Санкт-Петербургский государственный университет

Кафедра Системного Программирования

Группа 22.Б10-мм

Рефакторинг кода средствами Clang

Сергеев Игнатий Федорович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
Старший преподаватель кафедры ИАС К. К. Смирнов

Санкт-Петербург
2024

Оглавление

Введение **3**

1. Постановка задачи **4**

2. Обзор **5**

2.1. Требуемые преобразования 5

2.2. Критерии оценки инструментов рефакторинга 8

2.3. Описание Clang 8

2.4. Сравнение инструментов Clang для рефакторинга 9

2.5. Описание Clang-Refactor 10

3. Реализация **12**

3.1. Базовое требование необходимых рефакторингов 12

3.2. Интеграция методов преобразований из Clang-Transformer в Clang-Refactor 13

3.3. Действия необходимых рефакторингов 15

4. Тестирование **17**

4.1. Тестирование Clang-Refactor 17

4.2. Поддержка опции локации в тестировании 17

4.3. Тесты 18

5. Заключение **19**

Список литературы **20**

Введение

Область интегрированных средств разработки широко развивается, однако на рынке IDE собрались в ниши, которые специализированы на конкретные нужды конечных пользователей. Тем не менее некоторым пользователям недостаточно функций, которые предлагают существующие IDE, и им нужен подход, ориентированный на их требования.

Одной из таких групп пользователей являются заказчики IDE от компании ООО Софтком. Специально для их нужд была разработана IDE на базе Eclipse. Описанная среда разработки содержит множество функций, но одной из основных недостающих является рефакторинг шаблонов кода.

В рамках этой функции необходимо проверять код на предложенные заказчиками ситуации и локально для каждого участка предлагать его полуавтоматическое исправление пользователю. Проверка кода уже реализована внутри IDE, а исправление участков — нет.

Стоить отметить, что IDE ориентирована на работу с инструментами Clang [4], в связи с чем требуется выполнять рефакторинги средствами этого проекта.

Добавление авто-рефакторингов делится на три основные задачи — выбор инструмента, добавление в него требуемых заказчиками рефакторингов и интеграция этого инструмента с IDE. В рамках данной практики будут рассматриваться первые две задачи. Третья задача реализуется другим участником проекта.

1. Постановка задачи

Цель работы — реализовать автоматические изменения кода, используя инфраструктуру Clang. Для выполнения этой цели были поставлены следующие задачи:

1. провести обзор проекта Clang, его инструментов рефакторинга кода и выбрать подходящий;
2. дополнить проект необходимыми рефакторингами;
3. протестировать написанные трансформации.

2. Обзор

2.1. Требуемые преобразования

Заказчики требуют большой список преобразований, но в данной работе будут реализованы следующие:

2.1.1. RedundandIf

```
int a = 1;
if (true) {
  int b = 1;
}
```

преобразуется в

```
int a = 1;
{
    int b = 1;
}
```

2.1.2. RedundandWhile

```
int a = 1;
while (false) {
  int b = 1;
}
```

преобразуется в

```
int a = 1;
```

2.1.3. UnusedField

```
struct Str {
    int field;
    int unusedField;
};

void func() {
    Str obj;
    obj.field = 1;
}
```

преобразуется в

```
struct Str {
    int field;
};

void func() {
    Str obj;
    obj.field = 1;
}
```

2.1.4. UnusedMethod

```
struct Str {
    int method() {
        return 1;
    };
    int unusedMethod() {
        return 1;
    };
```

```
};

void func() {
    Str obj;
    int a = obj.method();
}
```

преобразуется в

```
struct Str {
    int method() {
        return 1;
    };
};

void func() {
    Str obj;
    int a = obj.method();
}
```

2.1.5. SimplifyExpr

```
void func() {
    bool f = false;
    bool a = f && true;
}
```

преобразуется в

```
void func() {
    bool f = false;
```

```
3     bool a = true;
4 }
```

2.2. Критерии оценки инструментов рефакторинга

Требуется выбрать инструмент, который будет использоваться для рефакторинга. В связи с этим будет рассматриваться выполнение следующих требований:

- простота добавления рефакторингов;
- возможность добавления произвольных проверок кода перед применением рефакторинга;
- проведение минимального количества встроенных проверок перед применением рефакторинга.

2.3. Описание Clang

Clang [4] — проект, содержащий инструменты работы с кодом написанном на C/C++/Obj C. Он является частью проекта инфраструктуры для создания компиляторов LLVM [7].

Clang содержит следующие инструменты для рефакторинга кода:

- Clang-Tidy [10];
- Clang-Rename [9];
- Clang-Refactor;
- Clang-Transformer.

2.3.1. Clang-Rename

Clang-Rename — это инструмент проекта Clang, ориентированный на переименование символов. Он не предполагает добавление других рефакторингов, однако использует библиотеку clangToolingRefactoring,

которая позволяет делать произвольные проверки перед изменением кода. Также этот инструмент не производит лишних проверок.

2.3.2. Clang-Transformer

Clang-Transformer — это инструмент для изменения шаблонов кода, в нем очень просто добавлять новые рефакторинги, используя абстракции ASTMatcher и ASTEdit, однако он не поддерживает добавление произвольных проверок перед изменением кода.

2.3.3. Clang-Tidy

Clang-Tidy — это статический анализатор проекта Clang. Он поддерживает добавление преобразований через модификацию специализированной библиотеки, и через плагины. Также позволяет писать произвольные проверки перед изменениями, однако содержит проверки, которые невозможно отключить [5].

2.3.4. Clang-Refactor

Clang-Refactor — это инструмент проекта Clang, специализирующийся на выполнении рефакторингов. Данный инструмент использует библиотеку clangToolingRefactoring, которая поддерживает произвольные проверки перед преобразованиями. Также clang-refactor не производит лишних проверок, но добавление новых рефакторингов заметно сложнее по сравнению с другими инструментами.

2.4. Сравнение инструментов Clang для рефакторинга

Как видно из таблицы 1, ни один из инструментов полностью не соответствует требованиям, однако Clang-Refactor имеет наибольшее соответствие. По этой причине он и был выбран. Однако простота расширения Clang-Transformer также была отмечена, и по этой причине было решено интегрировать инструмент этих расширений в Clang-Refactor.

Критерий	Clang-Refactor	Clang-Tidy	Clang-Transformer	Clang-Rename
Расширяемость	+-	+	+	-
Произвольные проверки	+	+	-	+
Отсутствие доп проверок	+	-	+	+

Таблица 1: Сравнение инструментов

2.5. Описание Clang-Refactor

Инструмент Clang-Refactor реализует преобразования через специальный движок — RefactoringEngine [11] из библиотеки clangToolingRefactoring.

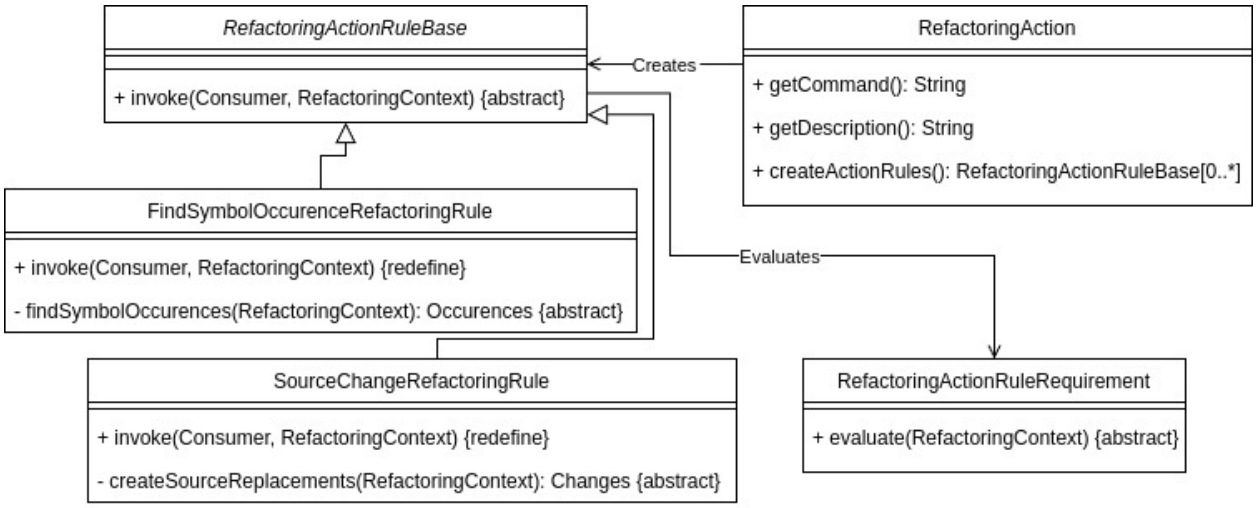

Рис. 1: Архитектура RefactoringEngine

RefactoringEngine состоит из 3-х основных компонент:

- действий;
- правил;
- требований.

Самая большая единица абстракции — это действие(наследник класса RefactoringAction на рисунке 1). Оно и представляет собой преобразование. При вызове инструмента из командной строки в первую очередь указывается имя действия.

Действие определяет последовательность правил(наследников классов FindSymbolOccurencesRefactoringRule и SourceChangeRefactoringRule на рисунке 1), которые нужно применить для выполнения рефакторинга.

И минимальная единица — это требование(наследник класса RefactoringActionRuleRequirement на рисунке 1). Это абстракция необходимого параметра, который нужен для выполнения правила. Например, этот параметр может быть значением аргумента командной строки. Выполняемые требования определяются после вызова инструмента, но до вызова действия, так как при невыполнении требования, преобразование может быть нереализуемо.

3. Реализация

3.1. Базовое требование необходимых рефакторингов

Как можно увидеть на рисунке 1, выполнение требований выполняется в некотором контексте, этот контекст должен содержать все параметры, которые пользователь передает в инструмент.

Задача требует указание места в котором необходимо выполнить преобразование. Для этого в контекст была добавлена абстракция локации (изменения можно увидеть на рисунке 3), дополнительно был модифицирован интерфейс командной строки Clang-Refactor — была добавлена опция location (изменения можно увидеть на рисунке 2).

И наконец было добавлено само требование (рисунок 4).

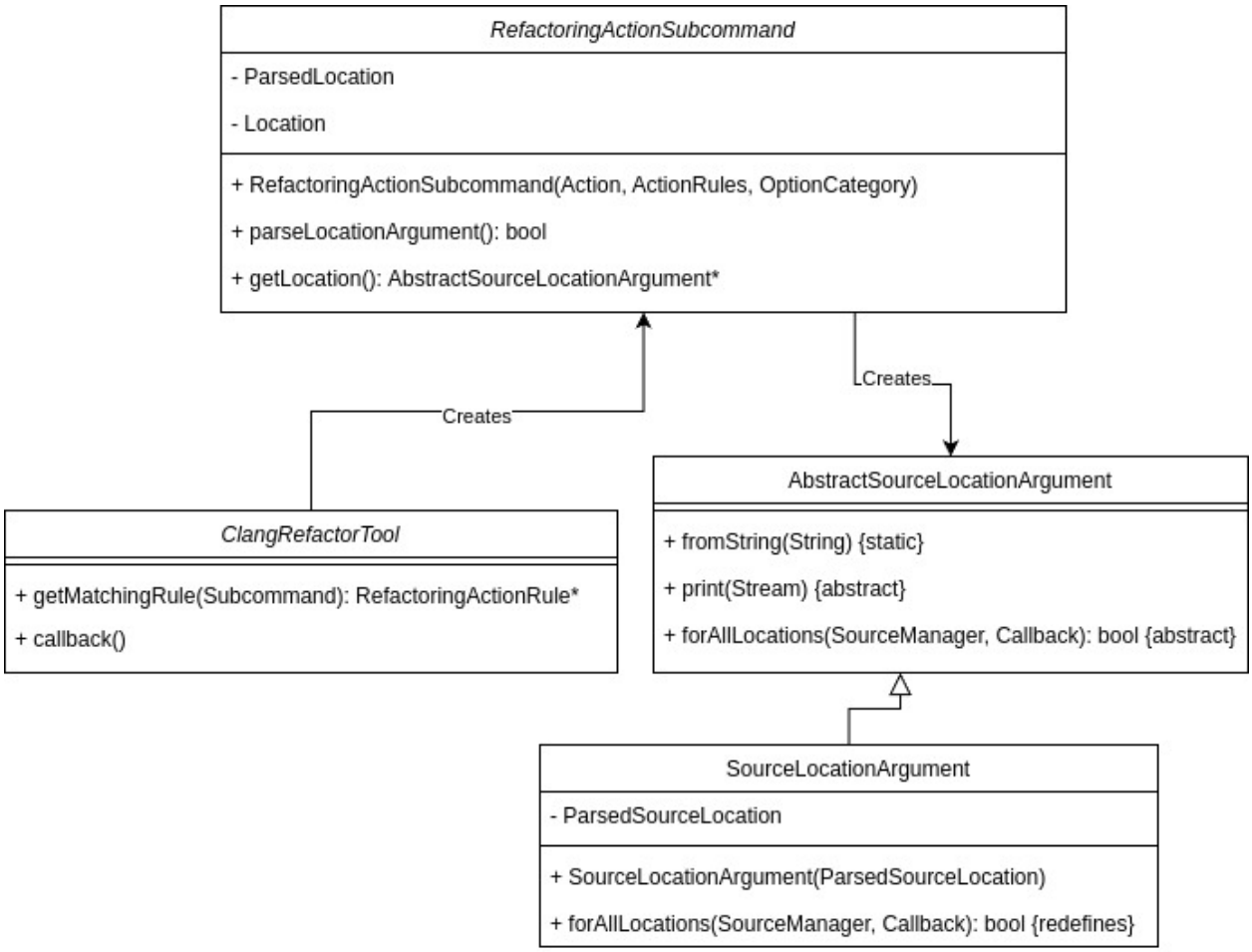

Рис. 2: Добавление опции location

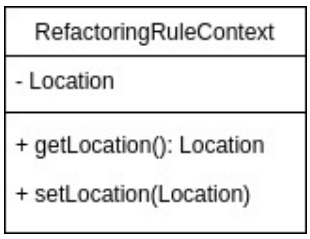

Рис. 3: Изменения в контексте

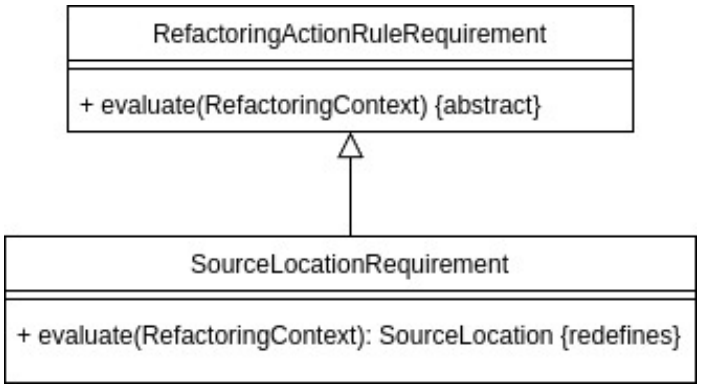

Рис. 4: Добавление требования

3.2. Интеграция методов преобразований из Clang-Transformer в Clang-Refactor

Инструмент Clang-Transformer использует специальные абстракции для поиска и замены поддеревьев AST - AST Matcher [3] и AST Edit [1]. Они очень удобные для добавления простых рефакторингов, поэтому использование этих абстракции было решено добавить в правила RefactoringEngine.

3.2.1. Пример AST Matcher

ifStmt(varDecl().bind(”var”)) — используется для нахождения условий в теле которых декларируется переменная.

3.2.2. Пример AST Edit

changeTo(node(”var”), cat(””)) — используется для удаления найденной в предыдущем AST Matcher декларации.

3.2.3. Требования для AST Matcher и AST Edit

Для использования AST Matcher и Edit в правилах, были добавлены требования предоставляющие их, они представлены на рисунке 5. Стоит отметить, что требование для AST Matcher наследовалось от SourceLocationRequirement для интеграции локализации проверок. Также вместо AST Edit было использовано его обобщение AST EditGenerator [2], которое позволяет использовать более сложные выражения.

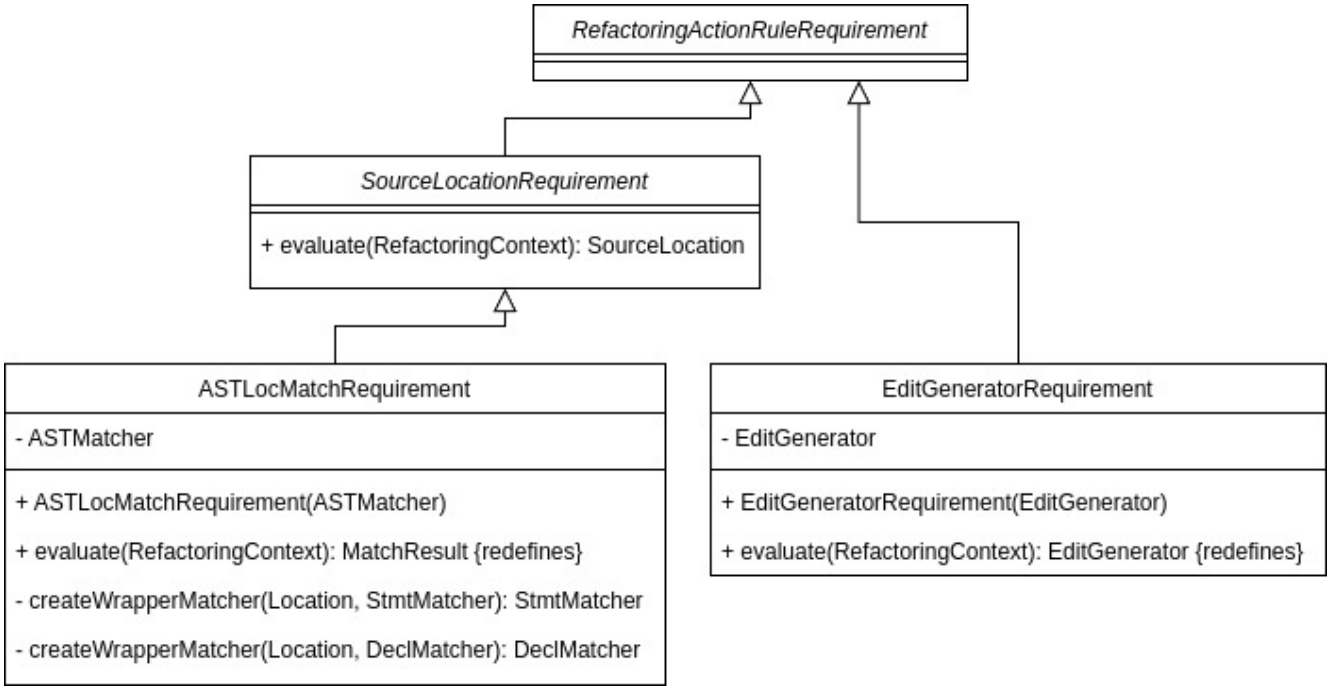

Рис. 5: Требования для AST Matcher и AST Edit

3.2.4. Правило использующее AST Matcher и AST Edit

Для использования AST Matcher и Edit в трансформациях было добавлено правило, которое генерирует изменения исходного кода согласно EditGenerator и MatchResult (рисунок 6).

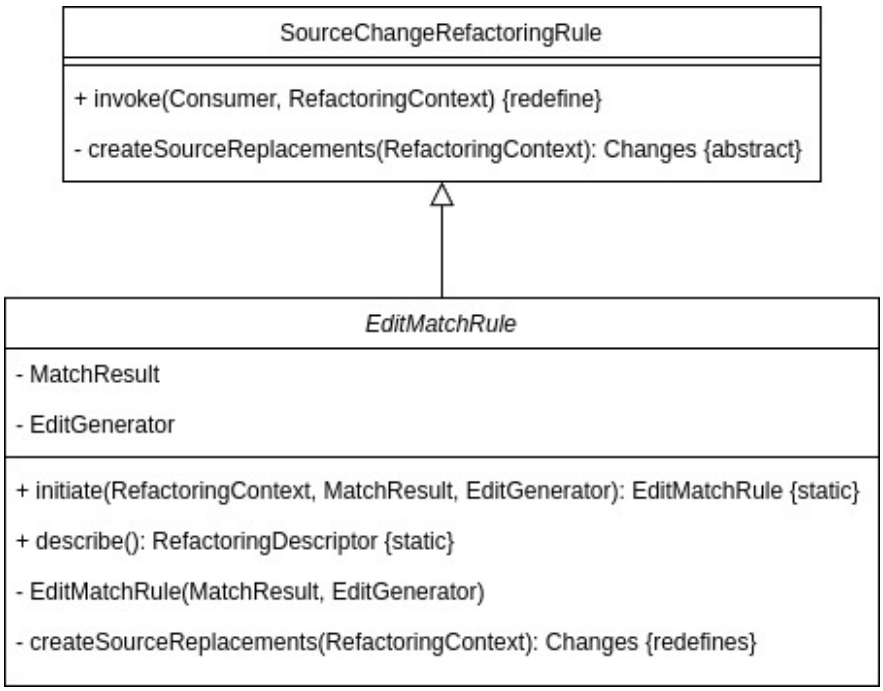

Рис. 6: Правило EditMatch

3.3. Действия необходимых рефакторингов

После добавления правила EditMatch все требуемые рефакторинги стали легко реализуемы. Для них были добавлены следующие действия:

- redundand-if — разворачивает if, расположенный вокруг данной локации, если условие принимает всегда одно значение(истинное или ложное);
- redundand-while — убирает while, расположенный вокруг данной локации, если условие всегда ложно;
- simplify-expr — упрощает бинарное выражение с операторами && и ||, расположенное вокруг данной локации;
- unused-field — убирает поле, расположенное в данной локации, если оно не нигде используется;
- unused-method — убирает метод, расположенный вокруг данной локации, если он не нигде используется.

Отличия между действиями только в используемых AST Matcher и AST

EditGenerator при создании правил, поэтому на рисунке 7 приведен пример одного рефакторинга.

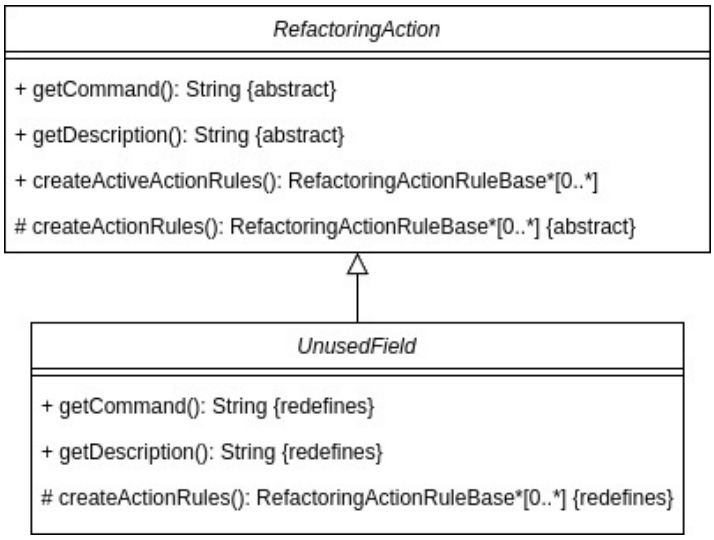

Рис. 7: Действие unused-field

4. Тестирование

4.1. Тестирование Clang-Refactor

В проекте LLVM основное тестирование Clang-Refactor проводится через интеграционные тесты с использованием LIT(LLVM Integrated Tester) [8] и FileCheck [6]. К коду для тестирования добавляются комментарии команд LIT для запуска, и FileCheck для корректности вывода.

4.2. Поддержка опции локации в тестировании

Использование в LIT обычных команд запуска Clang-Refactor с нетривиальными опциями, например селекции в коде, требует явного указания локации в исходном файле. Для решения этого неудобства в случае опции селекции в инструменте Clang-Refactor добавлена возможность указывать её через комментарии, разбивать на группы, и исполнять каждую группу по отдельности с помощью специального потребителя. Для опции локации были аналогично добавлены эти возможности(рисунок 8).

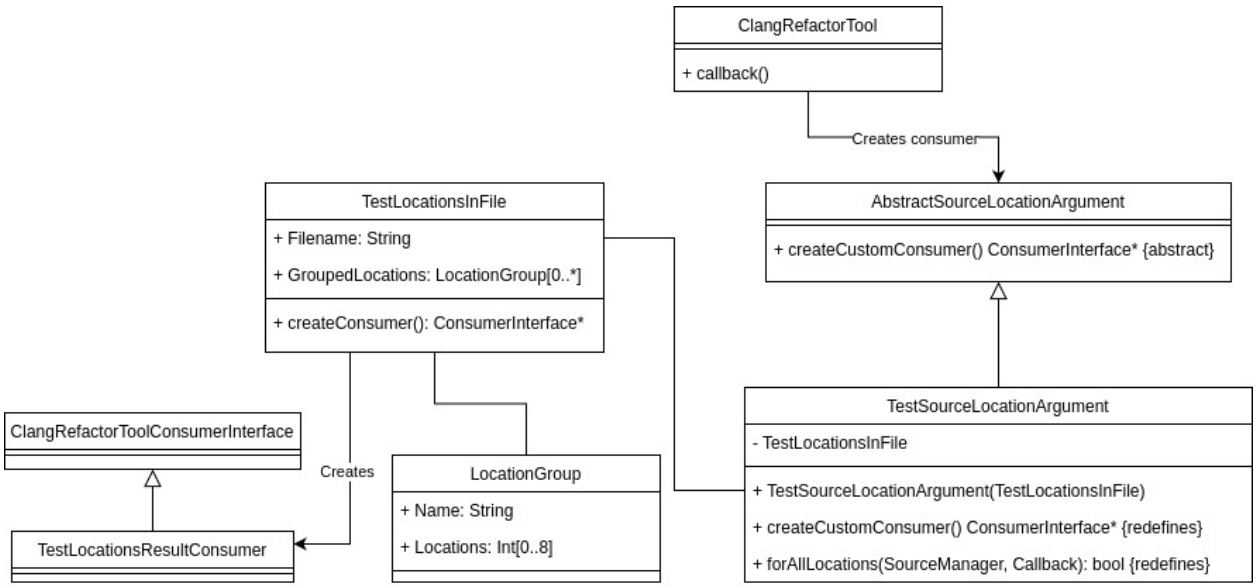

Рис. 8: Изменение в поддержке тестов

4.3. Тесты

После добавления поддержки указания опции локации в комментариях написание тестов заключается в написании тестируемого кода, указания в нем через комментарии локаций для тестирования, указания комментария запуска команды clang-refactor с помощью LIT, и описание ожидаемого вывода через комментарии FileCheck. Ниже представлен пример тестов для рефакторинга unused-field.

```
// RUN: clang-refactor unused-field -location=test:%s %s -- 2>&1
// RUN: | grep -v CHECK | FileCheck %s

struct Bar {
    /*loc used=*/int member;
    /*loc unused=*/int unusedMember;
};

void foo() {
    Bar var;
    var.member = 1;
}

// CHECK: 1 'unused' results:
// CHECK: struct Bar {
// CHECK-NEXT: /*loc used=*/int member;
// CHECK-NEXT: /*loc unused=*/
// CHECK-NEXT: };

// CHECK: 1 'used' results:
// CHECK-NEXT: refactoring action can't be initiated without a matching location
```

5. Заключение

В результате проведения данной практики были выполнены следующие задачи:

- был проведен обзор проекта Clang и его инструментов для рефакторинга;
- проект был дополнен необходимыми рефакторингами;
- написанные трансформации были протестированы.

Код изменений доступен в GitHub репозитории в следующих пул реквестах:

- https://github.com/IgnatSergeev/llvm-project/pull/3;
- https://github.com/IgnatSergeev/llvm-project/pull/4;
- https://github.com/IgnatSergeev/llvm-project/pull/5.

Список литературы

[1] AST Edit. — URL: https://releases.llvm.org/14.0.0/tools/clang/docs/ClangTransformerTutorial.html#edits (дата обращения: 1 июня 2024 г.).

[2] AST EditGenerator. — URL: https://releases.llvm.org/14.0.0/tools/clang/docs/ClangTransformerTutorial.html#editgenerators-advanced (дата обращения: 1 июня 2024 г.).

[3] AST Matcher. — URL: https://clang.llvm.org/docs/LibASTMatchersReference.html (дата обращения: 1 июня 2024 г.).

[4] Clang. — URL: https://clang.llvm.org/ (дата обращения: 1 июня 2024 г.).

[5] Clang-Tidy core checks. — URL: https://clang.llvm.org/docs/analyzer/checkers.html#core (дата обращения: 1 июня 2024 г.).

[6] FileCheck. — URL: https://llvm.org/docs/CommandGuide/FileCheck.html (дата обращения: 1 июня 2024 г.).

[7] LLVM. — URL: https://llvm.org/ (дата обращения: 1 июня 2024 г.).

[8] LLVM Integrated Tester. — URL: https://llvm.org/docs/CommandGuide/lit.html (дата обращения: 1 июня 2024 г.).

[9] Team The Clang. Clang-Rename. — URL: https://clang.llvm.org/extra/clang-rename.html (дата обращения: 1 июня 2024 г.).

[10] Team The Clang. Clang-Tidy. — URL: https://clang.llvm.org/extra/clang-tidy/ (дата обращения: 1 июня 2024 г.).

[11] Team The Clang. Refactoring engine. — URL: https://clang.llvm.org/docs/RefactoringEngine.html (дата обращения: 1 июня 2024 г.).