Архитектурно-управляемая модернизация существующего программного обеспечения

Н. Н. Мансуров

Аннотация. Отдел CASE-технологий института системного программирования (ИСП) занимается разработкой CASE-инструментов следующего поколения, основанных на формальных методах. Разрабатываемые инструменты и методы ориентированы на практическое усовершенствование процесса создания и поддержки программного обеспечения (ПО) в индустрии разработки систем реального времени и встроенных систем. В частности мы определяем *ускоренную методологию* спецификации, проектирования, тестирования и обратной инженерии ПО. Методология основана на широком применении формальных методов и формальных языков описания программ *на самых ранних стадиях* разработки, а также на автоматическом *извлечении* формальных моделей *при помощи обратной инженерии* из существующего кода. Основные направления в исследовательской работе отдела – восстановление и перестроение (рефакторинг) архитектуры ПО.
Эта статья посвящена созданию методов так называемой архитектурно-управляемой модернизации существующего кода в Object Management Group (OMG), совместно с нашим партнером – компанией klocwork.

1. Введение

Отдел CASE-технологий (Computer Aided Software Engineering tools) ИСП занимается исследованием и разработкой в области современной инструментальной поддержки разработки ПО. В отделе разрабатываются инструменты для работы с моделями следующего поколения, которые ускоряют разработку ПО за счет использования методов формального моделирования, автоматических трансформаций моделей, включая передовые технологии генерации кода, управляемую архитектуру ПО, валидацию и верификацию, особенно на ранних стадиях процесса разработки. Также осуществляется разработка методов и инструментов в следующих основных направлениях:

- Инструментальная поддержка управляемой архитектуры ПО. Это направление охватывает методы архитектурно-ориентированной разработки ПО, инструментальную поддержку разработки групп программных продуктов (Software Product Lines), автоматическое извлечение формальных моделей из существующего кода, а также

  инструменты для ускоренной поддержки и модуляризации существующих программных систем (архитектурно-ориентированные «раскопки» и рефакторинг)
- Исполняемые сценарии. Это направление охватывает методы системной инженерии, анализ требований к ПО, а также разработку ПО, основанную на сценариях использования.

Основные цели исследований отдела:

- разработать **архитектурно-ориентированный ускоренный подход**, включающий в себя набор методов спецификации, проектирования, тестирования и обратной инженерии ПО, основанных в значительной степени на использовании формальных методов и формальных языков описания ПО на самых ранних стадиях процесса разработки, а также автоматическом извлечении формальных моделей из существующего кода, уделяя особое внимание архитектурным вопросам и «крупномасштабному программированию»
- создать **открытую архитектуру**, включающую в себя языки и инструменты, поддерживающие описанный подход, и позволяющие его, в как можно большей, степени автоматизировать, для того чтобы сделать формальные методы и языки более доступными для разработчиков.

Отдел CASE-технологий тесно сотрудничает с компанией Klocwork, Inc. (http://www.klocwork.com), располагающейся в Оттаве (Канада). Klocwork – член Object Management Group (OMG) (http://www.omg.org), где является сопредседателем группы (Special Interest Group) по архитектурно-управляемой модернизации существующего кода (http://legacy.omg.org). В сотрудничестве с Klocwork исследовательская группа отдела CASE-технологий активно участвует в процессе стандартизации формальных методов разработки ПО в OMG.

Направление исследований отдела определяется несколькими факторами. В современной индустрии разработки программных систем реального времени, а также встроенных систем, существует достаточно много различных успешных организаций и подходов. Но по мере того, как возникают новые требования и технологии, и все больше участников вступает в игру, появляется потребность в усовершенствовании существующих подходов [4]. В частности, время выхода продукта на рынок («time-to-market») становится доминирующим фактором успеха в индустрии разработки ПО. Другие цели более традиционны и включают в себя улучшение качества программного продукта, соотношения цена/производительность и снижение расходов на производство ПО [4]. Общепризнанно, что использование методов формального описания (Formal Description Techniques - FDT), при поддержке CASE-инструментов – это важная составляющая в процессе достижения этих целей.

Установленное, используемое и работающее ПО представляет большую ценность для современной экономики. Большая часть существующих прикладных систем, т.е. всего эксплуатируемого ПО, независимо от платформы, на

которой оно исполняется, языков программирования, на которых оно написано и времени эксплуатации, перешла на стадию поддержки и эволюции [12].

Поддержка существующего ПО включает в себя несколько направлений [9]: (1) корректирующая поддержка (исправление дефектов); (2) адаптационная поддержка (адаптация системы к изменениям исполняемой платформы); (3) превентивная поддержка (поддержание и улучшение внутренней структуры системы для предотвращения «распада» архитектуры); (4) расширение функциональности (добавление новых возможностей). Добавление новой возможности в существующую систему зачастую оказывается более сложной задачей, чем разработка аналогичной по функциональности системы с нуля, из-за ограничений, которые накладывает на дизайнерские решения существующее ПО. Таким образом, существует ряд проблем, специфичных для фазы поддержки, которые не возникают в процессе разработки новых систем:

- необходимо учитывать большее количество информации (унаследованные дизайнерские и программные решения, анализ существующего исходного кода);
- недостаток экспертных знаний из-за того, что ведущие разработчики уходят из проекта;
- необходимость обучения нового персонала;
- географически распределенная среда разработки, например, когда поддержка ПО осуществляется посредством аутсорсинга;
- недостаточная гибкость, сопротивление системы при попытке внесения изменений, вызванное «распадом» архитектуры (обычно сопротивление возрастает со временем).

Эволюция существующего ПО зачастую затруднена, поскольку его стратегическая ценность и способность приспосабливаться ослаблена факторами, напрямую не связанными с его функциональностью. Примерами таких факторов может послужить неэффективность анализа и поддержки системы с экономической точки зрения, затрудненная интероперабельность, или зависимость системы от устаревших технологий или архитектур. Необходимость анализа и эволюции существующего ПО возникает в задачах документирования, развития, модификации, взаимодействия, переноса на другую платформу, перепроектирования и переустановки. Эти задачи в совокупности можно рассматривать как «модернизацию» ПО. Модернизация становится актуальной в тех случаях, когда существующие традиционные практики не удовлетворяют нуждам бизнеса [12].

Проблема поддержки и эволюционного развития существующего ПО достаточно долго игнорировалась ведущими методологами, разработчиками и стандартизационными комитетами. В результате между разработкой новых проектов (green-field projects, т.е. проектов разрабатываемых с нуля) и поддержкой существующего ПО образовался значительный разрыв в подходах и инструментальной поддержке. В частности, поддержка существующего ПО традиционно считалась одним из «потерянных миров» для методов

формального моделирования и CASE-инструментов, основанных на формальных методах. Современные подходы, использующие формальные языки и продвинутую инструментальную поддержку, зачастую плохо масштабируются применительно к задачам поддержки существующего ПО. Предварительное моделирование больших существующих систем обходится слишком дорого, и, поэтому, потенциальные преимущества современных подходов и инструментов не доступны для разработчиков, имеющих дело с модернизацией существующих систем. Инструменты поддержки обычно разрабатываются для узкого класса задач и слабо распространены в индустрии разработки ПО.

С другой стороны задачи, специфичные для поддержки ПО (такие, как устранение дефектов, адаптация и анализ существующих дизайнерских решений) составляют значительно большую часть жизненного цикла программной системы, чем задачи прямой инженерии (такие как анализ исходных требований, проектирование и создание исходного кода системы). Как уже было показано в [15], начиная с 90-х годов, в процесс устранения дефектов и расширения функциональности существующих программных систем вовлекается больше разработчиков, чем в разработку новых проектов «с нуля» (см. Таблицу 1).

Год	Новые проекты	Расширение функциональности	Устранение дефектов	Всего
1990	3 (43%)	3 (43%)	1 (14%)	7
2000	4 (40%)	4.5 (45%)	1.5 (15%)	10
2010	5 (35%)	7 (50%)	2 (14%)	14

Таблица 1. Прогноз количества разработчиков (млн. чел.) и распределение по решаемым задачам.

Таким образом, большую важность имеет разработка ускоренных подходов, применимых в задачах поддержки ПО, и основанных на некоторых методологических принципах. Это является главной целью нашего исследования.

Остальная часть статьи организована следующим образом. Раздел 2 описывает мероприятия, проводимые в рамках OMG, направленные на усовершенствование процесса поддержки и эволюции существующего ПО. В разделе 3 обсуждается своевременность разработки подходов к модернизации программных систем. Раздел 4 описывает первый запрос на внесение предложений (RFP – Request for Proposal) группы по архитекурно-управляемой модернизации OMG, вносящее предложения по разработке KDM (Knowledge Discovered Meta-Model). Разделы 5-8 представляют наши исследования в области архитектурно-ориентированного восстановления знаний о програм-мной системе и трансформации (рефакторинга) существующего ПО. В частности, раздел 5 содержит краткое введение в подход, основанный на интерфейсных моделях и архитектурных «раскопках», применяемый в Klocwork. Раздел 6 описывает архитектурно-ориентированный рефакторинг. Раздел 7 обрисовывает концепцию управляемых архитектур и

описывает соответствующую инструментальную поддержку. Раздел 8 содержит обсуждение того, как архитектурно-ориентированные трансформации могут использоваться для миграции современного ПО в направлении MDA (Model-Driven Architecture – модельно управляемая архитектура). Раздел 9 содержит некоторые выводы и заключение.

2. Почему OMG?

Object Management Group (OMG) - это открытая некоммерческая организация, деятельность которой направлена на разработку и поддержку спецификаций для взаимодействующих промышленных приложений. Список членов OMG включает много организаций из числа наиболее успешных и новаторских компаний компьютерной индустрии, равно как и из числа тех, кто находится на переднем крае использования новых технологий для получения конкурентных преимуществ в своем бизнесе. Все они взяли на себя обязательства активно участвовать в формировании будущего промышленных систем, интернет-систем, систем реального времени и встроенных систем.

OMG разработала несколько очень известных в индустрии и наиболее влиятельных спецификаций, включая CORBA, OMG IDL, IIOP, UML, MOF, XMI, CWM, OMA и Domain Facilities, в таких отраслях, как здравоохранение, промышленное производство, телекоммуникации и многие другие. Model Driven Architecture (MDA) опирается на успех этих технологий, обеспечивая всестороннюю основу взаимодействия для создания взаимосвязанных систем будущего.

Однако, в прошлом, индустрия прямой инженерии практически полностью игнорировала существующие программные ресурсы, в то время как обратная инженерия, в свою очередь, игнорировала новые методы прямой инженерии. Со временем, это породило существенный разрыв в инструментальной и методологической базе, используемой в новых проектах, и проектах, занимающихся поддержкой и эволюцией существующего кода. Прилагаемые усилия направлены на то, чтобы преодолеть барьеры между инструментами прямой и обратной инженерии, и, таким образом, помочь организациям с существующими программными ресурсами использовать преимущества современных технологий обратной инженерии таких, как Unified Modeling Language (UML) [5] и Model-Driven Architecture (MDA) [14].

MDA обеспечивает открытый, независимый от производителей подход к проблеме интероперабельности, основывающийся и усиливающий значимость принятых стандартов моделирования. MDA направлен на отделение бизнес логики и логики приложений от технологий целевой платформы. MDA пропагандирует новый способ разработки ПО, в котором платформо-независимая модель становится долгосрочно поддерживаемым артефактом. Платформо-независимые приложения, построенные с использованием MDA и сопутствующих стандартов, могут быть реализованы автоматически на множестве открытых и проприетарных платформ включая CORBA, J2EE, .NET

и Web Services или на других Web-платформах, с помощью компиляторов моделей. По мере появления платформ и технологий, MDA гарантирует будущую защиту инвестиций в разработку. Таким образом, MDA абстрагируется от промежуточного ПО (middleware) для обеспечения всестороннего, структурированного решения для обеспечения будущей интероперабельности и переносимости приложений.

Платформо-независимые модели MDA могут быть одной из целей модернизации. Для организаций, которые приняли решение о переходе на MDA, модернизация больше не является переносом с одного языка на другой, или с одной платформы на другую. Модернизация в контексте MDA создает модель кода с целью импортировать ее в MDA-ориентированную среду разработки. Стандарты модернизации становятся важной составляющей успешного применения современных технологий прямой инженерии, таких как MDA и UML.

В июне 2003 года OMG анонсировала создание Platform Special Interest Group (PSIG) с целью создания спецификаций и поддержания консенсуса в индустрии по вопросу модернизации существующих приложений (http://legacy.omg.org/) [11].

Главной целью рабочей группы является создание стандартов модельно-управляемой модернизации, определенной как процесс понимания и развития ресурсов существующего ПО, с акцентом на архитектурные изменения. Благодаря созданию стандартов модернизации пользователи могут продлить жизнь своего ПО, впоследствии увеличив эффективность инвестиций (Return on Investment, ROI). Это сделает возможным взаимодействие между различными производителями ПО, а также обеспечит, что однажды модернизированное ПО, будет возможно подвергнуть подобной трансформации в будущем (доказательство функциональности). Стандартизация будет гарантировать, что конечные пользователи делают инвестиции не только в конкретные инструменты, а скорее в координированную стратегию.

Для достижения своих целей группа планирует применить существующие стандарты моделирования OMG, включая MDA, к успешной трансформации и возвращения к жизни существующих приложений. В конечном счете, стандартизация модельно-управляемой модернизации поможет индустрии ПО и частному бизнесу уменьшением рисков инициатив по улучшению своего ПО, уменьшением времени, рисков и стоимости трансформации ПО с одновременным увеличением эффективности инвестиций инструментов разработки ПО.

В ноябре 2003 года, в Лондоне, члены OMG начали работу над двумя новыми стандартами в этой области. В первую очередь Special Interest Group (SIG), работающая над модельно-управляемой модернизацией, опубликовала запрос на внесение предложений (RFP) по стандартизации фундаментальной модели (в технической терминологии, метамодели) для анализа функциональности существующего ПО.

В рамках другого направления работ, группа разработала запрос комментариев (RFC) на спецификацию повторно используемых ресурсов (Reusable Asset

Specification или RAS). RAS определяет стандартный способ сборки повторно используемых программных ресурсов. Это является важной частью процесса разработки ПО, основанной на программных ресурсах (Asset-based Development или ABD), способствуя повторному использованию ресурсов разработки ПО в индустрии.

3. Почему теперь?

Есть несколько факторов, демонстрирующих своевременность консолидации практик модернизации ПО и стандартизации в этой области [12]:

- Современная экономика требует от организаций более эффективного использования доступных ресурсов, включая повторное использование существующих программных ресурсов.
- Неагрессивные проекты интеграции достигли пределов своей способности приносить экономическую выгоду.
- "Закон выживания" главной инфраструктуры и разработчиков инструментов; унификация связующего ПО (middleware) и оборудования порождает интерес к портированию на новые платформы.
- Повсеместное принятие UML и MDA, обеспечивающее методологию, равно как и цель для модернизационных проектов.
- Развитие технологий анализа и повторного использования унаследованного ПО и выполнения трансформаций унаследованного ПО; критическая масса индивидуальных решений разработчиков ПО.
- Разрыв между технологиями и методологиями в индустрии, что требует от успешных разработчиков программных продуктов возвращаться и повторно исследовать существующие решения и практики (эволюция программных систем, вырастающих из первоначальных прототипов и слияний/поглощений).
- Требование к интероперабельности промышленного ПО.

Существующие системы, которые в процессе эволюции приобрели противоречивую или сложную архитектуру, и чья модификация требует трудоемкого ручного кодирования, могут существенно затруднить подвижность организации. Компании, вынужденные противостоять слияниям и поглощениям, учитывать появление новых продуктов, изменение требований рынка, правительственное регулирование и другие бизнес-факторы, нуждаются в гибких инфраструктурах ПО для того, чтобы конкурировать или даже просто выжить. Модернизация может помочь удовлетворить эти бизнес-требования быстрее, за меньшую стоимость, и с лучшим качеством.

Для большого числа организаций необходимость модернизации своих промышленных информационных систем обычно побуждается определенными внешними событиями. Такими, как необходимость обеспечить Web-доступ к существующему приложению, или необходимость отказаться от неподдерживаемой платформы. Для компаний, выпускающих зависящие от технологии

или продукты, в которых программное обеспечение является важной частью, эволюция существующего кода является жизненной частью ежедневных бизнес-требований.

ADM обеспечивает средства, направленные на удовлетворение этих требований, используя систематические и мало-рискованные подходы.

При помощи анализа, категоризации и управления ресурсами существующего ПО, бизнес может продлить полезность функциональности существующего ПО и максимизировать эффективность инвестиций в разработку ПО.

4. RFP по метамодели обнаружения знаний (KDM – Knowledge Discovery Model)

Первая из спецификаций в серии Архитектурно-управляемая модернизация (Architecture-Driven Modernization или ADM) позволит пользователям начать проекты по модернизации их программного обеспечения, обеспечив их уверенность в *интероперабельности* программных инструментов различных поставщиков [13].

RFP по KDM учитывает существование спецификаций OMG, которые связаны с ADM. Предложения по этому RFP должны задействовать эти спецификации или обосновать их неадекватность. Поощряется использование и расширение существующих OMG стандартов сторонами, отвечающими на этот RFP, с тем, чтобы сделать эти стандарты более подходящими для проблемной области, в которой действует ADM.

RFP по KDM запрашивает предложения по метамодели для представления информации о существующих программных ресурсах и их операционной среде (в дальнейшем она именуется «*обнаружением знаний*» - «*knowledge discovery*»). Одной из общих черт различных инструментов, отвечающих на вопросы, которые ставит ADM, является то, что они *анализируют* существующие программные ресурсы (например, исходный код модулей, описание баз данных, сценарии сборки, и т.д.) с целью обнаружения знаний о системе. Каждый из этих инструментов извлекает порцию знаний о существующих ресурсах ПО. Подобные знания, зависящие от конкретного инструмента, могут быть неявными («зашитыми» в самом инструменте), ограниченными некоторым языком программирования, некоторым видом преобразования и/или операционной средой. Все это может стать помехой обеспечения интероперабельности этих инструментов. Метамодель обнаружения знаний обеспечит общую структуру репозитария, которая облегчит обмен данными, хранимых внутри моделей отдельных инструментов и представляющих ресурсы существующего ПО. Метамодель будет описывать физические и логические ресурсы на различных уровнях абстракции. Основной задачей, решаемой метамоделью является продвижение общего формата обмена, который обеспечит интероперабельность между различными инструментами модернизации ПО, службами и соответствующими моделями.

4.1. Выгода, приносимая интероперабельностью

Существует развитая отрасль поставки программных инструментов и услуг, которая обуславливает востребованность ADM . Существует потребность в стандартизации, которая обеспечит возможность интеграции и интероперабельности решений от различных поставщиков решений, основанных на ADM. Стандартизация увеличит интероперабельность между различными программными инструментами путем создания открытого каркаса для подобных инструментов (framework). Это может привести к появлению нового поколения решений, от чего выиграет вся индустрия в целом [12].

ADM стандарты позволят пользователям начать процесс модернизации их программных проектов, гарантируя интероперабельность решений от различных поставщиков программных инструментов. Таким образом, стандартизация будет гарантировать конечным пользователям, что они вкладывают деньги не в отдельный инструмент, а, скорее, в некоторую выбранную стратегию.

4.2. Итоговые выгоды от стандартизации

Стандартизация метамодели ADM поможет как всей индустрии в целом, так и отдельным компаниям путем *сокращения рисков,* возникающих при принятии инициатив по улучшению программного обеспечения. Возможность разделять информацию между различными проектами, в которых используются различные инструменты и процессы, сократит время, риски и цену преобразования программного обеспечения. В свою очередь, это улучшит качество инструментов поддержки обратной инженерии и ADM инструментов, обеспечит появление новых возможностей и увеличит возврат от капиталовложений (ROI – Return Of Investment) для инструментов разработки программного обеспечения.

4.3. Область поиска предложений

Своим первым RFP по архитектурно-управляемой модернизации SIG запрашивает предложения по метамодели обнаружения знаний, призванной обеспечить обмен информацией, связанной с трансформацией существующих ресурсов программного обеспечения и их операционных сред [13].

KDM представляет структуру существующего программного обеспечения и связанных с этой структурой артефактов. Такая метамодель обеспечит возможность документирования существующих систем, выявления компонентов, годных для повторного использования в существующем ПО, обеспечит поддержку портирования ПО на другие языки и на MDA, и сделает потенциально возможными другие преобразования.

Метамодель также позволит обмениваться информацией об артефактах существующего программного обеспечения между различными инструментами. Это позволит поставщикам решений, специализирующимся на

определенных языках, платформах или типах трансформаций поставлять потребителям решения, сочетающиеся с решениями других производителей.

4.4. Аспекты обнаружения знаний

Существуют по меньшей мере четыре аспекта знаний, обнаруживаемых в рамках ADM процесса:

- Реализация
- Дизайн
- Архитектура
- Бизнес-правила

В метамодели важно представить все основные артефакты существующего ПО, которые, в общем виде, могут быть описаны как *сущности* (структурные элементы), *отношения* между ними, и *атрибуты*.

Метамодель должна быть достаточно общей, с тем, чтобы она могла обеспечивать все описанные выше уровни трансформации. Метамодель не должна быть ограничена ни одним из языков реализации или платформой. Это утверждение особенно важно для аспектов, связанных с существующими знаниями, относящихся к языковому уровню. Особенно важно, чтобы метамодель могла описывать поведенческие артефакты *вплоть до процедурного уровня, но не ниже его*. В качестве примера процедуры можно привести параграф в языке COBOL или метод в языке Java. Обоснованием этого ограничения области описанного выше RFP является тот факт, что сущности и их отношения ниже процедурного уровня зачастую чересчур тесно связаны с языком и нижележащей платформой. К сущностям языкового уровня относятся, например, методы, файлы с исходным кодом, классы, элементы данных, записи, таблицы или транзакции. В качестве отношений на языковом уровне могут выступать, например, связи вида “функция использует переменную”, “класс является наследником другого класса”, или “к файлу подключен заголовочный файл”. К атрибутам языкового уровня относятся, например, такие атрибуты как имя, права доступа, версия, язык, последнее время просмотра, или тип отношения.

Аспекты дизайна системы могут широко варьироваться в зависимости от принятого методологического подхода. В общем случае информация о дизайне может включать в себя дизайн структур данных, системный дизайн, дизайн интерфейсов и программный дизайн. При этом, к аспектам, поддающимся обнаружению относятся, например, избыточность, сцепление (cohesion), удобство и простота использования, доступность (accessibility), производительность и фрагментация. KDM должен облегчать процесс обнаружения подобных аспектов.

Архитектурные аспекты особенно важны для ADM. На архитектурном уровне существуют несколько различных типов структур:

- Физические структуры (например, файлы, директории, отношения между файлами, связи времени сборки ПО, и т.д.)
- Логические структуры (например, подсистемы, модули, уровни системы, компоненты и их зависимости, различные архитектурные представления и т.д.)

Те физические структуры, которые связаны с процессом сборки существующего ПО, зачастую имеют существенную роль в ADM проектах. К архитектурным сущностям относятся, например, модули, подсистемы, архитектурные уровни, компоненты или библиотеки. Архитектурные отношения включают в себя, например, отношения вида “файл содержится в директории”, “компонент предоставляет метод”, “компонент использует интерфейс другого компонента”, и т. д. К архитектурным атрибутам можно отнести, например, имя компонента.

Для обеспечения применимости ADM на архитектурной основе, KDM должна быть способна описывать как физические, так и логические структуры. Метамодель должна обеспечивать возможность связывать элементы логической структуры с элементами физической структуры и, также, со структурами языкового уровня.

Рассматривая бизнес-правила, следует отметить, что знания о функциях, выполняемых в системе, должны быть представимы в KDM. Сущность, содержащаяся в бизнес-правилах, дает функциональное описание лишь некоторого фрагмента системы, которое может быть дополнено системным аналитиком. Отношение, идущее от заданного бизнес-правила может вести к одному или более физическому аспекту системы. На уровне бизнес-правил KDM не должен быть привязан ни к одной из специфических проблемных областей.

Для совмещения всех этих аспектов ADM, метамодель должна быть создана таким образом, чтобы облегчить ее использование, как в автоматических, так и ручных способах и средствах анализа.

Далее в этой статье описываются проведенные исследования в области архитектурно-ориентированного обнаружения знаний и переработки существующего кода (рефакторинга), выполненные в интересах нашего «производственного» партнера – компании Klocwork.

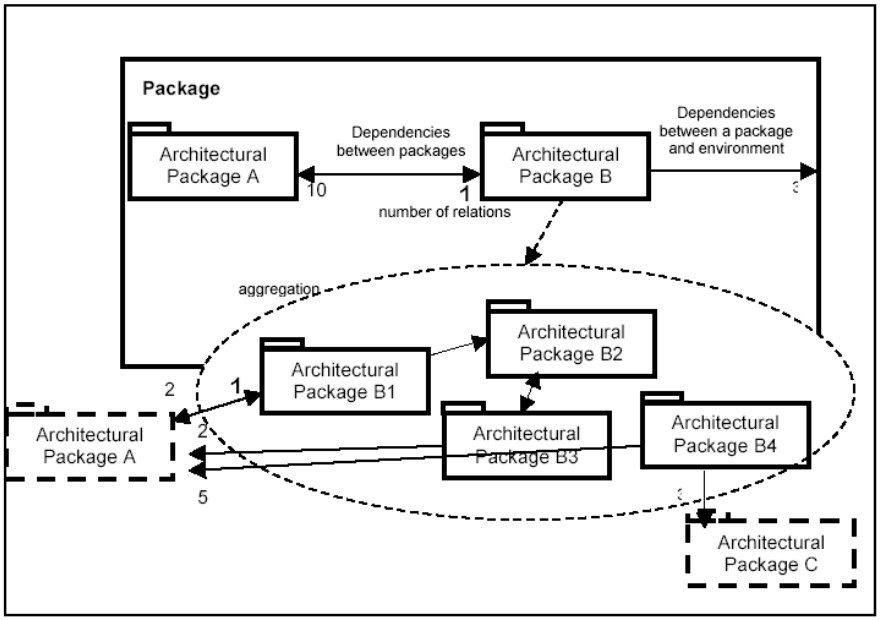
5. Архитектуро-ориентированная методика раскопки знаний компании Klocwork

В этой части описывается систематический процесс создания высокоуровневых архитектурно-значимых компонентных моделей существующего программного обеспечения – так называемый метод архитектурных раскопок Klocwork [7, 8]. Цель этого метода – создание формальной модели архитектуры сущес-твующего ПО, достаточно высокого уровня как для проведения анализа, так и для ее передачи команде разработчиков. Отсюда следует, что такая модель также может использоваться

для управления архитектурой существующего ПО, для анализа проблем в текущей архитектуре и для планирования и коор-динации оптимизационных инициатив. Данный подход чрезвычайно полезен, когда информация об архитектуре ПО устарела, отсутствует или неточна.

Мы предлагаем концепцию «моделей с объединенными интерфейсами», являющуюся масштабируемой, поддерживающую точность интерфейсов компонент и сохраняющую связь с существующим кодом ПО. Подробнее об использовании моделей с объединенными интерфейсами рассказывается в работе [8] метод архитектурных раскопок компании Klocwork.

В основе моделей с объединенными интерфейсами лежат компоненты и их интерфейсы. Так как компоненты могут состоять из подкомпонентов, модели с объединенными интерфейсами допускают неограниченное масштабирование. Модели с объединенными интерфейсами допускают трансформацию компонент (путем переноса символов между компонентами), по ходу трансформации сохраняется точность интерфейсов между компонентами. Это означает, что при появлении абстрактных компонент, объединяющих подкомпоненты, интерфейсы между компонентами остаются точными. Модель направляет процесс раскопки [7, 8].



Изначальная модель автоматически извлекается из исходного кода. Затем модель редактируется вручную, серией трансформаций, повышающих уровень абстракции компонентов и удаляющих случайные зависимости между компонентами. В процессе трансформации сохраняются точность информации, извлеченной из исходного кода. Отсюда, на каждом шаге трансформации, модели с объединенными интерфейсами точны по отношению к реализации.

Результирующая модель может быть использована для анализа и управления разработкой.

В основе моделей с объединенными интерфейсами лежат UML диаграммы пакетов и объектов [3]. Пакетная модель представляет собой высокоуровневые “модули” и зависимости между ними. Модель иерархична, что также находит отражение в одном из представлений модели. В основании иерархии пакетов лежат файлы. Далее, файлы содержат части диаграмм объектов, состоящих из символов, как для классов, так и для отдельных процедур, переменных, типов.

Зависимости между пакетами (модулями существующего ПО) в Klocwork-диаграммах являются производными от отношений между отдельными символами (элементарными языково-зависимыми сущностями исходного кода, такими как типы, процедуры, переменные, макросы и т.п.)

Когда же одна группа сущностей (например, классов, файлов, директорий и т.д.), группируется и объединяется в один пакет, а другая группа – в другой пакет, то отношения между отдельными символами двух пакетов прибавляются к зависимостям между этими пакетами. Это означает, что любая диаграмма пакетов в точности отображает интерфейсы пакетов. Подробнее интерфейсы между пакетами можно изучить, путешествуя по зависимостям между паке-тами и просматривая списки отдельных составляющих зависимостей, или, даже, получить полный список отношений между символами-составляющими пакетов.

Каждое отношение связывается с определенным положением в исходном коде. От любой высокоуровневой диаграммы пакетов возможно спуститься на уровень исходного кода, например посмотреть, откуда берется то или иное отношение. Пакет инструментов Klocwork использует блок-схемы для представления исходного кода, независимо от того, на каком языке написан код программы.

Архитектурные раскопки осуществляются применением к моделям с объединенным интерфейсом одной из двух основных операций: объединение и рефакторинг. Объединение – визуальная операция, в ходе которой несколько выбранных блоков группируются в новый пакет, располагающийся на уровень выше в иерархии пакетов. В модели автоматически пересчитываются зависимости между пакетами на всех уровнях. Тем не менее, агрегация сама по себе не обеспечивает увеличения уровня абстракции диаграмм архитектуры, если она не поддерживается операцией обрамления.

Рефакторинг – визуальная операция, позволяющая переносить блоки между пакетами. Обычно мы переносим отдельные символы с одной низкоуровневой диаграммы объектов из одного модуля в другой. Это означает, что рефакторинг включает в себя “виртуальное” редактирование файлов (конечно же, редактируется только соответствующая часть модели). Рефакторинг позволяет удалить зависимости между высокоуровневыми компонентами, вызванные случайным размещением функциональности на уровне файла. Это

происходит слишком часто, когда физическая архитектура программного обеспечения не соблюдается должным образом.

В ходе архитектурных раскопок изначальная модель ПО подвергается значительным трансформациям. Таким образом, между архитектурными раскопками, анализом и оптимизацией архитектуры нельзя провести четкой границы. Обычно раскопка архитектуры имеет дело с небольшими случайными проблемами, затрудняющими понимание «настоящей» архитектуры и, следовательно, затрудняющими выявление «настоящих» проблем из-за чрезмерной сложности сырых (необработанных) данных.

Архитектурные раскопки в большой степени включают в себя собственно понимание исследуемой программной системы (включая экспертизу предметной области, общее знание о программном обеспечении и здравый смысл) с целью выбрать группу символов для объединения и для рефакторинга. Дополнительные представления модели могут облегчить понимание этих проблем. Дополнительные представления позволяют временно уменьшить сложность, и, следовательно, «разморозить» процесс Архитектурных раскопок. Дополнительное представление модели позволяет обозреть набор объектов, включенных в некоторое первоначальное представление. Как правило, это обеспечивает достаточное понимание архитектурных абстракций. Часть представления может быть эффективно извлечена и, в последствии, применена в более сложном контексте.

6. Архитектурная переработка существующего кода klocwork

Эволюция существующего кода - неотъемлемая часть бизнеса компаний, предоставляющих высокотехнологические программные продукты. Компании, разрабатывающие программные решения, часто модернизируют свои продукты.

Модернизация может быть вызвана следующими факторами:

- Добавление новой или изменение существующей функциональности;
- Улучшение использования ценных частей программного кода путем усиления модульности, особенно при существовании группы программных продуктов (Software Product Line - SPL);
- Миграция к стандартным COTS-компонентам (компонентам готовым к использованию);
- Изменение или обновление платформы;
- Модернизация технологии разработки;
- Улучшение устойчивости программного продукта;
- Реорганизация компании, привлечение внешних ресурсов для разработки проекта.

Для поддержки модернизации необходимы эффективные методы компонентизации существующего программного кода – выделение существующих

компонентов с постепенным определением их границ и интерфейсов. В данной статье описан подход к компонентизации, основанный на применении высокоуровневых моделей существующего кода, сохраняющих, тем не менее, точность по отношению к этому программному коду. Повышение уровня абстракции модели существующего кода – это основной подход, используемый для донесения «вычищенной» архитектуры до команд разработчиков, а также для анализа, оптимизации и управления архитектурой программного продукта.

Основные усилия в исследовании и разработке направлены на увеличение скорости разработки и предоставлению конечному пользователю новых систем. С другой стороны, развитию существующих программных средств уделяется значительно меньше внимания. Развитие существующих программных средств затрагивает разработку новых возможностей, равно как и изменение существующего исходного кода (legacy code). Разработка новой возможности существующей программной системы может быть значительно сложнее разработки эквивалентной системы «с нуля», поскольку все решения прямой инженерии ограничены решениями, реализованными в существующем коде.

Для выживания в условиях усиления давления рынка, организациям, которые занимаются разработкой программного обеспечения, необходимо, рано или поздно, уменьшать стоимость обновления программных средств. Типичный сценарий для телекоммуникационных проектов (а также большинства проектов систем реального времени): только ядро (основная часть) системы появилась как реализация новых идей, и затем, в процессе быстрого накопления опыта, а также по мере появления инвестиций в проект, система разрасталась хаотически. Качество базовой инфраструктуры, как правило, приносится в жертву активному добавлению новых возможностей системы для удовлет-ворения рыночных запросов. Начальная структура программного средства *распадается,* и добавление новых возможностей становится все более сложным. Проблемы поддержки необходимого уровня качества начинают расти. Объединения и приобретения могут еще больше усугубить ситуацию. Все вышеперечисленные проблемы создают дополнительные трудности для улучшения инфраструктуры программного средства, которое может быть описано как *компонентизация* существующего программного средства.

Эти мероприятия представляют собой *архитектурную переработку* существующего кода. Компонентизация также является *архитектурной* активностью, которая определяет и улучшает *модульность* программного средства. Для улучшения модульности существующего программного средства следует произвести следующие действия:

- выделить компоненты,
- определить границы будущих компонентов путем агрегации подкомпонентов,
- представить новый компонент одним графическим символом и

- исследовать интерфейсы между компонентами для определения проблем модульности,
- изменить компоненты добавлением или удалением содержимого,
- по мере того как появляются лучшие модули, более продвинутые архитектурные механизмы могут быть использованы для более четкого определения границ модулей и определения межмодульных интерфейсов.

Инструменты, автоматизирующие процесс компонентизации, требуют точной информации о взаимосвязях между компонентами. В настоящий момент существует большое количество так называемых *инструментов обратной инженерии*, извлекающих подобную информацию из существующего исходного кода. Однако существующие инструменты обратной инженерии не могут решить проблему компонентизации. По нашему мнению, это происходит из-за недостаточности высокоуровневого представления существующего кода, получаемого при помощи традиционных инструментов. Другими словами, существующие инструменты обратной инженерии извлекают информацию об индивидуальных взаимосвязях между элементами существующего програм-много средства, а не работают в терминах модульных интерфейсов.

Основной целью исследований и разработок, проводимых отделом CASE-инструментов для klocwork, является уменьшение последствий вышеописанной проблемы. Наш подход к развитию существующего программного кода основан на, так называемой, методологии архитектурных раскопок [2].

7. Управляемые архитектуры

В этой части статьи мы обсудим, как архиеткурно-ориентированное обнаружение знаний о ПО (архитектурные раскопки) и архитектурная переработка могут существенно улучшить возможности различных организаций по разработке программных систем. Мы продемонстрируем, как использование архитектурных инструментов может облегчить переход от управления реализацией к управлению архитектурой [10].

Что такое управляемая *архитектура программного средства*? Мы будем использовать следующее определение, основанное на широко известной модели SEI CMM. В действительности, это проекция модели SEI CMM в домен архитектурных моделей, так называемой Architecture Capability Maturity Model (ACMM).

ACMM покрывает единственный аспект SEI CMM – архитектуру существующего программного кода.

Уровень 1: Начальная архитектура

Любое программное средство имеет некоторую *структуру* – вне зависимости от того, определена она или нет, понятна она (разработчикам) или нет. Она состоит из некоторых компонентов (может быть из одного монолитного компонента), у компонентов есть некоторые *зависимости*, компоненты объединяются в *конфигурации*.

Уровень 2: Повторяемая архитектура

Часто организации используют *повторяемые шаблоны* конструирования и поставки программных систем: некоторые правила компоновки, использование некоторых библиотек, повторное использование кода. Обычно шаблоны появляются на физическом уровне. Любая организация, обладающая большим количеством программных ресурсов, включая *вариации* (любое встроенное программное обеспечение), начинает интересоваться вопросами *архитектуры программных средств*, с момента попытки определения составляющих успеха группы программных продуктов. Существуют и другие мотивы, побуждающие организации уделять больше внимания качеству архитектуры их программных продуктов и начать работы по ее *оптимизации*.

Уровень 3: Определенная архитектура

На этом уровне компоненты, их интерфейсы и конфигурации формально *определены и поддерживаются* вместе с остальным кодом. При этом используется компонентное окружение, средства моделирования вроде Rational Rose. Существует несколько различных сценариев перехода организаций на этот уровень. Некоторые начинают «с чистого листа», и определяют архитектуру до начала разработки. Другие делают это позже, когда линейка программных продуктов становится достаточно конкурентоспособной.

Как бы то ни было, определение архитектуры программного средства является первым шагом к настоящему управлению программным средством, с точки зрения архитектуры.

Уровень 4: Управляемая архитектура

Архитектура программы становится *управляемой,* когда организация получает своевременную, точную и количественную информацию о компонентах, а также их зависимостях и конфигурациях. Для достижения этого уровня требуется:

- визуализация существующего программного кода;
- обратная связь между «разработанной» архитектурой и «построенной» архитектурой;
- использование метрик существующей архитектуры.

Организации, получающие эту информацию, используют ее для увеличения целостности архитектуры их программных продуктов.

Уровень 5: Оптимизирующая архитектура

Архитектурная целостность *усилена* и *улучшена*. Действия по улучшению архитектуры становятся неотъемлемой частью общего процесса разработки.

Есть существенное различие между Определенными Архитектурами и Управляемыми Архитектурами с точки зрения используемых программных систем. На уровне Определенной Архитектуры используются автоматизированные средства моделирования. На уровне Управляемых Архитектур

используются инструменты, обычно относящиеся к инструментам обратной инженерии или инструментам обеспечения качества ПО.

Одна из крупнейших проблем – расширить средства моделирования для применения их во всех ситуациях, встречающихся в существующих программных ресурсах.

Типичная *роль ответственного за архитектуру* скорее *реактивная*: сначала формулируются некоторые требования, затем они передаются разработчикам для реализации, и затем оцениваются результаты. Обычно, цикл конструирования является наиболее длинным, поэтому, когда проблемы выявляются – уже слишком поздно. В таких случаях устранить обнаруженные дефекты, не нарушая архитектурную целостность системы, практически невозможно. В связи с этим мы часто говорим об *эрозии* или *распаде архитектуры* – процессе деградации межкомпонентных зависимостей с течением времени, когда ответственный за архитектуру теряет контроль над ситуацией.

Средства автоматизации управления архитектурой программного обеспечения сокращают цикл оценки результатов.

Однако упреждающий подход к управлению архитектурой программной системы должен интегрировать действия по усилению целостности архитектуры в сам процесс разработки. Например, определенные «архитектурные правила» должны проверяться «на лету», когда разработчик пытается обновить некоторый модуль в системе управления конфигурацией.

Цель инструментальной поддержки существующего ПО в ускорении разработки программной системы, использующей существующий программный код. Klocwork inSight использует современные, запатентованные алгоритмы анализа программного кода для выделения программной архитектуры, взаимодействия компонентов, логических последовательностей и исполняемых потоков непосредственно из исходного кода как системы целиком, так и ее частей [1]. Средства Klocwork позволяют анализировать, автоматически контролировать и управлять архитектурой при помощи графической визуализации, задания архитектурных правил и автоматического слежения за ними. Средства Klocwork также включают инструмент для поиска логических ошибок «на лету» для их раннего обнаружения и удаления. Посредством интуитивного, дружественного интерфейса, они быстро и точно снабжают архитекторов и проектировщиков уникальной, востребованной, ранее не доступной информацией, что уменьшает временной цикл разработки и улучшает качество ПО. Klocwork также имеет инновационные и автоматизированные решения для оценки программного обеспечения, с возможностью составления отчета по системе в целом, или отдельным ее частям, со 100% точностью.

8. Архитектурно-ориентированный переход к MDA

Поддержка существующего ПО – только один аспект более широкого понятия *эволюции* программного обеспечения. Инфраструктура современных промыш-

ленных вычислительных систем быстро изменяется [18]. Это означает, что развитие программного обеспечения не должно отставать от этих изменений, что, в свою очередь, требует реструктуризации большей, чем просто разбиение на модули. Становится очевидным, что исходный код не всегда подходит для поддержки и эволюции, поскольку он обладает как платформо-независимыми, так и специфичными для некоторой платформы свойствами.

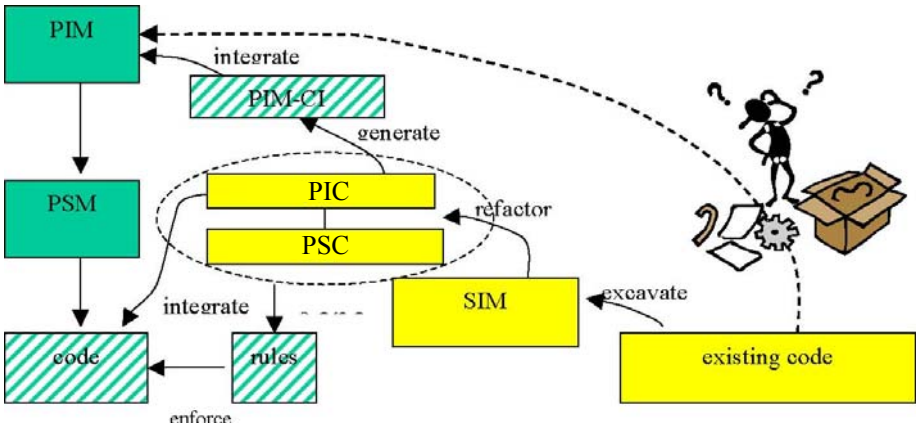
Подход, основанный на модельно-управляемой архитектуре (MDA), который в настоящее время стандартизован OMG (http://www.omg.org/mda), может быть рассмотрен как применение принципа управляемой архитектуры программ (Managed Software Architectures) для новых (green field) проектов, созданных прямым моделированием с использованием UML, где модель является основным поддерживаемым артефактом.

Ключевое понятие MDA – переход от поддержки кода к моделированию. MDA также выступает за разделение платформо-независимых и платформо-зависимых моделей (PIM, PSM). Реализация для выбранной платформы получается из PIM через PSM с использованием автоматической генерации кода. Это очень перспективный подход.

Однако, при переходе от существующего кода к MDA возникает проблема унаследованного кода, так как требуется “раскопка” (excavation) PIM из существующего кода.

Мы полагаем, что наши обобщенные интерфейсные модели (summary interface models), которые поддерживают неограниченную агрегацию, рефакторинг и связи с исходным кодом, делают возможным создание архитектурно значимых моделей существующего ПО, которые могут быть постепенно переработаны в PIM для существующих модулей. Таким образом, становится возможной интеграция существующих модулей в ориентированный на MDA процесс разработки ПО. Обобщенные модели хорошо соотносятся с моделями, используемыми в процессе прямой инженерии, поэтому они позволяют преодолеть проблему унаследованного кода для лучшего внедрения современных технологий разработки программных средств в промышленности.

Рисунок, приведенный ниже, иллюстрирует архитектурно-ориентированный процесс миграции в направлении MDA. На левой части рисунка приведены трансформации, входящие в MDA (переход от PIM к PSM, и от PSM к коду). Эти трансформации предполагают наличие детальной исполняемой модели. Проблема интеграции существующих программных систем в “мир MDA” заключается в том, что такая модель обычно отсутствует, и должна быть восстановлена с наименьшими трудозатратами. По нашему мнению, “прямой подход” перехода от исходного кода к PIM трудно реализовать по причине сложности артефактов существующего программного обеспечения. Вместо этого, мы выступаем за следующий архитектурно-ориентированный подход.



Во-первых, переход к управляемой архитектуре производится с помощью раскопки (excavating) обобщенной интерфейсной модели (summary interface model, SIM) существующего кода. Как было показано ранее, этот переход обычно включает некоторый рефакторинг архитектуры с целью восстановления интеллектуального контроля над “распадающейся” архитектурой.

Во-вторых, из обобщенной интерфейсной модели в дальнейшем выделяются две части: платформо-независимые компоненты (Platform-Independent Components, PIC) и платформо-зависимые компоненты (Platform-Specific Components, PSC).

В-третьих, модель компонентных интерфейсов, соответствующая PIC, может быть получена и экспортирована в инструментарий MDA. Дальнейшая разработка на основе существующего кода может увеличить детализацию модели. Таким образом, архитектурно-ориентированная модель существующего программного обеспечения может быть интегрирована в модель MDA.

Конечно, эта архитектурно-ориентированная модель не полностью поддерживает MDA, поскольку в ней содержатся только ссылки на интересующие компоненты и их интерфейсы, но не их поведение. Существующие компоненты все еще необходимо интегрировать с новыми компонентами, созданными с помощью средств MDA.

Преимущество этого подхода заключается в том, что целостность модели может быть усилена при использовании инструментов, описанных выше.

9. Заключение

Мы обобщаем наш опыт в определении стандартов архитектурно-управляемой модернизации в Object Management Group (OMG) с помощью нашего партнера Klocwork. В частности, мы описали нашу методологию архитектурно-ориентированного обнаружения и рефакторинга знаний. Применение этого подхода организацией-разработчиком программного обеспечения может привести к

увеличению производительности за счет облегчения перехода от управления реализацией (существующим кодом) к управлению архитектурой ПО. Мы считаем это существенным шагом для увеличения пригодности ПО к модернизации с использованием MDA, SOA (service-oriented architecture) или еще какой-нибудь архитектурно-управляемой модернизации. Мы обозначили миграционный процесс, в котором архитектура становится управляемой, а целостность архитектуры усиливается, чтобы затем сделать возможным создание моделей интерфейсов компонентов (component interface models, CIM) и их интеграцию с новым процессом разработки, поддерживающим MDA.

Литература:

1. N. Rajala, D. Campara, N. Mansurov, inSight Reverse Engineering CASE Tool, in Proc. of the ICSE 99, Los Angeles, USA, 1998.
2. Н. Мансуров, А. Рагозин, Генерация кода с простой и читабельной структурой по SDL-92, Вопросы кибернетики: System Programming Applications, N. 3, Москва, 1997 (in Russian)
3. N. Mansurov, A.Ragozin, Using declarative mappings for automatic code generation from SDL and ASN.1, in Proc. 9th SDL Forum, Montreal, Canada, June 21- 26, 1999, Elsevier Science Publishers B.V. (North-Holland).
4. R. Probert, N. Mansurov, Improving time-to-market using SDL tools and techniques (tutorial), Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999.
5. J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Addison-Wesley, 1999
6. A. Ragozin, Automatic generation and execution of programs from SDL specifications, PhD thesis, Moscow State University, 1999
7. N. Mansurov, A Systematic Approach to Recovering Architecture from Existing Software, SD Expo West, San Jose, 25th April, 2002
8. N. Mansurov, D. Campara, Extracting High-Level Architecture From Existing Code with Summary Models, in Proc. IASTED Conf. On Applied Informatics, Innsbruck, Austria, 2003
9. N. Mansurov, D. Campara, Using Message Sequence Charts to accelerate maintenance of existing systems, in Proc. 10th SDL Forum, Copenhagen, 2001, LNCS, Springer Verlag, 2001
10. N. Mansurov , Using Metrics to enforce quality of Managed Architectures , in industrial presentations proc. of int. Conf. Metrics-2002, Ottawa, Canada, 2002
11. OMG, Legacy Transformation Working Group Forms, Draws Number of New OMG Members, OMG press release, July 2003
12. OMG, Why do we need legacy transformation standards?, OMG whitepaper, 2003
13. OMG Architecture-Driven Modernization: Knowledge Discovery Meta-model RFP, 2003
14. D. Frankel, et. al. OMG Model-Driven Architecture, Addison-Wesley, 2003
15. A.van Deursen, P. Klint, C. Verhoef, Research issues in the Renovation of Legacy Systems, CWI reseaarch report P9902, April 1999