

PitestCracker

Рядом с файлом, ну или в комплекте к файлу, идёт два архива - **PitestCrackLastTry** и **PyParserImplement**. Пойдём по порядку

PitestCrackLastTry

В этом архиве лежат файлы и попытки пересобрать, собственно, Pitest. Попытка заключалась в следующем - написать какой-то свой класс-мутатор и подписать его в какой-то конфигурационный файл. В качестве кастомного класса был написан (хотя это громко сказано) класс **CustomMutator**. По сути это просто копия какого-то существовавшего класса, но с другим названием, однако на первых попытках кастомизации класс обладал уникальным поведением, однако я от этого ушёл, потому что для корректной работы нужны были какие-то особые методы/поля для нормальной работы. На этой стадии было всё равно на функционал мутатора, так как нужно было понять, что класс вообще подгружается и работает, а уникального названия для этого более чем достаточно.

```
.
├── pitest-jar
│   └── pitest-dev-SNAPSHOT.jar
├── pom.xml
├── src
│   ├── main
│   │   └── java
│   │       └── ru
│   │           └── choomandco
│   │               └── pitest
│   │                   ├── Triangle.java
│   │                   ├── TriangleM1.java
│   │                   ├── TriangleM2.java
│   │                   ├── TriangleM3.java
│   │                   ├── TriangleM4.java
│   │                   ├── TriangleM5.java
│   │                   ├── TriangleM6.java
│   │                   └── TriangleM7.java
│   └── test
└── target
    ├── classes
    └── generated-test-sources
```

```
├─ jacoco.exec
├─ maven-status
├─ surefire-reports
├─ test-classes
│   └─ ru
│       └─ choomandco
│           └─ pitest
│               ├─ TriangleM1Test.class
│               ├─ TriangleM2Test.class
│               ├─ TriangleM3Test.class
│               ├─ TriangleM4Test.class
│               ├─ TriangleM5Test.class
│               ├─ TriangleM6Test.class
│               ├─ TriangleM7Test.class
│               └─ TriangleTest.class
```

В древе выше видна краткая структура проекта. Файлы с названием **Triangle__Test.class** определенным образом покрывают соответствующие классы **TriangleTest.class**. Именно относительно этих классов производится мутационное тестирование.

Вообще стоит вернуться к цели данной работы - добавить новый мутатор. В файле **pitest-jar**

/pitest-dev-SNAPSHOT.jar

лежит байт-код "Питеста". Если покопаться в декомпилированных папках/файлах можно найти директорию **/org/pitest/mutationtest/engine/gregor/mutators**, где лежит какое-то количество каких-то мутационных классов, в том числе и **CustomMutator**.

Следующим шагом после добавления класса стало добавление этого класса в перечень остальных мутаторов в файле **MethodMutatorFactory**. Сделано это было по наитию, так как если тут написаны названия всех файлов, то нужно добавить и наш новый мутатор. Однако сейчас, буквально перед написанием этого текста, я нашёл ещё один перечень мутаторов в классе **/mutationtest/engine/gregor/config/Mutator.class** происходит ещё один перечень классов, но их я не трогал, лишь обнаружил.

Однако, как мне помнится, эти попытки не увенчались успехом, возможно стоит ещё раз попробовать заменить исходный плагин нашим модифицированным.

PyParserImplement

Возможности этого проекта я заново описывать не буду, просто скопирую старый README.

Этот проект содержит пример использования библиотеки Pitest для генерации мутаций и тестирования на Java.

Запуск проекта

Для запуска проекта вам потребуется Maven и Java JDK 13 версии.

1. Убедитесь, что у вас установлены Java JDK-13 и Maven.
2. Соберите проект с помощью Maven: `mvn clean install`

Запуск библиотеки Pitest

Библиотека Pitest используется для генерации мутаций и проверки эффективности написанных модульных тестов.

1. Убедитесь, что проект собран с помощью Maven.
2. Для запуска Pitest выполните следующую команду: `mvn org.pitest:pitest-maven:mutationCoverage`

Эта команда запустит генерацию мутаций и сформирует отчет. Результаты будут доступны в формате XML и HTML в каталоге `target/pit-reports`.

В случае ошибки запуска мутационного тестирования

Если после вызова команды из пункта 2 произошли ошибки в терминале:

1. Перейдите в директорию `/test`, в класс `RangeValodator`.
2. ПКМ по классу (или по каждому unit-тесту), далее `More Run/Debug` → `Modify Run Config`.
3. В окне справа находится выпадающая вкладка `Modify Options`, необходимо добавить параметр `Enable branch coverage and test tracking`.
4. Далее из пункта 2 в `More Run/Debug` необходимо запустить тестирование `with Coverage`.
5. Пропишите команду в терминале из раздела "Запуск библиотеки Pitest", пункт 2, или найдите команду во вкладке Maven (Plugins → pitest → `pitest:mutationCoverage`).

Структура проекта

- `src/main/java` : Исходный код проекта.
- `src/test/java` : Модульные тесты.

- `pom.xml`: Файл конфигурации Maven, включая зависимости и настройки плагинов.

Пример использования

Приведенная ниже функция `isValid` проверяет, находится ли входное число в заданном диапазоне.

```
class RangeValidator {
    boolean isValid(int input) {
        return input > 0 && input <= 100;
    }
}
```

Модульные тесты для этой функции расположены в классе `RangeValidatorTest`.

```
class RangeValidatorTest {

    private RangeValidator cut;

    @BeforeEach
    void setUp() {
        cut = new RangeValidator();
    }

    @Test
    @DisplayName("Should return true given 50")
    void fifty_isValid_returnsTrue() {
        assertThat(cut.isValid(50)).isTrue();
    }

    @Test
    @DisplayName("Should return false given 200")
    void twoHundred_isValid_returnsFalse() {
        assertThat(cut.isValid(200)).isFalse();
    }

    @Test
    @DisplayName("Should return true given 100")
    void hundred_isValid_returnsTrue() {
        assertThat(cut.isValid(100)).isTrue();
    }
}
```

```
@Test
@DisplayName("Should return false given 0")
void zero_isValid_returnsFalse() {
    assertThat(cut.isValid(0)).isFalse();
}
}
```