Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 20.Б10-мм

Волынец Арсений Викторович

Плагин CLion для генерации тестов на C/C++

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ст. пр. Е.К. Куликов

Консультант:
разработчик ООО Яндекс, В.Е. Володин

Санкт-Петербург
2022

Оглавление

1. Введение **3**

2. Постановка задачи **4**

3. Обзор **5**

3.1. Плагина для VSCode 5

3.2. Обзор используемых технологий 6

3.3. Технологии разработки 8

3.4. gRPC 8

4. Реализация **10**

4.1. Архитектура плагина 10

4.2. Модуль Requests 11

4.3. Модуль Handlers 12

4.4. Отображение логов сервера и клиента 13

4.5. Панель для выбора таргета 14

4.6. Графические элементы для запуска тестов в редакторе . 15

4.7. Отображение покрытия 16

4.8. Взаимодействие с сервером 17

4.9. Настройки 19

4.10. Визард 20

4.11. Интеграционные тесты 20

5. Заключение **23**

Список литературы **24**

1. Введение

Тестирование – это важная часть разработки любого программного продукта. Тестирование не только позволяет проверить поведение программы при определённых обстоятельствах, но и повышает ее ценность.

UTBotCpp[1] – инструмент для автоматической генерации юнит-тестов с максимизацией покрытия, разрабатываемый компанией Huawei. За счет использования данного инструмента можно сократить время, уходящее на рутинное написание тестов и посвятить его другим задачам. Для работы с UTBotCpp существуют интерфейсы командной строки и gRPC. gRPC[2] – набор библиотек для удаленного вызова процедур (remote procedure call – rpc), с открытым исходным кодом, первоначально разработанный компанией Google. В свою очередь удаленный вызов процедур – это класс технологий, позволяющий вызывать процедуры из другого адресного пространства, например удаленного узла. Подробнее про удаленный вызов см. [18].

В компании Huawei был разработан плагин для Visual Studio Code [14], позволяющий запускать генерацию тестов непосредственно из среды разработки, взаимодействующий с сервером через gRPC.

CLion – это другая среда разработки, предназначенная для языков C/C++. Согласно опросу [1], данная IDE, является одной из самых популярных для этих языков и имеет примерно такую же аудиторию как VSCode. Стоит отметить, что CLion предоставляет некоторые возможности, которых нет в VSCode, например рефакторинг кода и различные инспекции во время статического анализа кода.

Поскольку есть люди, которые пользуются именно CLion, и не пользуются VSCode, возникает потребность в добавлении возможностей использовать UTBotCpp из CLion. В связи с этим была поставлена задача разработать плагин CLion для взамодействия с UTBotCpp.

[1] github.com/UnitTestBot/UTBotCpp – репозиторий с сервером для генерации тестов и плагином для VSCode.

[2] https://grpc.io – официальный сайт gRPC

2. Постановка задачи

Целью работы является реализация плагина CLion для генерации тестов на C/C++.

Для её выполнения были поставлены следующие задачи:

1. Спроектировать архитектуру плагина.
2. Реализовать пользовательские интерфейсы:
    - Настроек плагина
    - Запуска генерации тестов
    - Отображение прогресса при генерации тестов
    - Отображения покрытия
    - Панели для выбора таргета (библиотеки или исполняемого файла, от которого зависит генерация тестов).
    - Графические элементы для запуска тестов в редакторе
    - Окно вывода для отображения логов сервера и клиента пользователю
    - Визард, который помогает настроить плагин при первом использовании
    - Прочие элементы пользовательского интерфейса: статус подключения к серверу, уведомления по завершении генерации тестов и др.
3. Реализовать взаимодействие с сервером UTBotCpp через gRPC, поддержав все удаленные процедуры, предлагаемые сервером. Реализовать обработку данных, которые поступают с сервера.
4. Написать интеграционные тесты, проверяющие корректность взаимодействия с сервером.

3. Обзор

В данной главе проводится обзор функциональных возможностей плагина для взаимодействия с сервером UTBotCpp, особенностей его реализации а также обзор используемых технологий.

3.1. Плагина для VSCode

На рис. 1 представлен пользовательский интрейфейс плагина VSCode.

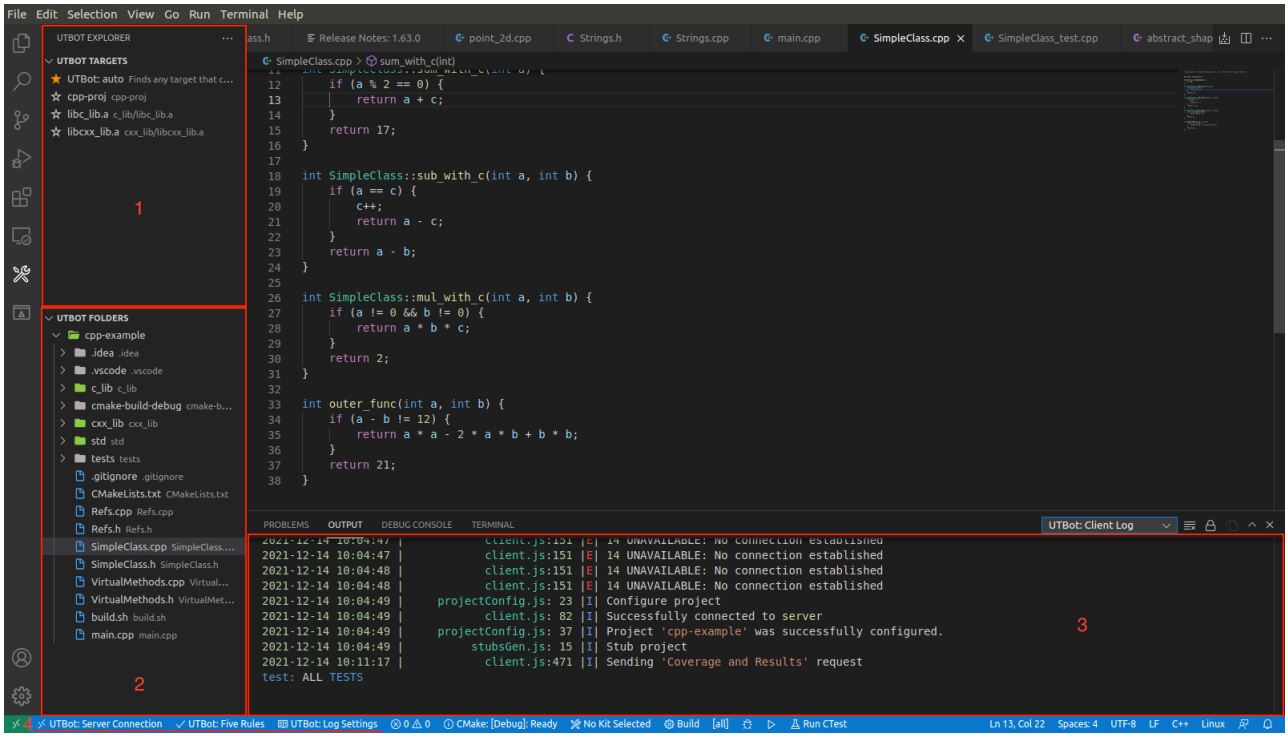

Рис. 1: Общий вид интерфейса

На рис. 1 цифрой 1 отмечена панель для выбора таргета (библиотеки или исполняемого файла, см. [17], от которого зависит с какими файлами, библиотеками надо компилировать и линковать файл, для которого запущена генерация). Цифрой 2 отмечена панель, в которой изображено дерево проекта. В ней можно помечать папки, под которыми лежат исходные файлы проекта (данные помеченные папки передаются серверу при генерации тестов для проекта). Цифрой 3 отмечена консоль, куда выводится лог плагина при отправке и получений сообщений от сервера. Цифрой 4 отмечены элементы строки состояния, отображаю-

щие установлено ли подключение к серверу, текущие настройки для уровня логгирования, и настройка «UTBot: Five Rules». Если она активирована, плагин использует рекомендованные его разработчиками настройки для генерации тестов.

Для отправки запросов на генерацию тестов пользователь может кликнуть правой кнопкой в текстовом редакторе и в появившемся контекстном меню выбрать “UTBot: Generate Tests” и выбрать интересующий вариант, см. рис 2.

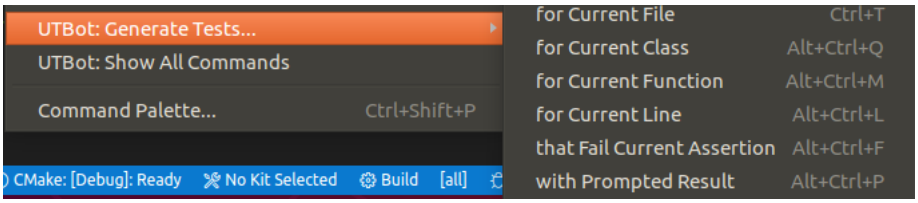

Рис. 2: Запуск генерации тестов

3.2. Обзор используемых технологий

3.2.1. IntelliJ Platform

Для создания плагинов для продуктов IntelliJ необходимо использовать IntelliJ Platform – платформу с открытым исходным кодом для создания IDE и плагинов к ним. Существуют три способа создать проект плагина:

1. Использовать готовый шаблон [19]
2. Использовать gradle [16]
3. Использовать плагин для IntelliJ IDEA – plugin devkit [15]

Был выбран первый способ, как самый быстрый для начала работы. Также в таком случае, чтобы опубликовать плагин в JetBrains Marketplace, достаточно запустить в gradle задание (task) publishPlugin.

Для реализации самого плагина необходимо использовать API IntelliJ Platform. Для поиска необходимого API, можно использовать документацию [12], исходный код IntelliJ-Community [13], а также плагины с открытым исходным кодом в JetBrains MarketPlace.

В зависимости от назначения плагина, могут потребоваться различные компоненты IntelliJ Platform. При разработке использовались такие подсистемы как Actions, различные элементы пользовательского интерфейса, PersistingStateComponent, PSI.

3.2.2. Actions

Механизм Actions позволяет пользователю запускать доступные в данном контексте функции плагина (контекст – это то место откуда вызывается Action *(далее действие)*, например, это может быть текстовый редактор или дерево проекта.

Для создания действия необходимо унаследоваться от одного из существующих действий (например, это может быть AnAction) и перегрузить необходимые методы. Наиболее часто это методы `update(e: AnActionEvent)` и `actionPerformed(e: AnActionEvent)`. `update` вызывается самой IDE и определяет, доступно ли действие пользователю для вызова. `actionPerformed` – вызывается, когда действие доступно и пользователь вызвал его. `AnActionEvent` содержит информацию о контексте, в котором было вызвано действие, например из него можно получить номер строчки, на которой находился курсор, если действие было вызвано из текстового редактора. Также действие необходимо зарегистрировать в конфигурационном файле плагина plugin.xml с указанием, где данное действие отображается для вызова, например это может контекстное меню редактора или контекстное меню в дереве проекта. Также в данном файле необходимо указать другие параметры, например, название действия.

3.2.3. PersistingStateComponent

Этот интерфейс позволяет сохранять данные между запусками IDE. Данные сериализуются в xml файл и при следующем запуске извлекаются из него. Подробности того как реализовывать данный интерфейс можно найти в документации [3].

3.2.4. PSI

PSI – program structure interface, структура программы. Данный механизм позволяет интроспектировать структуру программы во время работы плагина. Например, с помощью PSI можно узнать, существует ли функция с определенными параметрами или узнать, находится ли сейчас курсор внутри функции или внутри класса. Более подробное описание PSI можно найти в документации [6].

3.2.5. Extension point

Extension point (точка расширения) – это основной способ расширения функциональнасти IDE, помимо действий. Он представляет собой класс или интерфейс, который нужно реализовать. Классы реализующие точки расширения нужно указать в конфигурационном файле plugin.xml. Во многих случаях IDE будет сама создавать экземпляры данных классов и вызывать их методы. Подробнее см. [2]. Так в плагине используются точки расширения для реализации окон интруметов (tool window), отображения покрытия, отображения иконок для запуска тестов в редакторе код. Полный список можно посмотреть в plugin.xml.

3.3. Технологии разработки

Для создания плагинов на основе IntelliJ Platform, можно использовать Java, Kotlin, Scala. Поскольку по производительности данные языки прибилизительно одинаковы, в качестве языка программирования был выбран Kotlin, как более выразительный и простой в использовании.

3.4. gRPC

Как было написано выше, gRPC – это набор библиотек для удаленного вызова процедур. Сначала определяется набор процедур, которые можно вызывать удаленно, с указанием принимаемых и возвращаемых

параметров. Сервер – удаленный узел, который реализует данные процедуры. У клиента имеется *стаб (можно перевести как помощник на стороне клиента)* – сущность, которая предоставляет те же процедуры, что и сервер и генерируется gRPC. Стаб перенаправляет вызовы процедур на сервер и возвращает результат клиенту. Для пересылки сообщений между сервером и клиентом, а также для указания самих процедур используется язык protocol-buffers. В файлах util.proto, testgen.proto [11] содержатся определения сообщений и процедур, что предоставляет сервер UTBotCpp.

При генерации стабов на Kotlin в gRPC используется библиотека kotlinx.coroutines для асинхронного взаимодействия с сервером.

4. Реализация

В данной главе представлена архитектура приложения, а так же изложены достигнутые результаты и некоторые детали реализации. В частности, описаны классы, отвечающие за взаимодействие с сервером (Client), запуск генерации тестов (Actions), настройки плагина (ProjectsSettings), отображение визарда, обработку ответов от сервера.

4.1. Архитектура плагина

На рис. 3 представлена архитектура плагина в виде модулей, и их зависимостей между друг другом.

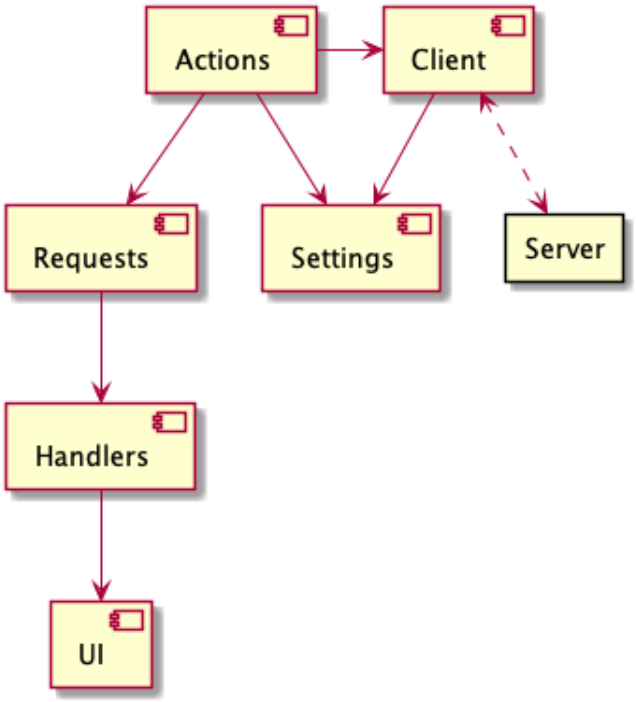

Рис. 3: Архитектура плагина

Модуль Actions содержит действия для запуска генерации тестов (действия GenerateFor...), запуска выполнения тестов и отображения покрытия (RunWithCoverageAction), настройки проекта (ConfigureProjectAction), генерации необходимых файлов для настройки проекта (GenerateJsonFilesAction), и другие действия (*внутренние действия*), используемые внутри плагина для разных целей (например FocusAction, MarkSourceFolderAction и др.).

Все действия кроме внутренних работают по общей схеме: сначала пользователь вызывает их из IDE, затем они, используя данные из настроек (модуль Settings), создают экземпляр класса Request и передают его экземпляру класса Client в метод executeRequest (подробнее про этот метод см. раздел ??).

Модуль Client отвечает за создание grpc канала взаимодействия с сервером и за предоставление стаба в методе executeRequest для вызова удаленных процедур. Для создания канала используются настройки порта и имени сервера.

Классы из модуля Requests вызывают один из методов стаба для получения сообщений от сервера и используют классы из модуля Handlers для обработки ответов.

Классы из Handlers отвечают за обработку сообщений от сервера и могут обращаться к компонентам из модуля UI (например, для показа уведомлений, отображения покрытия).

Модуль UI также содержит реализацию различных компонентов пользовательского интерфейса таких как панели инструментов (tool window), элементов строки состояния (status bar), визарда, графического отображения настроек.

4.2. Модуль Requests

Данный модуль содержит классы запросов – обертки над классами сообщений, которые генерируются gRPC. Все запросы реализуют общий интерфейс:

```
interface Request {
    suspend fun execute(stub: TestsGenServiceGrpcKt.
       TestsGenServiceCoroutineStub, cancellationJob: Job?)
}
```

Метод execute отправляет запрос на сервер, используя stub, и обрабатывает ответы. cancellationJob это Job корутины которую нужно отменить, если данный запрос был отменен пользователем.

4.3. Модуль Handlers

В данном модуле содержатся классы, отвечающие за обработку сообщений от сервера. В зависимости от запроса обработка ответов может включать: показ уведомлений пользователю об ошибке или об успешной генерации тестов, создание файлов с генерированными тестами, отображение покрытия, отображение прогресса выполнения запроса.

На рис. 4 представлена диаграмма классов модуля.

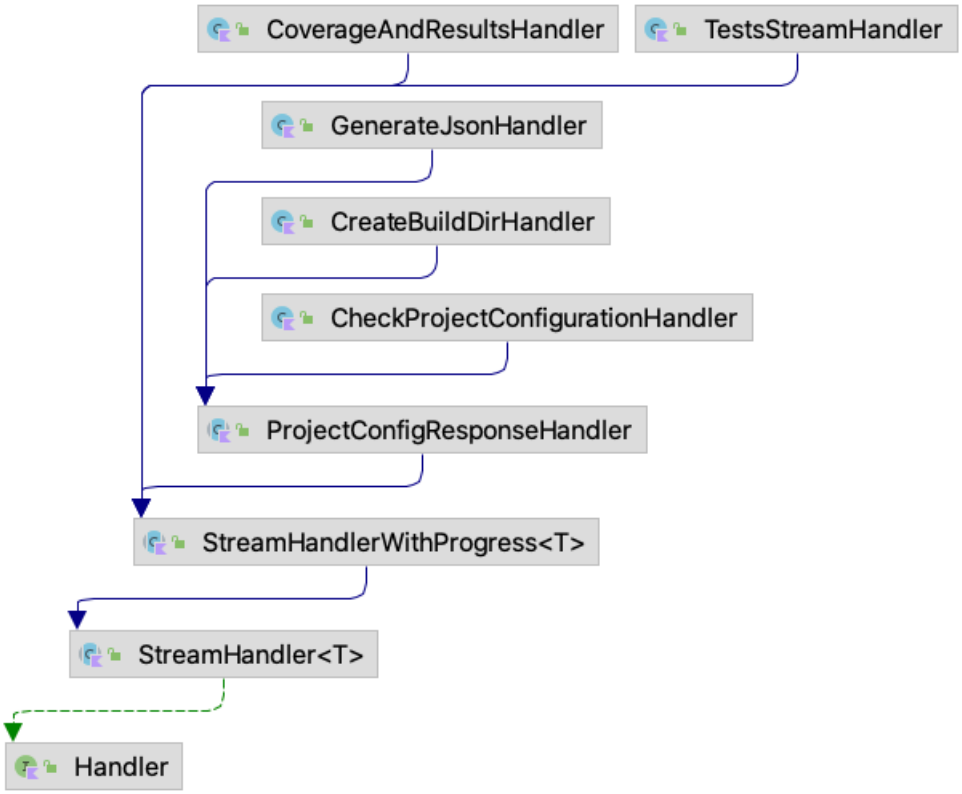

Рис. 4: Диаграмма классов модуля Handlers

Интерфейс Handler имеет только один метод `suspend fun handle`() . Поскольку сообщения могут приходить асинхронно функция handle помечена модификатором suspend.

Класс StreamHandler обрабатывает поток сообщений от сервера [10]. При его реализации используется паттерн шаблонный метод: в методе handle задается общий алгоритм обработки сообщений от сервера через функции onStart(), onData(), onException() и др., которые могут быть переопределены в наследниках.

Класс StreamHandlerWithProgress добавляет функциональность по отображению прогресса пользователю. Подавляющее большинство удаленных процедур сервера, возвращающих поток сообщений, передают во всех сообщениях кроме последнего только прогресс выполненной работы, а последнее содержит результат удаленной процедуры. На основе этого этот класс отображает прогресс пользователю, и отменяет обработку сообщений, если пользователь отменил выполнение данного запроса. Для отображения прогресса пользователю используется класс UTBotProgressIndicator.

Остальные класса данного модуля содержат логику по обработке того или иного запроса. CoverageAndResultsHandler отображает покрытие пользователю и показывает уведомление. TestsStreamHandler создает файлы с сгенерированными тестами, и показывает уведомление. CheckProjectConfigurationHandler в зависимости от ответа сервера на запрос о настройке проекта предлагает пользователю создать папку для сборки или сгенерировать файлы не обходимые для сборки.

4.4. Отображение логов сервера и клиента

Поскольку тесты запускаются удаленно на сервере, необходимо предоставить пользователю доступ к логам при запуске тестов для просмотра результов и исключений, которые могут случиться при выполнении тестов. Также ошибки могут происходить на стороне сервера при генерации тестов, для отладки которых также необходимо смотреть на логи. Помимо этого чтобы проверить, что сообщение было отправлено на сервер, или для отладки ошибок при подключению к серверу возникает необходимость показывать логи со стороны плагина (клиента). Сервер предоставляет две rpc, которые отсылают логи при генерации и запуске тестов: openGTestChannel, openLogChannel. При подключении к серверу Client отпраляет запрос на получение этих логов. Перед этим необходимо выполнить запросы closeGTestChannel, closeLogChannel соответственно, это связано с тем, как логгирование реализовано на стороне сервера.

Для реализации отображения логов использовался класс ConsoleViewImpl, который используется внутри IDE для отображения консоли. Сами консоли отображаются внутри ToolWindow (см. [9]), где каждая консоль это вкладка. На рис 5 показано вид окошка с консолями.

Рис. 5: Консоли для вывода логов

4.5. Панель для выбора таргета

Генерация тестов зависит от таргета. На рис. 6 представлена диаграмма классов, которые связаны с выбором таргета. TargetsToolWindow это UI класс, который отображает список таргетов. TargetsController - контроллер, который является посредником между моделью и пользовательским интерфейсом. Когда пользователь выбирает новый таргет, контроллер устанавливает новый таргет в настройках. Он также подписывается на события от CLient, и при изменении подключения запрашивает список таргетов у CLient, для чего Client делает запрос на сервер.

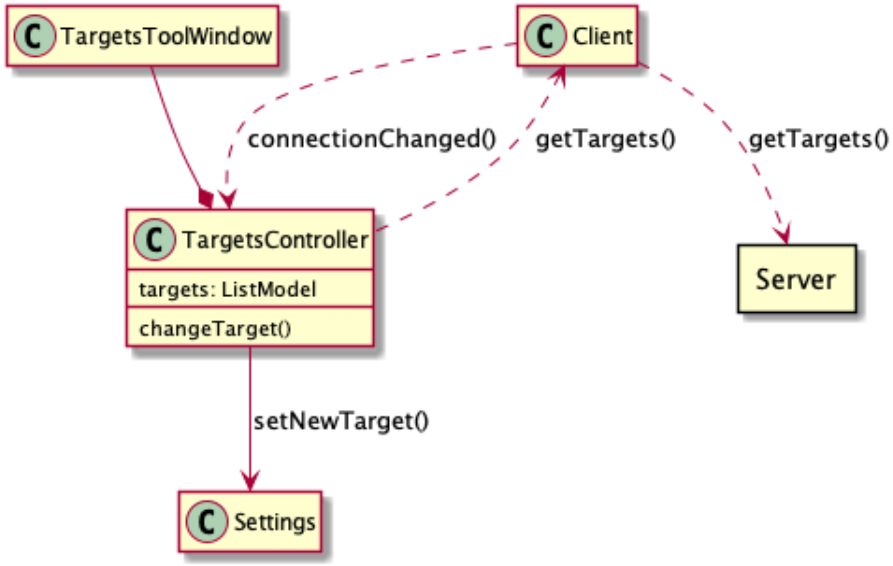

Рис. 6: Классы относящиеся к выбору таргета

4.6. Графические элементы для запуска тестов в редакторе

После того, как тесты были сгенерированы, для их запуска необходимо вызвать удаленную процедуру на сервере, которая помимо результатов выполнения тестов возвращает покрытие.

В данной главе рассматривается реализация пользовательского интерфейса для запуска выполнения запроса.

Были рассмотрены два варианта графического интерфейса для запуска тестов: это опция в контекстном меню и значок возле имени теста в текстовом редакторе. Был выбран второй способ, поскольку он требует меньшего числа действий от пользователя. Для отображения значков запуска тестов в редакторе была использована точка расширения LineMarkerProvider, у которой необходимо реализовать метод getLineMarkerInfo. Во время подсветки (highlighting) редактора иде проходит по всем зарегистрированным LineMarkerProvider, и у каждого из них вызывает getLineMarkerInfo(PsiElement) для каждого PsiElement, данный метод возвращает информацию в виде LineMarkerInfo<T: PsiElement> об строке на которой находится PsiElement.

Реализация getLineMarkerInfo возвращает UTBotLineMarkerInfo, если текст элемента есть TEST или UTBot. UTBotLineMarkerInfo в свою

очередь определяет, как иконка будет отображаться в редакторе и что при нажатии будет выполняться действие RunWithCoverageAction. Отображение иконки зависит от того какой был последний результат выполнения теста. Результаты выполнения тестов хранятся в TestsResultsStorage.

На рис. 7, 8 показано отображение иконок в IDE:

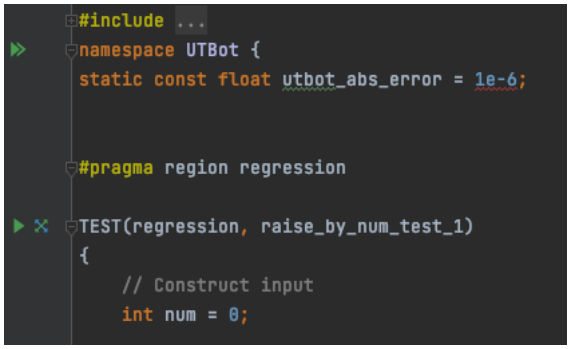

Рис. 7: Отображение иконок для запуска тестов в редакторе до запуска тестов

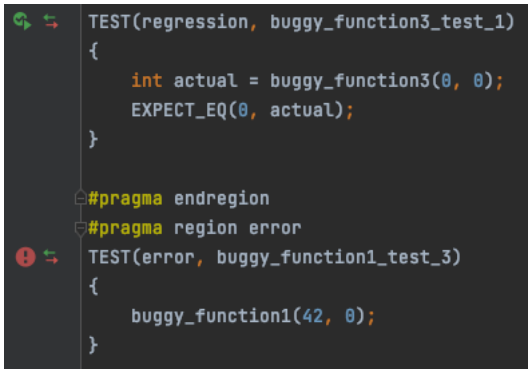

Рис. 8: Отображение иконок для запуска тестов в редакторе после запуска тестов

4.7. Отображение покрытия

Для отображения покрытия были реализованы следующие интерфейсы: CoverageRunner, CoverageEngine, CoverageSuite, CoverageFileProvider. Данное api, предполагает, что после того как тесты были исполнены, информация о покрытии была сохранена локально в некотором файле, который предоставляет CoverageFileProvider. CoverageSuite хранит

мета-информацию о покрытии, а также настройках для отображения покрытия.

Поскольку покрытие отправляется сервером по запросу пользователя, использовать файлы для сохранения покрытия не имеет особого смысла. Вместо этого покрытие сохраняется в UTBotCoverageSuite. Что потом используется в UTBotCoverageRunner.

Когда пользователь делает запрос на запуск тестов и получение покрытия сервер возвращает последовательность асинхронных ответов. Только последний ответ содержит покрытие и результаты тестов. После чего создается UTBotCoverageSuite, в котором сохраняется информация о покрытии. И запускается процесс обработки покрытия в IDE: CoverageDataManager.coverageGathered(UTBotCoverageSuite). Далее будет вызван метод UTBotCoverageRunner.loadCoverageData(), который используя покрытие в UTBotCoverageSuite, возвращает ProjectData - представление покрытия, используемое IDE. После чего покрытие будет отображено в открытых редакторах.

На рис. 9 показано отображение покрытия.

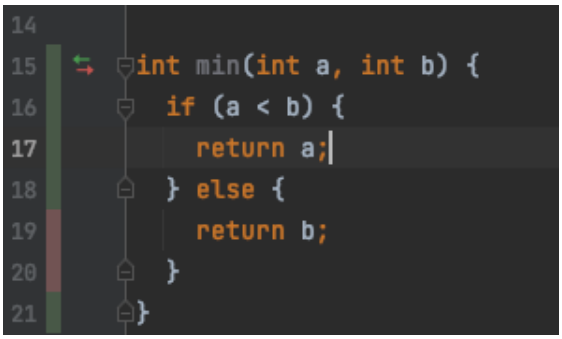

Рис. 9: Отображение покрытия

4.8. Взаимодействие с сервером

За взамодействие с сервером отвечает класс `Client`. На рис. 10 представлено взаимодействие класса с другими компонетами.

Класс GrpcClient отвечает за создание и настройку стаба для отправки запросов. Класс Client оборачивает запросы в корутины и предоставляет стаб для вызова удаленных процедур. Также данный класс

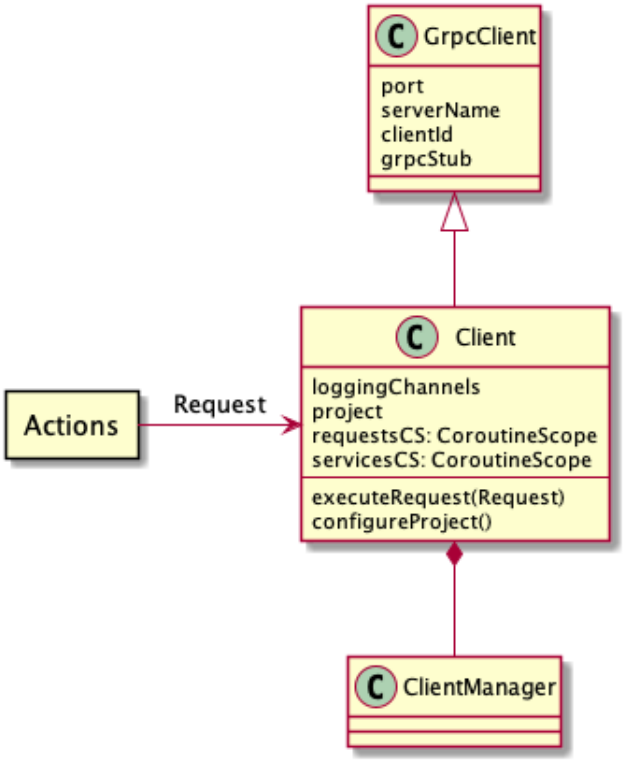

Рис. 10: Client – предоставляет контекст для выполнения запросов.

делает выполняет ряд удаленных процедур не относящихся непосредственно к запросам инициированных пользователем (получение логов от сервера, периодическая проверка подключения с сервером). Для этих процедур используется отдельный CoroutineScope: servicesCS. Это позволяет отслеживать запросы инициированные пользователем отдельно.

ClientManager создает новую сущность класса Client, когда пользователь в настройках меняет порт или имя сервера.

4.8.1. Отображение прогресса

Генерация для больших проектов может занимать продолжительное время, и пользователю нужно понимать какая доля работы была выполнена. Многие процедуры, что предоставляет сервер UTBotCpp, возвращают не один ответ, а последовательность асинхронных ответов [10]. Сообщения, возвращаемые сервером, содержат процент выполненной работы и текущий этап.

Стандартный способ отображения прогресса в IntelliJ Platform – это использование методов `ProgressManager` [7]. Но данные методы не

предназначены для работы с корутинами, которые используются в стабе при отправке сообщений на сервер и при получении ответов. Непосредственное использование корутин и методов класса `ProgressManager` ведет к блокировке потока. По этой причине было сделано следующее: в исходниках IntelliJ-Community, найден класс (`StatusBar`), отвечающий за отображение прогресса пользователю, а также способ обратиться к нему. Данная логика реализована в классе UTBotRequestProgressIndicator. На рис. 11 показано отображение прогресса при генерации тестов.

Рис. 11: Отображение статуса подключения в статус баре

4.8.2. Отображения статуса подключения

Для отображения статуса подключения к серверу, каждые 500 мс выполняется `rpc Heartbeat`, проверяющий что подключение есть. Сам статус отображается в строке состояния, см. рис. 12.

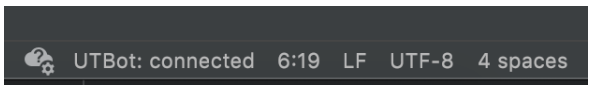

Рис. 12: Отображение статуса подключения в статус баре

4.8.3. Настройка проекта

Перед непосредственной отправкой запросов на генерацию тестов, выполняются следующие `rpc: RegisterClient` (только при первом запуске плагина) и `ConfigureProject`, чтобы сервер настроил внутренние параметры для генерации тестов. В случае если настройка проекта не удалась, клиенту будет выведено сообщение, с возможными дальнейшими действиями, например см. рис. 13.

4.9. Настройки

Для сохранения настроек между запусками среды разработки классы `GeneratorSettings` и `ProjectSettings` реализуют интерфейс `Persistent`

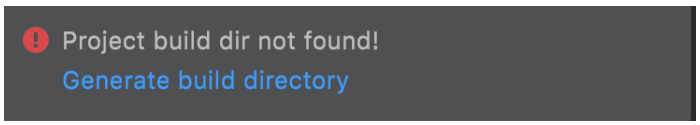

Рис. 13: Пример сообщения, когда не получилось настроить проект

\—`State`\—`Component<T>` [3], при этом IDE сама считывает данные из класса и сереализует его, сохраняя в xml файл. Также требуется, чтобы сами классы являлись сервисами (service [4]) – IDE следит за тем, чтобы загружался только один экземпляр класса.

4.10. Визард

Для того чтобы пользователю было легче настроить плагин, была поставлена задача реализовать пошаговую настройку проекта. Плагин Visual Studio Code для отображения текста использует html. Было принято решение переиспользовать html, чтобы плагины имели общий вид. На рис. 14 показаны классы отвечающие за визард.

UTBotWizard создает UTBotStep и располагает их в нужном порядке. Step – это каждый отдельный шаг в настройке, у него есть своя панель, и после того как пользователь переходит на следующий шаг, вызывается метод commit. UTBotStep - общий класс для шагов в настройке плагина. Каждый наследник реализует метод init, который инициализирует UI и возможно вызывает метод addHtml() для добавления к панеле html компонента.

На рис. 15, 16, 17 показаны некоторые шаги в визарде.

4.11. Интеграционные тесты

Для проверки взаимодействия с сервером были написаны интеграционные тесты. Тесты в IJ Plantform проводятся в среде без UI, с использованием реальных компонентов. Для написания тестов использовался фреймворк JUnit5, поскольку у него довольно подробаня документация и активная поддержка. Также использовалось непосредственно api IntelliJ Platform для написания тестов (подробнее про данное api

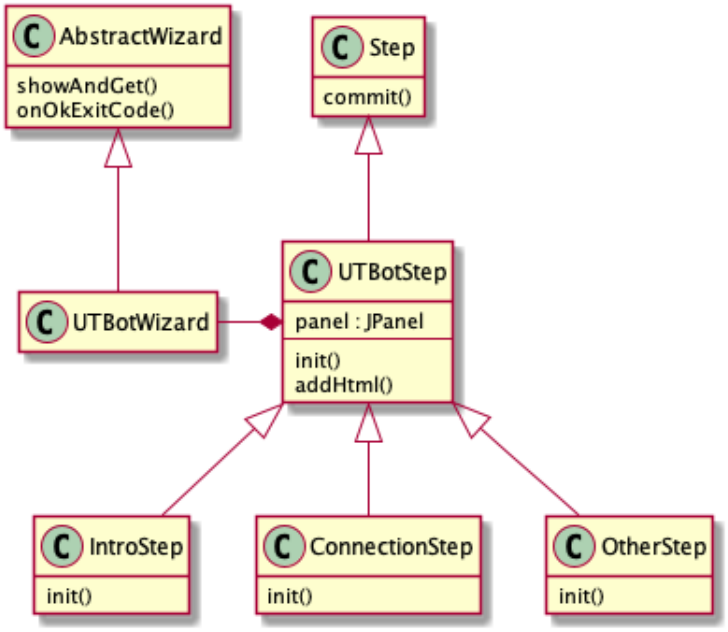

Рис. 14: Классы отвечающие за визард

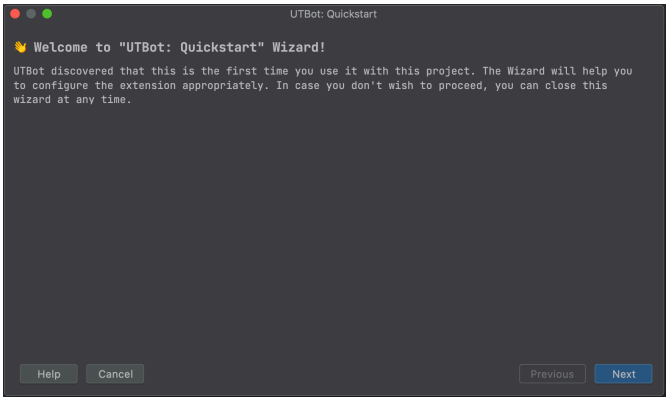

Рис. 15: Начальный шаг. Текст данного и других шагов был взят из плагина для VSCode

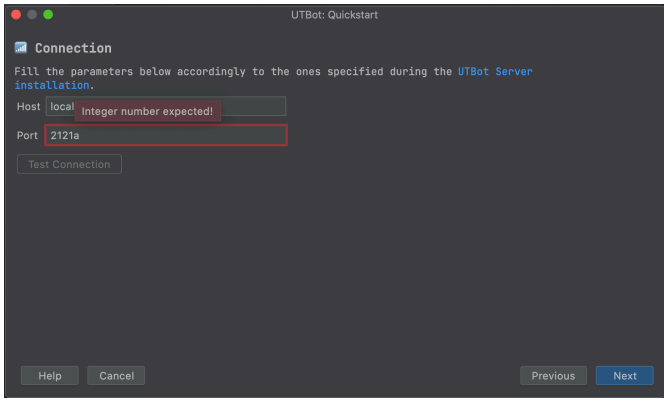

Рис. 16: Настройка подключения к серверу

Рис. 17: Завершение настройки

см. [8]):

- CodeInsightTestFixture - создает тестовый проект
- TmpDirTestFixture - предоставляет директорию, в которой лежит тестовый проект. Стандартная реализация использует временную директорию и копирует туда файлы проекта. Чтобы избежать копирования была написана другая реализация TestFixtureProxy, который использует директорую внутри проекта UTBotCpp/integration-tests

Также методы setUp, tearDown вызываются по одному разу на класс. В данных методах происходит соотвественно создание и освобождение CodeInsightTestFixture, что является затратной операцией (см. [5]).

Тесты были встроенны в CI проекта UTBotCpp.

5. Заключение

В ходе учебной практики было сделано следующее: [3]:

- Предложена архитектура плагина.
- Реализован пользовательский интерфейс для:
  - Настроек плагина
  - Запуска генерации тестов
  - Отображение прогресса при генерации тестов
  - Отображения покрытия
  - Панели для выбора таргета (библиотеки или исполняемого файла, от которого зависит генерация тестов).
  - Графические элементы для запуска тестов в редакторе
  - Окно вывода для отображения логов сервера и клиента пользователю
  - Визард, который помогает настроить плагин при первом использовании
  - Прочие элементы пользовательского интерфейса: статус подключения к серверу, уведомления по завершении генерации тестов и др.
- Реализовано взаимодействие с сервером UTBotCpp через gRPC, поддержаны все удаленные процедуры, предлагаемые сервером. Также была реализована обработка данных, которые поступают с сервера.
- Были написаны интеграционные тесты, проверяющие корректность взаимодействия с сервером. Данные тесты были добавлены в CI проекта UTBotCpp.

[3] https://github.com/UnitTestBot/UTBotCpp/pull/103 – ссылка пуллреквест с плагином. Юзернейм под которым велась работа: vol0n.

Список литературы

[1] C++ developer ecosystem state. — URL: https://www.jetbrains.com/lp/devecosystem-2020/cpp/.

[2] Extension points. — URL: https://plugins.jetbrains.com/docs/intellij/plugin-extensions.html (online; accessed: 2022-02-06).

[3] IntelliJ Platform PersistingStateComponent. — URL: https://plugins.jetbrains.com/docs/intellij/persisting-state-of-components.html (online; accessed: 2021-14-12).

[4] IntelliJ Platform Service. — URL: https://plugins.jetbrains.com/docs/intellij/plugin-services.html (online; accessed: 2021-14-12).

[5] Light and heavy tests. — URL: https://plugins.jetbrains.com/docs/intellij/light-and-heavy-tests.html (online; accessed: 2022-02-06).

[6] PSI. — URL: https://plugins.jetbrains.com/docs/intellij/psi.html (online; accessed: 2021-14-12).

[7] Showing progress in IntelliJ Platform. — URL: https://plugins.jetbrains.com/docs/intellij/general-threading-rules.html#background-processes-and-processcanceledexception (online; accessed: 2021-14-12).

[8] Testing plugins. — URL: https://plugins.jetbrains.com/docs/intellij/testing-plugins.html (online; accessed: 2022-02-06).

[9] Tool windows. — URL: https://plugins.jetbrains.com/docs/intellij/tool-windows.html (online; accessed: 2022-02-06).

[10] gRPC server streaming rpc. — URL: https://grpc.io/docs/what-is-grpc/core-concepts/#server-streaming-rpc (online; accessed: 2021-14-12).

[11] util.proto, testgen.proto. — URL: https://github.com/UnitTestBot/UTBotCpp/tree/main/server/proto (online; accessed: 2021-14-12).

[12] Документация IntelliJ Platform. — URL: https://plugins.jetbrains.com/docs/intellij/welcome.html (online; accessed: 2021-14-12).

[13] Исходный код IntelliJ-Community. — URL: https://github.com/JetBrains/intellij-community (online; accessed: 2021-14-12).

[14] Плагин для VSCode. — URL: https://github.com/UnitTestBot/UTBotCpp/tree/main/vscode-plugin (online; accessed: 2021-14-12).

[15] Создание проекта через devkit. — URL: https://plugins.jetbrains.com/docs/intellij/using-dev-kit.html (online; accessed: 2021-14-12).

[16] Создание проекта через gradle. — URL: https://plugins.jetbrains.com/docs/intellij/gradle-build-system.html (online; accessed: 2021-14-12).

[17] Таргеты в CMake. — URL: https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html#id15 (online; accessed: 2021-14-12).

[18] Удалённый вызов процедур. — URL: https://ru.wikipedia.org/wiki/Удалённый_вызов_процедур (online; accessed: 2021-14-12).

[19] Шаблон для создания проекта. — URL: https://plugins.jetbrains.com/docs/intellij/github-template.html (online; accessed: 2021-14-12).