Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Генерация модульных тестов с конкретизацией типов для Spring-специфичных приложений

ШИШИН Кирилл Александрович

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2023

Оглавление

1. Введение

Тестирование программного обеспечения является его важной и неотъемлемой частью. Притом, что написанные руками тесты зачастую покрывают довольно маленький процент путей исполнения программы, само их написание отнимает много сил и времени у программистов и тестировщиков. В связи с этим последние десятилетия активно разрабатываются решения для автоматизации тестирования, которые призваны помочь существенно повысить покрытие программы тестами, многократно снизив время, потраченное на их написание. Довольно известным является эксперимент [2] над проектом Coreutils [3], показывающий насколько эффективной и полезной может быть автоматизация тестирования (с ее помощью удалось значительно увеличить покрытие кода тестами и узнать о множестве багов, некоторые из которых существовали более пятнадцати лет). Актуальность темы подтверждается в том числе и ежегодно проводимыми соревнованиями для автоматических генераторов модульных тестов [5].

Среди автоматических генераторов, показывающих хорошие[1] результаты на соревнованиях [14], хотелось бы выделить проект с открытым исходным кодом – UnitTestBot Java [15], разрабатываемый компанией Huawei в Saint-Petersburg Research Center. Это Command Line Tool и плагин для IntelliJ IDEA [4], предназначенный для генерации модульных тестов для Java приложений. В основе проекта лежит две техники анализа кода: символьное исполнение и фаззинг.

UnitTestBot Java умеет генерировать тесты для “чистой” Java, но как известно, для этого языка написано множество фреймворков и библиотек, накладывающих дополнительные требования к анализу пользовательского кода и виду генерируемых тестов. Один из самых популярных фреймворков на сегодняшний день [19] – Spring [12]. Он используется при разработке большинства Java проектов. В связи с этим важно, чтобы UnitTestBot Java умел генерировать тесты для приложений, использующих этот фреймворк.

[1] по проценту покрытия путей исполнения программы тестами и читаемости сгенерированных тестов

Spring – многообразный инструмент. Однако его главными задачами считаются – Dependency Injection и Inversion of Control. В Spring есть контейнер DI/IoC, хранящий в себе управляемые объекты – бины. Конфигурация контейнера может как определяться на основе аннотаций в пользовательском коде, так и задаваться посредством специальных конфигурационных классов и xml файлов. Кроме того, в фреймворке есть механизм реализации паттерна MVC, подходы к тестированию сервисов и контроллеров которого, как правило, отличаются. Все это делает автоматизированное тестирование Spring-приложений весьма трудоемкой задачей.

UnitTestBot Java уже умеет генерировать какие-то тесты для Spring-приложений, однако в этих тестах, как и при тестировании обычных Java приложений, мокируется все, что находится вне тестируемого класса или пакета (в зависимости от выбранной стратегии мокирования). Такой подход к тестированию нельзя назвать Spring-специфичным, так как тестирование Spring-приложений подразумевает использование меньшего количества моков благодаря информации, полученной в ходе анализа пользовательской конфигурации приложения.

Рассмотрим пример генерации тестов для Spring-приложения с помощью UnitTestBot Java и посмотрим на Spring-спеицифичный тест, который хотелось бы генерировать:

Листинг 1: интерфейс Animal и его реализация

```
public interface Animal {
    String getSpecies();
}

public class Cat implements Animal {
    public String getSpecies() {
       return "cat";
    }
}
```

Листинг 2: тестируемый сервис

```
@Service
public class SpeciesService {
    @Autowired @Setter
    private Animal animal;

    public String getSpecies() {
        return animal.getSpecies();
    }
}
```

Пусть пользовательское приложение сконфигурировано следующим образом:

Листинг 3: конфигурация приложения

```
public class AnimalConfiguration {
    @Bean
    public Animal animal(){
        return new Cat();
    }
}
```

Посмотрим на тест, сгенерированный с помощью UnitTestBot Java для метода getSpecies() из класса SpeciesService:

Листинг 4: тест для getSpecies(), сгенерированный с помощью UnitTestBotJava

```
public void testGetSpecies() {
    SpeciesService speciesService = new SpeciesService();
    Animal animalMock = mock(Animal.class);
    (when(animalMock.getSpecies())).thenReturn(((String) null));
    speciesService.setAnimal(animalMock);
    String actual = speciesService.getSpecies();
    assertNull(actual);
}
```

Данный тест корректен, однако исходя из наших знаний о конфигурации приложения, более отражающим реальное поведение программы был бы тест:

Листинг 5: тест для getSpecies() с конкретизацией типа

```
public void testGetSpecies() {
    SpeciesService speciesService = new SpeciesService();
    Cat animal = new Cat();
    speciesService.setAnimal(animal);
    String actual = speciesService.getSpecies();
    assertEquals("cat", actual);
}
```

Замену абстрактных типов конкретными реализациями, основанную на конфигурации приложения, мы будем называть конкретизацией типов.

Конечно, конкретизировать тип можно не всегда и не всегда необходимо это делать, однако в ряде случаев конкретизация типов позволяет получать более выразительные тесты, описывающие реальные сценарии исполнения программы, поэтому возможность генерировать тесты с конкретизацией типов является желательной опцией для автогенератора.

Также важно отметить, что поскольку модульные тесты не предполагают запуск приложения и инициализацию его контекста, так как это может быть опасно для пользовательских данных, а тестируют компонент в изоляции, то ожидается, что и в процессе генерации тестов, который включает в себя анализ пользовательской конфигурации, эти условия безопасности будут соблюдены. Это создает дополнительные трудности при решении задачи конкретизации типов.

2. Постановка задачи

Целью данной работы является поддержка генерации модульных тестов с конкретизацией типов для Spring-специфичных приложений в UnitTestBot Java

Для достижения цели были выделены следующие задачи:

1. Исследовать возможности генерации Spring-специфичных тестов в существующих инструментах автоматизации тестирования
2. Разработать механизм анализа конфигурации приложения, извлекающий информацию о бинах без инициализации контекста самого приложения
3. Модернизировать символьный движок, так чтобы он позволял:
   (a) определять, возможно ли конкретизировать тип
   (b) конкретизировать тип при необходимости
4. Реализовать механизм коммуникации между анализатором конфигурации и символьным движком
5. Покрыть тестами основные сценарии работы анализатора и включить эти тесты в CI плагина
6. Апробировать предложенную функциональность на реальных проектах с целью определения ее эффективности

3. Обзор

3.1. Существующие решения

Существует ряд инструментов, в той или иной степени решающих проблему автоматизации тестирования кода на Java. Все они используют одну или несколько основных техник анализа кода: символьное исполнение, фаззинг и AI. Самыми известными решениями с открытым исходным кодом являются EvoSuite [17] и Randoop [10]. А среди решений с закрытым исходным кодом известны Parasoft Jtest [9], Diffblue Cover [16] и Machinet [7].

Среди них только малая часть умеет генерировать тесты для Spring-приложений. Например, Parasoft Jtest генерирует только шаблоны тестов, а затем пользователю нужно самому заполнить значения аргументов тестовых методов.

Листинг 6: шаблон теста, сгенерированный Parasoft Jtest

```
@Test
public void testGetPerson() throws Throwable {
    MockedStatic<ExternalPersonService> mocked =
        mockStatic(ExternalPersonService.class);
    mocks.add(mocked);
    Person getPersonResult = null; // UTA: default value mocked.when(() ->
        ExternalPersonService.getPerson(anyInt())).
    thenReturn(getPersonResult);
    // Given
    PeopleController underTest = new PeopleController();
    // When
    int id = 1;
    Model model = mock(Model.class);
    ResponseEntity<Person> result = underTest.getPerson(id, model); // Then
    assertNotNull(result);
    assertNotNull(result.getBody());
}
```

Diffblue Cover же генерирует “реальные” тесты для Spring-приложений:

Листинг 7: тесты, сгенерированные Diffblue Cover для метода, считающего сумму квадратов чисел

```
@ContextConfiguration(classes = {CalculatorService.class})
@ExtendWith(SpringExtension.class)
class CalculatorServiceUnitTests {
    @MockBean
    private Calculator calculator;

    @Autowired
    private CalculatorService calculatorService;

    @Test
    void testCalcSquareSum() {
        when(calculator.multiply(anyInt(), anyInt())).thenReturn(5L);
        assertEquals(10L, calculatorService.calcSquareSum(3, 3));
        verify(calculator, atLeast(1)).multiply(anyInt(), anyInt());
    }

    // Another tests...
}
```

Однако, и Diffblue Cover никаких механизмов конкретизации типов не предлагает.

3.2. Технологии, используемые для разработки

Выбор стека технологий был обусловлен интеграцией решения в продукт UnitTestBot Java.

Использовались:

- Kotlin [6] – язык программирования, который был выбран для разработки в рамках курсовой работы.
- Framework rd [11] – фреймворк, который был выбран для организации межпроцессного взаимодействия между движком и анализатором Spring-специфичной информации.

Поддерживались:

- JUnit4 [18], JUnit5 [1], TestNG [13] – фреймворки для тестирования Java приложений.
- Mockito [8] – фреймворк для мокирования, который в том числе содержит специальные аннотации, удобные при тестировании Spring-приложений.

Также нельзя не рассказать подробнее о некоторых особенностях Spring-framework, которые существенны при решении поставленной задачи. Этому посвящен следующий раздел.

3.2.1. Spring-framework

Spring предоставляет огромную функциональность, позволяющую пользователю детально сконфигурировать свое приложение. Все это влечет за собой огромное количество сценариев использования фреймворка, которые должны быть поддержаны при реализации механизма анализатора конфигураций.

Конфигурация Spring-приложений предназначена для заполнения контейнера DI/IoC и происходит через специальные конфигурационные классы или XML файлы.

Посмотрим на пример настройки контейнера с помощью конфигурационного класса:

Листинг 8: конфигурационный класс

```
@Configuration
public class AppConfig {
    @Bean
    public MyInterface myInterface() {
        return new MyInterfaceImplementation();
    }
}
```

Хоть сейчас в большинстве Spring приложений для настройки контейнера DI/IoC и используются конфигурационные классы, однако

возможность конфигурировать Spring приложения таким способом появилась только начиная с Spring core 3.0. В более старых версиях были доступны только XML конфигурации, что важно принять во внимание во время реализации анализатора.

Ниже приведен пример конфигурирования Spring приложения с помощью XML:

Листинг 9: конфигурационный Xml файл

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myInterface" class="com.example.MyInterfaceImplementation"/>
</beans>
```

Также в Spring предусмотрена возможность разделения конфигурации с помощью профилей. Например: конфигурация для разработки и конфигурация для продакшена.

Ниже приведен пример использования профилей при конфигурации приложения:

Листинг 10: конфигурационный класс с использыванием профилей

```
@Configuration
public class AppConfig {
    @Bean
    @Profile("dev")
    public MyInterface myInterfaceDev() {
        return new MyInterfaceDevImplementation();
    }

    @Bean
    @Profile("prod")
    public MyInterface myInterfaceProd() {
        return new MyInterfaceProdImplementation();
    }
}
```

Кроме того, более детальная конфигурация контейнера может происходить с использованием properties или yml файлов.

Ниже приведен пример properties файла:

Листинг 11: properties файл

```
database.url=postgres:5432
database.username=postgres
database.password=pass
```

Ниже приведен пример использования properties файла в конфигурационном классе:

Листинг 12: использование properties файла в конфигурационном классе

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {
    @Value("${database.url}")
    private String databaseUrl;

    @Value("${database.username}")
    private String databaseUsername;

    @Value("${database.password}")
    private String databasePassword;
}
```

Также помимо “чистого” Spring пользователь может использовать в своем приложении Spring Boot – расширение для Spring, имеющее всю ту же функциональность, но при этом существенно упрощающее использование фреймворка, например, посредством механизма автоматической конфигурации Spring приложения. Возможность использования пользователем Spring Boot также нужно учитывать при написании анализатора конфигураций.

Заключение

В ходе выполнения учебной практики были решены следующие задачи:

1. Исследованы возможности генерации Spring-специфичных тестов в существующих инструментах автоматизации тестирования
2. Разработан механизм анализа конфигурации приложения, извлекающий информацию о бинах без инициализации контекста самого приложения
3. Модернизирован символьный движок, так чтобы он позволял:
   (a) определять, возможно ли конкретизировать тип
   (b) конкретизировать тип при необходимости
4. Реализован механизм коммуникации между анализатором конфигурации и символьным движком

Планируется решить следующие задачи:

1. Покрыть тестами основные сценарии работы анализатора и включить эти тесты в CI плагина
2. Апробировать предложенную функциональность на реальных проектах с целью определения ее эффективности

Список литературы

[1] The 5th major version of the programmer-friendly testing framework for Java and the JVM. — URL: https://github.com/junit-team/junit5 (дата обращения: 21 декабря 2023 г.).

[2] Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — OSDI'08. — USA : USENIX Association, 2008. — P. 209–224.

[3] Coreutils. — URL: https://www.gnu.org/software/coreutils/ (дата обращения: 21 декабря 2023 г.).

[4] IntelliJ IDEA – the Leading Java and Kotlin IDE. — URL: https://www.jetbrains.com/idea/ (дата обращения: 21 декабря 2023 г.).

[5] Jahangirova Gunel, Terragni Valerio. SBFT Tool Competition 2023 - Java Test Case Generation Track // 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT). — 2023. — P. 61–64.

[6] The Kotlin Programming Language. — URL: https://kotlinlang.org (дата обращения: 21 декабря 2023 г.).

[7] Machinet: AI Assistant for Developers. — URL: https://www.machinet.net (дата обращения: 21 декабря 2023 г.).

[8] Most popular Mocking framework for unit tests written in Java. — URL: https://github.com/mockito/mockito (дата обращения: 21 декабря 2023 г.).

[9] Parasoft Jtest for Java Unit Testing. — URL: https://www.parasoft.com/products/parasoft-jtest/java-unit-testing/ (дата обращения: 21 декабря 2023 г.).

[10] Randoop: Automatic unit test generation for Java. — URL: https://randoop.github.io/randoop/ (дата обращения: 21 декабря 2023 г.).

[11] Reactive Distributed communication framework for .NET, Kotlin, C++. — URL: https://github.com/JetBrains/rd (дата обращения: 21 декабря 2023 г.).

[12] Spring framework. — URL: https://spring.io (дата обращения: 21 декабря 2023 г.).

[13] TestNG testing framework. — URL: https://github.com/testng-team/testng (дата обращения: 21 декабря 2023 г.).

[14] UTBot at the SBFT 2023 Java Tool Competition / Dmitry Ivanov, Alexey Menshutin, Maxim Pelevin et al. // 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT). — 2023. — P. 68–69.

[15] UnitTestBot/UTBotJava: Automated unit test generation and precise code analysis for Java. — URL: https://github.com/UnitTestBot/UTBotJava (дата обращения: 21 декабря 2023 г.).

[16] What is Diffblue Cover? | Diffblue. — URL: https://www.diffblue.com/products/ (дата обращения: 21 декабря 2023 г.).

[17] What is EvoSuite? — URL: https://github.com/EvoSuite/evosuite (дата обращения: 21 декабря 2023 г.).

[18] A programmer-oriented testing framework for Java. — URL: https://github.com/junit-team/junit4 (дата обращения: 21 декабря 2023 г.).

[19] “Java Programming - The State of Developer Ecosystem in 2022 Infographic,”. — JetBrains: Developer Tools for Professionals and Teams. — URL: https://www.jetbrains.com/lp/devecosystem-2022/ (дата обращения: 21 декабря 2023 г.).