т.п. Текущие компонентные технологии используют формальные средства спецификации только функциональных свойств компонентов, включающих описание синтаксиса операций и атрибутов, а для описания нефункциональных свойств компонентов формальный аппарат пока отсутствует.

Более крупным образованием компонентов, используемым в практике программирования, являются паттерны и каркасы [2].

Паттерны определяют повторяемые решения для проблемы объединения компонентов в структуры. Для каждого объединения определяется абстракция общения (взаимодействия) определенной совокупности объектов в кооперативной деятельности, для которой задаются абстрактные участники, их роли, взаимоотношения и распределение полномочий. Они классифицированы по трем уровням абстракции. На *верхнем уровне* – архитектура системы, которая скомпонована из компонентов, называется архитектурным паттерном, охватывающим общую структуру и организацию ПС, набор подсистем, роли и отношения между ними. На *среднем уровне* абстракции паттерн уточняет структуру и поведение подсистем, компонентов ПС и связей между ними.
На *нижнем уровне* паттерн – абстракция определенной цели, которая зависит от выбранной парадигмы его представления и ЯП.

Каркас представляет собой типичную повторно возникающую ситуацию на уровне модели, в которой определенная структура проекта имеет не доопределенные элементы с пустыми слотами для занесения в них доопределенных свойств компонентов. При сборке отдельно построенных программных частей в каркас, компоненты инкапсулируются и определяются контекстом сборки, после чего каркас становится контейнером, применяемым далее как повторный компонент или ПИК.

Спецификация каркаса – это определение требуемых компонентов, как вариантов наполнения контейнера. Каркас концентрирует общие свойства и правила инкапсулированных компонентов, а контракты задают спецификацию отношений между конкретными компонентами, которые отличаются от спецификации компонентов как частей композиции. Конструктивно специфицированный интерфейс и функциональные свойства компонентов значительно повышают их надежность и устойчивость работы, заданные в нефункциональных характеристиках компонентов.

Между каркасами и паттернами связаны определенными отношениями: каркасы физически реализованы с помощью одного или более паттернов, которые могут рассматриваться как инструкция для реализованных проектных решений.

11.1.1. Средства ЯП JAVA для описания и интеграции компонентов

В языке JAVA в качестве готовых компонентов используются beans компоненты, которые включают описание функциональности, интерфейса и шаблона развертывания, как средства интеграции их в новые ПС. Он может повторно использоваться в разных средах для выполнения своих функций самостоятельно или в составе с другими компонентами. Класс можно сделать Beans компонентом, внеся небольшие изменения с помощью специальной утилиты системы BDK (Bean Development Kit) [3-5].

Компоненты beans подразделяются на три категории:
1. Компоненты сеансов, которые поддерживают правила бизнеса–логики, ориентированы
на состояния и могут быть связаны с конкретным клиентским сеансом;
2. Компоненты сущностей используются для связи с БД непосредственно, представляют данные в объектной форме;
3. Компоненты, которые управляются событиями, функционируют для получения сообщений, поступающих от системы обмена сообщениями JMS (Java Messaging System), и реагируют на них.

При создании bean компонентов используется интерфейсы: Home для управления ЖЦ компонента, интерфейс Remote для вызова и рализации компонента в среде виртуальной машины JVM (Java Virtual Machine). Каждый компонент beans имеет свой контейнер, который вызывает и регулирует все аспекты ЖЦ, а также интерфейс.

Основной особенностью beans компонентов в JAVA является отображение – т.е.способность анализировать самого себя и описывать свои возможности динамично во время выполнения, а не во время компиляции. Пакет JAVA java.lang.reflect входит в ядро API, поддерживает отображение разных компонентов и содержит один интерфейс – Member, определяющий методы получения информации о полях, структуре классов.

Существуют два способа задания свойств, событий и методов beans компонентов. Первый способ – использование согласованных имен, другой – создание дополнительного класса для предоставления требуемой информации.

Bean компонента – это подмножество состояний, которые определяют поведение и внешний вид компонента. Эти свойства делятся на простые, булевы и индексированные. *Простые* свойства имеют одиночные значения, могут быть идентифицированы проектными шаблонами (например, свойства для read/write, read–only, write–only). *Булевы* свойства принимают значение true или false и идентифицируются проектными шаблонами. *Индексированные* свойства состоят из множества индексированных значений, задаваемых проектным шаблоном.

Bean компоненты могут иметь, так называемые *связанные свойства,* которые отражают событие без изменения свойства компонента. В стандартном шаблоне BeanInfo непосредственно содержатся информационные массивы свойств, событий и методов PropertyDescriptor, EventSetDescriptor, MethodDescriptor), при реализации которых разработчик может обеспечивается предложение пользователя.

Bean компонент с *ограниченным* свойством генерирует событие и изменяет значение этого свойства, затем это событие отсылается объектам, которые могут отклонить изменение свойств или поддержать в зависимости от среды выполнения. Инструментарий BDK позволяет сохранять компоненты с помощью пунктов меню (File, Save) в JAR архиве следующей последовательностью шагов:

– создать каталог для нового Bean компонента;
– создать одних или несколько исходных JAVA файлов, которые реализуют компонент;
– скомпилировать эти файлы;
– создать файл описания свойств компонента;
– сгенерировать JAR файл;
– запустить BDK инструментарий для сохранения нового компонента;
– протестовать компонент.

Для взаимодействия разных компонентов используется механизм вызова удаленного метода RMI, который дополнен к языку JAVA через стандартную модель EJB (Enterprise Java Beans) компании Sun. К ней подключены классы языка JAVA, определения их атрибутов, параметров среды и свойств группирования компонентов в прикладную программу для выполнения на виртуальной машине JVM. Механизм развертывания JAVA–компонентов типа beans на сервере базируется на программах в исходном языке, а сервер создает для них оптимальную среду для выполнения задач EJB.

Для реализации и повторного использования ПИК типа beans разработаны следующие шаблоны в Java for Forte:
– Bean для создания нового beans компонент и формирования каркас компонента с простыми свойствами и возможностью автоматического изменения этих свойств;
– BeanInfo для интеграции beans компонентов и обеспечения взаимодействия;
– Customizer создает панель, на которой размещаются элементы, которые со временем можно использовать для управления конфигурацией beans компонента;
– Property Editor создает класс, который используется во время проектирования и редактирования свойств beans компонентов.

Фирма Microsoft Active разработала ряд Beans компонентов, которые формируют интегрированную архитектуру приложения. Контейнеры этих компонентов поддерживаются Internet Explorer, Microsoft Office и Visual Basic. Кроме того, на сайте **java.sun.com** можно загрузить инструментальную систему Bridge for Active в целях применения Java Beans компонентов в контейнерах Active, а также загрузить инструментарий Java Beans Migration Assistant for Active для анализа элементов управления Active и генерации каркаса JavaBean компонентов.

Таким образом, JAVA поддерживает стандартные механизмы для работы с компонентами как со строительными блоками со следующими особенностями:

– большинство свойств, событий и методов в ПС являются управляемыми;
– реинженерия компонента основывается на параметрах времени разработки;
– параметры отладки конфигурации сохраняются и используются в заданное время;
– beans компоненты регистрируют принятые сообщения о событиях или сами генерируют события;
– ПИК сохраняется в архиве JAVA с помощью шаблона JAR Contents самостоятельно либо в виде группы;
– каждый компонент может описываться в разных языках.

Для всех классов компонентов в Forte for JAVA компании Sun Microsoft Systems, Inc., предлагаются шаблоны интеграции в программную разработку [7].

11.1.2. Типы компонентов и средства их интеграции в JAVA

Интерфейс является видимой частью спецификации компонента, предназначенной для интеграции компонента в среду для повторного использования. Для описания интерфейса компонента применяется Inspector Components, позволяющий изменять необходимые параметры интерфейса компонента с помощью визуальной таблицы, отображающей все параметры. Интерфейс в Inspector Components задается в виде пары – имя параметра и значение параметра, которые могут изменяться автоматически без вмешательства в код компонента.

Для разных типов компонентов Inspector Components содержит неизменную часть представленных параметров, которая может быть включена в инвариант спецификации, к которому относятся параметры: тип компонента, имя компонента, входные, выходные данные, типы атрибутов и параметров методов компонента.

Для описания и инициализации разных типов компонентов и интеграции их в новый проект используются специализированные шаблоны. Тип компонента имеет функциональность и поддерживается стандартным набором методов JAVA, которые обеспечивают инициализацию, запуск, функционирование и уничтожение компонента.

Инструментарий NFTW (New From Teplate Wizard) обеспечивает поиск, селекцию и подключение компонентов к выбранному пакету или проекту. Верхний уровень иерархии задает основные шаблоны для построения новых компонентов и интеграции их в проекты. Этот уровень определяется с помощью основных типов компонентов в языке программирования JAVA, к которым относятся: проекты, формы (AWT компоненты), beans компоненты, COBRA компоненты, RMI компоненты, стандартные классы–оболочки, базы данных, JSP компоненты, сервлети, XML документы, DTD документы, файлы разных типов и их групп [3-5]. Рассмотрим основные типы компонентов

Шаблон развертывания представляет собою скрытую часть и необязательную часть абстракции компонента, который может быть повторно использован в одном или многих средах и для этого он имеет несколько шаблонов отладки. К спецификации компонента могут добавляются новые шаблоны его интеграции или изменяются старые шаблоны. В некоторых классах ПИК параметры интегрирования в новое среды включаются в интерфейс компонента, что ограничивает способность компонента адаптироваться к новым средам и тем самым сужается круг задач, в которых он может повторно использоваться.

Для селекции и подключения нового компонента избранного типа используется механизм NFTW в JAVA. Набор параметров для интегрирования нового компонента в определенном пакете варьируется в зависимости от типа компонента.
Приведем дальше краткое описание функциональности и обзор шаблонов развертывания для отдельных типов компонентов в JAVA.

Проекты как средство композиции компонентов. Создание нового проекта состоит в конфигурации системы с помощью компонентов JAVA и обеспечении их взаимодействия следующими шагами:

– скомпилировать разные файлы с разными JAVA компонентами одной командой; установить основной компонент (класс) в проекте, который задает шаблон кооперации других компонентов в проекте;
– установить уникальную конфигурацию для каждого отдельного проекта,
– поддержать соответствующую файловую систему,
– установить уникальные типы компилирования, выполнения и отладки;
– подключить к работе иерархию окон.

Базовые операция проекта – это создание нового проекта, импорт компонентов из другого проекта, создание новых компонентов с помощью “Мастера шаблонов”, компиляция, выполнение и отладка группы подключенных к проекту компонентов как единой композиции. Проект является мощным средством разработки, сохранения и корректировки шаблона для поддержки взаимодействия разных типов компонентов для решения одной задачи и для последующего повторного использования.

Для реализации ПИК типа проект JAVA предлагает ряд шаблонов для развертывания компонентов:

- BlankAntProject создает проект, который не содержит в себе ни одного класса или пакета классов, разрешает подключать новые классы и пакеты в схему проекта;
- SampleAntProject разрешает сконфигурировать общую схему проекта с помощью иерархии системы файлов как корневой узел схемы нового проекта. Этот шаблон создает первичный эскиз проекта. Потом предоставляется возможность добавлять компоненты к проекту, делать их пакетирование и просматривать более детально для дostройки отдельных компонентов.
- CastomTask. разрешает создать новый проект, начиная с формирования первоначального класса в этом проекте.

Классы – основа ЯП JAVA, порождаются с помощью ключевого слова Extends, после которого указывается тип компонента (например, JApplet). В проектах используются основной класс, с которого начинается выполнение проекта, и вторичный класс. К основному классу относится Class, Main, Empty (пустой класс) и шаблоны типа:

– exception применяется для создания класса, его исключений и соответствующих сообщений об ошибках, которые могут случиться в программе;
– persistence–Capable разрешает отобразить реляционную схему БД и использовать ее для создания БД без подключения к MySQL;
– interface – шаблон, который помогает создать новый JAVA интерфейс и в дальнейшем использовать его любым классом с помощью ключевого слово implements.

При построении классов с помощью шаблонов применяются стандартные классы–оболочки (Boolean, Character, BigInteger, BigDecimal, Class), класс работы из строчными переменными, класс–коллекция (Vector, Stack, Hashtable, Collection, List, Set, Map, Iterator) и класс–утилита (Calendar, работа с массивами, работа со случайными числами).

Формы. Интерфейсы компонентов содержат методы работы с графическими объектами и классы, реализующие эти методы, подключаются к AWT библиотеке классов, каждый из которых описывает отдельный графический компонент, применяемый независимо от других элементов. В AWT существует класс Component, а графический компонент является экземпляром этого класса. При выводе графического элемента на экран он размещается в окне дисплея, как потомок класса Container.

Библиотека AWT содержит формы, каждая из форм представляет собою контейнер для размещения графических элементов интерфейса пользователя, а также систему классов Abstract Window Toolkit для построения абстрактного окна.

Различаются AWT формы и Swing формы. AWT формы построены на базе“тяжелых” интерфейсов (peer–интерфейс), а Swing формы – на базе “легких” интерфейсов. В разных средах AWT компоненты имеют вид, специфический для данной среды, а Swing компоненты выглядят одинаково в разных средах и сохраняют этот вид (“plaf” – Pluggable Look and Feel) за счет того, что они разработаны средствами ЯП JAVA независимо от платформы. Swimg и AWT библиотеки используются самостоятельно.

Все упомянутые окна применяются как контейнеры, к которым можно добавлять более простые графические элементы интерфейса с пользователем (кнопок, полос прокрутки, разных типов меню и т.п.). Интеграция простых компонентов в программный код происходит с помощью панели с изображением всех графических компонентов, изменение которого выполняются автоматически. Необходимые методы обработки форм подключаются к коду с помощью окна Inspector Components.

Аплет представляет собою небольшую программу, доступную на Internet сервере, автоматически устанавливается и выполняется WEB браузером или программой просмотра аплета appletviewer пакета JDK (Java developer Kit). Аплети не выполняются JAVA интерпретатором, а работают в консольном режиме. После компиляции аплет подключается к HTML файлу, использующий тег <applet>. Компонент JAVA Applet обеспечивается набором стандартных методов инициализации, запуска, подключения аплета в требуемый WEB контекст для работы с аудиоклипами, с URL адресами, с объектами типа Image и др.

Диалоговая форма создается в виде окна для поддержки диалога с пользователем, имеет механизм открытия и закрытия в зависимости от интерфейса с пользователем, может существовать при условии, если оно принадлежит определенному окну–фрейма.

Каждое такое окно может быть модальным (из него невозможно выйти, пока пользователь не выполнит все приписанные ему действия) и немодальное, из которого можно выйти в любой момент времени.

Фрейм представляет собою окно со строкой заголовка, которое может быть встроено в апелет или существовать само по себе в программе. Чаще всего фрейм используют для того, чтобы сделаться собственником других окон, которые имеют свой порядок открытия и закрытия, но могут существовать только как подокна Frame.

Панель – это область окна (фрейм или диалоговое окно), в которой могут быть собраны разные элементы, открываемые и закрываемые вместе с панелью. Swing формы представляют набор компонентов интерфейса пользователя, подобных функциональности AWT формам, но реализованных на языке JAVA, что дает Swing компонентам быть независимым от платформы компонентов.

Для создания наиболее употребляемых форм в языке JAVA используются шаблоны:
– Application, который создает фрейм, в состав которого входит трехуровневое меню;
– MDI Application служит для создания фрейма, в состав которого входит меню и панель с заведомо определенными в ней элементами;
– OkCancelDialog создает диалоговое окно, которое имеет обязательно две кнопки – Ok и Cancel.

11.1.2. Система CORBA и средства описания объектов и компонентов

Архитектура CORBA (Common Object Request Broker Architecture) предоставляет распределенный обмен данных с помощью специальных объектов–брокеров ORB (Object Request Broker), применяется для обработки запросов. Эта технология позволяет прикладной программе запрашивать сервисы другой программы, вызывая методы ее удаленных объектов. Для реализации взаимодействия объекта–брокера и программы (клиента и сервиса) используется язык интерфейса IDL (Interface Definition Language). JAVA включает ORB и компилятор idltojava, который генерирует код по IDL спецификациям [8 –10].

Работа в системе CORBA сводится к генерации JAVA файлов на основе IDL файла интерфейса для стороны сервера или для стороны клиента или для обеих. Со стороны клиента требуется специфическая IOR форма (Interoperable Object Reference), которая поддерживает именование сервера. CORBA использует браузер для просмотра имен сервисов, генерации кода и вставки его в файл класса клиента. Со стороны сервера дополняется код, который связывает экземпляры сервентов с именами сервисов. Браузер может сгенерировать этот код и вставить его в файл класса сервера. IDL файл компилируется и полученная программа запускается для выполнения.

В системе CORBA имеются следующие шаблоны *интеграции* компонентов:
– Client class для вызова метода, который будет выполнен сервером.
– Stub class для конвертации метода, инициирующего работу клиента в Wire формате, используемом при связывании в сети на стороне клиента;
– ORB class управляет методами передачи данных и вызовами методов между процессами;
– Implementation class содержит деловую логику сервера, экземпляр этого класса сервент регистрируется в ORB и может использоваться клиентом для запуска другого процесса;

– Server class создает сервент и ссылку IOR, доступную ему, которую он записывает в стандартный выходной файл;
– Skeleton class конвертирует инициирующий метод с Wire форматом с помощью ORB брокера в формат, который может прочитать экземпляр сервента.

Для реализации ПИК типа CORBA компоненты используют как шаблон, так и мастер создания CORBA компонентов. Система CORBA предлагает шаблоны для создания и реинженерии компонентов.

Шаблон поддержки работы адаптера POA (Portable Object Adapter) является корнем каждой иерархии серверов и доступен разным ORB. Адаптер порождает следующие типы объектов: пустой сервер (Empty), основной класс сервер (ServerMain), класс клиент (ClientMain) и простой Simple (сервер с минимальными инициированными элементами).

При использовании мастера инициализации CORBA компонентов пользователь должен использовать три параметра (value, title, type), каждый из которых задается переменной строкового типа, значения которого понимает мастер. Например:

<server– binding name = ‘Proprietary Binder’ template–tag = ‘SERVER_BINDING’>
<wizard requires–value = ‘/* FFJ_COBRA_TODO_SERVER_NAME*/’
title = ‘Server name:’ type = ‘string’/>

Сервлет – это небольшая программа, которая выполняется на серверной стороне WEB, расширяет функциональные возможности WEB сервера, облегчает доступ к ресурсам и разрешает процессу читать данные из HTTP, запрашивать WEB–сервер и записывать данные из сервера в http, как ответ. Они выполняются в границах адресного пространства WEB–сервера. Сервлеты – альтернатива CGI (Common Gateway Interface), которые базируются на взаимодействии процесса запроса клиента с WEB–сервером. CGI программы разрабатываются на разных ЯП и являются необходимыми при создании отдельного процесса обработки каждого запроса клиента. Сервлеты не требуют больших ресурсов памяти и процессора, описываются на языке JAVA независимо от платформы, размещаются в разных средах и используют библиотеку классов JAVA для получения параметров инициализации, активизации и регистрации событий, а также для доступа к информации и формирования ответа клиенту.

Создание сервлетов в языке Java осуществляется с помощью инструментария Servlet Development Kit (JSDK) с применением следующих шаблонов создания и интеграции:

– WebModule – элемент WEB ресурса, который может разворачиваться в прикладной программе для поддержки функционирования сервлета, использует спецификации сервлетов и серверных страниц. Может содержать в себе файлы с Java классами, которые поддерживают сервлети, beans компоненты с тем же назначением, страницы сервера, статические HTML документы и аплеты;
– WebModuleGroup – шаблон для создания группы взаимодействующих WebModule на WEB сервере и поддержки создания сервлетов.
– Servlet;
– HTML File.

Методы спецификации компонентов. Спецификация – это описание компонента и способа вызова компонентов из другой среды, или из электронной библиотеки, или

репозитария. Спецификация включает интерфейс, контракт и нефункциональные свойства.

Интерфейс компонента может быть определен как спецификация точек доступа к компоненту. Клиент получает сервис, который предоставляет компонент, через эти точки доступа. Так как интерфейс не дает реализацию его операций, а предоставляет их описание имеется возможность изменять реализацию без изменения интерфейса, а также добавлять новые интерфейсы (и реализацию).

Семантика интерфейса может быть представлена с помощью контрактов, в которых определяются глобальные ограничения, которые поддерживает компонент т.е инвариант. Контракт может определять ограничения операций, которые должны быть выполнены клиентом перед вызовом операции (пред–условия), и условия, которые предлагает установить после завершения операции (пост–условия). Вместе пред–условия, инвариант и пост–условия составляют спецификацию поведения компонента.

Каждый интерфейс состоит из набора операций (сервисов, которые он предлагает или требует). С каждой операцией связанный набор пред–условия и пост–условия, зависящие от состояния, которое получает компонент. Контракты и интерфейс связаны между собою, но наполняются по–разному. Интерфейс отражает функциональные свойства и состоит из набора операций для спецификации сервисов, а контракт отражает семантику и описание поведения компонента, зависящее от взаимодействия с другими компонентами.

Нефункциональные свойства задают общие черты поведения компонента, которые не могут быть выражены через стандартные интерфейсы. Примерами нефункциональных свойств являются надежность, живучесть, доступность и др. Все эти свойства можно предоставлять как специальные расширения интерфейса, которые обеспечивают сервисы описания компонента, динамической конфигурации, динамического взаимодействия с другими компонентами или средой.

11.1.2.1. Язык описания интерфейсов в системе CORBA

Для задания взаимодействия объектов в системе CORBA используется язык описания интерфейсов IDL, который независим от языка описания самого объекта, а именно С, С++, Паскаль и др. Интерфейсы объектов в IDL-языке запоминаются в репозитории интрфейсов (Interface Repository), а реализации объектов -- в репозитории реализаций (Implementation Repository). Независимость интерфейсов от реализаций объектов позволяет их использовать статически и динамически разными приложениями.

Объект-клиент и объект-сервер обмениваются между собой с помощью запросов, каждый из которых исполняется брокером ORB с помощью компонентов, создаваемых на основе описания интерфейсов клиента, сервера и ядра ORB.

Интерфейс клиента (Client Interface) обеспечивает взаимодействие с объектом-сервером с помощью ORB и состоит из трех интерфейсов:

– stub-интерфейса, содержащего описание внешне видимых параметров и операций объекта в IDL-языке, генерируется в статическую часть программы клиента и хранится в репозитарии интерфейсов;

– интерфейса динамического вызова (Dynamic Invocation Interface – DII) объекта, определяемого во время выполнения программы клиента посредством поиска описания интерфейса в репозитории интерфейсов или в репозитории реализаций;
– интерфейса сервисов ORB (ORB Services Interface), содержащего набор сервисных функций, которые клиент запрашивает у сервера через брокера.

Stub-интерфейс – клиентский интерфейс, обеспечивает взаимосвязь клиента с ORB. Прикладная программа клиента через посредника stub – статической части программы клиента посылает в запросе параметры, которым сопоставляются соответствующие описания из репозитория интерфейсов.

Интефейс DII обеспечивает доступ (извлечение) объектов и их интерфейсов во время выполнения. Этот интерфейс становится известным во время выполнения и доступен благодаря вызова брокера ORB. В каждом вызове указывается тип объекта, тип запроса и параметры. Такую информацию посылает прикладная программа либо она извлекается из репозитория интерфейсов или репозитория реализаций.

Компонент обеспечения сервиса - *объектный адаптер* (*Object-Adapter*) позволяет экземплярам объектов обращаться к большинству сервисных функций ORB, включая генерацию и интерпретацию ссылок на объект, вызов методов, защиту, активизацию (поиск и выполнение объекта), отображение ссылок в экземпляры и регистрацию объектов. Существует несколько видов адаптеров:

– базовый адаптер (*Basic Object Adapter -- BOA*), который может обеспечивать выполнение объектов независимо от брокера;
– библиотечный адаптер (*Library Adapter*), обеспечивающий выполнение объектов, хранящихся в библиотеке объектов и вызываемых из прикладной программы клиента;
– адаптер БД (*Database Adapter*), обеспечивающий доступ к объектно-ориентированным БД.

11.1.2.2. Язык описания интерфейсов объектов

Язык IDL позволяет описывать типы данных, интерфейсы объектов и модули, которые вызываются для выполнения, а также предоставляет средства для описания параметров объектов, передаваемых в сообщении другим объектам. В нем описываются интерфейсные программы клиента и сервера (клиент-stub и сервер-skeleton), а сами программы описываются в ЯП С++ или JAVA.

Описание интерфейсов начинается с ключевого слова **interface**, за которым следует идентификатор (имя интерфейсной программы), образующие вместе заголовок, и тело, содержащее описание типов параметров для обращения к объекту. Пример описания заголовка описания интерфейса:

interface A { ... }
interface B { ... }
interface C: B,A { ... }.

Тело интерфейса содержит описание: типов данных (type_dcl), констант (const_dcl), исключительных ситуаций (except_dcl), атрибутов параметров (attr_dcl), операций (op_dcl).

Описание типов данных начинается ключевым словом *typedef*, за которым следует базовый или конструируемый тип и его идентификатор. В качестве константы может

быть некоторое значение типа данного или выражение, составленное из констант. Типы констант могут быть: integer, boolean, string, float, char и др.

Описание операций op_dcl включает: атрибуты операции, тип результата, наименование операции интерфейса, список параметров (от нуля и более) и др.

Атрибуты параметров могут начинаться следующими служебными словами:
in - при отсылке параметра от клиента к серверу;
out - при отправке параметр-результатов от сервера к* клиенту;
inout - при передаче параметров в оба направления (от клиента к серверу и от сервера к клиенту).

Описание интерфейса может наследоваться другим объектом, тогда такое описание интерфейса становится базовым. Пример базового интерфейса приведен ниже:

```
const long l=2
interface A  {
    void f (in float s [l]);
 }
interface  B {
    const long l=3
 }
interface C: B,A {  }.
```

В нем интерфейс С использует интерфейс В и А и их типов данных, которые по отношению к С – глобальные. Имена операций могут использоваться во время выполнения интерфейсного посредника (Skeleton) для динамического вызова интерфейса. Пример описания интерфейса для динамического вызова приведен ниже:

```
interface Vlist  {
status add_item (
    in Identifier item_name,
    in typeCode   item_type,
    in void       * value,
    in long       value_len,
    in Flags      item_flags
           );
status free ( );
status free_memory( );
status get_count (
    out long count);
           };
```

Описание модуля в языке IDL начинается с ключевого слова module, за которым следует имя модуля и описание его тела.

Средства описания типов. Язык IDL позволяет описывать типы данных, которые задают параметры, передаваемые от объекту к объекту. Типы данных подразделяются на базовые, ссылочные и конструируемые.

К базовым типам относятся фундаментальные типы данных:

16- и 32-битовые (короткие и длинные) знаковые и беззнаковые двухкомпонентные целые;
32- и 64-битовые числа с плавающей запятой, что соответствует стандарту IEEE;
символьные;
8-битовый непрозрачный тип данных, обеспечивающий преобразование данных в момент пересылки между объектами;
булевые (TRUE, FALSE);
строка, которая состоит из массива одинаковых длин символов, допустимых во время выполнения;
перечисляемый тип, включающий упорядоченную последовательность идентификаторов;
произвольный тип any, который представляет любой базовый или конструируемый тип данных.

Конструируктивные типы создаются из базовых типов и включают:
– запись, состоящая из множества упорядоченных пар (имя-значение);
– структура, состоящая из совокупности разнородных базовых элементов;
различительное объединение, содержащее дискриминатор, за которым располагается подходящий тип и значение;
– последовательность, представляющая собой массив, компоненты которого имеют переменную длину и одинаковый тип;
– массив, состоящий из компонентов фиксированной длины одинакового типа;
– интерфейсный тип, специфицирующий множество операций, которые клиент может послать в запросе.

Каждому типу данных соответствует значение, которое задается в запросе клиента или объекта, отправляющего ответ на запрос.

11.1.2.3. Интегратор объектов – брокер объектных запросов

Системную функцию «интегратора» в CORBA выполняет брокер ORB и механизм их удаленного вызова. В рамках CORBA определена эталонная объектная модель, для объектов которой определяются свойства, характеристики и типы данных. Объекты, обладающие одинаковыми свойствами группируются в классы. Каждому объекту соответсвует одна или несколько операций вызова его методов. После выполнения операции объект приобретает некоторое состояние, которое влияет на его поведение. Эталонная модель включает:

– язык IDL и транслятор интерфейса компонентов приложений (Application Interface);
– общий объектный сервис (Common Object Services) для управления событиями, транзакциями, интерфейсами, запросами и др.;
– общие средства (Common facilities), необходимые для групп компонентов и приложений (электронная почта, телекоммуникация, управления информацией, эмулятор программ и др.);
– брокер объектных запросов;

При выполнении сервисных функций брокер ORB выполняет запрашиваемые объектами или приложениями сервисы, общая характеристика которых приведена ниже.

Общие объектные сервисы обеспечивают базовые операции для логического моделирования и физического хранения объектов, определяют совокупность операций, которые могли бы реализовывать или наследовать все классы. Сервисы описываются с помощью специального сервиса спецификаций (Common Object Services Spesification – COSS), который определяет набор объектов, их имена, события, взаимодействие и т.п . Операции, предоставляемые объектными сервисами, становятся доступными приложению через ORB, поддерживают работу с объектами, их существование и независимость от приложений, которые к ним обращаются.

Общие средства обслуживания содержат возможности, облегчающие построение приложений для функционирования в среде ORB. Для конечных пользователей эти средства обеспечивают унифицированную семантику общих компонентов и взаимодействие с другими объектами через брокер ORB и объектный интерфейс.

Объектные приложения – это приложения, разрабатываемые независимыми разработчиками на основе объектного подхода в виде связанного набора прикладных функций, имеют доступ к сервису и услугам CORBA через стандартный интерфейс.

Каждому объекту приложения соответствует метод, который реализует некоторую функцию, имеет доступ к другим объектам, изменяет данные и создает результат. Параметры метода могут быть классифицированы так:

in–параметр (Input) для описания входных констант, массивов, ссылок и т.п.;
in – атрибут –атрибуты класса, которые не изменяются методом;
out – параметр – значение, которое возвращает первичный метод;
out – вторичный – значение, которое возвращает вторичный метод;
out – атрибут – атрибут класса, который изменяет метод;
in – вторичный – возврат результата первичному методу, как **out**–параметры.

Объект сервера считается **in**–вторичным, если вторичный метод прямо или опосредованно изменяет его состояние. Влияние определенного метода на смену состояния объектов отмечается присущей нему комбинацией классов его параметров: не меняющих состояние, сменяющих состояние непосредственно (прямо). Типичные случаи таких комбинаций можно рассматривать как паттерны [10] потоков данных (data flow).

11.1.3. Средства унифицированного процесса RUP

RUP (Rational Unified Process) – это процесс моделирования и построения ПС из объектов с применением языка UML. Он включает теоретический и прикладной аспекты представления и толкования создаваемых моделей для проектируемой предметной области [12, 13].

Теоретический аспект процесса моделирования моделей ПС поддерживается методами и понятиями формальных теорий. Формализация моделей в RUP обеспечивается средствами UML и дает возможность строго описывать требования и преобразовывать их к готовому продукту.

Основу процесса моделирования составляют *прецеденты* – варианты использования для определения требований к системе. Главный элемент проектирования – модель вариантов использования, на основе которой разрабатываются модели анализа, проектирования и реализации системы. Каждая модель анализируется на соответствие

модели вариантов использования, в которую входят входные данные для поиска и спецификации классов и подсистем, для подбора и спецификации тестов, а также при планировании итераций разработки и интеграции ПС. В процессе моделирования создаются следующие модели:

– вариантов использования, отражающих взаимодействие между пользователями и ПС; анализа, обеспечивающего спецификации требований к системе и описание вариантов использования как кооперации между концептуальными классификаторами;
– проектирования, ориентированного на создание статической структуры и интерфейсов системы, реализацию вариантов использования в виде набора коопераций между подсистемами, классами и интерфейсами;
– реализации, включающей компоненты системы в исходном виде на ЯП;
– тестирования;
– размещения компонентов и выполнение в операционной среде компьютеров.

Эти модели представляются разными видами диаграмм, например, в модели вариантов использования диаграммы use–case, в моделях анализа – диаграммы классов, кооперраций и состояний. Данные модели – взаимосвязанные, семантически пересекаются и определяют систему как единое целое. Например, вариант использования в соответствующей модели может иметь отношение зависимости к кооперации в модели проектирования, задающей реализацию. Модели, определенные на каждой итерации процесса RUP, уточняются или расширяют модели предыдущих итераций процесса

Типы моделей и их связи показаны на рис.11.1, каждая из моделей задается соответствующими диаграммами. Например, модель анализа состоит из диаграмм классов, состояний и кооперации.

Артефакты одной модели связаны между собой и должны быть совместимы друг с другом. Отношения между моделями являются не полностью формальными, поскольку части моделей специфицированы на языке метамодели, а другие описаны только неформально, на естественном языке. Спецификации диаграмм UML является также полуформальними.

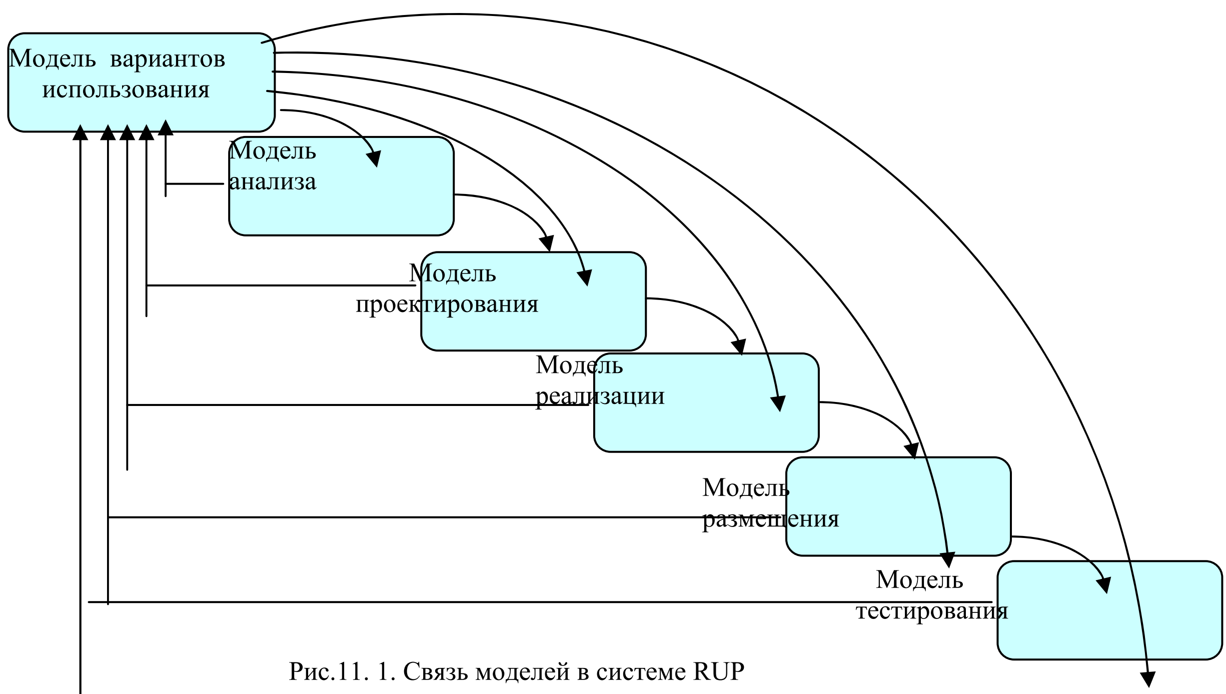

Рис.11. 1. Связь моделей в системе RUP

Основу модели анализа составляет диаграмма варианта использования, которая включает в себя диаграммы классов, взаимодействия, задающие возможные сценарии вариантов использования в терминах взаимодействия объектов на этапе анализа.

Варианты использования специфицируют тип отношений между действующим лицом (актером), пользователем и системой. На высоком уровне абстракции они представляются упорядоченной последовательностью действий или альтернатив.

Вариант использования в UML – это разновидность классификатора, операциями которого являются сообщения, получаемые экземплярами конкретного варианта использования. Методы задают реализацию операций в терминах последовательностей действий, выполняемых экземплярами варианта использования. Пример.

Пусть *uc* – вариант использования (uc – use case), операция которого выполняется над учетной записью и имеет следующее определение:
uc.operations = <*op1*>
op1.name = запрос и обновление учетной записи
op1.method.body = {< проверка идентификации пользователя, наличия сервиса, запроса о долгах, обновление учетной записи >,
< проверка идентификации пользователя, отклонение учетной записи >,
< проверка идентификации пользователя, наличия сервиса, отклонение учетной записи >,
< проверка идентификации пользователя, проверка наличия сервиса, запроса о долгах, запроса на оплату, обновление учетной записи >}

Тело метода – процедура, специфицирующая реализацию операций в виде последовательности действий *op.method.body* или *op.action Sequence*. Между именами действий варианта использования и именами действий в кооперации устанавливается отображение, что обеспечивает гибкость в процессе разработки и модификацию имен действий. Между кооперацией и вариантом использования *uc* создается отношение реализации.

Вариант использования реализуется кооперацией, если роли классификаторов в ней взаимодействуют для обеспечения поведения. Если кооперация имеет более сложное поведение, чем специфицированное вариантом использования, то этот вариант использования – частичная спецификация поведения кооперации. Варианты использования специфицируют действия, видимые за пределами системы, но не специфицируют внутренних действий (создание и удаление экземпляров классификаторов, взаимодействие между экземплярами классификаторов и т.д.).

Определение расширения включает как условие расширения, так и ссылку на точку расширения в целевом варианте использования, которая является позицией внутри варианта использования. Как только экземпляр варианта использования достигает точки расширения, на которую ссылается это отношение, проверяется его условие. Если условие выполняется, последовательность, удовлетворяющая условиям в экземпляре варианта использования, расширяется таким образом, чтобы включить в себя последовательность расширяемого варианта использования.

С практической точки зрения RUP представляется упорядоченным набором шагов и этапов ЖЦ, которые выполняются итеративно. Этот процесс является управляемым,

как в смысле задания требований, так и реализации функциональных возможностей ПрО с заданным уровнем качества и гарантированными затратами согласно графика работ. Оценка качества всех шагов и действий участников процесса базируется на определенных критериях.

Шаги при выполнении RUP управляются прецедентами, т.е. технологическим маршрутом от делового моделирования и требований до испытаний. Экземпляр прецедента – это последовательность действий, выполняемых системой с наблюдаемым результатом для конкретного субъекта. Функциональные возможности системы определяются набором прецедентов, каждый из которых представляет некоторый поток событий. Описание прецедента определяет то, что произойдет в системе, когда прецедент будет выполнен. Каждый прецедент ориентирован на задачу, которую он должен выполнить. Набор прецедентов устанавливает все возможные пути (маршруты) выполнения системы. Прецеденты играют роль в каждом из пяти основных работ процесса: формирование требований, анализ, проектирование, реализация и испытание.

RUP содержит пять основных этапов, выполняемых на всех фазах процесса разработки ПС. Завершение этих этапов называется итерацией, при которой заканчивается выпуском промежуточного продукта. На каждой итерации цикл повторяется, начиная со сбора и уточнения требований.

Этап формирования требования. На этом этапе проводится сбор функциональных, технических и прикладных требований к проекту. На основе требований заказчика и пользователей система описывается так, чтобы достичь понимания между ними и проектной группой. Информация собирается с учетом особенностей существующих систем и документов, подготовленных заказчиком, и включает:
– модель ПрО;
– модель схем использования с описанием функциональных и общих требований в форме результатов опрашивания, наборов диаграмм и детального описания каждой схемы;
– дизайн и прототип интерфейса пользователя для каждого актера;
– список требований, которые не относятся к конкретным схемам использования.

Этап анализа. Сформулированные требования уточняются и отображаются в модели сценариев использования. Кроме того, создается аналитическая модель системы, которая включает формализмы для анализа внутренней структуры системы, определения классов и превращения этой модели в проектные концепции и схемы их реализация.

Этап проектирования служит для уточнения классов и описания их относительно четырех уровней: пользовательского интерфейса, бизнесов–решений, уровня доступа и уровня данных. Создаваемая проектная модель системы состоит из структуры подсистем, их распределения между уровнями, интерфейсов классов и объектов, связей классов с узлами развертывания (модель развертывания).

На этапе реализации выполняется построение прототипа из компонентов; создания тестов по схемам использования; тестирование и интеграция компонентов; проверка архитектуры; переход к следующей итерации. При каждой итерации тестовая модель уточняется путем исключения неактуальных тестов, создания схемы регрессионного тестирования и добавления тестов для собираемых компонентов. Каждый тест

создается с помощью вариантов использования и реализует конкретный метод проверки функций системы на входных данных.

RUP – методология оформлена и размещена в Web базе знаний поисковой системы. В ней регламентированы этапы разработки ПО, документы и инструментальные средства для обеспечения каждого этапа ЖЦ.

11.2. Энциклопедия инструментов создания ПС из объектов и компонентов

Основным инструментом проектирования приложений является CASE–система Rational Rose [14], основанная на применении языка UML для представления архитектуры ПС с помощью разных видов его диаграмм. Средства системы на проектирование и моделирование общей модели (абстрактной) предприятия, постепенно уточняемой до конкретной (физической) модели классов создаваемой ПС. Допускается доработка созданной системы. Результатом моделирования является визуальная логическая модель системы.

Rational Rose – средство для проектировщиков, аналитиков, разработчиков объектно–ориентированных информационных систем на языке UML, представляемых в виде файлов логической модели, используемой при кодировании средствами конкретного ЯП (C++, Ada, Java, Basic, Xml, Oracle). Имеются специальные мосты связи с системой Delphi, что дает возможность связывать программный код системы с БД. Допускается обратное проектирование, т.е. преобразование готовой информационной системы (например, на C++) или база данных (в Oracle) в наглядную визуальную, структурную модель ПС.

Управление требованиями. RequisitePro – инструмент для ввода и управления требованиями. Продукт позволяет получать, создавать и структурировать наборы вводимых требований в наглядной форме. Предусмотрен набор атрибутов требований для расширения схожих (готовых) в рамках одного или нескольких проектов. Для каждого требования сохраняется его история, позволяющая отследить, какие изменения были внесены в требование. Все документы и данные, относящиеся к требованиям, централизованно ведутся. Их, а также сценарии, функциональные и нефункциональные спецификации и планы тестирования представлено в удобном для использования в управлении проектом.

Управление проектом и версиями. ClearCase – система управления программным проектом, сохраняет в архиве все изменения и версии, которые были внесены в проект. Эти данные хранятся в масштабируемых репозиториях, а также хранятся исходные тексты программных модулей, исполняемые и объектные модули, библиотеки DLL. Система позволяет работать как по командам, так и по времени выхода из нее, впоследствии объединяются данные с общим проектом. Можно выводить подробные характеристики измененных файлов в наглядной графической форме, представлять все изменения в виде дерева версий. При наличии MultiSite открывается возможность обмена данным между группами разработчиков, географически удаленных друг от друга. При этом ClearCase берет на себя все рутинные операции по пересылке/приемке информации.

ClearCase рекомендован для контроля команды разработчиков, объединяющим всех участников проекта в единую среду, хранящую всю возможную информацию по проекту. При получении последней версии редактируемых файлов, проводится их

контроль, управление рабочим пространством с помощью уникального инвариантного подхода. При применении ClearCase команда разработчиков может ускорить циклы разработки системы и создать новые, надежные в эксплуатации продукты, а также дорабатывать и поддерживать ранее реализованные продукты без изменения среды, инструментов и метода разработки.

Каждый участник проекта может иметь доступ, как ко всем файлам проекта, так и к только определенной его части. Для достижения подобного эффекта ClearCase использует мощную систему настраиваемых фильтров, скрывающих ненужную информацию. Система позволяет осуществить параллельную разработку, а также отдельному участнику проекта выходить из общего состава разработки, забирая работу “на дом”, а после всех внесенных изменений вернуть ее снова в проект. При этом ClearCase осуществит автоматическое слияние версий.

В ClearCase проводится контроль конфигураций и версий на основе сохранения всей истории каждого файла проекта, включая миграцию файлов между независимыми проектами. Продукт ориентирован на всех участников команды: директоров, менеджеров, разработчиков, аналитиков, тестировщиков, технических писателей.

Ведение изменений. ClearQuest для архивирования всех изменений и создания БД в MS SQL, MS Access, Sybase SQL Anywhere или Oracle. Имеется возможность добавлять собственные SQL–запросы уже к готовой базе данных. Основные задачи, решаемые ClearQuest:
– управление изменениями, возникающими в ходе процесса разработки ПО;.
– оптимизация пути прохождения запросов и связанных с ними форм и процедур;
– поддержка связи объектов, разделенных территориально через World Wide Web;
– внедрение надежного и проверенного процесса CRM, либо изменение уже существующего процесса для удовлетворения специфическим требованиям;
– визуальный анализ проекта с помощью графического представления информации и отчетов;
– интеграция со средствами конфигурационного управления (Rational’s ClearCase), позволяющая создавать связи между запросами на изменение и развитием кода;
– связь с Sybase, Oracle, Microsoft;.
– интеграция со средствами тестирования Rational (TeamTest, VisualTest, Purify, PureCoverage, Quantify и Robot);
– создание отчетов на базе Crystal Reports (из составаProfessional);
– интеграция через COM с MS Word и MS Excel.

Инструменты измерения. Rational Quantify – инструментальное средство для идентификации "узких мест" в разрабатываемых приложениях, учета производительности, идентифицирующее и выявляющее части приложения, которые замедляют скорость его выполнения. Это средство генерирует в табличной форме список всех вызываемых в процессе работы приложения функций, указывая временные характеристики каждой из них; предоставляет статистику по всем вызовам (внешним и внутренним). Сбор данных осуществляется посредством технологии OCI (Object Code Insertion) путем подсчета циклов, вставки счетчиков в код тестируемой программы. Уникальность данного подхода заключается в тестировании исходного кода, внутреннего представления, а также всех используемых компонентов. Статистическая информация по вызовам может быть перенесена в Microsoft Excel для построения графиков и сводных таблиц для разных запусков приложения.

Основные свойства продукта:
– предоставление точной информации о производительности созданного приложения;
– определение узких мест пользовательских функций, системных вызовов, разделяемых библиотек и библиотек других фирм.

В состав данного средства входит дополнительный набор инструментов, повышающих производительность моделирования отдельных аспектов ПС:
– графическое средство Call Graph из Rational Quantify – наглядное представление данных о критических функциях системы, требующих наибольшего времени для выполнения;
– средства Function List и Function Detail выдает в визуальных окнах данные в табличной форме для непосредственного внесения изменений в участки кода, а также для построчного просмотра данных в аннотированных копиях окон Source Code;
– выход в DCE, систему Solaris и библиотеки SunOS;
– сбор данных о производительности по всем используемым инструментам
– использование языков (C, C++, FORTRAN, Ada и Java) и соответствующих систем программирования.

Тестирование. Visual Test обеспечивает функциональное, не зависящее от языка реализации тестирование 32–битных приложений, написанных для Windows, а также компонентов ActiveX, DLL, сервера автоматизации OLE (OLE Automation server) или приложения на основе Web. Продукт имеет интерфейс с VisualStudio компании Microsoft. Кроме того, он дает возможность создавать поддерживаемые, расширяемые и пригодные для повторного применения компоненты тестирования, а также приспосабливать их во многие версии других проектов.

Rational Robot – средство функционального тестирования, базирующееся на объектно–ориентированной технологии, что позволяет существенно превзойти традиционные средства GUI–тестирования (тестирования графического интерфейса), так как здесь тестируются сотни и тысячи свойств всех объектов приложения (даже скрытых объектов), вместе и для каждого в отдельности. Программа работает в двух режимах: в автоматическом и ручном. В ручном режиме пользователь сам задает на специальном языке сценарий тестирования, в автоматическом режиме система Robot автоматически генерирует необходимый скрипт для дальнейшего повторного тестирования.

LoadTest – средство автоматизированного тестирования характеристик распределенных сетевых приложений на платформах Windows и Unix. При тестировании производительности типично используется нагрузка сервера большим количеством виртуальных пользователей. Например, можно установить таймер для одного VU, чтобы определить время выполнение запроса при посылке запросы на тот же самый сервер в то же самое время множества других VU.

Термин “тесты производительности” включает нагрузочные, стрессовые, конкурирующие и конфигурационные тесты. Совокупность этих тестов позволяет ускорить цикл тестирования производительности. Нагрузочное тестирование в LoadTest выполняется тогда, когда нужно определить время отклика серверов или клиентских приложений при изменяющейся нагрузке и используется тогда, когда нужно вычислить максимальное количество транзакций, которые может выполнить сервер за определенный временной отрезок. Если клиент/серверная система использует распределенную архитектуру, то нагрузочное тестирование может быть использовано для проверки правильности выбранных методов в целях балансирования или конструирования системы.

Нагрузочное тестирование выполняется с использованием режима виртуальных пользователей для измерения время отклика серверной части, а также с использованием пользовательского графического интерфейса для измерения времени отклика системы в конкретном клиентском приложении.

Pure Coverage предназначен для выявления участков кода, пропущенных при тестировании приложения Windows NT и компонентов, написанных на Visual Basic, Visual C++ или Java. Инструмент собирает статистику о тех участках программы, которые во время тестирования не были пройдены выявляет не исполнившиеся строки и анализирует причины их невыполнения.

Управление качеством. Для улучшения качества производимого кода в методологии Rational предусмотрен определенный набор средств: Visual Test, Rational Quantify, Purify, Pure Coverage. Инструмент Visual Test используется для высокоуровневого тестирования ПС и интерфейсов компонентов. Rational Robot и LoadTest обеспечивают загрузочное тестирование приложений клиент–сервер.

Инженерия и реинженерия. Rational Rose Professional имеет набор изобразительных средств и в зависимости от выбранного ЯП осуществляет прямое и обратное проектирование ПС. Результат проектирования – шаблон информационной системы, который необходимо запрограммировать на ЯП. Rose Enterprise для проектирования предприятия с использованием многих перечисленных выше инструментов.

Rose DataModeler компонент Rational Rose, предназначен для проектирования системы и баз данных без кодогенерации.

Rose RealTime – специализированная версия для проведения полной (100%) кодогенерации и реинженерии систем на С и С++ с использованием набора диаграмм UML.

Анализ состояния среды. Purify направлен на разрешение проблем, связанных с утечками памяти и Run–time ошибками. Программа собирает данные о любых потерях в памяти. К ним можно отнести и невозвращение блока, и не использование указателей, и остановку исполнения программы с выводом состояния среды при возникновении ошибки run–time. Разработчик ПС имеет возможность не только видеть состояние исполнения (предупреждения, ошибки) ПС, но и переходить к соответствующим внутренних вызовов других компонентов.

Документация. SoDa – инструмент автоматизации документов и подготовки отчетов по заранее установленному шаблону, по которому компилируется документация в одном документ с текстовыми и графическими данными. Допускается использование стандартных шаблонов, созданных пользователем при помощи Wizard и меню Word.

11.3. Средства и методы разработки архитектуры MSF

Micposoft Solutions Framework (MSF) – комплекс средств и методов процесса разработки проекта из скоординированного набора элементов (программно–технических средств, документации, обучения и сопровождения) для удовлетворения производственной архитектуры [15].

Базисом управления проектом построения производственной архитектуры предприятия является база знаний PMBOK, содержащая следующие области знаний:

– управление объемом работ в проекте,
– управление временем,
–управление стоимость,
– управление качеством,
– управление персоналом,
– управление коммуникациями,
– управление закупками и контрактами,
– управление рисками.

В рамках общего процесса управления проекта используется модель архитектуры предприятия, обеспечивающая планирование корпоративного развития предприятия с учетом четырех основных аспектов: бизнес, приложение, информация, технология.

Под реализацией производственной архитектурой понимается скоординированный технологический план создания и развития информационной системы (ИС) из главных ее элементов, соответствующих приоритету архитектуры и получению максимального эффекта при минимуме затрат с соблюдением баланса между целями и требованиями ИС, главными проектными решениями и человеческими и финансовыми ресурсами организации. Архитектор должен доказать, что затраты времени на разработку плана производственной архитектуры сэкономит время на создание всего проекта, при условии, что планирование, разработка и сопровождение могут осуществляться параллельно.

Так как любая организация имеет сложившуюся производственную архитектуру, то при ее оценке и применяется архитектурно–ориентированный метод для планирования, создания и сопровождения проекта на базе архитектуры более высокого уровня. После определения существующего уровня архитектуры организации можно приступить к планированию более совершенной архитектуры и определить направления работ для достижения цели.

Поэтому жизненно важно выполнять рационализацию производственных процессов, усовершенствовать структуру организации и внедрять новые технологии. Создаваемая ИС должна удовлетворять потребностям клиентов, одновременно поддерживать задачи производства с , приспособлением к технологическим изменениям.

Метод создания производственной архитектуры основывается на приоритетных потребностях бизнеса, принятии выгодных технических решений и возможности изменения технологии и организации производства.

Целью разработки производственной архитектуры есть логически связанный, цельный план работ и скоординированных проектов для преобразования сложившейся структуры ИС и приложений организации в состояние, определенное на основе текущих и перспективных задач и процессов.

Метод MSF компании Microsoft обеспечивает анализ и разработку требований к ПО, а также проектирование проектных решений, основанных на базовых концепциях предприятия и приоритетности архитектуры. Метод включает в себя построение производственной архитектуры, ориентированной на получение бизнеса, и организацию процесса разработки системы для предприятия в условиях, когда архитектура еще не сформирована.

Этот метод включает набор моделей для организации и эффективного создания информационных технологий в бизнесе:
– производственной архитектуры;
– проектной группы;
– процесса разработки ПО;
– управления рисками;
– процесса проектирования;
– приложения.

Модель производственной архитектуры – это набор принципов, обеспечивающих создание версии производственной архитектуры предприятия. Главным ее разработчиком является архитектор, который определяет направления создания и развития ИС исходя из приоритетов предприятия. На основе анализа существующей структуры организации определяются направления достижения поставленных целей создания проекта.

Данная модель является структурной, включает четыре перспективы: бизнес, приложение, информацию и технологию (рис.11.2).

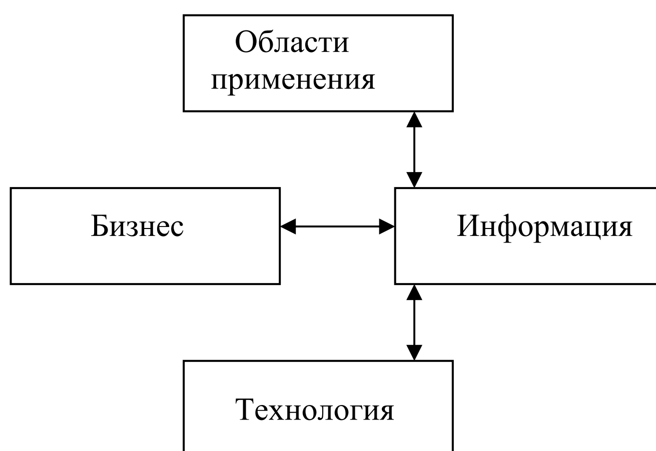

Рис.11.2. Перспективы производственной архитектуры

Рассматриваемая модель состоит из четырех перспектив: бизнеса, приложения, информации и технологии, которые связаны между собой разными зависимостями и взаимодействиями. Основной задачей этой модели является приспособление производственной архитектуры к бизнес–целям организации путем итерационного, поэтапного выпуска серии последовательных версий, ориентированных на указанные приоритеты, выполнение отдельных проектов для постепенного решения задач и последовательной корректировки производственной архитектуры.

Бизнес–перспектива включает стратегии и планы перехода к лучшему состоянию предприятия, а именно глобальные цели и задачи организации; виды продуктов и услуг организации; бизнес–процессы реализации основных функций организации и связи между ними.
Прикладная перспектива (приложение) включает услуги и сервисы, информацию и функции, которые требуются для связи пользователей, в том числе описание сервисов поддержки процессов в бизнес–перспективе; описание взаимодействий и зависимостей корпоративных приложений; совершенствование существующих и развитие новых приложений бизнес–перспективы.

Информационная перспектива основывается на возможностях организации автоматизировать бизнес–задачи с помощью персональных компьютеров, серверов и др. оборудования; ОС, общесистемных средств и сетевых компонентов; принтеров и другого периферийного оборудования; данных, которые хранятся в БД и документов, таблиц, созданных в процессе работы организации.

Технологическая перспектива включает технологию работы с аппаратным и программным обеспечением для регламентации действий разработчиков по созданию производственной архитектуры в рамках имеющейся среды разработки. Эта перспектива направлена на логическое описание инфраструктуры и системных компонентов, которые необходимы для поддержки прикладной и информационной перспектив (топологии, среды разработки, средств защиты), а также на определение перечня технологических стандартов и сервисов для выполнения задач организации.

Модель проектной группы определяет роли, обязанности каждого участника проекта и распределение между ними ответственности. Эта модель служит для формирования эффективной команды и приведения в соответствие содержания проекта с размером группы и квалификацией участников. Члены проектной группы анализируют планы (разработки, тестирования, эксплуатации, мер безопасности и обучения), выявляют взаимосвязи между ними, создают сводный календарный план, в котором предусматриваются версии проекта и проверку их на функциональность на имеющихся ресурсах. Члены группы выполняют определенную роль по оценки состава проектных решений, рисков и ресурсов и корректировки приоритетов.

Модель процесса разработки ПО определяет организационную структуру процесса и руководство им в течение всего времени жизни проекта. Отличительные особенности модели – поэтапность, итеративность и гибкость. Модель определяет фазы, этапы, виды деятельности и результаты процесса разработки приложения и их связь с моделью проектной группы. Использование этой модели обеспечивает контроль за ходом разработки проекта, минимизацию рисков, повышение качества и сокращение сроков выполнения проекта.

На этапе разработки создаются: код приложения, скрипты установки и конфигурации, окончательная функциональная спецификация, спецификации и сценарии тестирования. Все эти работы выполняют члены проектной группы. Они создают инфраструктуру и документ на конфигурацию.

Инфраструктура предприятия предназначена для выполнения требований клиентов к структуре выпуска продукции, а также проведения анализа рынков для продажи продуктов и т.п.

Задачи инфраструктуры состоят в следующем:
– привлечение клиентов к созданию приложения;
– связь с корпоративной сетью;
– сохранение данных на разных компьютерах, которые расположены на разных территориях предприятия;
– выдача информации о продукте через компьютерную сеть и т.п.

Соответственно этим задачам проводится:
– согласование информационных технологий с целями бизнеса;
– обоснование изменений и соответствующих затрат для планирования будущих инвестиций;

– усовершенствование внутренних и внешних связей между подразделениями для повышения эффективности работы с заказчиками, поставщиками и партнерами и т.п..

Модель управления рисками предназначена для управления рисками проекта. Она определяет порядок и условия реализации упреждающих решений и мер для постоянного выявления наиболее существенных моментов риска и реализации стратегии их устранения. Использование этой модели и ее основных принципов помогает команде сосредоточиться на наиболее важных моментах разработки ПО. Модель управления рисками является основой выявления, планирования и мониторинга рисков.

Выявление состоит в анализе и формулировке имеющихся рисков, причиной которых могут быть неучтенные особенности проекта и среды, в которой будет разрабатываться проект. Выявленные риски классифицируются и составляется база знаний о рисках на уровне предприятия.

Формулировка рисков включает рассмотрение условий возникновение рисков и последствий, которые они вызывают. Устанавливаются причинно–следственные связи рисков, проводится приоритезация рисков, составление плана мониторинга рисков и документа с описанием возможных рисков в проекте, в котором определены меры вероятности возникновения риска, схема оценки типа «почти невозможно», «маловероятно», «возможно» и денежные компенсации за предотвращение рисков. В плане графике предусматривается мониторинг рисков – своевременное исполнение превентивных мер для снятия появляющихся угроз риска.

Модель процесса проектирования определяет цели и задачи процесса разработки производственной архитектуры с параллельным и итерационным выполнением отдельных работ. Процесс включает три основные фазы разработки – концептуальное, логическое и физическое проектирование. Переход от концептуального проекта к физическому способствует превращению созданного набора сценариев в совокупность компонентов и сервисов, образующих приложение и реализующих требования заказчика и пользователей.

Процесс проектирования – это систематический способ от абстрактных концепций к конкретным техническим решениям. На этапе выработки концепции формируется набор сценариев использования (usage scenarios), в каждом из которых моделируется выполнение операции определенным пользователем системы. Сценарии разбиваются на последовательность действий – примеров использования (use cases), которые необходимо выполнить пользователю для выполнения операции. Существует три уровня процесса проектирования концептуальный, логический и физический. Процесс проектирования заканчивается описанием функциональных спецификаций.

Модель приложения определяет трехуровневую структуру и сценарный метод проектирования и разработки приложения. Применение этой модели обеспечивает наглядность разработки, параллельное выполнение работ на процессах и различные удобства при эксплуатации и развертывании компонентов приложения на компьютерах и в различных серверах.

Таким образом, **м**етодология MSF обеспечивает проектирование приложения для предприятия с помощью приведенных принципов, моделей и методов решения задач этого предприятия.

Контрольные вопросы и задания

1. Дайте характеристику спецификации компонента.
2. Определите языковые средства описания компонентов.
3. Представьте объекты языка JAVA.
4. Определите методы интеграции объектов языка JAVA.
5. Определите основные характеристики объектов в системе CORBA.
6. Приведите структуру описания спецификации интерфейса в языке IDL.
7. Расскажите об особенностях описания объектов в системе COM.
8. Проведите сравнение средств CORBA и COM.
9. Для каких целей создано Rational Rose.
10. Назовите инструменты Rational Rose какими Вы пользовались.
11. Дайте перечень диаграмм языка моделирования UML.
12. Определите процесс разработки ПС с помощью UML.
13. Для каких целей разработан метод MSF?.
14. Назовите основные модели MSF.
15. Как решаются вопросы управления проектом и рисками в системе MSF.
16. Цели и задачи проектной группы в MSF.

Литература к теме 11.

1. *Crnkovic I., Hinch B., Jonsson T., Kiziltan Z.* Specification, Implementation and Deployment of Components // Communications of the ACM.–2002.–vol.45.–№ 10.–P.35–40.

2. *Гамма Э, .Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно–ориентированного проектирования. Паттерны проектирования.– СПб:Питер, 2001.–368с.

3. *Монсон-Хейфел Р*. Enterprise JavaBeans. – СПб: Символ-Плюс, 2002. – 672 с.

4. *Барлет Н., Лесли А., Симкин С.* Программирование на JAVA. Путеводитель.– Киев.-1996.- 736с.

5. *Холл М.* Сервлеты и JavaServer Pages. Библиотека программиста. – СПб.: Питер, 2001. – 496 с.

6. *Бабенко Л.П., Поляничко С.Л.* Модели классификации объектов программной инженерии// Проблемы программирования.–Вып. 1.–С.25–32.

7. *Грищенко В.Н., Лаврищева Е.М.* Компонентно–ориентированное программирование. Состояние, направления и перспективы развития // Проблемы программирования.–2002.–№ 1–2.–с. 80–90.

8. *Орфали Р., Харки Д., Эрварс Д*. Основы CORBA .– М: НАЛИП, 1999. – 317с.

9. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.

10. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных систем, Киев, Изд. «Наукова думка», 1997г.–228с.

11. *Роджерсон Д*. Основы COM. Руск..пер.- Microsoft Press.- 361с.

12. *Кендалл Скотт.* Унифицированный процесс. Основные концепции.–Москва–С–Петербург–Киев.–2002.–157.

13. *UML*. Специальный выпуск. Питер.–Санкт–Петербург–Москва–Харьков–Минск.–2002. –552с.

14. .*Трофимов С.А*. CASE-технологии: практическая работа в Rational Rose.– 2–е изд.–М.: Бином–Пресс, 2002.–288с.

15. *С.Ф.Уилсон, Б.Мєйлс, Т.Ленгрейв*. Принципы проектирования и разработки программного обеспечения. Учебный курс MCSD.–Пер. с анг.–М.: Из–во торговый дом «Русская редакция», 2000.–608с.

ПРИЛОЖЕНИЕ 1

Словарь терминов программной инженерии

Абстракция – способность отделять существенное от второстепенного, видеть идею, которая определяет реализацию.

Абстрактная архитектура – декомпозированная структура задач домена на подсистемы или иерархию подсистем, на каждом уровне которой фиксируются выделенные параметры и ограничения, определяющие варианты состава выделенных подсистем.

Агрегация – объединение ряда понятий в новое понятие (отношение типа "часть–целое"), общие признаки которого могут быть суммой признаков компонентов или существенно новыми признаками

Актеры – действующие лица, для которых создается система.

Анализ требований – отображения функций системы и ее ограничений в модели предметной области.

Артефакт – любой продукт деятельности специалистов по разработке ПО.

Архитектура программной системы – определение системы в терминах вычислительных подсистем и интерфейсов между ними, отображающая правила декомпозиции проблемы..

Ассоциация – наиболее общее и существенное отношение, которое утверждает наличие связи между понятиями без уточнения их содержания и размеров.

Белого ящика метод – исследование внутренний структуры программы в целях выявления ошибок путем исчерпывающего тестирования всех путей и потоков передач управления.

Валидация – проверка соответствия разработки программной системы требованиям заказчика.

Верификация – проверка правильности реализации системы заданным требованиям на каждом этапе жизненного цикла.

Взаимодействие объектов – связь между объектами через посылку сообщений друг другу.

Водопадная (каскадная) модель – схема работ, в которой каждая из работ выполняется один раз и в том порядке, который указан в модели жизненного цикла.

Гарантия качества программного обеспечения – действия на каждом этапе жизненного цикла по проверке и подтверждении достигаемого качества соответственно стандартам и процедурам.

Дефект – это ошибочное событие в результате неверного описания спецификаций требований, исходных или проектных спецификаций, документации и т.п.

Диаграмма – графическое представление моделирования системы с помощью классов, сценариев, состояний и т.п.

Динамическое тестирование – выполнение программ для обнаружения ошибок, установления их причины и устранения.

Домен предметной области – спектр задач проблемной области, которые допускают похожие приемы их решения.

Жизненный цикл системы (ЖЦ) – непрерывный процесс, который начинается с момента принятия решения о необходимости ее создания и заканчивается в момент ее полного изъятия из эксплуатации.

Задача системы – описание способа (технологии) достижения цели, содержащее указание на цель с конкретными числовыми (в том числе временными) характеристиками.

Имитационное моделирование – моделирование поведения системы в различных аспектах и в разных внешних и внутренних условиях с анализом динамических характеристик и анализом распределения ресурсов

Инженерия – применение научных результатов и дисциплины управления программированием задач в целях получения пользы от свойств продуктов, способов взаимосвязи и выполнения.

Инженерия качества – процесс управления предоставлением продуктам программного обеспечения свойств качества (надежность, сопровождаемость и т.п.).

Инженерия требований – сбор, анализ, оформление условий и ограничений на разработку системы в виде спецификации, согласованной как заказчиком, так и исполнителем.

Интенсивность отказов – это частота появления отказов или дефектов в программной системе при ее тестировании или эксплуатации.

Инспекция кода – формальная проверка описания, используемых типов и структур данных в проекте системы на их соответствие требованиям.

Информационная модель повторно используемых компонентов (ПИК) – модель, с помощью которой отображается информационный (поисковый) образ ПИК в каталоге.

Информационная система – система, которая проводит сбор, обработку, сохранение и производство информации с использованием автоматических процессоров и людей.

Информационное обеспечение – набор средств для предоставления информации пользователям о содержании и условиях ее применения.

Интерфейсные объекты – связка (стыковка) программ, представленная в виде описания передаваемых через сообщения параметров для выполнения.

Интерфейсные объекты – связка (стыковка) программ, представленная в виде описания передаваемых через сообщения параметров для выполнения.

Инцидент – абстрактное событие, влияющее на изменение состояния объекта.

Каркас (фрейм) – разновидность абстрактной архитектуры для определения выделенных в структуре компонентов.

Качество программного обеспечения – совокупность свойств, которое определяют пригодность программного обеспечения удовлетворить заказчика в соответствии его требований к разработке.

Компонент – тип, класс, проектное решение, документация или иной продукт программной инженерии приспособленный для практического использования.

Компонентная разработка – конструирование программного обеспечения путем композиции готовых компонентов, сохраняемых в каталогах.

Конкретизация – добавление существенных признаков, благодаря чему содержание понятия расширяется, а объем понятия сужается.

Конечные пользователи системы – профессиональные лица, для потребностей которых заказывается компьютерная система.

Конфигурация – вариант (версия) изготовленной программной системы из отдельных экземпляров компонентов и подсистем.

Концептуальное моделирование – процесс построения модели проблемы, ориентированной на ее понимание человеком.

Критерий – количественная или качественная характеристика состояния системы, позволяющая оценить степень достижения цели и сформулировать решающие правила выбора средств (способов, технологий) достижения цели.

Критерий эффективности – критерий, позволяющий оценить степень достижения цели с учетом произведенных затрат различных ресурсов.

Менеджмент – профессиональное управление коллективами работников (персоналом).

Метрика – количественная мера и шкалы измерения характеристик программы.

Модель жизненного цикла – типичная схема последовательности работ на процессах разработки программного продукта.

Модель процессов – определенная последовательность действий, сопровождающая изменение состояния программного объектов.

Модель состояний – отображения динамики изменения состояния объекта класса, которое изменяют его поведение.

Модель качества – четырехуровневая структура, которая отображает характеристики, атрибуты, метрики и оценочные элементы программного обеспечения.

Надежность программной системы – это способность системы сохранять свои свойства (безотказность, устойчивость и др.) в процессе преобразования исходных данных в результаты в течение определенного промежутка времени при определенных условиях эксплуатации.

Нефункциональные требования – требования, которое характеризуют организационные, исполнительские, операционные аспекты работы программной системы в среде реализации.

Наследование – конкретизация в подклассе отдельных свойств для использования другими объектами суперкласса.

Отладка – проверка программы на наличие в ней ошибок и их устранение без внесения новых.

Отказ – переход программы из работающего состояния в неработающее в связи с ошибками или дефектами в ней.

Обобщение – сужения истинных признаков понятия для расширения свойств охваченных этим понятием объектов.

Ошибка – недостатки в операторах программы или в технологическом процессе ее разработки, которые приводят к неправильной интерпретации исходной информации и к неверному решению.

Объекты управления (по Джекобсону) – это функции преобразования объектов интерфейса в объекты сущности, часто отображают алгоритмы обработки данных в системе.

Объект–сущности (по Джекобсону) – долго живучие объекты, которые отвечают реальным предметам мира и сохраняют свое состояние после выполнения сценария.

Объектно–ориентированная модель – структура из совокупности объектов, которые взаимодействуют между собою, обладают свойствами и поведением.

Онтология – совокупность элементарных понятий, терминологии и парадигмы их интерпретации в среде проблемы, которую требуется разработать.

Оценочный элемент метрики – количественная или качественная мера для оценка соответствующего показателя с учетом его веса в системе оценки качества.

Оценивание качества – действия, направленные на определение степени удовлетворения программного обеспечения требованиям, соответствующим его предназначению.

Пакет – программная структура с общим механизмом организации элементов (объектов, классов) в группы, начиная от системы (стереотип "система") и к ее подсистемам различного уровня детализации.

Переносимость системы – возможность изменять сервис системы (ОС, связи, , сетевые коммуникации, данные СУБД и т.п.) путем настройки модулей на новые условия среды или платформы.

План тестирования – описание стратегии, ресурсов и график тестирования отдельных компонентов и системы в целом.

Поведение домена – переход элементов домена из состояния к состоянию во времени.

Повторное использование – использование в качестве готовой порции любых формализованных знаний, полученных при реализации программных систем.

Повторно используемый компонент (ПИК) – фрагмент знаний о минувшем опыте программирования системы, представленный так, чтобы его можно использовать не только его разработчиками, но и пользователями после соответствующей адаптации к новой среде.

Прикладная система – продукт программной инженерии, предназначенный для выполнения конкретных задач конечного пользователя.

Прецедент (класс) определяет набор экземпляров прецедента, где каждый экземпляр – это последовательность действий, выполняемых системой, которая выдает наблюдаемый результат, ценный для конкретного субъекта.

Прецедент (экземпляр) – последовательность действий, выполняемых системой, которая выдает результат, ценный для конкретного субъекта.

Принципы – базовые концепции, лежащие в основе всей области программирования.

Приложение – область применения, в которых принципы и практика находят свое наилучшее выражение.

Программная инженерия – система методов, средств и дисциплины планирования, разработки, эксплуатации и сопровождения программного обеспечения, способного к массовому воспроизводству.

Процесс приобретения – действия, которые инициируют определенный цикл анализа для определения покупателем программной системы или сервиса.

Процесс разработки – действия разработчика по инженерии требований, проектированию, кодирование и тестированию программного продукта

Процесс сдачи – действия по передаче разработанного продукта покупателю.

Процесс эксплуатации – действия по обслуживанию системы пользователем.

Процесс сопровождения – действия по управлению модификациями и поддержкой системы в актуальном состоянии при выполнении функций системы или изъятие системы из употребления.

Проектирование – преобразования требований в последовательность проектных решений и их в архитектуру из программных компонентов.

Проектирование концептуальное – уточнение понимания и согласование деталей требований к системе.

Проектирование архитектурное – определение структурных особенностей строящейся системы.

Проектирование техническое – отображение требований среды функционирования и разработки системы путем определения всех конструктивных элементов и их композиций.

Проектирование детальное – определение подробностей реализации функций для заданной среды и связей между соответствующими компонентами системы.

Реализация программной системы – преобразования проектных решений в работающую систему (синонимы: кодирование, конструирование) .

Родовые знания – знания относительно всех задач семейства домена, которые представляются в виде, пригодном для обеспечения решения любой задачи, относящейся к данному семейству.

Сертификация программного продукта – процесс для установления соответствия программной продукции (процесса или услуг) конкретному стандарту или техническим условиям со специальным знаком или свидетельством.

Семейство прикладных систем – множество прикладных систем с общими функциональными свойствами и управлением.

Связь (Relationship) – поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области.

Спецификация – описание алгоритма, правил, ограничений действий объектов с учетом стандартов, критериев качества и др.

Спиральная модель ЖЦ – модель процессов в жизненном цикле разработки системы, позволяющей возвращаться к любому предыдущему процессу с целью переработки элементов сделанного продукта.

Событие – явление, которое провоцирует смену определенного состояния и переход к другому состоянию в системе.
Состояние (домена, системы, объекту и тому подобных) – фиксация определенных свойств на определенный момент или интервал времени.
Статическое тестирование – анализ и рассмотрение спецификаций компонентов на правильность представления без их выполнения на компьютере.
Стереотип – указатель категории элемента моделирования UML.
Сопровождение – работы по внесению изменений в программную систему после того, как она передана пользователю для эксплуатации.
Структура системы – множество элементов и отношений между ними.
Субъект (экземпляр) – кто–то или что–то, вне системы, что взаимодействует с системой.

Сущность (Entity) – реальный либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области, информация о котором подлежит хранению.

Сценарий – конкретная последовательность действий, которая иллюстрирует поведение и выполнение экземпляра прецедента.

Тест – некоторая программа, предназначенная для проверки правильности ее работы и выявления в ней ошибочных ситуаций.
Тестовые данные – данные, которые готовятся на основе документов программы или спецификаций для проверки работы программной системы.
Тестирование – способ семантической отладки (проверки) программы, который состоит в выполнении последовательности различных контрольных наборов тестов и сверка с известным результатом.
Требование – соглашение или договор между заказчиком и исполнителем системы относительно ее работы.
Унаследованная система – существующая действующая система, созданная за любыми и методами и технологиями для поддержки некоторых процессов бизнеса.
Управление качеством – комплекс способов и системной деятельности по планированию, управлению и оценке качества программного обеспечения.
Упрятывание информации – принятие решения о том, что следует сообщить всем о программе, а что оставить при себе – не показывать.
Фасад – механизм доступа к тем свойствам системы компонентов, которые необходимы при реинженерии с точка зрения пользователя системы.
Функция – содержание действий, выполнение которых возлагается на элемент системы при заданных требованиях, условиях и ограничениях.
Функциональные требования – требования, которые определяют цели и функции системы и принципы их выполнения на компьютере.
Функциональная полнота – атрибут, показывающий степень достаточности основных функций для решения специальных задач соответственно назначению программного обеспечения.

Функциональная структура – структура, элементами которой являются функции, реализуемые подразделениями предприятия, а отношениями – связи, обеспечивающие передачу предметов труда.
Характеристики качества – функциональность (functionality), надежность (realibility), удобство (usability), эффективность (efficiency), сопровождаемость (maitainnability), переносимость (portability) и тому подобное.
Черного ящика метод – тестирование реализованных функций путем проверки соответствия реального поведения функций с ожидаемым поведением исходя из спецификаций требований.

Экземпляризация – зависимость между параметризованным абстрактным классом–шаблоном (template) и реальным классом через определение параметров шаблона.
Эксплуатация – действия по выполнению готовой программной системы.
UML (Unified Modeling Language) – диаграммный способ (язык) для спецификации, визуализации, конструирования и документирования продуктов на процессах ЖЦ.

ПРИЛОЖЕНИЕ 2

Характеристика стандартов разработки автоматизированных систем (АС)

2.1 Характеристика стандарта ГОСТ 34.601–90 для разработки АС

Данный стандарт определяет стадии и этапы разработки автоматизированных систем (АС). В зависимости от конкретных условиях стадии и этапы могут объединяться друг с другом. В стандарте определено восемь этапов разработки АС, на каждом из которых формируются документы, описываемые ниже.

1 этап. Формирование требований к автоматизированной системе (АС). Проводится обследование объекта и обоснование необходимости создания АС. Формулируются требования пользователя к АС, оформляются отчет о выполненной работе. Выясняется документооборот, формы начальных и исходящих документов, методики расчета отдельных показателей. Отчет об обследовании создается в произвольной форме, является основой разработки технического проекта АС, в приложениях к отчету приводятся формы документов и методики расчета экономических показателей АС.
Сформулированные требования к системе – заявка на разработку технического задания.

2 этап. Разработка концепции АС. Определяются путей и оценки возможностей реализации требований пользователя, а также методы, которые будут положены в основу расчетов и подходы к решению конкретных задач системы. Заканчивается этап созданием и утверждением отчета о научно–исследовательской работе, в котором дается оценка необходимых для реализации ресурсов, вариантов разработки и оценки качества АС.

3 этап. Разработка технического задания (ТЗ). ТЗ – это основной документ, который определяет требования и порядок создания (развития или модернизации) АС. На основании ТЗ ведется разработка АС, ее прием во время ввода в действие. Дополнительно могут быть разработаны ТЗ на отдельные части АС.

4 этап. Разработка эскизного проекта. На основе проектных решений ко всей системе или ее частям определяется перечень задач, которые будут выполняться в системе, концепция информационной базы, функции и параметры основных программных средств. Для каждой задачи приводятся согласованные с заказчиком формы первичных и исходящих документов, структура информационных массивов или их перечень, основные алгоритмы обработки информации.

5 этап. Разработка технического проекта (ТП). В ТП дается описание разработанных проектных решений для системы и ее частей, документации на АС и комплектации АС или технических требований на нее, задач проектирования смежных частей проекта для проектные решения определяют организационную структуру, функции персонала в АС, структуру технических средств, языки программирования, СУБД, общие характеристики ПО, система классификации и кодирование, а также варианты информационной базы.

6 этап. Разработка рабочей документации. На этом этапе создаются проектные документы, которые определяются государственными стандартами и включают: постановки задач, алгоритмы их решения, организация информационного, технического и программного обеспечения. Эти проектные документы могут оформляться как отдельные документы, а могут входить в технический проект как

отдельные разделы. К документам рабочего проекта относятся общее описание системы, описание технологического процесса обработки информации, инструкции по выполнению отдельных операций технологического процесса, руководство пользователя, описание программ и т.п.

7 этап. Введение АС в эксплуатацию. При создании рабочего проекта проводится разработка и отладка программ или адаптация, готовых программ, которые разрабатывалось для других объектов, их описание или паспорт. На этапе ввода в эксплуатацию выполняется: подготовка объекта к вводу в эксплуатацию, комплектация АС, установка технических и программных средств, выполнение монтажных работ и проведения приемочных испытаний. Готовится приказ об изменениях в структуре объекта, документообороте, а также о распределении обязанностей персонала при переходе на новую технологию обработки информации. Параллельно с подготовкой персонала ведутся работы по установке технических и программных средств, а также разрабатываются средства охраны и защиты АС.

Предварительные испытания системы выполняет разработчик на основе контрольного примера или реальных данных. По результатам опытной эксплуатации программного обеспечения могут вноситься изменения, которые могут повлечь за собой доработку технического проекта системы.

После завершения опытной эксплуатации проводятся приемочные испытания соответственно ТЗ специально созданной комиссией. В результате создается акт введения системы в эксплуатацию.

8 этап. Сопровождение АС. Во время сопровождения устраняться недостатки, которые обнаружены во время эксплуатации и создается акт о выполненных работах. По мере внесения изменений в рабочую документацию могут вноситься изменения и в технический проект и, как правило, самими разработчиками.

АС может создаваться на основе готовых типовых программных средств, которые ориентированы на предметную область этой АС. Программные средства могут продаваться разработчиком или его представителем. В этом случае работы для заказчика выполняются в одну стадию — введение в эксплуатацию АС. Проводится экспертиза, принимается решение о закупке необходимых программных средств. Кроме того, готовится приказ об изменении технологии работы на отдельных участках проекта АС и определяются ответственные за внедрение новой технологии. Разработчик передает заказчику системы рабочую документацию на АС или на ее отдельные части. Все перечисленные работы создаются по договору между заказчиком и разработчиком.

2.2. Стандарт разработки документации на АС – ГОСТ 34.201–89

Стандарт 34.201–89 («Информационная технология. Виды, комплектность и типы документов при создании АС») определяет набор разных видов документов и их комплектность для разрабатываемой АС. К документам относятся: отчеты об обследовании предметной области, результаты научно–исследовательской работы, техническое задание, эскизный проект, технический проект и рабочий проект.

Отчеты об обследовании и научно–исследовательской работе, а также эскизный проект АС создаются в произвольной форме, их структура и содержание согласуются с заказчиком и разработчиком системы. Содержание и структура технического задания, технического и рабочего проектов определяют соответствующие государственные стандарты, например, ГОСТ 34.601–90.

Техническое задание для АС – это основной документ, который определяет требования и порядок его создания или модернизации и может содержать такие разделы:
1. Общие сведения. ,
2. Назначение и цель создания системы.
3. Характеристика объектов автоматизации.
4. Требования к системе.
5. Состав и содержание работ по созданию системы.
6. Порядок контроля и приемки системы.
7. Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу в действие.
8. Требования к документации.
9. Источники разработки.

В техническое задание разрешается вносить некоторые новые разделы или их объединять и детализировать отдельно.

1. Общие сведения. Раздел знакомит со структурой организации заказчика и разработчика, определяет источники финансирования разработки, сроки начала и окончания работ, порядок оформления результатов проектных работ.

2. Назначение и цель создания системы. Раздел содержит описание целей создаваемой АС, ее назначение и возможности ее автоматизации с применением средств вычислительной техники. Дается обоснование необходимости создания АС и решения о выделении необходимого финансирования.

3. Характеристика объекта автоматизации. Раздел содержит важнейшие сведения об объекте (или ссылка на документы, где такие сведения можно найти). Например, информация о наличии вычислительной техники, размещение подразделов, основные их функции и т.п.

4. Требования к системе. В разделе приводятся требования к структуре АС, численности и квалификации персонала, режимам работы системы. Среди требований могут быть требования к техническому обслуживанию системы, к сохранению и защите информации от несанкционированного доступа и др., а также требования к системе в целом, к функциям системы и к отдельным видам обеспечения.

5. Состав и содержание работ по созданию системы. В разделе указывается перечень этапов создание системы, сроки начала и окончания каждого этапа и исполнители работ. В требованиях к составу и содержанию работ перечисляются мероприятия, которые предшествуют внедрению системы, а именно:
– приведение информации к виду, пригодному для обработки на ЭВМ;
– создание необходимых для функционирования информационной системы подразделов;
– срок и порядок комплектования и обучения персонала.

6. Порядок контроля и приемки системы. Описывается правила контроля и приемки созданной АС.

7. Требования к документации системы. Раздел содержит согласованный с заказчиком перечень документов, которые должны быть разработаны и включают систему учетно–статистической, учетной, финансовой и другой документации. Эта документация следующие виды информации:

Информация, которая используется, т.е. приводится перечень массивов информации, которые формируют входные сообщения и сохраняются для реализации данного алгоритма. Для каждого массива приводится его название, идентификатор и возможное количество записей.

Результативная информация – описание назначения результатов, перечень массивов информации, которые участвуют в формировании выдаваемых сообщений и сохранении ее для решения других задач.

Математическое описание основных показателей задачи, описание процесса и объектов, перечень предварительных оценок разработанной модели при разных условиях работы системы.

Алгоритм решения подается в виде схемы в соответствии с требованиями ГОСТ 19.701–90, дополненное текстом в соответствии с ГОСТ 24.301–80 «Система технической документации на АСУ. Общие требования к выполнению текстовых документов».

Информационное обеспечение содержит описание: характеристик, организации сбора и передачи информации на обработку, системы классификации и кодирования; структуры документов и информационных массивов.

Организационное обеспечение содержит схему технологического процесса автоматизированного сбора, обработки информации, передачи данных с перечнем соответствующей документации.

Программное обеспечения включает его характеристику, схема взаимодействия программ и схемы самих программ.

В состав документов рабочего проекта входят такие документы:
– описание программ решения задачи;
– инструкции по технологическому процессу или руководству пользователя;
– классификаторы технико–экономической информации.

9. Источники разработки. В разделе перечисляются документы и информационные материалы, которые использовались во время разработки ТЗ, а также те, которые потребуются во время создания информационной подсистемы.

ПРИЛОЖЕНИЕ 3

Жизненный цикл компонентной разработки ПС

Существует много различных методологий компонентной разработки ПС, которые отличаются методами интеграции, способами описания компонентов, языками определения интерфейсов, способами построения интегрированной среды и др. К основным этапам компонентной разработки ПС относятся:

1) разработка требований (Requirements) к ПС согласно с компонентной методологией;
2) анализ поведения (Behavioral Analysis) для ПС для определения сервисных аспектов программы и соответствующие процессы обработки данных;
3) спецификация интерфейсов и взаимодействия компонентов (Interface and Interaction Specification);
4) интеграция компонентов в единую среду для ПС на основе новых компонентов и компонентов повторного применения (Application Assembly and Component Reuse);
5) развертывание (System Deployment) ПС у пользователя;
6) поддержка и сопровождение (System Support and Maintenance) ПС.

Под интегрированной средой понимается результат завершения работы на этапе интеграции, т.е. макета ПС, которая будет функционировать у пользователя. Программная система – это результат работы после этапа развертывания, характеризуется привязкой к компьютерной и сетевой среде пользователя. Кроме этого, компоненты системы могут приобретаться пользователем в отдельности и выполняться самостоятельно.

3.1. Этап разработки требований

Разработка требований – это формирование и описание функциональных, технологических, организационных и нефункциональных свойств ПС. Суть этапа состоит в определении бизнесов–процессов, выполнение которых должна обеспечить ПС. Бизнесы–процессы связаны с профессиональной деятельностью конечного пользователя и их выполнение поддерживается функционированием ПС. Согласно объектно–ориентированного подхода на этом этапе выполняются следующие процессы:

– определение бизнесов–процессов;
– формирование прецедентов;
– спецификация требований к бизнесам–процессам, условий и особенностей их выполнения.

Определение бизнесов–процессов. Это наименее формализованный процесс на этапе разработки требований. Начальное описание делается заказчиком ПС с использованием обычного языка представления понятий относительно бизнесов–процессов и ПрО. На практике существуют несколько итераций в формировании этого описания, чтобы заказчик и разработчик пришли к соглашению и поняли друг друга.

Метод декомпозиции предусматривает постепенную и планомерную детализацию функционального, пользовательского и технологического аспектов и т.п. Детализация проводится несколькими итерациями и может закончиться на следующем этапе процесса.

Формирование прецедентов. Прецедент описывает последовательность событий в профессиональной деятельности пользователя, где применяется ПС. Прецеденты описывают варианты использования системы. Совокупность всех прецедентов определяет общую картину применения ПС для решения задач определенной ПрО. Поскольку они формируются различными категориями пользователей, могут существовать определенные противоречия в их описаниях. Для их предотвращения при анализе таких прецедентов, им даются определенные приоритеты, согласно которых они ранжируются и структурируются. Важным моментом в формировании прецедентов – согласование с описанием бизнесов–процессов.

Спецификация требований к бизнесам–процессам. Спецификация требований не являются конечными, они выполняться на стадии анализа поведения ПС. Далее проводится переход от общих системных требований к требованиям относительно отдельных компонентов. Детализация системных требований к уровню компонентов является основой их спецификации.

3.2. Этап анализа поведения ПС

Основная задача этого этапа – определить, что система непосредственно будет делать. Если предыдущий этап описывает бизнесы–процессы и специфицирует разные их аспекты, то на этом этапе анализируются детали этих процессов, определяются подходы и методы их поддержки в системе. Данный этап включает такие процессы:

– моделирование предметной области;
– построение объектной системы;
– адаптация объектной системы.

Моделирование предметной области определяется степенью формализации и уровнем детализации в предоставлении знаний относительно предметной области. Главные задачи этого процесса являются типичными и состоят:
– в определении базовых понятий и терминов предметной области, а также реальных объектов, которые им отвечают;
– в определении отношений и связей между этими понятиями соответственно отношениям и связях между реальными объектами, которые важные для функциональности будущей ПС;
– в построении моделей, которые отображают различные аспекты предметной области: структурные, поведенческие, последовательности действий, потоки данных и пр.

Результаты моделирования могут уточняться и детализироваться на иных этапах с целью наиболее полного описания предметной области. В частности, важным моментом является согласование терминов и понятий по результатам моделирования и терминологии описания бизнесов–процессов.

Построение объектной системы состоит в определении структурных свойств для совокупности объектов, ассоциации между ними, описание их атрибуты и соответствующих значений. Далее определяются характеристики поведения для объектной системы – состояние системы, переходы между состояниями, граф переходов, эволюция объектов и системы в целом.

Адаптация объектной системы. Этот процесс не типичный, его сущность состоит в оптимизации объектной системы возможностям покрытия общих функций за счет функций существующих программных компонентов. То есть, система

трансформируется таким образом, чтобы объекты, которые присутствуют в ней будут со своими сервисными возможностями будут приближены к функциональным возможностям компонентов.

3.3. Этап спецификации интерфейсов и взаимодействия компонентов

Этот этап есть ключевым моментом, когда специфика компонентного подхода начинает играть главную роль в создании ПС. Предыдущие этапы повторяют соответствующие этапы объектно–ориентированного подхода. Основу компонентного подхода составляют иные аспекты. Для объектов существуют много реализованных и готовых к применению компонентов и их необходимо предоставить так, чтобы можно было задействовать механизмы их поиска, выборки и интеграции в единую среду. Эта цель и определяет главную задачу этапа спецификации интерфейсов и взаимодействия компонентов, что соответствует следующим процессам:

– распределение ролей компонентов;
– проектирование и спецификация отдельных интерфейсов;
– описание взаимодействий компонентов.

Распределение ролей компонентов. Интерфейсы специфицируются исходя из ролей и физических реализаций соответствующих объектов системы. Распределение ролей в определяет поиск и выбор компонентов, которые имеют соответствующие сервисные возможности и необходимые функции.

Проектирование и спецификация интерфейсов. Проектирование интерфейсов происходит соответственно ролям компонентов. Важно придерживаться концепции оптимальности в проектировании – интерфейсов для компонента не должно быть много, но в тот же время не нужно проектировать мало, но большие по размеру интерфейсы. Каждый из типов интерфейсов – клиентский или сервисный – проектируется в отдельности, идентифицируется и определяется состав поддерживаемых ими операций. Описание отдельных интерфейсов проводится в языке IDL для модели CORBA.

Каждый интерфейс составляется из определенного количества операций с типами параметров и результатов, представленных в терминах пред – и постусловий и определения возможных нестандартных результатов их выполнения с помощью специального объекта, который содержит данные о нестандартных ситуациях.

Описание взаимодействия компонентов выполняется в контексте последовательности действий (workflow), поддерживающих определенные бизнес–процессы. Если результат проектирования интерфейсов определяет пары взаимодействующих компонентов, то результатом этого процесса является совокупность последовательностей операций всех компонентов для достижения целей выполнения бизнесов–процессов.

3.4. Этап интеграции

Главными задачами этого этапа является определение, поиск и выбор всех необходимых компонентов, их адаптация, определение плана компонентной конфигурации, создание и тестирование интегрированной среды для ПС. Поиск и выбор компонентов объединяются в одну задачу, что называется квалификацией

(Component Qualification), а интеграция компонентов определяется как композиция, которая разделяется на несколько видов в зависимости от комбинаций компонентов. Входными данными процесса являются описания:

– интерфейсов компонентов на языке описания интерфейсов;
– взаимодействие компонентов описывается соответственно последовательностями операций для выполнения функций в ПС;
– дополнительные условия и данные для интеграции и управления компонентами.

На этапе интеграции соответственно выполняются следующие процессы:
– поиск компонентов соответственно описанию интерфейсов;
– выбор совокупности компонентов, которые обеспечивают необходимую функциональность;
– адаптация существующих компонентов к требованиям построения интегрированной среды;
– создание новых компонентов, для которых результаты поиска, выбора и адаптации не дали требуемых результатов;
– инсталляция компонентов для потребностей интеграции;
– определение полной совокупности правил и условий интеграции;
– непосредственное построение проекта;
– тестирование интегрированной среды.

Поиск компонентов. Этот процесс предусматривает существование информационных хранилищ с описаниями компонентов. Для реализации больше качественного и оптимального поиска целесообразно иметь систему классификации программных компонентов.

Для предоставления и поиска информации могут быть примененные поисковые машины сети Интернет, как, например, AltaVista,. проект Agora, нацеленный на разработку поисковой машины, для которой критериями поиска есть компонентные модели и их свойства.

Выбор компонентов. Результаты поиска могут быть неопределенными, т.е. могут существовать несколько компонентов для определенного интерфейса или некоторый компонент почти не подходит. Для таких случаев необходимо выполнить процедуры оценки и выборки, например, основанные на системах показателей качества.

Адаптация компонентов. Для расширения возможностей компонентов с одновременным сохранением их основных функциональных и технологических характеристик и обеспечения полной реализации интерфейсов, условий взаимодействия и особенностей интегрированной среды проводится адаптация. Основой механизма реализации такой трансформации является свойство к усовершенствованию и расширении своей функциональности и иных возможностей. К корректным механизмам относятся:

– расширение существующих интерфейсов компонентов с целью соответствия интерфейсам в интегрированной среде;
– обеспечение дополнительных реализаций компонентов с сохранением своих интерфейсов;
– создание контейнеров, которые вмещают иные компоненты с одновременным предоставлением их как самостоятельных компонентов интегрированной среды с необходимой функциональностью и интерфейсами.

Создание компонентов. Этот процесс выполняется, когда предыдущие процессы поиска, выбора, адаптации для определенных компонентов дают отрицательные результаты. В этом случае необходимо создавать новые компоненты с заранее определенными интерфейсами и функциональностью. Для правильного построения новых элементов используется модель Enterprise java beans (EJB), в состав которой входят:

– сервер EJB, как прикладной сервер, в среде которого загружаются и выполняются один или больше контейнеров EJB;
– контейнер EJB, который отвечает за создание среды и условия функционирования соответствующего компонента;
– специальной интерфейс, который поддерживает выполнение условий жизненного цикла для экземпляров компонента, в частности их создание;
– специальные интерфейсы, которые обеспечивают дополнительные сервисы функционирования компонентов в условиях распределенной обработки для обеспечения целостности выполнения функций, для которых задействованы операции в нескольких компонентах.

Инсталляция компонентов для потребностей интеграции. Этот процесс не отличается от инсталляции отдельных компонентов этапа развертывания. Единое отличие состоит в том, что определенные компоненты специально созданы самым разработчиком ПС для ее потребностей, а, следовательно, условия и процедуры инсталляции он может оптимизировать.

Определение правил и условий интеграции. Правила и условия интеграции зависят от компонентной модели, практическое применение имеют три:
– COM/DCOM/COM+, реализованная в операционных системах семейства WINDOWS;
– система CORBA [11], которая поддерживается на различных платформах;
–Javabeans и enterprise javabeans [13] на базе JAVA–технологий, которые функционируют на платформах, для которых реализованы виртуальные JAVA–машины.

Построение проекта. Главная цель процесса – выполнить все действия относительно подготовки интегрированной среды к функционированию и создание плана конфигурации компонентов ПС.

Тестирование интегрированной среды. При тестировании решаются две задачи:
– проверка функциональности ПС;
– проверка возможности совместимой работы компонентов системы.

Первая задача традиционная, для создания любой программы или ПС проводится проверка корректности функционального назначения и организации взаимодействия с пользователем.

Вторая задача – специфична для компонентного подхода, она имеет факторы, которые влияют на проверку совместной работы компонентов:

1. Инициирование работы интегрированной среды для ПС.

2. Определение инфраструктуры для создания возможности динамического взаимодействия компонентов во время выполнения и распределения клиент–серверных запросов в системе.

3. Для каждой пары компонентов, которые взаимодействуют, один с них является клиентом, а второй – сервером. Серверы функционируют в режиме ожидания, пока клиенты к ним не обратятся. Механизм поддержки режима функционирования на уровне операционной среды является механизмом сервисов.

4. Схема взаимодействия компонентов в ПС является ориентированным графом. Приведенные факторы и определяют порядок и содержание второй задачи для проведения тестирования:

– проверка процедуры старта ПС;
– проверка функционирования и создания инфраструктуры для поддержки компонентной модели;
– проверка корректности регистрацииї сервисов и возможности взаимодействия с ними с различных компьютеров сети;
– проверка корректности последовательностей инициирования сервисов.

3.5. Этап развертывания компонентов системы

На выполнение задачи этого этапа влияет ориентация ПС на конечного пользователя. В случае, когда ПС создается для конкретного заказчика, что часто он и является пользователем, некоторые задачи развертывания решаются на предыдущих этапах. К ним относятся:

– проектирование и интеграция ПС с ориентацией на конкретные компьютерные условия заказчика, расположение, наличие компьютерных сетей, коммуникаций и др.;
– инсталляция отдельных компонентов на определенных компьютерах для целей интеграции;
– создание компонентной конфигурации на этапе интеграции с ориентацией на ее дальнейшее применение на этапе сопровождения.

Пользователь компонентной системы должен пройти все процессы этапа развертывания на своей собственной базе соответственно свой конфигурации компьютерных и общесистемных средств.

В состав этапа развертывания входят:

– оптимизация плана компонентной конфигурации соответственно имеющей компьютерной базе пользователя, ее свойств и топологии;
– развертывание отдельных программных компонентов;
– создание целевой компонентной конфигурации.

Оптимизации плана компонентной конфигурации. План компонентной конфигурации является результатом тапа интеграции, он определяет абстрактную модель предоставления интегрированной среды ПС. Все взаимосвязи и взаимодействия компонентов представлены на логическом уровне, описываются сами факты связей и взаимодействий без учета реального расположения компонентов.

Главная задача процесса – определения реальной конфигурации компонентов и оптимизационная задача с данными:

– компьютерные мощности и способы коммуникации у пользователя;
– требования к ресурсам со стороны отдельных компонентов;
– план компонентной конфигурации.

В качества предельных условий выступает необходимость выполнения функциональных, технологических и иных требований к программной системе.

Развертывание программных компонентов путем инсталляции компонентов на конкретных компьютерах, расположения которых определено в результате предыдущего процесса. Процесс инсталляции для каждого отдельного компонента может иметь свои собственные отличия, которые зависят от метода и способов инсталляции.

Создание целевой компонентной конфигурации. Результаты предыдущих процессов являются основой для создания целевой конфигурации, в задачи которой входит:

– расположение отдельных компонентов (сетевые адреса, имена каталогов и файлов);
– характеристики компонентов (например, размеры, необходимые ресурсы и условия функционирования и др.);
– описание взаимосвязей компонентов (например, описание последовательностей инициирования и окончания работы).

Этот процесс завершает создание целевой конфигурации. Предыдущая информация дополняется:

– организационными требованиями (приведение в порядок категорий пользователей, предоставление им определенных прав доступа к отдельным компонентам, к сервисам и др.);
– технологическим регламентом (например, по времени и периодичности инициирования компонентов, выполнению определенных задач и др.);
– описанием процесса инициирования ПС в целом (например, может быть специально сформированный пакет для запуска и выполнения системы).

Вся приведенная информация включает описание целевой конфигурации и применяется для поддержки функционирования ПС на этапе сопровождения. При наличии системы управления конфигурацией приведенные данные описываются соответствующими способами.

3.6. Этап сопровождения

В сравнении с традиционными методологиями разработки ПС этап сопровождения в компонентной методологии характеризуется следующими особенностями.

1. Обслуживающий персонал ПС не имеет доступа к коду компонентов. В связи с этим при необходимости изменения ПС традиционные подходы и методы адаптируются к возможностям новых условий функционирования, тестирования, выявления и исправление ошибок, модификации отдельных элементов и др.

2. Вся политика модернизации, усовершенствования, расширения ПС должна строиться на компонентной основе, в которой главными механизмами могут лишь быть:

– замена существующих компонентов новыми компонентами с сохранением интерфейсов и сервисных возможностей;
– расширение функциональных и технологических возможностей отдельных компонентов на основе их свойств и сохранение существующих интерфейсов.

3. Отдельные компоненты, которые применяются в ПС, могут быть созданы посторонними разработчиками и использоваться в данной ПС, как готовые. Соответственно с этим производителями проводится собственная политика относительно поддержки, усовершенствования, развития таких компонентов. При сопровождении ПС такие ситуации необходимо учитывать как в технологическом, так и в организационно–правовом аспекте (например, охрана авторских прав на программное обеспечение).

Эти особенности существенным образом влияют на традиционные задачи этапа сопровождения и процессов, которое их поддерживают. К основным процессам этого процесса относятся:
– модификация компонентной конфигурации;
– адаптация новых компонентов к требованиям и условиях интегрированной среды;
– анализ отказов функционирования, обнаружение дефектов, поиск и исправления ошибок в программной системе;
– тестирование ПС.

Кратко остановимся на общей характеристике этих процессов.

Модификация компонентной конфигурации. Этот процесс отвечает за следующее:

– добавление и исключение определенных компонентов;
– замещение существующих компонентов новыми как с тождественной функциональностью и интерфейсами, так и с расширенными характеристиками.

Необходимыми условиями для этого процесса является возможность манипуляций с компонентами как отдельными объектами с сохранением свойств и характеристик разных частей ПС. Это достигается благодаря применению систем управления конфигурациями, с помощью которых отслеживаются и выполняются все изменения в конфигурации системы.

Адаптация новых компонентов к требованиям и условиям среды. Данный процесс, по сути и по содержанию, почти не отличается от соответствующего процесса этапа интеграции. Имеющиеся отличия носят непринципиальный характер. К ним, в частности, можно отнести то, что в случае неудовлетворительной адаптации, всегда имеется возможность вернуться к существующему компоненту и программная система остается без перемен.

Кроме этого, сам процесс адаптации может выполняться обслуживающим персоналом пользователя (при наличии специалистов с необходимой квалификацией), а не разработчиком ПС.

Анализ отказов функционирования, поиск и исправления ошибок в ПС. Если при определенных условиях в программной системе появляются отказы функционирования или ошибки программирования, то главной задачей их локализации является нахождение компонентов, которые ненормально работают. В большинстве случаев

обслуживающий персонал не в состоянии исправить код компонента, к которому нет доступа.

Для исправления ошибок используются следующие механизмы:
– обращение к разработчику компонента и, если он был специально созданный для этой системы, дождаться от него исправления ошибки, а потом заменить соответствующий компонент;
– если компонент является коммерческим продуктом, который создан сторонним производителем, то ему должны сообщить соответствующие разработчики и дождаться официальной версии компонента, в котором исправлена ошибка, и есть возможность заменить этот компонент;
– не дожидаясь исправления ошибки другими разработчиками, провести замену локализованного ошибочного компонента другим правильным с соответствующей функциональностью и интерфейсами.

На период исправления ошибки последовательности взаимодействий компонентов, которые были определены на предыдущих этапах, выключаются из функционирования путем внесения адекватных изменений в компонентную конфигурацию.

Тестирование ПС. Тестирование проводится периодически для проверки правильности функционирования системы и в случаях внесения изменений в компонентную конфигурацию или замены отдельных компонентов. Под этим процессом понимается тестирование системы на компонентном уровне, т.е. покомпонентное тестирование.

Для проведения периодического тестирования применяются тесты, которые передаются разработчиком ПС пользователю. Главная цель такого тестирования – подтвердить правильность функционирования системы, оценить ее эффективность, быстродействие и другие технологические характеристики. Порядок и условия процесса тестирования для ПС отображаются в соответствующих документах.

Для тестирования проведенных изменений в компонентную конфигурацию и отдельные компоненты необходимо иметь:
– сами компоненты в готовом к применению виде;
– гарантию относительно достаточно полного тестирования компонентов их разработчиками, иметь информацию о результатах тестирования, в частности перечень еще неисправленных ошибок;
– четко сформулированные условия применения компонентов, как с функциональной точки зрения, так и с технологической (в частности, иметь данные о ресурсах, необходимых для нормальной работы компонентов системы).

При разработке тестов учитывается:
– последовательности взаимодействий компонентов, в состав которых входят те компоненты, которые проходят тестирование;
– информация о самостоятельном тестировании компонентов (используется для уменьшения объема тестирования);
– условия для нормального функционирования компонентов.

После проведения тестирования и анализа результатов, при наличии ошибок разного рода, проводятся соответствующие изменения, как в компонентную конфигурацию, так и в отдельные компоненты системы, в которых обнаружены ошибки.

ПРИЛОЖЕНИЕ 4

Кодекс этики программной инженерии

Своим появлением программная инженерия обязана деятельности мощных профессиональных объединений – The Association for Computer Machinery (ACM) и Institute of Electrical and Electronics Engineers Computer Society (IEEE Computer Society). Общими усилиями этих двух объединений разработан кодекс этики программной инженерии, который фокусирует мораль, правила и нормы поведения профессионалов, их обязательства и ответственность по отношению к обществу и один к другому [1, 2].

Этика инженерной деятельности в программировании отличается от этики прикладных исследований, где исследователи работают в прикладной науке, направляют свои усилия на реализацию возможностей и отвечают определенным требованиям.

Инженерная деятельность в программной инженерии, кроме сказанного, включает технические умения, ответственность перед пользователями, умение управлять и приводить к удачному завершению больших программных проектов. Иначе говоря, инженеры должны хорошо знать, что есть риск сделать реализацию быстро или высококачественно, но с малым риском.

Кодекс состоит из преамбулы и восьми принципов, которых должны придерживаться профессионалы с программной инженерии. В преамбуле профессионал определяет как специалист, который принимает непосредственно участие в деятельности или в обучении анализу, спецификации, проектированию, разработке, сертификации, сопровождению и тестированию программных систем.

Сформулированные принципы декларируют здоровье, безопасность и благосостояние общества как главный фактор, который необходимо принимать во внимание при принятии решений в профессиональной деятельности программной инженерии.

В кодексе задекларировано восемь принципов, которые касаются соответственно:
1) согласование профессиональной деятельности с интересами общества;
2) взаимоотношение между клиентом, работодателем и исполнителем разработки;
3) достижение соответствия качества продукта лучшим профессиональным стандартам;
4) соблюдение честности и независимости при профессиональных оценках;
5) соблюдение этических норм в менеджменте и в сопровождении разработок;
6) поддержка становления профессии в соответствия с кодексом этики;
7) соблюдение этических норм во взаимоотношениях между коллегами;
8) усовершенствование специальности.

Каждый с приведенных принципов имеет детальные пояснения относительно различных спектров его соблюдения.

Литература

1. Jotterbarn D., Miller K., Rogerson S. Software Engineering Code of Ethies is Approved / Communications of the ACM.– v.42. – №1O. – 1999.– p.1O2 –1O7.
2. www. asm. org / serving /se / code.hfm

ПРИЛОЖЕНИЕ 5

Стандарты программной инженерии

Существует определенное количество организаций, профилирующими направлениями деятельности которых есть разработка и сопровождение стандартов. В зависимости от области применения стандарт может иметь статус международного, ведомственного или стандарта предприятия. Главным органом установления международных стандартов является международная организация по стандартизации (The International Standards Organization или сокращенно ISO), которая работает в сотрудничестве с международной электротехнической комиссией (The International Electrotechnical Commission или сокращенно IEC). Все утвержденные ими совместно стандарты имеют идентификатор, который состоит из префикса ISO/IEC, серийного номера стандарта и даты выпуска, например: ISO/IEC 12207: 1995 – 08–11.

Каждый стандарт ISO/IEC имеет также название, которое при ссылках указывается после идентификатор. ISO/IEC имеет десятки технических профильных комитетов, в частности технический комитет "Информационные технологии", одним из технических подкомитетов которого является подкомитет по программной инженерии. Существуют также международные объединения по отдельным проблемным областям, которые выпускают стандарты для соответствующих приложений. Каждое цивилизованное государство имеет свои национальные органы стандартизации.

Большинство национальных комитетов по стандартизации признает стандарты ISO/IEC и входит в ее состав, и проводят для стандартов ISO/IEC процедуру гармонизации, т.е. их приспособление к национальным условиям и особенностям применения (как например, национальные алфавиты, метрические системы, валютные знаки и т.п.).

Главным источником стандартов является профессиональные объединения, в частности для программной инженерии – IEEE Computer Society. На данное время существует свыше 300 стандартов IEEE для программной инженерии, значительная часть которых принимается во внимание широким кругом разработчиков программных систем. Практически большинство стандартов программной инженерии исторически появляются как стандарты IEEE, а со временем, после испытания опытом использования, вносятся как кандидаты в стандарты ISO/IEC.

Процедура утверждения стандартов ISO довольно сложная. Имеется несколько стадий прохождения кандидата в стандарт, для любой из них предусмотрена рассылка предложений на экспертизу всем национальным комитетам, сбор замечаний, их обработка и голосование для создания новой версии.

Эта процедура для некоторых стандартов может длиться годами относительно официальных стандартов или стандартов де–юре. Бюрократизированная процедура приводит к тому, что технически доведенные до кондиции стандарты могут за время утратить свою значимость для индустрии программного обеспечения и соответствие действующему уровню технологии.

Тем временем индустрия создает так называемые "стандарты де–факто", которые фактически находят массовое использование независимо от того, утверждены они компетентными плановыми органами или нет, так как они являются наиболее актуальными в индустрии программных систем.

Термином "стандарт де–факто" обозначаются спецификации на проект стандарта (или внутренний стандарт), которые публикуются некоторым консорциумом (группой

учреждений, которые официально работают для общей цели стандартизации) или фирмой, получившие общее одобрение от других разработчиков, и этот проект стал восприниматься как необходимое современное направление технологии.

Например, проектные решения консорциума Object Management Group (OMG) по управлению транзакциями стали стандартом де–факто и утверждены комитетом ISO, модель UML фирмы Rational Rose определена как стандарт моделирования артефактов программной инженерии, язык JAVA – претендент на стандарт, который активно используется на рынке, и др.

Стандарты де–факто, благодаря своему соответствию актуальным потребностям рынка, распространяются и внедряются довольно быстро, потому что их отработка и апробация проходит внутри групп, которые их создают, а публикация этих стандартов обозначает, как "зрелые" решения.

Учитывая вышесказанное, влиятельные международные организации и, в первую очередь ISO, которые являются разработчиками стандартов де–юре, признали, что они не являются монопольными и компетентными источниками всех стандартов и поэтому ввели процесс преобразования стандартов де–факто на стандарты де–юре, которые в этом случае получили название общедоступной спецификации (Public–Available Specifications – сокращенно PAS).

Объекты стандартизации. Согласно эталонной модели программной инженерии, в ней определены такие главные элементы :
– процессы разработки ПО;
– продукты разработки;
– ресурсы, которые используют процессы для создания программного продукта.

Важным элементом моделирования проблем предметной области является клиент (заказчик), который заказывает программную систему. Требования заказчика определяют состав и качество указанных элементов. Объектами стандартизации любого стандарта в программной инженерии являются аспекты указанных выше элементов или их соединений.

Больше всего стандартов существует для разных видов процессов. Так в стандарте ISO/SEC 12207 базовых процессов – 42, а всего процессов в нем более 200. Надо сказать при этом, что усовершенствование процессов, как правило, ведет к усовершенствованию создаваемых продуктов и эффективному использованию ресурсов.

Определен детальный перечень процессов и действий, которые составляют процессы, для всех этапов жизненного цикла разработки и большинства аспектов рассмотрения указанных этапов. Наибольшего успеха и широкого использования приобрел стандарт [2] для процессов жизненного цикла программного обеспечения. Этот стандарт стал определенным каркасом для рассмотрения всех проблем программной инженерии.

Первым измерением классификации является отношение стандарта к продукту, процессу, ресурсу или взаимодействию с заказчиком. В [3] предложено второе измерение классификации стандартов по уровням обобщения регламентаций, которые подаются в стандарте.

Отдельные группы разработчиков стандарта в программной инженерии создают:
– терминологию,
– общие рекомендации,
– принципы действий,
– стандартизированные элементы,
– рекомендации для прикладных применений,
– рекомендации относительно инструментов и методов разработки (например, классификация аномалий выполнения),
– планы управления качеством,
– планы валидации и верификации.

В соответствии с предложенной классификацией ниже приведены наиболее значимые и существующие стандарты программной инженерии.

ISO/IEC 12207:1996 Information Technology – Software life–cycle processes;
12207/FPDAM 1.2 – Software Engineering – Life Cycle Processes // ISO/IEC JTC1/SC7 N2413, Software & System Engineering Secretariat, Canada. – 2001. – 62p.

DTR 15271 Information Technology – Guide for ISO/IEC 12207 (Software Life Cycle Processes);

IEEE/IEA Std. 12207.1:1997 Software Life Cycle processes – Life Cycle data;

ISO/IEC JTC1/SC7 N2172a (Draft ISO/IEC TR 16326:1999). Software Engineering – Guide for the Application of ISO/IEC 12207 to Project Management;

ISO/IEC TR 15504 Information Technology – Software Process Assessment Part 1 – Part 9;

IEEE Std. 1058:1998 IEEE Standard for Software Project Management Plans

IEEE Std. 1044:1993 IEEE Standard Classification for Software Anomalies

IEEE Std. 1044–1:1995 IEEE Guide to Classification for Software Anomalies

ISO/IEC 15026:1998 Information Technology – System and Software Integrity Levels;

ISO/IEC JTC1/SC7 N2207:1999 IEC 60300: Dependability management – Part 3–13: Application Guide – Project risk management;

ISO/IEC JTC1/SC7 N2198:1999 2nd Working Draft On Risk Management Terminology и др.