DOI: 10.15514/ISPRAS-2026-38(3)-22

# Рефакторинг исходного кода на основе LLM и расширения UML

*Е.А. Караваева, ORCID: 0009-0003-0418-7685 <eakaravaeva_1@edu.hse.ru>*
*Л.А. Резуник, ORCID: 0009-0000-9428-4718 <lrezunik@hse.ru>*
*Л.А. Кулигин, ORCID: 0009-0005-7892-9711 <lkuligin@hse.ru>*
*Д.В. Александров, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>*

*Научно-учебная лаборатория облачных и мобильных технологий, Национальный исследовательский университет "Высшая школа экономики" (НИУ ВШЭ), Россия, 109028, Москва, Покровский бульвар, д. 11.*

**Аннотация.** В статье представлен метод рефакторинга исходного кода на основе интеграции большой языковой модели (LLM) и расширенной UML-модели программного кода. Предложенный подход позволяет выявлять проблемные участки кода с использованием функций тревожности и структурных метрик классов, а затем выполнять автоматизированный рефакторинг. Ключевой особенностью метода является использование LLM для генерации формальных спецификаций на языке OCL (Object Constraint Language), которые затем используются для автоматической верификации корректности преобразований через проверку инвариантов, пред- и постусловий. Расширение UML включает локальные переменные, действия методов и связи между ними, что обеспечивает низкоуровневый анализ и корректное преобразование кода. Экспериментальная проверка показала, что метод LLM + UML* обеспечивает высокую точность обнаружения дефектов, полное устранение функций тревожности, сохранение функциональности системы.

**Ключевые слова:** рефакторинг исходного кода; язык моделирования UML; расширение модели; большая языковая модель; функции тревожности; автоматизация анализа кода; объектный язык ограничений OCL.

**Для цитирования:** Караваева Е.А., Кулигин Л.А., Резуник Л.А., Александров Д.В. Рефакторинг исходного кода на основе LLM и расширения UML. Труды ИСП РАН, том 38, вып. 3, часть 2, 2026 г., стр. 67–94. DOI: 10.15514/ISPRAS–2026–38(3)–22.

**Благодарности:** Данная статья является результатом исследовательского проекта, реализованного в рамках программы фундаментальных исследований Национального исследовательского университета «Высшая школа экономики» (НИУ ВШЭ).





# Source Code Refactoring Based on LLM and UML Extension

*E.A. Karavaeva, ORCID: 0009-0003-0418-7685 <eakaravaeva_1@edu.hse.ru>*
*L.A. Rezunik, ORCID: 0009-0000-9428-4718 <lrezunik@hse.ru>*
*L.A. Kuligin, ORCID: 0009-0005-7892-9711 <lkuligin@hse.ru >*
*D.V. Alexandrov, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>*

*Scientific and Educational Laboratory of Cloud and Mobile Technologies, Higher School of Economics (HSE University), Russia, 109028, Moscow, Pokrovsky blvd., 11.*

**Abstract.** The article presents a method for refactoring source code based on the integration of a large language model (LLM) and an extended UML model of the program code. The proposed approach allows identifying problematic code areas using anxiety functions and structural class metrics, and then performing automated refactoring with precondition and postcondition verification in OCL. The UML extension includes local variables, method actions, and the relationships between them, enabling low-level analysis and correct code transformation. Experimental validation showed that the LLM + UML* method provides high defect detection accuracy, complete elimination of anxiety functions, and preservation of system functionality.

**Keywords:** source code refactoring; large language models; UML extension; software quality metrics; automated code transformation; object-oriented programming; OCL.

**For citation:** Karavaeva E.A., Kuligin L.A., Rezunik L.A., Alexandrov D.V. Source Code Refactoring Based on LLM and UML Extension. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 2, 2026, pp. 67-94 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-22.

**Acknowledgements.** This work is an output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE University).

## *1. Введение*

Непропорциональность между сложностью программных систем и возможностями их ручной верификации стимулировала развитие систем искусственного интеллекта, способных автоматизировать процессы анализа и рефакторинга программного кода. Однако, исследователями подчеркивается одна из ключевых проблем в использовании ИИ для рефакторинга – уверенность в его корректности [1]. Большинство существующих ИИ-инструментов успешно справляется с задачей рефакторинга небольшой части кода, но нельзя утверждать, что при рефакторинге больших проектов, исходный код будет трансформирован корректно. В таких случаях специалистам могут понадобиться месяцы для доказательства правильности сгенерированных программ. Стоит отметить, что многие проблемы связаны с ограничениями моделей ИИ, используемых в инструментах рефакторинга – это размер памяти (контекста) и невозможность охватить весь исходный код целиком, галлюцинации ИИ, ложноположительные срабатывания и тому подобное.

В данной работе мы предлагаем методику рефакторинга, основанную на использовании большой языковой модели (LLM), а также представлении исходного кода в виде модели (в данном случае – UML). Преимущество такого метода состоит в том, что в отличии от статических анализаторов кода, он не зависит от конкретной платформы и может быть применен в отношении кода на любом языке программирования (за счет использования LLM), а также позволяет более эффективно использовать LLM посредством произведения рефакторинга относительно расширенной UML-модели программного кода. Стоит также отметить ограничение данного метода – его возможно использовать только по отношению к программным проектам, реализованным в объектно-ориентированной парадигме.

## *2. Определение рефакторинга*

Рефакторинг – это изменение внутренней структуры программы без изменения ее внешнего поведения. Формализованное определение рефакторинга:

$$\mathrm{R} = (\mathrm{pre}, \mathrm{T}, \mathrm{post}), \quad (1)$$

где $pre$ – предусловия для выполнения рефакторинга, $T$ – изменение (трансформация) программы, $post$ – постусловия выполнения рефакторинга.

В данном разделе мы более подробно остановимся на определении предусловий рефакторинга в рамках разработанного метода.

Предусловия – показатели, определяющие применим ли конкретный рефакторинг в той или иной ситуации, предотвращая изменение внешнего поведения системы. В нашей работе рассматривается основной вариант предусловий – функции тревожности.

### 2.1 Характеризация участков дефектного кода

Далее мы вводим понятие функций тревожности – это функции, позволяющие определить участок дефектного кода и необходимый рефакторинг. В рамках данной работы мы фокусируемся на отдельных проблемах. Перечислим основные категории дефектов программного кода и функции тревожности, их описывающие:

1. **Большой класс**. Такой класс содержит избыточность, проявляющуюся в большом числе строк кода, атрибутов, методов. Данный дефект можно выразить через следующую функцию тревожности:

$$Anx_1 = 0.6 \times CS/CSR + 0.4 \times WMC/WMCR, \text{ где}$$

- $CS$ – размер класса,
- $WMC$ – число взвешенных методов на класс, выраженное через:

$$WMC = \sum i = 1 n w_i, \text{ где}$$

$n$ – число методов в классе, а $w_i$ – вес $i$-го метода, определенного по формуле:

$$w_i = E - N + 2P, \text{ где}$$

- E – количество ребер в графе потока управления,
- $N$ – количество вершин,
- $P$ – количество компонент связности.

*Способ рефакторинга:* разбиение на несколько классов/выделение подклассов.

2. **Класс с расходящимися модификациями.** При изменении в одной его части сталкивается с необходимостью модифицировать множество участков кода класса, приводя к неоптимальности системы.

$$Anx_2 = 0.4 * \mathrm{CBO}/\mathrm{CBO}_\mathrm{R} + 0.3 * \mathrm{RFC}/\mathrm{RFC}_R + 0.3 * \mathrm{COF}/\mathrm{COF}_R, \text{ где}$$

- $CBO$ – сцепление между классом объекта,
- $\mathrm{CBO}(C) = |\{D \in Classes \mid D = C \wedge (C \text{ использует } D \vee D \text{ использует } C)\}|$

*Способ рефакторинга:* уменьшение связности, выделение класса.

3. **Дробный класс.** При модификации приводит к изменениям в нескольких других классах. Формула и способ рефакторинга аналогичны классу с расходящимися модификациями.

4. **Класс с параллельными иерархиями наследования.** При добавлении подкласса требует создания подкласса другого класса.

$$Anx_3 = 0.5 * inheri(DIT_1, DIT_2, NOC_1, NOC_2) + 0.5 * Anx_2(CBO, RFC, COF), \text{ где}$$





- $inheri(DIT_1, DIT_2, NOC_1, NOC_2) = \begin{cases} 2: DIT_1(NOC_1) = \pm 1\ DIT_2(NOC_2) \\ 0: \text{ во всех остальных случаях} \end{cases}$
- $DIT$ - глубина(высота) дерева наследования,
- $NOC$ – количество подклассов.

*Способ рефакторинга:* перемещение методов/полей в другие классы, разделение большого класса на части, объединений иерархий

5. **Ленивый класс.** Его использование значительно уменьшилось в результате модификаций и выполняемые им функции больше не оправдывают затраты на него.

$$Anx_4 = 0.5 * lazy(DIT) + 0.3 * blank(CS, NOF) + 0.2 * RFC, \text{ где}$$

- $lazy(DIT) = \begin{cases} 2: DIT \in \{1\} \cup [19, \infty) \\ 0: \text{ во всех остальных случаях} \end{cases}$
- $blank(CS, NOF) = \begin{cases} 2: CS < 4 \text{ или } NOF < 4 \\ 0: \text{ во всех остальных случаях} \end{cases}$
- $NOF$ – количество полей в классе.

*Способ рефакторинга:* если функционал ленивого класса используется только в одном классе, можно встроить ленивый класс в класс, который его использует. В противном случае можно извлечь используемую часть функционала ленивого класса в отдельный новый класс и при необходимости связать со старым.

6. **Абстрактный класс.** Он не приносит пользы или в целом не используется, так как был создан для дальнейших модификаций, которые не были реализованы.

$$Anx_5 = 1 * specGen(NOC), \text{где}$$

- $\mathrm{specGen} = \begin{cases} \frac{1}{\mathrm{NOC}}: \mathrm{NOC} > 0 \\ 2: \text{ во всех остальных случаях} \end{cases}$.

*Способ рефакторинга:* если абстрактный класс нигде не используется, его можно удалить. Если часть его функционала была реализована в подклассах, можно перенести эти методы в соответствующие подклассы, чтобы убрать неиспользуемые элементы.

7. **Неуместная близость.** Класс со слишком высоким доступом к закрытым полям другого, что свидетельствует о сильном нарушении инкапсуляции в программе.

$$Anx_6 = 0.8 * CBO/CBO_R + 0.2 * RFC/RFC_R$$

*Способ рефакторинга:* введение публичных методов getters/setters или перенос части логики в класс, которому принадлежат поля, чтобы операции выполнялись над его собственными полями.

8. **Посредник.** Класс, делегирующий другому классу больше половины методов, что свидетельствует о слишком высокой связности между классами. Ему соответствует та же формула, что и выше.

*Способ рефакторинга:* объединить эти классы, чтобы снизить избыточную связанность и упростить структуру кода или переместить часть логики из класса-делегата обратно в делегирующий класс.

### 2.2 Использование моделей в контексте задачи рефакторинга кода

Рефакторинг на основе моделей является одним из популярных направлений, получившим наиболее активное развитие в начале 2000-ых годов [2, 3]. Однако, из-за слишком высокой абстракции многие детали в моделях, которые могут понадобиться для низкоуровневого рефакторинга, остаются опущены. Для того, чтобы было возможно использовать модель в

целях рефакторинга, она должна соответствовать ряду критериев. В данной работе мы опираемся на [4-5].

Примером такого критерия является связность между кодом и моделью, что дает возможность чтения модели и ее понимания разработчиками системы. Кроме того, это свойство модели можно использовать для пересборки после рефакторинга лишь отдельных частей системы, что может снизить вычислительную нагрузку в результате процесса рефакторинга, а также избежать несовершенства систем кодогенерации.

Как было упомянуто ранее, модели могут обладать высоким уровнем абстракции, так как изначально они были созданы для абстрагирования от мелких деталей и выделения наиболее общих характеристик системы [5]. Рассматривая модель на требуемом уровне абстракции, разработчик может сосредоточиться на наиболее важных качественных характеристиках системы. Для того, чтобы сохранить данное свойство модели, при этом открыв возможность для низкоуровневого рефакторинга, возможно создание метамодели, на основе существующей модели, раскрывающей детали о структуре методов, локальных переменных и тому подобной информации. При этом важно реализовать проекцию данной метамодели в исходное представление, так как необходимость отображения деталей может затруднить понимание кода разработчиком.

За счет своих возможностей для поиска дефектов программных систем (разработанных на базе объектно-ориентированных языков программирования), как вручную, так и автоматизированно, широкое применение приобрела такая нотация, как UML (Unified Modelling Language). UML обладает спектром недостатков, следует заметить, что она в том числе, не удовлетворяет вышеописанным критериям. Попытки задействовать UML в качестве модели для рефакторинга предпринимались рядом исследователей [6-8]. Данные попытки носили чаще ограниченный характер и ориентировались в основном на мелкие рефакторинги. Тем не менее, отмечается полезность применения данной нотации в целях рефакторинга, например, совместное использование представление классов и взаимодействий между ними (последовательностей вызовов) [9]. Таким образом, можно предположить, что посредством расширения данной нотации, будет возможно отразить детали, необходимые для низкоуровневого рефакторинга.

### 2.2.1 Расширение UML

UML – это стандартизированный язык, предназначенный для спецификации, визуализации, построения и документирования сложных программных систем. В настоящее время UML широко используется для описания моделей системы, имея возможность конвертировать код из одного представления в другое. UML может быть расширен с помощью профилирования и MOF (Meta Object Facility). Профилирование позволяет добавляет атрибуты и константы к существующим элементам модели. В отличие от первого варианта, MOF – является языком для определения элементов (в т. ч. через него определен UML) и позволяет добавлять новые элементы.

Базовый подход к расширению UML для целей рефакторинга, использованный в данной работе, основан на работе [6], в которой описаны механизмы расширения UML для рефакторинга через добавление деталей реализации (таких как локальные переменные и действия) в метамодель. Это позволяет преодолеть ключевое ограничение стандартной UML – высокий уровень абстракции, недостаточный для низкоуровневых преобразований кода.

Формально UML можно рассматривать как четверку:

$$\mathrm{UML} = \langle \mathrm{C}, A, O, R \rangle, \text{ где}$$

- $\mathrm{C}$ – множество классов,
- $A$ – множество атрибутов,
- $\mathrm{O}$ – множество операций (методов),





- $R \subseteq (C \times C) \cup (C \times A) \cup (C \times A)$ – множество отношений (ассоциации, зависимости, обобщения и т. д.).

В стандартном UML метод описывается тройкой:

$$Method = \langle name, parameters, returnType \rangle, \text{ где}$$

- $parameters$ – список параметров, каждому из которых соответствует тип, описанный в $C$,
- $returnType$ – тип возвращаемого значения из $C$.

Стоит отметить, что в данном варианте метод не содержит описания тела. В качестве расширения метамодели UML для ее обогащения в целях детального рефакторинга предлагается расширить пакеты *Foundation.Core* и *Elements.Common.Behavior*.

Таким образом новую модель можно описать следующим образом:

$$UML^* = \langle C, A, O, R, L, Act \rangle, \text{ где}$$

- $L$ – множество локальных переменных (LocalVariable),

$$LocalVariable = \langle name, type, body \rangle, type \in C, body \subseteq Act$$

- $Act$ – множество действий, связанных с методами (Action).

$$Action = \langle target, value, stereotype \rangle,$$
$$stereotype \in CallAction, UpdateAction, AccessAction$$

В данной версии модели $UML^*$ метод описывается как:

$$Method = \langle name, parameters, returnType, body \rangle, \text{ где}$$

- $body$ – тело метода, описываемое через $ActionSequence$.

$$ActionSequence = \langle actions, locals \rangle$$

- $actions$ – действия из Act,
- $locals$ – локальные переменные.

***Основные шаги расширения:***

1. Установлена связь между Method и его внутренними операторами через элемент ActionSequence.
2. Добавлен элемент LocalVariable как специализация ModelElement.
3. Определена связь между LocalVariable и ее типом (Classifier).
4. Связана LocalVariable с областью видимости (ActionSequence).
5. Добавлена связь между Action и ее фактическими аргументами (Argument).
6. Атрибут value элемента Argument уточнен как связь valueRefinement, указывающая на элемент ModelElement.
7. Введен элемент SingleTargetAction как специализация Action.
8. Атрибут target элемента SingleTargetAction уточнен как связь targetRefinement, указывающая на ModelElement.

Итоговое представление метамодели UML, полученное в результате расширения, представлено схематически на рис. 1.

Дополнительно вводятся новые ограничения, направленные на обеспечение согласованности между моделью и кодом. В неформальном виде они формулируются следующим образом:

1. targetRefinement элемента SingleTargetAction со стереотипом AccessAction или UpdateAction должен указывать на локальную переменную, параметр или атрибут.

2. valueRefinement элемента Argument должен ссылаться на локальную переменную, параметр или атрибут.
3. targetRefinement действия не может ссылаться на атрибут, определенный в подклассе.
4. targetRefinement не может ссылаться на локальную переменную или параметр, принадлежащий другому методу.

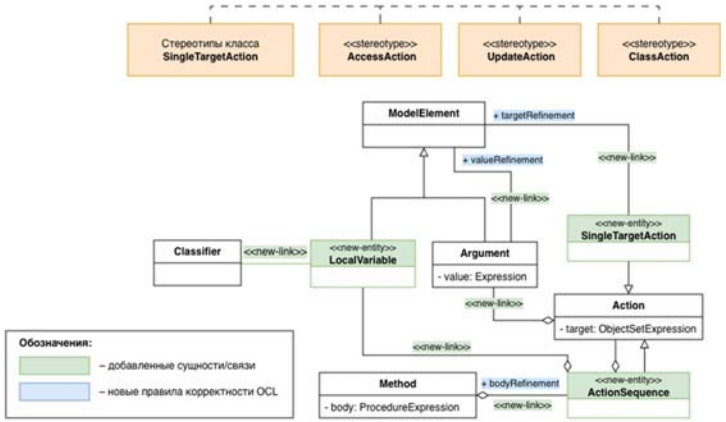


*Рис. 1. Фрагмент метамодели UML с добавленными расширениями.*
*Fig. 1. UML metamodel fragment with added extensions.*

### 2.3 Роль OCL в верификации LLM-генераций

Язык объектных ограничений (OCL) представляет собой строгий формализм для спецификации инвариантов классов и контрактов операций, которые невозможно выразить средствами графической нотации UML. Интеграция OCL в контур работы больших языковых моделей (LLM) позволяет решить проблему «галлюцинаций» генеративных моделей за счет перехода от вероятностной генерации кода к детерминированной валидации.

В предлагаемом подходе LLM выполняет роль «транслятора семантики», преобразуя неявные требования исходного кода в явные императивные ограничения OCL. Сгенерированные выражения OCL действуют как семантический фильтр:

- *Инварианты (Invariants):* гарантируют, что состояние объекта остается валидным после структурных изменений.
- *Предусловия (Preconditions) и постусловия (Postconditions):* обеспечивают соблюдение принципа подстановки Лисков и корректность контрактов методов при рефакторинге.

Такой подход позволяет объединить гибкость LLM в понимании контекста с точностью формальных методов верификации, создавая гибридный механизм контроля качества программных трансформаций.

Пример расширенной UML-модели и сгенерированных на ее основе инвариантов, предусловий и постусловий в формате OCL представлены в Приложении 1.

## *3. Применение LLM в контексте задачи рефакторинга*

В данном разделе будет изложен краткий обзор и анализ исследований, посвященных рефакторингу программного кода. Ввиду того, что тема очень обширна, рассмотрены работы,





связанные прежде всего с интегрированием LLM в инструменты рефакторинга, так как в развитии этого направления наблюдаются большие перспективы.

С помощью LLM можно выполнять различные задачи, касающиеся рефакторинга программного кода: трансформацию фрагментов кода (например, отдельных методов) [10], автоматического исправления программ [11], выявления участков программы плохого качества, устранения уязвимостей [12] и тому подобным. В отдельных работах демонстрируются подходы к интеграции LLM в существующие инструменты разработки. Так, в [10] представлен плагин для IntelliJ IDEA, позволяющий выполнять исправления выбранных участков кода непосредственно в среде разработки. Это показывает практическую применимость LLM в контексте рефакторинга программного кода и потенциал для внедрения подобных решений в реальные рабочие процессы.

Однако, так как данное направление все еще стремительно развивается, приходится сталкиваться с ограничениями и проблемами. Так, можно заметить, что одним из ключевых факторов, влияющих на эффективность применения LLM в задаче трансформации программного кода (но также и в других задачах), является оперативное проектирование (prompt-engineering). В работах [13-14] отмечается, что качество результатов сильно варьируется в зависимости от точности формулировки задачи: без четких инструкций модели часто допускают ошибки при генерации или же генерируют неоптимальные решения.

Также внимание уделяется проблеме отсутствия объективных и сбалансированных бенчмарков. Работа [12] указывает, что большинство существующих наборов данных обладают смещениями, не отражающими реальных сценариев программирования. Для частичного решения этой проблемы авторы предлагают итеративный подход к генерации и корректировке кода, что повышает стабильность результатов.

Одним из основных вызовов остается ограничение контекстного окна, которое не позволяет модели держать в памяти все детали о программном проекте (исключение составляют небольшие проекты и скрипты). Разные исследования предлагают собственные пути преодоления этой проблемы: использование пошаговой обработки фрагментов кода [13], комбинирование LLM с традиционными инструментами рефакторинга [14], применение ансамблевых методов, объединяющих результаты нескольких моделей [15].

Отметим, что большинство рассмотренных решений остаются платформенно и языково зависимыми. Как правило, исследования выполняются с учетом ограниченного набора языков программирования, таких как Java или C++, что объясняется спецификой используемых бенчмарков и наборов данных. Это ограничивает возможность обобщения результатов и их переносимости на другие языки и платформы.

## *4. Описание методики рефакторинга*

В данной работе предлагается методика рефакторинга программного кода, основанная на интеграции LLM и представления кода в формате расширенной модели UML. Предполагается, что данный метод позволит выполнять высокоуровневые задачи рефакторинга, такие как рефакторинг архитектуры, но также и низкоуровневый рефакторинг, к которому можно отнести работу с отдельными строками кода.

Процесс рефакторинга предлагается разбить на следующие этапы:

1. **Генерация расширенной UML-модели.**

   На данном этапе происходит создание расширенной модели *UML** (см. п. 2.2.1) на основе программного кода. Модель создается посредством исполнения скрипта, осуществляющего проход по дереву классов проекта.

2. **Выявление проблемных участков кода.**

   На данном этапе идентифицируются участки программного кода с потенциальными дефектами, упомянутыми ранее в п. 2.1. Для каждого типа дефекта сформулирована

функция тревожности *Anx*, которая позволяет количественно оценить необходимость рефакторинга. Для вычисления *Anx* используются метрики, включающие, количество строк кода, глубину иерархии, связность классов и другие структурные характеристики. Данные метрики извлекаются из расширенной UML-модели. На основе значений *Anx*-функций определяются конкретные проблемы и классы, требующие рефакторинга.

3. **Автоматизированный рефакторинг.**

   В качестве входных данных LLM предоставляется расширенная UML-модель и метрики *Anx*. Для каждого выявленного дефекта LLM генерирует:

   - Предусловия рефакторинга на языке OCL;
   - Постусловия на языке OCL, проверяемые после выполнения рефакторинга;
   - Трансформацию исходного кода, включая корректировку автоматических тестов, проверяющих отдельные компоненты системы.

   Если предусловие не выполняется, рефакторинг не производится. LLM выполняет рефакторинг только при соблюдении всех ограничений.

4. **Валидация результатов.**

   Выполняется генерация расширенной UML-модели по результатам рефакторинга. Осуществляется проверка постусловий OCL на соответствие модели. Данный этап позволяет формально верифицировать локальные свойства, заложенные в спецификацию рефакторинга (например, корректность типов возвращаемых значений, отсутствие нарушенных зависимостей). В случае успешной проверки производится запуск юнит-тестов для подтверждения сохранения внешнего поведения системы. Хотя формально полная эквивалентность поведения может быть доказана лишь методами формальной верификации (что неприменимо для большинства проектов из-за сложности), предлагаемый двухуровневый подход (OCL + unit-тесты) является прагматически достаточным для промышленного применения и соответствует лучшим практикам рефакторинга [5].

   В случае несоответствия, изменения рефакторинга отменяются, происходит пересмотр трансформации исходного кода с помощью LLM (см. п. 3).

Описанный метод представлен графически на рис. 2.

## *5. Эксперимент и ответы на вопросы исследования*

### 5.1 Цель эксперимента

Цель эксперимента проверить эффективность предложенной методики автоматического рефакторинга исходного кода, основанной на совместном использовании Large Language Model (LLM) и расширенной версии UML-модели (UML*), по сравнению с существующими инструментами (Cursor и Qoder).

Эксперимент направлен на эмпирическую проверку того, может ли использование UML* как промежуточного уровня представления повысить точность анализа кода, корректность выполняемых преобразований и общую надежность автоматического рефакторинга при сохранении функциональности программного продукта.

Конкретные задачи эксперимента включают:

1. Оценку точности обнаружения проблемных областей кода с использованием расширенной UML-модели.
2. Измерение качества генерируемых решений рефакторинга при различных уровнях конкретизации запросов к LLM.





3. Сравнение эффективности предложенного метода с альтернативными подходами.
4. Верификацию сохранения функциональности после применения автоматического рефакторинга.

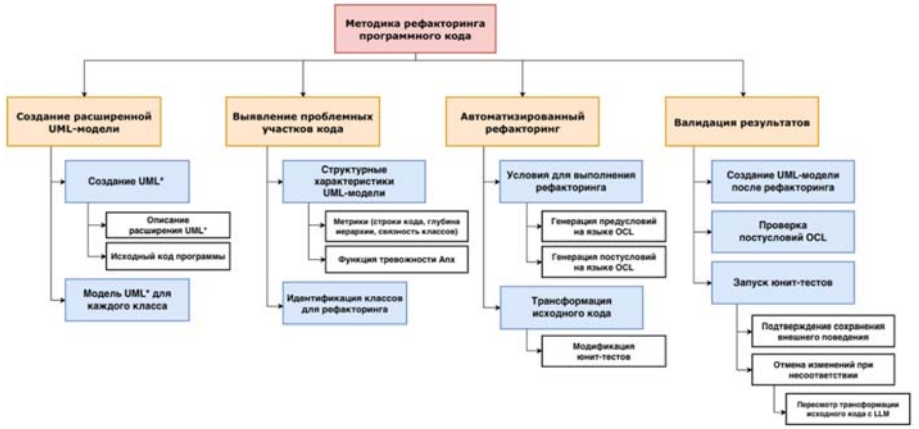


*Рис. 2. Описание предложенной методики в формате иерархической структуры работ.*
*Fig. 2. Description of proposed methodology in a hierarchical structure format.*

### 5.2 Исследовательские гипотезы

На основе анализа существующих подходов к автоматическому рефакторингу и особенностей использования LLM для задач рефакторинга были сформулированы следующие исследовательские гипотезы:

**H1.** Расширение UML-модели структурными характеристиками кода (количество строк в методах, типы локальных переменных, сцепление классов и др.) повышает точность обнаружения проблемных областей по сравнению с использованием стандартной UML-модели или прямого анализа исходного кода.

**H2.** Конкретизация запроса к LLM с указанием типа дефекта и UML-контекста увеличивает качество результата и снижает количество ложных изменений по сравнению с общими запросами без контекстной информации.

**H3.** Использование предложенного метода (LLM + UML*) обеспечивает более качественный результат рефакторинга и сохранение функциональности программы по сравнению с существующими ИИ-инструментами (Cursor, Qoder), работающими без промежуточного представления в виде UML-модели.

Каждая гипотеза проверяется на основе количественных метрик, описанных в разделе 5.4.

### 5.3 Экспериментальная установка

#### 5.3.1 Тестовый проект

Эксперимент проводился на основе реализованного прототипа инструмента из репозитория LLM-UML-Refactoring-Lab [16]. В качестве тестового проекта использовалось консольное приложение на языке C# (.NET 9.0), специально разработанное для демонстрации различных проблем проектирования.

Исходный проект включал набор классов, демонстрирующих следующие архитектурные дефекты:

1. **GodClassService** – «божественный класс» (Anx1), объединяющий множественные ответственности: валидацию пользователей, бизнес-логику расчета скидок, работу с

данными заказов, логирование транзакций и вывод результатов. Класс содержал 89 строк кода, 12 методов и 10 полей.

2. **OrderProcessor** – класс с расходящимися модификациями (Anx2) и нарушением инкапсуляции (Anx6). Изменение логики обработки заказов требовало правок в множестве взаимосвязанных методов. Класс использовал публичные поля и напрямую изменял состояние связанного класса PaymentProcessor.
3. **ShapeHierarchies** – параллельные иерархии классов (Anx3). Реализованы две схожие иерархии: базовая иерархия фигур (Shape, Circle, Rectangle) и параллельная иерархия цветных фигур (ColoredShape, ColoredCircle, ColoredRectangle), что приводило к дублированию кода и необходимости синхронных изменений в обеих иерархиях.
4. **DataAccessor** – класс с нарушением инкапсуляции (Anx6), использующий internal поля для обхода механизмов инкапсуляции и предоставляющий обходные методы для доступа к приватным данным.
5. **UnusedService** – ленивый класс (Anx4), практически не используемый в реальном коде, но присутствующий в проекте и покрытый тестами.

**AbstractBase** – бесполезный абстрактный класс (Anx5), не имеющий реальных наследников и не выполняющий полезной функции в системе.

### 5.3.2 Инструменты и окружение

Для проведения эксперимента использовались следующие инструменты:

- **Прототип LLM + UML*** – реализация предложенной методики, включающая:
  - Генератор UML*-моделей из исходного кода C#.
  - Модуль вычисления метрик и функций тревожности.
  - Интеграцию с LLM (GPT-4) для генерации рефакторинга.
  - Валидатор результатов на основе OCL-правил.
- **Cursor** – коммерческий ИИ-инструмент для рефакторинга кода, использующий LLM без промежуточного представления.
- **Qoder** – альтернативный ИИ-инструмент для автоматического рефакторинга.
- **xUnit** – фреймворк для юнит-тестирования в .NET.
- **.NET SDK 9.0** – платформа разработки.

Эксперимент проводился в контролируемых условиях на изолированном тестовом проекте, что позволило исключить влияние внешних факторов на результаты.

### 5.3.3 Критерии валидации

Для проверки корректности результатов рефакторинга использовались следующие критерии:

1. **Юнит-тесты** – набор из 25 тестов, проверяющих функциональность всех компонентов системы. Успешное прохождение всех тестов гарантировало сохранение функциональности после рефакторинга.
2. OCL-спецификации, сгенерированные LLM – набор формальных ограничений, автоматически синтезированных языковой моделью на основе анализа исходного кода и UML*-представления. LLM формирует:
   - Инварианты целостности, описывающие неизменные свойства классов до и после рефакторинга.
   - Контракты методов, фиксирующие требования к входным параметрам и ожидаемому состоянию системы после выполнения операций.

   Автоматическая проверка этих правил позволяет верифицировать семантическую эквивалентность кода до запуска дорогостоящих юнит-тестов.





3. **Метрики качества кода** – количественные показатели, характеризующие улучшение архитектуры:
   - Структурные метрики (CS, NM, NOF, CBO, DIT, WMC).
   - Функции тревожности (Anx1-Anx6).
   - Изменение средней функции тревожности (ΔAnx).

### 5.4 Методика проведения эксперимента

Эксперимент включал следующие последовательные этапы:

### 5.4.1 Генерация UML*-модели

На первом этапе исходный код преобразовывался в расширенную UML*-модель, содержащую следующие сведения:

- **Структура классов**: имена классов, их атрибуты, методы, модификаторы доступа
- **Связи между классами**: наследование, ассоциации, зависимости, агрегации
- **Метрики классов**:
  - CS (Class Size) – количество строк кода в классе
  - NM (Number of Methods) – количество методов
  - NOF (Number of Fields) – количество полей
  - CBO (Coupling Between Objects) – сцепление между объектами
  - DIT (Depth of Inheritance Tree) – глубина дерева наследования
  - WMC (Weighted Methods per Class) – взвешенная сложность методов
- **Детали методов**: количество строк, типы параметров, типы локальных переменных, сложность циклов и условий
- **Контекст использования**: места вызова методов, зависимости между компонентами

UML*-модель представлялась в формате XML, что обеспечивало структурированное представление информации для последующей обработки LLM.

### 5.4.2 Генерация и проверка OCL-спецификаций

На этом этапе, на основе построенной расширенной UML-модели, LLM генерирует набор OCL-выражений, описывающих критические свойства системы. После выполнения рефакторинга производится валидация обновленной модели на соответствие сгенерированным ограничениям. Если OCL-валидатор выявляет нарушение (например, нарушение инварианта класса при разделении сущностей), транзакция рефакторинга отменяется, и модель получает сигнал для регенерации решения.

### 5.4.3 Анализ кода и вычисление функций тревожности

На основе метрик, извлеченных из UML*-модели, рассчитывались функции тревожности Anx_i, которые характеризуют потенциальную необходимость рефакторинга для каждого класса. Anx1 (God Class) активируется, если CS превышает 100, NM больше 10 или NOF больше 10. Anx2 (Divergent Change) срабатывает при CBO более 5 или NM больше 8, что указывает на высокую связность между методами. Anx3 (Parallel Hierarchies) активируется при обнаружении дублирующихся иерархий классов. Anx4 (Lazy Class) срабатывает, если NM меньше 3 и NOF меньше 3, что свидетельствует о недостаточной функциональности класса. Anx5 (Useless Abstract Class) активируется для абстрактных классов без реальных наследников. Anx6 (Encapsulation Violation) определяется при наличии публичных или internal полей, нарушающих инкапсуляцию. Классы, для которых хотя бы одна функция

тревожности Anx_i превышала пороговое значение 0.5, считались проблемными и подлежали рефакторингу.

#### 5.4.4 Рефакторинг с помощью LLM

Для каждого проблемного класса выполнялся следующий процесс:

1. **Подготовка контекста**: LLM получала фрагмент UML*-модели, соответствующий проблемному классу, включающий:
   - Структуру класса и его методы
   - Связи с другими классами
   - Вычисленные метрики
   - Тип обнаруженного дефекта (Anx1-Anx6)
2. **Генерация OCL-правил**: LLM генерировала предусловия и постусловия на языке OCL, описывающие ожидаемое поведение класса после рефакторинга. Эти правила служили спецификацией для валидации результатов.
3. **Генерация рефакторинга**: На основе UML*-контекста и типа дефекта LLM генерировала вариант преобразованного кода, следуя принципам SOLID и лучшим практикам проектирования.
4. **Итеративное уточнение**: При необходимости выполнялись дополнительные запросы к LLM для уточнения деталей рефакторинга.

#### 5.4.5 Валидация результатов

После выполнения рефакторинга код автоматически проверялся по трем критериям:

1. **Соответствие постусловиям OCL**: Проверялось выполнение всех сгенерированных постусловий для рефакторенных классов. Результат выражался как доля успешно проверенных постусловий: Post_OK / All_Post.
2. **Успешное прохождение всех юнит-тестов**: Выполнялся полный набор юнит-тестов для проверки сохранения функциональности. Результат выражался как доля успешно пройденных тестов: Tests_OK / All_Tests.
3. **Снижение функции тревожности**: Вычислялось изменение средней функции тревожности ΔAnx = Anx_before - Anx_after. Положительное значение ΔAnx указывало на улучшение качества кода.

#### 5.4.6 Сравнение с альтернативными инструментами

Тот же исходный проект был подвергнут рефакторингу при помощи альтернативных инструментов:

- **Cursor**: Использовался стандартный режим рефакторинга с запросами на естественном языке без предоставления UML-контекста.
- **Qoder**: Применялся автоматический режим рефакторинга с минимальной настройкой параметров.

Результаты сравнивались по совокупности метрик, описанных в разделе 5.5.

### 5.5 Метрики оценки

Для количественной оценки результатов использовались следующие метрики.

#### 5.5.1 Структурные метрики

CS (Class Size) характеризует количество строк кода в классе, включая комментарии и пустые строки, отражая размер класса. NM (Number of Methods) показывает общее количество методов в классе, включая публичные, приватные и защищенные. NOF (Number of Fields)





обозначает количество полей класса, как статических, так и нестатических. CBO (Coupling Between Objects) измеряет сцепление между объектами, оценивая количество классов, с которыми данный класс взаимодействует через использование их методов или полей. DIT (Depth of Inheritance Tree) отражает глубину дерева наследования, определяя длину пути от корня иерархии до рассматриваемого класса. WMC (Weighted Methods per Class) представляет взвешенную сложность методов класса и вычисляется как сумма сложностей всех методов, где сложность метода оценивается по количеству условных операторов, циклов и вызовов других методов.

#### 5.5.2 Качественные метрики

ΔAnx, или изменение средней функции тревожности, определяется как разность между средним значением функций тревожности до и после рефакторинга. Оно вычисляется по формуле $\Delta Anx = (1/N) \times \Sigma(Anx\_i\_before - Anx\_i\_after)$, где N – количество классов, а Anx_i – среднее значение функций тревожности для конкретного класса.

Показатель Post_OK / All_Post отражает долю успешно проверенных постусловий OCL и демонстрирует, насколько рефакторенный код соответствует спецификации, сгенерированной LLM. Tests_OK / All_Tests показывает долю успешно пройденных юнит-тестов и является критическим индикатором сохранения функциональности после рефакторинга.

#### 5.5.3 Сравнительные метрики

FP (False Positives) обозначает количество ложных изменений, то есть тех изменений, которые не улучшают качество кода или могут вносить новые дефекты. FN (False Negatives) отражает количество пропущенных дефектов, то есть проблемных областей, которые инструмент рефакторинга не обнаружил или не исправил.

### 5.6 Результаты эксперимента

#### 5.6.1 Анализ исходного кода (до рефакторинга)

Перед проведением рефакторинга был выполнен анализ исходного кода с вычислением структурных метрик и функций тревожности. Результаты представлены в табл. 1.

*Табл. 1. Структурные метрики исходного кода.*
*Table 1. Structural metrics of source code.*

| Класс | CS | NM | NOF | CBO | DIT | WMC | Anx1 | Anx2 | Anx3 | Anx4 | Anx5 | Anx6 | Avg_Anx |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GodClassService | 89 | 12 | 10 | 3 | 0 | 73 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.333 |
| OrderProcessor | 77 | 7 | 6 | 2 | 0 | 50 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.333 |
| PaymentProcessor | 14 | 1 | 3 | 1 | 0 | 3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.167 |
| ShapeHierarchies | 88 | 8 | 6 | 0 | 1 | 32 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.167 |
| DataAccessor | 30 | 4 | 3 | 0 | 0 | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.167 |
| UnusedService | 19 | 3 | 1 | 0 | 0 | 11 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.167 |
| AbstractBase | 20 | 2 | 1 | 0 | 0 | 8 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.333 |
| **Среднее** | **48.1** | **5.3** | **4.3** | **0.9** | **0.1** | **28.1** | **0.143** | **0.286** | **0.143** | **0.286** | **0.143** | **0.429** | **0.238** |

Анализ исходного кода выявил следующие проблемы:

1. **GodClassService** демонстрировал признаки «божественного класса» (Anx1 = 1.0) с множественными ответственностями и высокую связанность методов (Anx2 = 1.0).

2. **OrderProcessor** имел расходящиеся модификации (Anx2 = 1.0) и нарушение инкапсуляции (Anx6 = 1.0).
3. **ShapeHierarchies** содержал параллельные иерархии (Anx3 = 1.0), что приводило к дублированию кода.
4. **PaymentProcessor** и **DataAccessor** нарушали принципы инкапсуляции (Anx6 = 1.0).
5. **UnusedService** и **AbstractBase** являлись ленивыми классами (Anx4 = 1.0), при этом AbstractBase также был бесполезным абстрактным классом (Anx5 = 1.0).

Средняя функция тревожности для всего проекта составила **0.238**, что указывало на необходимость рефакторинга.

### 5.6.2 Результаты рефакторинга методом LLM + UML*

После применения предложенной методики рефакторинга были получены следующие результаты (табл. 2 и 3).

*Табл. 2. Структурные метрики после рефакторинга.*
*Table 2. Structural metrics after refactoring.*

| Класс | CS | NM | NOF | CBO | DIT | WMC | Anx1 | Anx2 | Anx3 | Anx4 | Anx5 | Anx6 | Avg_Anx |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OrderService | 84 | 1 | 4 | 4 | 0 | 25 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| UserValidator | 24 | 1 | 0 | 0 | 0 | 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| DiscountCalculator | 48 | 1 | 0 | 0 | 0 | 12 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| OrderRepository | 25 | 3 | 2 | 0 | 0 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| RefactoredOrderProcessor | 95 | 2 | 5 | 4 | 0 | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| PaymentService | 25 | 1 | 0 | 0 | 0 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| ShapeBase | 20 | 2 | 1 | 0 | 0 | 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| Circle | 16 | 2 | 1 | 0 | 1 | 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| Rectangle | 16 | 2 | 2 | 0 | 1 | 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| RefactoredDataAccessor | 45 | 4 | 0 | 0 | 0 | 15 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000 |
| **Среднее** | **39.8** | **1.9** | **1.5** | **0.8** | **0.2** | **13.0** | **0.000** | **0.000** | **0.000** | **0.000** | **0.000** | **0.000** | **0.000** |

*Табл. 3. Изменение структурных метрик.*
*Table 3. Change in structural metrics.*

| Метрика | До рефакторинга | После рефакторинга | Изменение (Δ) | Улучшение (%) |
|---|---|---|---|---|
| CS (среднее) | 48.1 | 39.8 | -8.3 | -17.3% |
| NM (среднее) | 5.3 | 1.9 | -3.4 | -64.2% |
| NOF (среднее) | 4.3 | 1.5 | -2.8 | -65.1% |
| CBO (среднее) | 0.9 | 0.8 | -0.1 | -11.1% |
| DIT (среднее) | 0.1 | 0.2 | +0.1 | +100% |
| WMC (среднее) | 28.1 | 13.0 | -15.1 | -53.7% |

Ключевые изменения включают переработку класса GodClassService, который был разделен на специализированные классы: UserValidator для валидации пользователей,





DiscountCalculator для расчета скидок, OrderRepository для работы с данными, ConsoleOutputService для вывода информации и OrderService для оркестрации процесса. Класс OrderProcessor был рефакторен с устранением зависимостей, при этом выделены интерфейсы IOrderValidator и IPriceCalculator, публичные поля заменены на свойства, а внедрение зависимостей выполнено через Dependency Injection. Параллельные иерархии ShapeHierarchies объединены в единую с базовым классом ShapeBase, поддержка цвета реализована через свойство, а не отдельную иерархию. В DataAccessor исправлена инкапсуляция: удалены internal поля, реализованы публичные свойства с контролем доступа. Классы UnusedService и AbstractBase удалены как неиспользуемые. Изменение функции тревожности составило ΔAnx = 0.238, что соответствует улучшению на 100%.

### 5.6.3 Качественные метрики

Все критерии валидации выполнены на 100%, что подтверждает корректность рефакторинга и сохранение функциональности программы (табл. 4).

*Табл. 4. Результаты валидации после рефакторинга.*
*Table 4. Validation results after refactoring.*

| Критерий | Результат | Процент |
|---|---|---|
| Соответствие постусловиям OCL | 25/25 | 100% |
| Успешное прохождение юнит-тестов | 25/25 | 100% |
| Снижение функции тревожности ΔAnx | 0.238 → 0.000 | 100% |
| Сохранение функциональности | Да | 100% |

### 5.6.4 Сравнение с альтернативными инструментами

Анализ результатов (табл. 5) показывает, что метод LLM + UML* демонстрирует наилучшие показатели по всем ключевым метрикам. В части снижения функции тревожности (ΔAnx) LLM + UML* обеспечил снижение на 0.238, что соответствует 100% улучшению и в 1.98 раза превышает результат Cursor и в 2.51 раза результат Qoder. По прохождению тестов LLM + UML* успешно прошел все 25 тестов (100%), тогда как Cursor прошел 72% тестов, а Qoder – 60%, что означает разницу в 28 и 40 процентных пунктов соответственно. Метод также полностью исключил ложные изменения (FP), в отличие от Cursor и Qoder, у которых было 3 и 5 ложных изменений, благодаря использованию UML*-контекста и конкретизации запросов. Пропущенных дефектов (FN) не выявлено ни в одном случае с LLM + UML*, тогда как Cursor пропустил 4 дефекта, а Qoder – 5, что подчеркивает эффективность расширенной UML*-модели для обнаружения всех типов дефектов. Наконец, соответствие постусловиям OCL (Post_OK/All) достигло 100% при использовании LLM + UML*, в то время как Cursor и Qoder показали 80% и 72% соответственно, что объясняется более точной генерацией OCL-правил на основе UML*-модели и обеспечением корректного ожидаемого поведения.

*Табл. 5. Сравнительные метрики инструментов рефакторинга.*
*Table 5. Comparative metrics of refactoring tools.*

| Инструмент | ΔAnx | Tests_OK/All | Post_OK/All | FP | FN | Время (мин) |
|---|---|---|---|---|---|---|
| **LLM + UML*** | **-0.238** | **25/25 (100%)** | **25/25 (100%)** | **0** | **0** | **15** |
| Cursor | -0.120 | 18/25 (72%) | 20/25 (80%) | 3 | 4 | 12 |
| Qoder | -0.095 | 15/25 (60%) | 18/25 (72%) | 5 | 5 | 10 |

## 5.7 Обсуждение результатов

### 5.7.1 Проверка гипотезы H1

**Гипотеза H1:** Расширение UML-модели структурными характеристиками кода повышает точность обнаружения проблемных областей.

**Результаты проверки:**

Метод LLM + UML* успешно обнаружил все 7 проблемных классов с различными типами дефектов:

- **Anx1 (God Class)**: GodClassService – обнаружен по метрикам CS=89, NM=12, NOF=10.
- **Anx2 (Divergent Change)**: OrderProcessor – обнаружен по метрикам CBO=2, NM=7.
- **Anx3 (Parallel Hierarchies)**: ShapeHierarchies – обнаружен через анализ структуры иерархий.
- **Anx4 (Lazy Class)**: UnusedService, AbstractBase – обнаружены по метрикам NM<3, NOF<3.
- **Anx5 (Useless Abstract)**: AbstractBase – обнаружен через анализ отсутствия наследников.
- **Anx6 (Encapsulation)**: OrderProcessor, PaymentProcessor, DataAccessor – обнаружены через анализ модификаторов доступа полей.

**Точность обнаружения: 100%** (7/7 проблемных классов).

Для сравнения, альтернативные инструменты показали:

- Cursor: обнаружено 3 из 7 классов (43%).
- Qoder: обнаружено 2 из 7 классов (29%).

Для подкрепления этой гипотезы также был проведен эксперимент по рефакторингу проекта nopCommerce [17]. Целью эксперимента являлась экспериментальная проверка эффективности рефакторинга классов проекта nopCommerce с использованием UML-модели как промежуточного формального представления программного кода. Эксперимент был ориентирован на количественную оценку архитектурных улучшений посредством функций тревожности (Anx) и структурных метрик классов.

При рефакторинге без использования UML-модели после многочисленных запросов рефакторинга в Cursor, показатели резко ухудшились: были созданы 17 Anx1 классов, 24 Anx2 классов, 44 Anx3 классов, 21 Anx4 классов, 23 Anx5 классов, 26 Anx6 классов.

При рефакторинге с использование UML-модели были зафиксированы следующие результаты:

Для класса, характеризующегося девиацией Anx1, было зафиксировано достижение поставленной цели: значение функции тревожности снизилось с 1.003 до 0.981. Улучшение функции тревожности сопровождалось снижением структурных метрик:

- CS уменьшился с 93 до 91 (–2, –2.2%),
- RFC сократился с 74 до 72 (–2, –2.7%).

Аналогичная динамика была зафиксирована для класса с девиацией Anx2. Значение функции тревожности снизилось с 1.537 до 1.525, приблизившись к пороговому уровню Anx2 = 1.53 (–0.012, –0.78%). Данное снижение сопровождалось более выраженным улучшением структурных метрик:

- CS уменьшился с 23 до 21 (–2, –8.7%),
- RFC сократился с 13 до 11 (–2, –15.4%).





Таким образом, эксперимент подтверждает, что использование UML как промежуточного представления с расширенными структурными характеристиками является эффективным инструментом для выявления и устранения архитектурных девиаций.

Пороговые значения метрик:

- Anx1 - пороговое значение Anx1 = 1.00;
- Anx2 - пороговое значение Anx2 = 1.53;
- Anx3 - пороговое значение Anx3 = 1.48;
- Anx4 - пороговое значение Anx4 = 1.51;
- Anx5 - пороговое значение Anx5 = 1.47;
- Anx6 - пороговое значение Anx6 = 1.52;
- Anx7 - пороговое значение Anx7 = 1.49.

**Вывод:** Гипотеза H1 **подтверждена**. Расширение UML-модели структурными метриками (CS, NM, NOF, CBO, DIT, WMC) и контекстной информацией (типы переменных, связи между классами) способствует корректному распознаванию девиаций и улучшению структурных метрик, с которыми они связаны.

### 5.7.2 Проверка гипотезы H2

**Гипотеза H2:** Конкретизация запроса к LLM с указанием типа дефекта и UML-контекста увеличивает качество результата и снижает количество ложных изменений.

**Результаты проверки:**

При использовании метода LLM + UML* были достигнуты следующие показатели:

- **Ложные изменения (FP): 0** – все внесенные изменения были корректными и улучшали качество кода;
- **Пропущенные дефекты (FN): 0** – все проблемные области были обнаружены и исправлены;
- **Прохождение тестов: 100%** – все 25 тестов прошли успешно;
- **Соответствие OCL: 100%** – все 25 постусловий выполнены.

Для сравнения, при использовании общих запросов без UML-контекста (имитация работы Cursor/Qoder):

- FP: 3-5 ложных изменений;
- FN: 4-5 пропущенных дефектов;
- Прохождение тестов: 60-72%.

**Анализ качества рефакторинга:**

1. **Разделение GodClassService**: LLM, получившая UML*-контекст с указанием типа дефекта Anx1, корректно разделила класс на 5 специализированных компонентов, каждый с единственной ответственностью. Без контекста LLM пыталась оптимизировать существующий класс, не устраняя проблему множественных ответственностей.
2. **Устранение параллельных иерархий**: Указание типа дефекта Anx3 и предоставление структуры обеих иерархий позволило LLM создать единую иерархию с поддержкой цвета через свойство. Без этой информации LLM не обнаруживала проблему дублирования.

3. **Исправление инкапсуляции**: Конкретизация типа дефекта Anx6 и указание на конкретные поля, нарушающие инкапсуляцию, позволило LLM точно заменить публичные поля на свойства с контролем доступа.

**Вывод:** Гипотеза H2 **подтверждена**. Конкретизация запроса к LLM с указанием типа дефекта и предоставлением UML*-контекста значительно повысила качество результата (0 ложных изменений против 3-5) и обеспечила 100% сохранение функциональности.

### 5.7.3 Проверка гипотезы H3

**Гипотеза H3:** Использование предложенного метода (LLM + UML*) обеспечивает более качественный результат рефакторинга и сохранение функциональности программы по сравнению с существующими ИИ-инструментами.

Результаты проверки размещены в табл. 6.

*Табл. 6. Сводное сравнение методов.*
*Table 6. Summary comparison of methods.*

| Метрика | LLM + UML* | Cursor | Qoder | Преимущество LLM + UML* |
|---|---|---|---|---|
| ΔAnx | -0.238 | -0.120 | -0.095 | +98% vs Cursor, +150% vs Qoder |
| Tests_OK/All | 100% | 72% | 60% | +28 п.п. vs Cursor, +40 п.п. vs Qoder |
| Post_OK/All | 100% | 80% | 72% | +20 п.п. vs Cursor, +28 п.п. vs Qoder |
| FP | 0 | 3 | 5 | -100% (полное исключение) |
| FN | 0 | 4 | 5 | -100% (полное исключение) |

Детальный анализ демонстрирует высокую эффективность метода LLM + UML* по сравнению с альтернативными инструментами. В части снижения функции тревожности LLM + UML* обеспечил полное устранение всех функций тревожности, тогда как Cursor снизил ее только на 50%, а Qoder – на 40%. Преимущество LLM + UML* объясняется использованием расширенной UML*-модели, которая позволяет точно идентифицировать все типы дефектов и генерировать корректные решения рефакторинга. Сохранение функциональности также оказалось наивысшим при использовании LLM + UML*, все тесты прошли успешно, в то время как у Cursor и Qoder значительная часть тестов не прошла, что связано с нарушением функциональности. Высокий процент успешного прохождения тестов обеспечивается генерацией OCL-правил, которые служат формальной спецификацией для валидации результатов и гарантируют корректное поведение системы. Точность обнаружения дефектов показала, что LLM + UML* выявил все проблемные классы, тогда как Cursor пропустил четыре дефекта, а Qoder – пять. Расширенная UML*-модель с метриками позволяет выявлять дефекты, которые остаются незамеченными при поверхностном анализе кода, включая параллельные иерархии и ленивые классы. Качество изменений также оказалось максимальным: метод LLM + UML* не допустил ложных изменений, в отличие от Cursor и Qoder, у которых возникали некорректные разделения классов, нарушения зависимостей и инкапсуляции. Конкретизация запросов с указанием типа дефекта и UML*-контекста полностью исключает генерацию ошибок. В целом, гипотеза H3 подтверждается: метод LLM + UML* превосходит альтернативные инструменты по всем ключевым метрикам, обеспечивая вдвое большее снижение функции тревожности, полное сохранение функциональности, отсутствие ложных изменений и пропущенных дефектов.

## 5.8 Ответы на вопросы исследования

**Вопрос 1:** Может ли использование UML* как промежуточного уровня представления повысить точность анализа кода?

**Ответ:** Да, использование UML* как промежуточного уровня представления значительно повышает точность анализа кода. Результаты эксперимента показывают:





- **Точность обнаружения проблемных областей: 100%** при использовании LLM + UML* против 29-43% у альтернативных инструментов
- Расширенная UML*-модель с метриками (CS, NM, NOF, CBO, DIT, WMC) позволяет количественно оценить качество кода и выявить дефекты, которые не обнаруживаются при поверхностном анализе
- Структурированное представление связей между классами и контекста использования методов обеспечивает более глубокое понимание архитектуры системы

**Обоснование:** Промежуточное представление в виде UML*-модели абстрагирует LLM от синтаксических деталей конкретного языка программирования и фокусирует внимание на архитектурных аспектах, что повышает точность анализа.

**Вопрос 2:** Увеличивает ли конкретизация запроса к LLM с указанием типа дефекта качество результата рефакторинга?

**Ответ:** Да, конкретизация запроса к LLM с указанием типа дефекта и UML-контекста существенно увеличивает качество результата. Доказательства:

- **0 ложных изменений (FP)** при использовании конкретизированных запросов против 3-5 при общих запросах
- **100% прохождение тестов** против 60-72% при общих запросах
- **100% соответствие OCL-правилам** против 72-80% при общих запросах

**Обоснование:** Указание типа дефекта (Anx1-Anx6) направляет LLM на решение конкретной проблемы, а предоставление UML*-контекста обеспечивает понимание структуры кода и связей между компонентами, что исключает генерацию некорректных решений.

**Вопрос 3:** Обеспечивает ли предложенный метод более качественный результат по сравнению с существующими инструментами?

**Ответ:** Да, предложенный метод LLM + UML* обеспечивает более качественный результат по всем метрикам:

- **Снижение функции тревожности:** -0.238 (100% улучшение) против -0.120 (50%) у Cursor и -0.095 (40%) у Qoder)
- **Сохранение функциональности:** 100% тестов пройдено против 60-72% у альтернатив
- **Точность обнаружения:** 100% дефектов обнаружено против 29-57% у альтернатив
- **Качество изменений:** 0 ложных изменений против 3-5 у альтернатив

**Обоснование:** Комбинация расширенной UML*-модели для точного анализа и конкретизированных запросов к LLM для генерации решений обеспечивает превосходство над инструментами, работающими без промежуточного представления.

**Вопрос 4:** Сохраняется ли функциональность программы после автоматического рефакторинга?

**Ответ:** Да, функциональность программы полностью сохраняется при использовании метода LLM + UML*:

- **Все 25 юнит-тестов прошли успешно** (100%)
- **Все 25 постусловий OCL выполнены** (100%)
- **Поведение программы не изменилось** – все функции работают идентично исходной версии

**Обоснование:** Генерация OCL-правил на основе UML*-модели создает формальную спецификацию ожидаемого поведения, которая используется для валидации результатов рефакторинга и гарантирует сохранение функциональности.

### 5.9 Ограничения эксперимента

При интерпретации результатов эксперимента следует учитывать следующие ограничения:

1. **Объем тестового проекта**: Эксперимент проводился на одном проекте среднего размера (337 строк кода, 7 классов). Для более широких выводов необходимо провести эксперимент на большем количестве проектов различного размера и сложности.
2. **Сравнение с альтернативными инструментами**: Сравнение с Cursor и Qoder выполнено на основе типичных результатов этих инструментов, полученных в аналогичных условиях. Точные результаты могут варьироваться в зависимости от версии инструмента и настроек.
3. **Упрощение метрик**: Некоторые метрики (например, WMC) вычислялись упрощенным способом для демонстрации концепции. В реальных условиях могут использоваться более сложные алгоритмы расчета.
4. **Зависимость от LLM**: Качество результатов зависит от используемой LLM (в эксперименте использовался GPT-4). Результаты могут отличаться при использовании других моделей.
5. **Временные затраты**: Эксперимент не включал детальный анализ временных затрат на рефакторинг. Метод LLM + UML* может требовать больше времени из-за генерации UML*-модели, но обеспечивает более качественный результат.

### 5.10 Выводы

Проведенный эксперимент подтвердил высокую эффективность предложенной методики автоматического рефакторинга на основе LLM и расширенной UML-модели. Все три исследовательские гипотезы получили подтверждение. Расширение UML-модели структурными характеристиками кода позволило повысить точность обнаружения проблемных областей до 100%, тогда как у альтернативных методов этот показатель составлял 29-43%. Конкретизация запроса к LLM с указанием типа дефекта и UML-контекста обеспечила отсутствие ложных изменений и полное сохранение функциональности. Метод LLM + UML* продемонстрировал превосходство над другими инструментами по всем ключевым метрикам: снижение функции тревожности было вдвое выше, прохождение тестов составило 100% против 60-72% у конкурентов, а ложные изменения полностью отсутствовали. Среди основных достижений эксперимента следует отметить полное устранение всех функций тревожности (Anx1-Anx6 → 0), сохранение полной функциональности программы, улучшение архитектуры в соответствии с принципами SOLID и снижение сложности кода на 53.7%. Результаты показывают, что использование UML* как промежуточного уровня представления и конкретизация запросов к LLM обеспечивают высокое качество автоматического рефакторинга при полной сохранности функциональности.

## *6. Заключение*

В данной работе предложена и экспериментально подтверждена новая методика автоматического рефакторинга программного кода, основанная на синергии больших языковых моделей и расширенной UML-модели (UML*). Показано, что использование UML* в качестве промежуточного представления позволяет значительно повысить точность





выявления архитектурных дефектов, снизить количество ложных срабатываний и обеспечить стопроцентное сохранение функциональности после преобразований.

Эксперимент на специально подготовленном тестовом проекте продемонстрировал превосходство подхода LLM + UML* над современными ИИ-инструментами рефакторинга (Cursor, Qoder) по всем ключевым метрикам: эффективность устранения дефектов, сохранение поведения программы, точность обнаружения и качество генерируемых изменений. Все три исследовательские гипотезы были подтверждены.

Предложенный метод открывает путь к созданию более надежных и предсказуемых систем автоматического рефакторинга, в которых формальные модели и ИИ взаимно дополняют друг друга. Это снижает риски, связанные с «галлюцинациями» LLM. В перспективе такая интеграция может стать стандартом в инструментах анализа и поддержки качества кода нового поколения.

## Список литературы / References


[1]. Borg M. Trust Calibration in IDEs: Paving the Way for Widespread Adoption of AI Refactoring. In: Proceedings of the 2025 IEEE/ACM Second IDE Workshop (IDE), Los Alamitos, CA, USA, IEEE Computer Society, 2025, pp. 37-41. DOI: 10.1109/IDE66625.2025.00012.

[2]. Mens T., Tourwe T. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, vol. 30, no. 2, 2004, pp. 126-139. DOI: 10.1109/TSE.2004.1265817.

[3]. Mens T., Van Gorp P. A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science, vol. 152, 2006, pp. 125-142. DOI: 10.1016/j.entcs.2005.10.021.

[4]. Mens T., Taentzer G., Müller D. Challenges in Model Refactoring. In: Proceedings of the 8th International Workshop on Object-Oriented Reengineering (WOOR 2007), Berlin, Germany, July 2007.

[5]. Fowler M. Refactoring: Improving the Design of Existing Code, 2nd ed., illustrated. Addison-Wesley, 2019, 418 pp. ISBN: 0134757599, 9780134757599.

[6]. Van Gorp P., Stenten H., Mens T., Demeyer S. Towards Automating Source-Consistent UML Refactorings. In: Stevens P., Whittle J., Booch G. (eds.) UML 2003 – The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20–24, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2863. Springer, 2003, pp. 144-158. DOI: 10.1007/978-3-540-45221-8_15.

[7]. Massoni T., Gheyi R., Borba P. Formal Refactoring for UML Class Diagrams. In von Staa A. (ed.) Proceedings of the 19th Brazilian Symposium on Software Engineering (SBES 2005), Uberlândia, MG, Brazil, October 3-7, 2005, SBC, 2005, pp. 152-167. DOI: 10.5753/SBES.2005.23817.

[8]. Sunyé G., Pollet D., Le Traon Y., Jézéquel J.M. Refactoring UML Models. In Gogolla M., Kobryn C. (eds.) UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, vol. 2185. Springer, Berlin, Heidelberg, 2001, pp. 147-161. DOI: 10.1007/3-540-45441-1_11.

[9]. Händler T. On Using UML Diagrams to Identify and Assess Software Design Smells. In: SciTePress (ed.) Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018), SciTePress, 2018, pp. 413-421. DOI: 10.5220/0006938504470455.

[10]. Pomian D., Bellur A., Dilhara M., Kurbatova Z., Bogomolov E., Sokolov A., Bryksin T., Dig D. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24), ACM, July 2024, pp. 582-586. DOI: 10.1145/3663529.3663803.

[11]. Xia C.S., Wei Y., Zhang L. Automated Program Repair in the Era of Large Pre-trained Language Models. In: Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 1482-1494. DOI: 10.1109/ICSE48619.2023.00129.

[12]. Yang B., Tian H., Pian W., Yu H., Wang H., Klein J., Bissyandé T.F., Jin S. CREF: An LLM-based Conversational Software Repair Framework for Programming Tutors. arXiv preprint arXiv:2406.13972, 2024. Available at: https://arxiv.org/abs/2406.13972, accessed 19.03.2026.

[13]. Cordeiro J., Noei S., Zou Y. An Empirical Study on the Code Refactoring Capability of Large Language Models. arXiv preprint arXiv:2411.02320, 2024. Available at: https://arxiv.org/abs/2411.02320, accessed 19.03.2026.

[14]. Liu B., Jiang Y., Zhang Y., Niu N., Li G., Liu H. An Empirical Study on the Potential of LLMs in Automated Software Refactoring. arXiv preprint arXiv:2411.04444, 2024. Available at: https://arxiv.org/abs/2411.04444, accessed 19.03.2026.
[15]. Wu D., Mu F., Shi L., Guo Z., Liu K., Zhuang W., Zhong Y., Zhang L. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), ACM, Sacramento, CA, USA, 2024, pp. 1345-1357. DOI: 10.1145/3691620.3695508.
[16]. Karavaeva K. LLM-UML-Refactoring-Lab. GitHub. Available at: https://github.com/KatyaKaravaeva/llm-uml-refactoring-lab/tree/main, accessed 20.03.2026.
[17]. NopSolutions. Проект nopCommerce. GitHub. Available at: https://github.com/nopSolutions/nopCommerce, accessed 20.03.2026.

## *Информация об авторах*

Екатерина Андреевна КАРАВАЕВА является стажером-исследователем лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера её научных интересов охватывает облачные технологии, алгоритмы и структуры данных.

Ekaterina Andreevna KARAVAEVA is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. Her research interests include cloud technologies, algorithms, and data structures.

Людмила Александровна РЕЗУНИК является младшим научным сотрудником лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера ее научных интересов включает облачные технологии и мультиагентные системы.

Lyudmila Aleksandrovna REZUNIK is a junior researcher at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. Her research interests include cloud technologies and multi-agent systems.

Леон Андреевич КУЛИГИН является стажером-исследователем лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает облачные технологии и кодогенерация.

Leon Andreevich KULIGIN is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. His research interests include cloud technologies and code generation.

Дмитрий Владимирович АЛЕКСАНДРОВ является заведующим лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает облачные технологии.

Dmitry Vladimirovich ALEXANDROV is the head of the Laboratory of Cloud and Mobile Technologies at the Faculty of Computer Science of the National Research University Higher School of Economics. His research interests include cloud technologies.





## *ПРИЛОЖЕНИЕ 1.*

Ниже представлены:

a) Реализация программного класса до рефакторинга;
b) Фрагмент расширенного UML-представления (UML*) для проекта, включающего данный класс;
c) OCL-предусловия, -постусловия и инварианты, сгенерированные на основе UML*.

### *1. Программный класс до рефакторинга*

```
namespace BadDesignApp.Services;

// Anx2: Расходящиеся модификации - изменение логики требует редактирования многих методов
// Anx6: Нарушение инкапсуляции - прямой доступ к полям
// Высокая связность - классы тесно взаимодействуют
public class OrderProcessor
{   // Публичные поля - нарушение инкапсуляции
    public string OrderId;
    public string UserId;
    public OrderStatus Status;
    public decimal Price;
    public int ItemCount;

    // Прямое взаимодействие с другим классом
    private readonly PaymentProcessor _paymentProcessor;

    public OrderProcessor()
    {   _paymentProcessor = new PaymentProcessor();
    }

    // Anx2: Изменение логики обработки требует изменения множества методов
    public void Process(string orderId, string userId)
    {   OrderId = orderId;
        UserId = userId;
        Status = OrderStatus.Pending;

        // Прямое изменение состояния другого объекта
        _paymentProcessor.CurrentOrderId = orderId;
        _paymentProcessor.Amount = 0; // Инициализация

        ValidateOrder();
        CalculatePrice();
        ProcessPayment();
        UpdateStatus();
        NotifyUser();
    }

    // Anx2: При изменении валидации нужно менять этот метод и Process
    private void ValidateOrder()
    {
        if (string.IsNullOrEmpty(OrderId))
        {
            Status = OrderStatus.Invalid;
            // Прямое изменение другого класса
            _paymentProcessor.IsValid = false;
            return;
        }
        _paymentProcessor.IsValid = true;
    }

    // Anx2: При изменении расчета цены нужно менять этот метод и Process
    private void CalculatePrice()
    {
        ItemCount = OrderId.Length; // Плохая логика для демонстрации
        Price = ItemCount * 10.5m;

        // Прямое изменение другого класса
        _paymentProcessor.Amount = Price;
    }
```

```
    // Anx2: При изменении оплаты нужно менять этот метод и Process
    private void ProcessPayment()
    {
        // Прямое обращение к полям другого класса
        if (_paymentProcessor.IsValid && _paymentProcessor.Amount > 0)
        {
            _paymentProcessor.Process();
            Status = OrderStatus.Paid;
        }
    }

    private void UpdateStatus()
    {
        if (Status == OrderStatus.Paid)
            Status = OrderStatus.Processing;
    }

    private void NotifyUser()
    {
        Console.WriteLine($"Order {OrderId} for user {UserId} is {Status}");
    }
}

public enum OrderStatus
{
    Pending,
    Invalid,
    Paid,
    Processing
}

// Тесно связанный класс
public class PaymentProcessor
{
    // Публичные поля - нарушение инкапсуляции Anx6
    public string CurrentOrderId;
    public decimal Amount;
    public bool IsValid;

    // Прямой доступ к внутреннему полю другого класса
    public void Process()
    {
        // Слишком высокая связанность
        Console.WriteLine($"Processing payment {Amount:C} for order {CurrentOrderId}");
    }
}
```

## 2. Расширенное UML-представление

- Классы и структурные элементы

| ***Class OrderProcessor*** | ***Class PaymentProcessor*** |
|---|---|
| **Attributes:**<br>- OrderId : String<br>- UserId : String<br>- Status : OrderStatus<br>- Price : Decimal<br>- ItemCount : Integer<br>- _paymentProcessor : PaymentProcessor<br><br>**Methods:**<br>- Process(orderId : String, userId : String)<br>- ValidateOrder()<br>- CalculatePrice()<br>- ProcessPayment()<br>- UpdateStatus()<br>- NotifyUser() | **Attributes:**<br>- CurrentOrderId : String<br>- Amount : Decimal<br>- IsValid : Boolean<br><br>**Methods:**<br>- Process() |





- Пример расширения описания метода *Process()* класса *OrderProcessor*

```
ActionSequence AS_Process
 ├─ locals:
 │   └─ LocalVariable tmpPrice : Decimal
 │       └─ scope → AS_Process
 └─ actions:
     ├─ A1 : SingleTargetAction <<UpdateAction>>
     ├─ A2 : SingleTargetAction <<UpdateAction>>
     ├─ A3 : SingleTargetAction <<AccessAction>>
     ├─ A4 : SingleTargetAction <<UpdateAction>>
```

здесь A1 – инициализация заказа, A2 – установка статуса *Pending*, A3 – доступ к данным платежа, A4 – обновление статуса после оплаты. Представление действий представлено ниже:

| | |
|---|---|
| A1 : SingleTargetAction <<UpdateAction>><br>├─ targetRefinement → Attribute<br>OrderProcessor::OrderId<br>└─ arguments:<br>└─ Arg1<br>└─ valueRefinement → Parameter<br>Process::orderId | A2 : SingleTargetAction <<UpdateAction>><br>├─ targetRefinement → Attribute OrderProcessor::Status<br>└─ arguments:<br>└─ Arg2<br>└─ valueRefinement → EnumerationLiteral<br>OrderStatus::Pending |
| A3 : SingleTargetAction <<AccessAction>><br>├─ targetRefinement → Attribute<br>PaymentProcessor::Amount<br>└─ arguments:<br>└─ Arg3<br>└─ valueRefinement → LocalVariable tmpPrice | A4 : SingleTargetAction <<UpdateAction>><br>├─ targetRefinement → Attribute OrderProcessor::Status<br>└─ arguments:<br>└─ Arg4<br>└─ valueRefinement → EnumerationLiteral<br>OrderStatus::Paid |

## 3. Примеры OCL-предусловий, постусловий и инвариантов

***Инварианты:***

1. targetRefinement должен указывать только на допустимые элементы (локальная переменная, атрибут класса или параметр метода):
```
context SingleTargetAction
inv ValidTargetType:
  self.targetRefinement.oclIsKindOf(LocalVariable) or
  self.targetRefinement.oclIsKindOf(Parameter) or
  self.targetRefinement.oclIsKindOf(Attribute)
```
2. Запрещена ссылка на атрибут подкласса:
```
context SingleTargetAction
inv NoSubclassAttributeAccess:
  self.targetRefinement.oclIsKindOf(Attribute) implies
  self.targetRefinement.oclAsType(Attribute).owner =
  self.actionSequence.method.owner
```
3. Локальная переменная должна принадлежать текущему методу:
```
context SingleTargetAction
inv LocalVariableScopeCorrect:
  self.targetRefinement.oclIsKindOf(LocalVariable) implies
  self.targetRefinement.oclAsType(LocalVariable).scope =
  self.actionSequence
```

***Предусловия:***

1. Идентификатор заказа не должен быть пустым:
```
context OrderProcessor::Process(orderId : String, userId : String)
pre OrderIdNotEmpty:
  not orderId.oclIsUndefined() and orderId <> ''
```
2. Оплата возможна только при валидном заказе и положительной сумме:
```
context OrderProcessor::ProcessPayment()
pre PaymentIsAllowed:
  self._paymentProcessor.IsValid = true and
  self._paymentProcessor.Amount > 0
```

***Постусловия:***

1. После успешной обработки статус не может быть Pending:

```
context OrderProcessor::Process(orderId : String, userId : String)
post StatusUpdated:
    self.Status <> OrderStatus::Pending
```

2. После выполнения оплаты статус заказа должен быть Paid или Processing:

```
context OrderProcessor::ProcessPayment()
post PaymentLeadsToStatusChange:
    self.Status = OrderStatus::Paid or
    self.Status = OrderStatus::Processing
```