

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Реализация кодовой модели приложений на Java и Kotlin для генераторов кода, основанных на LLM

Демченко Артем Евгеньевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
к. ф.-м. н., доц. Куликов Е. К.

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Структура Explyt plugin	6
2.2. Обзор существующих способов представления пользова- тельского кода	7
2.3. PSI	8
3. Реализация кодовой модели	10
3.1. Архитектура	10
4. Тестирование	12
Заключение	13
Список литературы	14

Введение

Тестирование является неотъемлемой частью разработки программного обеспечения, так как оно позволяет контролировать качество создаваемых приложений. Когда тестирование осуществляется вручную, оно становится трудозатратным и дорогостоящим процессом, потому что требует большого количества человеческих усилий. Например, в крупных компаниях существуют целые команды тестировщиков, занимающихся верификацией программного обеспечения. В связи с этим в последние годы активно развиваются различные подходы к автоматизированному тестированию программ.

Одним из таких подходов является использование формальных методов, а именно символьного исполнения [14] и фаззинга [1]. Такие техники применяются, например, в `UnitTestBot Java` [15]. Тем не менее этот подход имеет минусы, такие как взрыв состояний, из-за чего генерация тестов может работать долго, и низкое качество генерируемого кода. В результате чего развиваются подходы, основанные на LLM¹, чему посвящено большое количество научных статей [13]. Основным плюсом этого метода является то, что нейросети предоставляют возможность генерировать максимально похожий на написанный человеком код.

В данный момент ведется разработка `Explyt plugin` для одной из самых распространенных сред разработки — `IntelliJ IDEA` [4]. Основная функциональность плагина — автоматическая генерация тестов, основанная на LLM. При таком подходе важную роль играет формируемый к нейросети запрос, так как от этого напрямую зависит качество сгенерированных тестов. Например, если пользователь хочет протестировать метод a класса B , то для запроса будет полезен не только исходный код метода a , но и поля класса B [10]. Таким образом, одной из ключевых задач является составление оптимальных запросов к LLM. При этом чтобы их осуществлять, команде AI необходимо получать информацию об исходном коде проекта пользователя.

¹LLM — Large Language Model, https://en.wikipedia.org/wiki/Large_language_model (дата обращения: 27 мая 2024 г.).

Для этого требуется реализовать сбор метаданных проекта и представление их в некоторой кодовой модели. Для достижения этой цели необходимо использовать IntelliJ Platform [5] — ядро с открытым исходным кодом, позволяющее пользоваться уже реализованными инструментами, такими как PSI [11], форматирование кода и т.п. Пока рассматривается поддержка языков Java и Kotlin, но также предполагается создание архитектуры кодовой модели таким образом, чтобы ее можно было расширить и для других объектно-ориентированных языков, поддерживаемых IntelliJ Platform. Для них кодовая модель должна содержать информацию об используемых в коде классах и их синтаксических и семантических свойствах.

1 Постановка задачи

Цель работы — реализовать кодовую модель для языков Java и Kotlin, содержащую информацию о нужных для создания запросов к LLM синтаксических и семантических свойствах пользовательского кода. Для достижения цели были поставлены следующие задачи:

1. Рассмотреть уже существующие способы представления пользовательского кода в виде кодовой модели;
2. Спроектировать архитектуру, позволяющую поддерживать языки Java и Kotlin и отражающую все необходимые для запросов к LLM синтаксические и семантические свойства;
3. Реализовать кодовую модель для Java и Kotlin, используя инструменты, предоставляемые IntelliJ Platform;
4. Протестировать полученную реализацию;
5. Внедрить кодовую модель в плагин.

2 Обзор

2.1 Структура Explyt plugin

Как уже отмечалось ранее, Explyt plugin разрабатывается для IntelliJ IDEA. На рисунке 1 схематично представлена архитектура плагина. Оранжевым цветом выделена часть, которую необходимо реализовать. Ядро (Core) содержит в себе компоненты, реализованные преимущественно на основе IntelliJ Platform. Так, для непосредственного запуска генерации тестов используется **Action** [6] — один из механизмов, позволяющий расширять базовую функциональность IntelliJ IDEA. Сбор контекста тестируемого кода должен осуществляться после срабатывания **Action**-а, но до самой кодогенерации. Таким образом, контекст, представленный в виде кодовой модели, передается в клиент, который формирует запрос к LLM и затем возвращает сгенерированные тесты. Предполагается, что кодовая модель будет представлять собой небольшую библиотеку, удовлетворяющую некоторому API, отражающему необходимую для создания запросов к LLM информацию. Далее в работе это API будем называть Codemodel API. Реализацию требуется написать на языке Kotlin, так как этот язык выбран командой как основной в проекте.

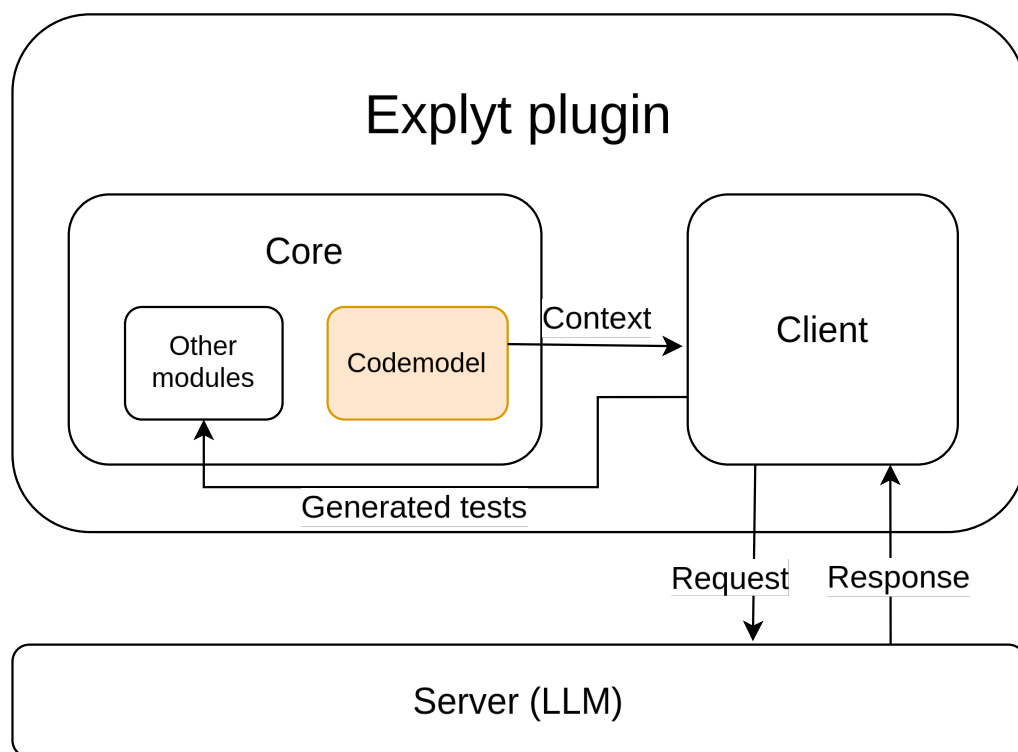


Рис. 1: Основная структура Explyt plugin

2.2 Обзор существующих способов представления пользовательского кода

В первую очередь был рассмотрен CodeGPT [9] — плагин для IntelliJ IDEA с открытым исходным кодом, интегрирующий возможности LLM непосредственно в среду разработки. Он предоставляет пользователю целый ряд функций: автодополнение кода, поддержка чата с нейросетью в самой IntelliJ IDEA, генерация сообщения коммита и генерация тестов. Был проанализирован принцип работы именно последней функции. Для этого изучался исходный код плагина, а также плагин был собран локально и запущен на тестовом проекте.

Процесс взаимодействия с плагином заключается в том, что пользователю необходимо выделить фрагмент кода, для которого требуется сгенерировать тесты. После этого происходит обращение к LLM, и пользователю предлагаются результаты генерации. Есть возможность указать собственный префикс для запроса, но по умолчанию в текст запроса к LLM не добавляется никакой дополнительной информации

об исходном коде. Добавить информацию можно, используя функцию `Include File in Context`, которая добавляет к запросу текст выбранного файла и путь к нему. В контекст можно добавить несколько файлов.

Таким образом, кодовой модели в том виде, который нужен в `Explyt plugin`, в `CodeGPT` нет, так как там она представляет собой набор исходных файлов. По этой причине была рассмотрена статья [2], описывающая подход к генерации тестов при помощи LLM, который используется в плагине `ChatUnitest` [3].

В `ChatUnitest` отдельным этапом генерации является подготовка — парсинг и анализ исходного кода. В результате парсинга код проекта представляется в виде абстрактных синтаксических деревьев, затем собираются данные о классах и методах. Например, для метода сохраняется его сигнатура, тело, доступ к полям класса, вызов других методов, имена зависимых классов. Полученное представление имеет гораздо более широкий спектр возможностей по сравнению с сохранением текста исходных файлов, так как процесс включения исходного кода в контекст становится более гибким. Поэтому в `Explyt plugin` при выборе архитектуры кодовой модели был выбран похожий подход. Тем не менее на данный момент `ChatUnitest` поддерживает только Java, а кодовая модель для `Explyt plugin` должна представлять как минимум 2 языка, поэтому она должна быть более универсальной.

2.3 PSI

PSI (The Program Structure Interface) — совокупность интерфейсов и инструментов `IntelliJ Platform`, отвечающих за парсинг и создание синтаксической и семантической модели кода, которая поддерживает большое количество уже реализованных инструментов `IntelliJ Platform`. Например, интерфейс `PsiFile` [8] используется для представления файла с исходным кодом. Его структура является PSI-деревом, вершины которого — PSI-элементы [7]. Так как необходимые синтаксические структуры (например, исходный код класса, сигнатура метода, список моди-

фикаторов видимости и т. п.) являются PSI-элементами, было решено использовать их в качестве базовых составляющих кодовой модели.

Одним из инструментов, облегчающих работу с PSI, является PSI Viewer [12] — плагин для IntelliJ IDEA, предоставляющий возможность просмотра PSI-дерева определенного исходного файла. Он позволяет сократить затраченное на разработку и тестирование время, потому что с его использованием отпадает необходимость запускать разрабатываемый Explyt plugin в режиме отладки и смотреть, как средства IntelliJ представят исходный файл в PSI. Таким образом, PSI Viewer активно использовался на протяжении всего времени работы над кодовой моделью.

3 Реализация кодовой модели

3.1 Архитектура

Классы, реализованные как часть кодовой модели и удовлетворяющие Codemodel API, будем называть обертками. Обертки требуется реализовать для всех основных единиц языка: классов, методов, функций верхнего уровня (для Kotlin), полей класса и т. д. Каждая обертка должна создаваться по соответствующему ей PSI-элементу. По дизайну Codemodel API обертки для одних и тех же единиц языка для Java и Kotlin должны имплементировать один и тот же интерфейс. Таким образом, одним из основных факторов, влияющих на архитектуру кодовой модели, является поддержка двух языков — Java и Kotlin.

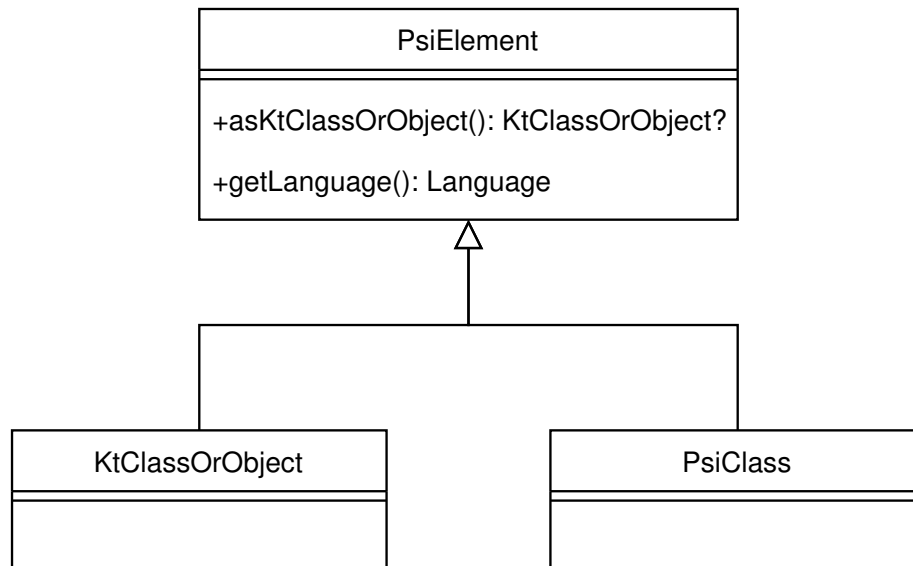


Рис. 2: Упрощенная иерархия классов и интерфейсов, представляющих Kotlin и Java классы в PSI

Для примера рассмотрим создание обертки класса. IntelliJ Platform предоставляет разные PSI-элементы для Kotlin и Java, что отражено на рисунке 2. Это логично, так как исходные коды классов Kotlin и Java отличаются синтаксически. Однако, в некоторых случаях парсинга Kotlin классы представляются в виде `PsiClass`. Чтобы получить `KtClassOrObject`, необходимо использовать метод `asKtClassOrObject()`. Таким образом, создание обертки для Kotlin

класса возможно как по `KtClassOrObject`, так и по `PsiClass`. При этом также необходимо рассмотреть возможность создания только по `PsiElement`, осуществив дополнительную проверку. Подобная ситуация происходит не только для классов, но и для методов. Поэтому было принято решение реализовать объект (в терминах Kotlin) `CodeModelFactory`, который осуществляет всю логику создания оберток по `PsiElement`, определяя язык по самому PSI-элементу. При этом создание оберток напрямую через их конструкторы было запрещено при помощи использования `annotation class`. Общая архитектура кодовой модели отражена на рисунке 3.

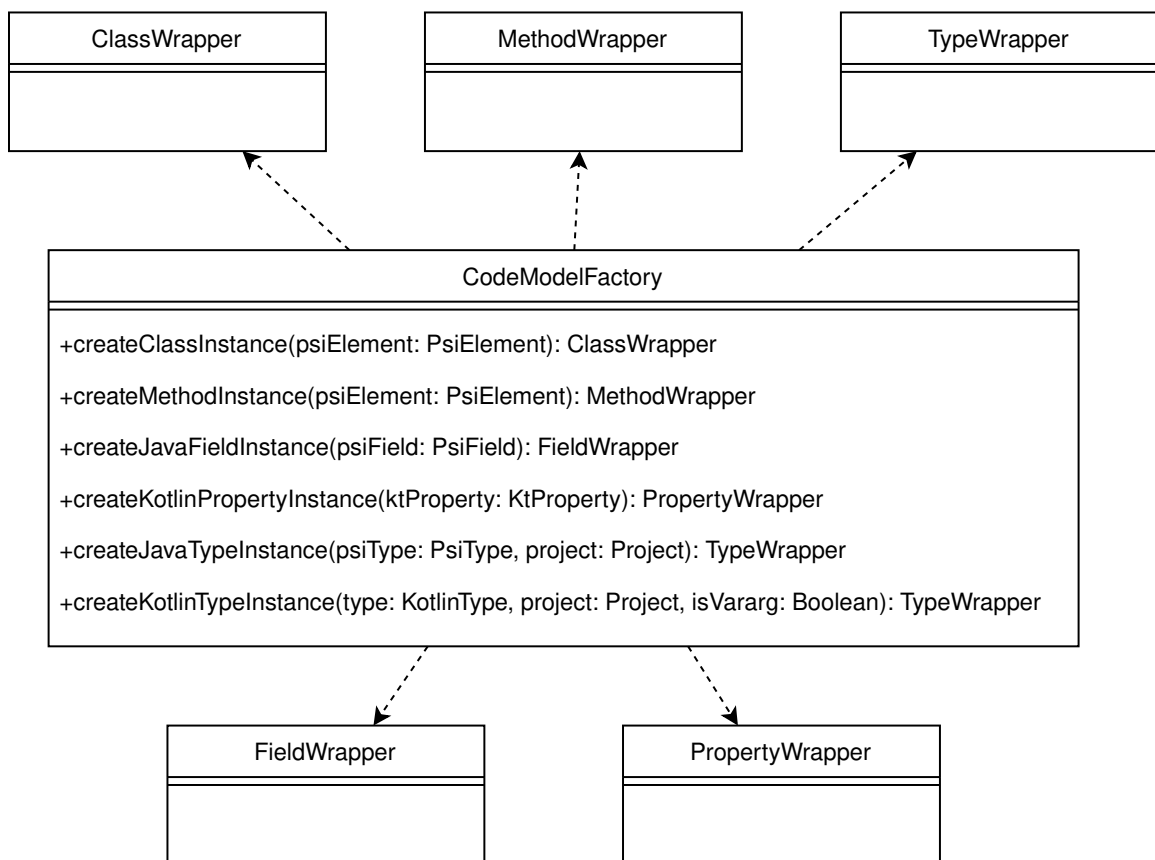


Рис. 3: Общая архитектура кодовой модели

Учитывая необходимость поддержки двух языков, основная цель, преследуемая при реализации оберток — добиться максимальной переиспользуемости кода.

4 Тестирование

Для тестирования реализованной функциональности необходимо написать unit-тесты на публичные методы оберток. Для этого нужен способ получения правильно сконфигурированного проекта с рассматриваемыми тестовыми случаями в самих тестах, что является нетривиальной задачей.

IntelliJ Platform предоставляет два основных механизма тестирования — Light Tests и Heavy Tests. Первый является более предпочтительным, так как при его использовании проект из предыдущего тестового запуска по возможности используется повторно, в то время как Heavy Tests создают новый проект для каждого теста, что является довольно затратной по времени операцией.

Учитывая специфику тестируемых сценариев и используемый фреймворк, был выбран подход, использующий Light Tests. В качестве основы для тестов было решено использовать `LightJavaCodeInsightFixtureTestCase5`, предоставляемый IntelliJ Platform.

Заключение

В ходе данной работы основная цель была достигнута — реализована кодовая модель для языков Java и Kotlin, содержащая информацию о нужных для создания запросов к LLM синтаксических и семантических свойств пользовательского кода. Результаты:

1. Рассмотрены уже существующие способы представления пользовательского кода в виде кодовой модели, на их основе выбран подход, который стал использоваться в плагине;
2. Спроектирована архитектура, поддерживающая языки Java и Kotlin и отражающая все необходимые для запросов к LLM синтаксические и семантические свойства;
3. Реализована кодовая модель для Java и Kotlin;
4. Написаны unit-тесты и протестирована полученная реализация;
5. Кодовая модель внедрена в плагин и используется в других модулях, имеется подписанный акт о внедрении.

Код находится под договором о неразглашении, поэтому не может быть предоставлен.

Список литературы

- [1] The Art, Science, and Engineering of Fuzzing: A Survey / Valentin J.M. Manès, HyungSeok Han, Choongwoo Han et al. // [IEEE Transactions on Software Engineering](#). — 2021. — Vol. 47, no. 11. — P. 2312–2331.
- [2] Chen Yinghao, Hu Zehao, Zhi Chen et al. ChatUniTest: A Framework for LLM-Based Test Generation. — 2024. — 2305.04764.
- [3] ChatUnitest. — URL: <https://github.com/ZJU-ACES-ISE/ChatUniTest> (дата обращения: 27 мая 2024 г.).
- [4] IntelliJ IDEA – the Leading Java and Kotlin IDE. — URL: <https://www.jetbrains.com/idea/> (дата обращения: 27 мая 2024 г.).
- [5] The IntelliJ Platform. — URL: <https://plugins.jetbrains.com/docs/intellij/intellij-platform.html> (дата обращения: 27 мая 2024 г.).
- [6] IntelliJ Platform Plugin SDK. Actions. — URL: <https://plugins.jetbrains.com/docs/intellij/basic-action-system.html> (дата обращения: 27 мая 2024 г.).
- [7] IntelliJ Platform. Psi Elements. — URL: <https://plugins.jetbrains.com/docs/intellij/psi-elements.html> (дата обращения: 27 мая 2024 г.).
- [8] IntelliJ Platform. Psi Files. — URL: <https://plugins.jetbrains.com/docs/intellij/psi-files.html> (дата обращения: 27 мая 2024 г.).
- [9] A JetBrains extension providing access to state-of-the-art LLMs. — URL: <https://github.com/carlrobertoh/CodeGPT> (дата обращения: 27 мая 2024 г.).

- [10] Yuan Zhiqiang, Lou Yiling, Liu Mingwei et al. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. — 2023. — 05. — URL: <https://arxiv.org/abs/2305.04207> (дата обращения: 27 мая 2024 г.).
- [11] Program Structure Interface (PSI). — URL: <https://plugins.jetbrains.com/docs/intellij/psi.html> (дата обращения: 27 мая 2024 г.).
- [12] A Program Structure Interface (PSI) tree viewer plugin for IDEA platform IDE's. — URL: <https://www.jetbrains.com/help/idea/psi-viewer.html> (дата обращения: 27 мая 2024 г.).
- [13] Software Testing With Large Language Models: Survey, Landscape, and Vision / Junjie Wang, Yuchao Huang, Chunyang Chen et al. // IEEE Transactions on Software Engineering. — 2023. — Vol. 50. — P. 911–936. — URL: <https://api.semanticscholar.org/CorpusID:259924919> (дата обращения: 27 мая 2024 г.).
- [14] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia et al. // ACM Computing Surveys (CSUR). — 2016. — Vol. 51. — P. 1 – 39. — URL: <https://api.semanticscholar.org/CorpusID:36536> (дата обращения: 27 мая 2024 г.).
- [15] UnitTestBot/UTBotJava: Automated unit test generation and precise code analysis for Java. — URL: <https://github.com/UnitTestBot/UTBotJava> (дата обращения: 27 мая 2024 г.).