

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
2024

Оглавление

1. Введение	3
2. Постановка задачи	8
3. Обзор	9
3.1. Существующие решения	9
3.2. <i>UnitTestBot Java</i>	11
3.3. Технологии, используемые для разработки	12
4. Реализация	16
4.1. Анализатор конфигураций <i>Spring</i> -приложения	16
4.2. Модернизация символьного движка	18
4.3. Взаимодействие символьного движка и анализатора кон- фигураций	19
5. Апробация	20
5.1. <i>Java Blog Aggregator: Boot</i>	20
5.2. <i>Mall</i>	22
Заключение	24
Список литературы	25

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 [1] над проектом *Coreutils*¹, показывающий, насколько эффективной и полезной может быть автоматизация тестирования. Хотя проект существует уже много лет и тесты в нем разрабатываются постоянно, созданному авторами эксперимента инструменту автоматизации тестирования всего за несколько часов удалось значительно увеличить покрытие кода тестами и узнать о дефектах, некоторые из которых существовали более пятнадцати лет. Актуальность темы подтверждается в том числе и ежегодно проводимыми соревнованиями автоматических генераторов [2]. Участвующие на этих соревнованиях инструменты осуществляют тестирование реальных проектов (например, *Guava*², *Seata*³, *Spoon*⁴). В результате выявляются победители по нескольким номинациям, среди которых эффективность обнаружения дефектов в коде, процент покрытия путей исполнения программы и человекочитаемость сгенерированных тестов.

Среди автоматических генераторов, показывающих хорошие результаты на соревнованиях хотелось бы выделить проект с открытым исходным кодом *UnitTestBot Java*⁵, разрабатываемый в Huawei Russian Research Institute. В 2022 году инструмент неплохо проявил себя на

¹Coreutils. – URL: <https://www.gnu.org/software/coreutils/>

²Guava: Google Core Libraries for Java. – URL: <https://github.com/google/guava>

³Seata: Simple Extensible Autonomous Transaction Architecture. – URL: <https://github.com/apache/incubator-seata>

⁴Spoon. – URL: <https://github.com/INRIA/spoon>

⁵UnitTestBot Java: Automated unit test generation and precise code analysis for Java. – URL: <https://github.com/UnitTestBot/UTBotJava>

соревнованиях [3], а 2023 году он и вовсе занял первое место по человекочитаемости генерируемых тестов и второе место в общем зачете по всем номинациям [4]. *UnitTestBot Java* это инструмент командной строки и плагин для *IntelliJ IDEA*⁶, предназначенный для генерации модульных тестов для *Java* приложений. Методологической основой проекта являются 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 [5] — *Spring*⁷. Он используется при разработке многих *Java* проектов. В связи с этим важно, чтобы *UnitTestBot Java* умел генерировать качественные тесты для приложений, использующих этот фреймворк.

Spring — многообразный инструмент. Одними из главных его задач являются *Dependency Injection (DI)* и *Inversion of Control (IoC)*. В *Spring* есть *DI/IoC* контейнер⁸, хранящий в себе управляемые объекты — бины. Конфигурация этого контейнера может определяться различными способами: на основе аннотаций в пользовательском коде, а также посредством специальных конфигурационных классов или XML файлов. Кроме того, пользователь вправе выбрать профиль, определяющий, как именно будет конфигурироваться его приложение. Помимо этого, в фреймворке предложен механизм реализации паттерна MVC, подходы к тестированию сервисов и контроллеров которого, как правило, отличаются. Эти и другие особенности фреймворка делают автоматизированное тестирование *Spring*–приложений весьма трудоемкой задачей.

UnitTestBot Java уже умеет генерировать какие-то тесты для *Spring*–приложений. В них, как и при тестировании обычных *Java* приложений, мокируется все, что находится вне тестируемого класса или пакета (в

⁶IntelliJ IDEA – the Leading Java and Kotlin IDE. – URL: <https://www.jetbrains.com/idea/>

⁷Spring framework. – URL: <https://spring.io>

⁸The IoC container. – URL: <https://docs.spring.io/spring-framework/reference/core/beans.html>

зависимости от выбранной стратегии мокирования). Однако конфигурация *Spring*-приложения может содержать информацию о реальных типах объектов. Используя эту информацию, можно генерировать тесты, которые будут в большей степени соответствовать реальному поведению программы и тем сценариям, которые проверялись бы для этого проекта при написании тестов вручную.

Рассмотрим минималистичный пример *Spring*-приложения, информация из конфигурации которого позволяет генерировать более качественные тесты.

Пусть дан класс `SpeciesService`, у которого есть метод `getSpecies`, а также поле типа `Animal`,

```
1 @Service
2 public class SpeciesService {
3     @Autowired
4     @Setter
5     private Animal animal;
6
7     public String getSpecies() {
8         return animal.getSpecies();
9     }
10 }
```

где `Animal` — представленный ниже интерфейс, имеющий две реализации — `Cat` и `Dog`:

```
1 public interface Animal {
2     String getSpecies();
3 }
4
5 public class Cat implements Animal {
6     public String getSpecies() {
7         return "cat";
8     }
9 }
10
11 public class Dog implements Animal {
12     public String getSpecies() {
13         return "dog";
14     }
15 }
```

Также, пусть приложение имеет конфигурацию, показывающую, что вместо интерфейса `Animal` используется конкретная реализация — `Cat`.

```
1 public class AnimalConfiguration {
2     @Bean
3     public Animal animal(){
4         return new Cat();
5     }
6 }
```

Предположим, что мы разрабатываем тесты для метода `getSpecies` вручную. Один из возможных подходов — использование мокирования:

```
1 public void testGetSpecies() {
2     SpeciesService speciesService
3         = new SpeciesService();
4
5     Animal animalMock = mock(Animal.class);
6     when(animalMock.getSpecies())
7         .thenReturn("mouse");
8
9     speciesService.setAnimal(animalMock);
10
11     String actual = speciesService.getSpecies();
12     assertEquals("mouse", actual);
13 }
```

Такой тест формально корректен, но далек от проверки реального поведения программы и поэтому едва ли имеет практическую ценность.

При этом, разрабатывая тесты, мы имеем возможность исследовать конфигурацию приложения и выяснить, что для данного интерфейса используется реализация `Cat` и написать гораздо более репрезентативных тест.

```
1 public void testGetSpecies() {
2     SpeciesService speciesService = new SpeciesService();
3
4     Cat animal = new Cat();
5     speciesService.setAnimal(animal);
6
7     String actual = speciesService.getSpecies();
8     assertEquals("cat", actual);
9 }
```

Теперь рассмотрим сценарий автоматической генерации тестов. Не обладая информацией из конфигурации *Spring*–приложения, указывающей, какая из реализаций **Animal** предпочтительней, мы можем либо сгенерировать уже упомянутый тест с моком, либо выбрать одну из реализаций интерфейса **Animal** случайно. Таким образом, может быть сгенерирован еще один формально корректный, но не имеющий практической ценности тест, использующий реализацию **Dog**. Однако, если проанализировать конфигурацию приложения и затем предоставить инструменту генерации тестов информацию об используемых типах, то можно сгенерировать тест, отражающий реальное поведение программы.

Несмотря на то, что описанную выше конкретизацию типов можно сделать не всегда (ввиду возможной неоднозначности их выбора) и не всегда необходимо ее делать, в ряде случаев конкретизация позволяет получать более выразительные тесты, описывающие реальные сценарии исполнения программы, поэтому возможность генерировать тесты с конкретизацией типов для *Spring*–приложений является желательной опцией для автогенератора.

Также важно отметить, что поскольку модульные тесты не предполагают запуск приложения и инициализацию его контекста, а тестируют компонент в изоляции, то ожидается, что и в процессе генерации тестов, который включает в себя анализ пользовательской конфигурации, полная инициализация контекста также не будет происходить. В противном случае генерация тестов может быть опасна для пользовательских данных: например, при инициализации контекста могут выставлены переменные среды, проведена загрузка данных из сторонних сервисов и т.п. Это создает дополнительные трудности при решении задачи генерации тестов для приложений, написанных с использованием *Spring*.

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:
 - (а) 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
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 или машинное обучение [6]. Самыми известными решениями с открытым исходным кодом являются *EvoSuite*⁹, *UnitTestBot Java* и *Randoop*¹⁰. А среди решений с закрытым исходным кодом известны *Parasoft Jtest*¹¹, *Diffblue Cover*¹² и *Machinet*¹³.

Среди них только малая часть умеет генерировать тесты для Spring-приложений. Например, *Parasoft Jtest* генерирует только шаблоны тестовых методов. Конечно, это сокращает общее время написания тестов, однако степень покрытия кода будет зависеть от самого пользователя, который подставляет значения аргументов в код сгенерированных шаблонов тестовых методов.

Ниже приведен пример генерации шаблона теста для *Spring*-приложения с помощью *Parasoft Jtest*, взятый с официального сайта *Parasoft* [7].

```
1 @Test
2 public void testGetPerson() throws Throwable {
3     MockedStatic<ExternalPersonService> mocked =
4         mockStatic(ExternalPersonService.class);
5     mocks.add(mocked);
6
7     Person getPersonResult = null; // UTA: default value
8     mocked.when(() ->
9         ExternalPersonService.getPerson(anyInt())
10        ).thenReturn(getPersonResult);
11
12    // Given
```

⁹What is EvoSuite? – URL: <https://github.com/EvoSuite/evosuite>

¹⁰Randoop: Automatic unit test generation for Java. – URL: <https://randoop.github.io/randoop/>

¹¹Parasoft Jtest for Java Unit Testing. – URL: <https://www.parasoft.com/products/parasoft-jtest/java-unit-testing/>

¹²What is Diffblue Cover? | Diffblue. – URL: <https://www.diffblue.com>

¹³Machinet: AI Assistant for Developers. – URL: <https://www.machinet.net>

```

12     PeopleController underTest = new PeopleController();
13
14     // When
15     int id = 1;
16     Model model = mock(Model.class);
17     ResponseEntity<Person> result = underTest.getPerson(id, model);
18
19     // Then
20     assertNotNull(result);
21     assertNotNull(result.getBody());
22 }

```

Еще одним примером инструмента, который умеет генерировать тесты для *Spring*-приложений является *Diffblue Cover*. Он уже умеет генерировать тесты, которые принимают во внимание специфику *Spring*-приложения.

Ниже приведен пример генерации теста для *Spring* приложения с помощью *Diffblue Cover*.

```

1 @ContextConfiguration(classes = {SpeciesService.class})
2 @ExtendWith(SpringExtension.class)
3 class SpeciesServiceUnitTests {
4     @MockBean
5     private Animal animal;
6
7     @Autowired
8     private SpeciesService speciesService;
9
10    @Test
11    void testGetSpecies() {
12        when(animal.getSpecies()
13            .thenReturn("cow");
14        assertEquals("cow",
15            speciesService.getSpecies());
16        verify(animal, atLeast(1)).getSpecies();
17    }
18 }

```

Однако, и *Diffblut Cover* не предлагает механизмы генерации тестов, опирающихся на конфигурацию *Spring*-приложения.

3.2. *UnitTestBot Java*

UnitTestBot Java — один из инструментов линейки *UnitTestBot*¹⁴, предназначенный для автоматизации тестирования *Java* приложений. Для генерации тестовых сценариев в инструменте разработано два механизма: движок символьного анализа и фаззер.

Символьный движок является одной из реализаций парадигмы символьного исполнения [8]. Он производит анализ возможных путей исполнения программы путем сопоставления набора ограничений пути каждой ветви исполнения. Эти ограничения выражаются в терминах логики предикатов, а затем с использованием SMT-решателя *Z3*¹⁵ определяются условия их выполнимости. Получение же возможных путей исполнения программы, а также их приоритизация, происходит из графа потока исполнения, получаемого из байт кода программы. Предварительно байт код преобразуется в *Jimple* представление при помощи фреймворка *Soot*¹⁶. Благодаря этому преобразованию, байт код принимает более простое представление, имеющее меньшее количество инструкций.

Фаззер, используемый в *UnitTestBot Java*, применяет технику *greybox* фаззинга, подразумевающую генерацию случайных входных значений, используемых для конкретного исполнения тестируемых методов. После каждого конкретного исполнения фаззер получает фидбек об изменении покрытия кода тестами, исходя из которого производит изменение входных значений для следующей итерации своей работы.

Обе эти техники анализа кода, как правило, комбинируются для достижения максимальной эффективности. В *UnitTestBot Java*, по умолчанию, 95% времени, отведенного на генерацию тестов, отдается символьному движку, а фаззер используется как дополнительный вспомогательный инструмент.

¹⁴UnitTestBot. – URL: <https://www.utbot.org>

¹⁵Z3. – URL: <https://github.com/Z3Prover/z3>

¹⁶Soot. – URL: <https://github.com/soot-oss/soot>

3.3. Технологии, используемые для разработки

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

Использовались:

- *Kotlin*¹⁷ — язык программирования, который был выбран для разработки в рамках курсовой работы.
- *RD framework*¹⁸ — фреймворк, который был выбран для организации межпроцессного взаимодействия между движком и анализатором конфигураций приложения.

Поддерживались:

- *JUnit4*¹⁹, *JUnit5*²⁰, *TestNG*²¹ — фреймворки для тестирования *Java* приложений.
- *Mockito*²² — фреймворк для мокирования, который в том числе содержит специальные аннотации, удобные при тестировании *Spring*-приложений.

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

3.3.1. Фреймворк *Spring*

Spring предоставляет огромную функциональность, позволяющую пользователю детально сконфигурировать свое приложение. Все это

¹⁷The Kotlin Programming Language. — URL: <https://kotlinlang.org>

¹⁸Reactive Distributed communication framework for .NET, Kotlin, C++. — URL: <https://github.com/JetBrains/rd>

¹⁹A programmer-oriented testing framework for Java. — URL: <https://github.com/junit-team/junit4>

²⁰The 5th major version of the programmer-friendly testing framework for Java and the JVM. — URL: <https://github.com/junit-team/junit5>

²¹TestNG testing framework. — URL: <https://github.com/testng-team/testng>

²²Most popular Mocking framework for unit tests written in Java. — URL: <https://github.com/mockito/mockito>

влечет за собой огромное количество сценариев использования фреймворка, которые должны быть поддержаны при реализации механизма анализатора конфигураций.

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

Посмотрим на пример настройки контейнера с помощью конфигурационного класса:

```
1 @Configuration
2 public class AppConfig {
3
4     @Bean
5     public MyInterface myInterface() {
6         return new MyInterfaceImplementation();
7     }
8 }
```

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:

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="http://www.springframework.org/schema/beans
4       http://www.springframework.org/schema/beans/spring-beans.xsd">
5
6     <bean id="myInterface" class="com.example.MyInterfaceImplementation"/>
7 </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:

```
1 @Configuration
2 public class AppConfig {
3     @Bean
4     @Profile("dev")
5     public MyInterface myInterfaceDev() {
6         return new MyInterfaceDevImplementation();
7     }
8
9     @Bean
10    @Profile("prod")
11    public MyInterface myInterfaceProd() {
12        return new MyInterfaceProdImplementation();
13    }
14 }
```

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

Ниже приведен пример properties файла:

```
1 database.url=postgres:5432
2 database.username=postgres
3 database.password=pass
```

Ниже приведен пример использования properties файла в конфигурационном классе:

```
1 @Configuration
2 @PropertySource("classpath:application.properties")
3 public class AppConfig {
4     @Value("${database.url}")
5     private String databaseUrl;
6
7     @Value("${database.username}")
8     private String databaseUsername;
9
10    @Value("${database.password}")
11    private String databasePassword;
12 }
```

Также помимо обычного *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 реализации анализатора конфигураций.

4. Реализация

В этом разделе приведена реализация анализатора пользовательских конфигураций, а также рассказано о модернизации символьного движка *UnitTestBot Java*, позволяющей конкретизировать типы во время генерации тестов исходя из информации, полученной от анализатора конфигураций.

4.1. Анализатор конфигураций *Spring*–приложения

Spring–специфичную информацию для конкретизации типов движок получает из анализатора конфигураций пользовательского приложения. Сбор информации осуществляется с использованием инструментов самого *Spring*, во время инициализации контекста *Spring*–приложения.

Инициализация *Spring*–контекста происходит в несколько этапов:

1) Сбор определений бинов. Во время этого этапа происходит парсинг и анализ всех конфигураций приложения. В результате, в частности, создаются *BeanDefinitions*, включающие в себя информацию о классе бина, его свойствах и о связях с другими бинами.

2) Конфигурация определений бинов (*BeanFactoryPostProcessor*). После того как информация о бинах собрана, *Spring* может модифицировать эти определения перед тем, как они будут использованы для самого создания бинов. Конфигурация определений бинов подразумевает установку зависимостей, указание области видимости, настройку жизненного цикла и других параметров, присущих бину.

3) Создание бинов и их настройка. На этом этапе происходит создание и последующая настройка экземпляров бинов на основе их определений. Создаются экземпляры классов и вызываются инициализирующие методы.

И если этапы сбора и конфигурации определений бинов являются безопасными для пользовательского приложения, то этап создания бинов может повлечь изменения в пользовательских данных. В связи с этим было принято решение внедриться в процесс инициализации *Spring*–контекста и реализовать собственный постпроцессор фабрики

определений бинов (*BeanFactoryPostProcessor*), который на этапе настройки определений бинов собирает всю нужную и доступную аналитатору информацию о типах, а затем останавливает дальнейшую инициализацию контекста, удаляя из него все определения бинов и тем самым, не допуская создание этих бинов. Общая схема сбора информации о типах во время инициализации контекста *Spring*-приложения показана на рисунке 1.

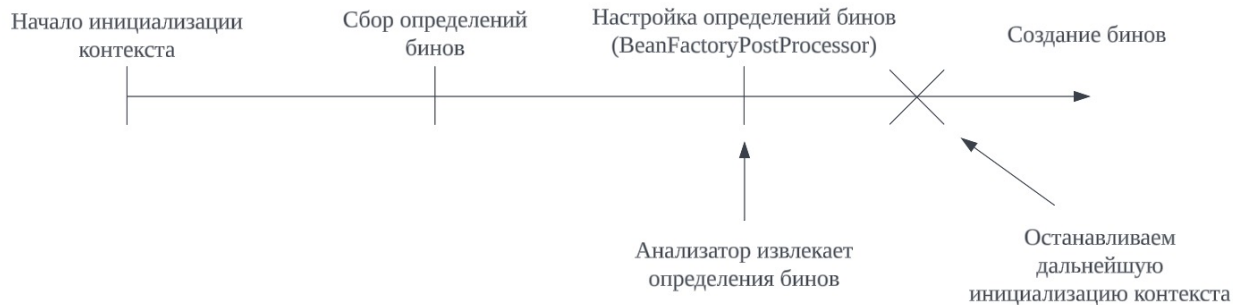


Рис. 1: Этапы инициализации *Spring*-контекста

Поскольку использование собственного постпроцессора не может происходить на стороне пользовательского приложения, было решено динамически создавать “гибридное” *Spring*-приложение, classpath которого комбинирует classpaths пользовательского приложения и модуля анализатора конфигураций. Важно отметить, что модуль анализатора конфигураций *Spring*-приложений имеет минимальное количество зависимостей, что позволяет избежать конфликтов зависимостей с пользовательским приложением. В частности, анализатор не зависит от какой-то конкретной версии *Spring*, а задействует рефлекссию для работы с версией фреймворка, используемой в пользовательском приложении. При старте “гибридного” *Spring*-приложения сначала определяется, использует ли пользователь обычный *Spring* или *Spring Boot* и, в зависимости от этого, выбирается нужный для создания *Spring*-приложения класс. Затем происходит динамический патчинг аннотаций, позволяющий запущенному *Spring*-приложению использовать выбранный для генерации тип конфигураций (конфигурационный *Java* класс или XML файл) и профиль.

Таким образом, получается схема создания “гибридного” Spring-приложения, показанная на рисунке 2, а процесс сбора *Spring*-специфичной информации для конкретизации типов анализатором конфигураций перерастает в целую цепочку действий, представленных на рисунке 3.

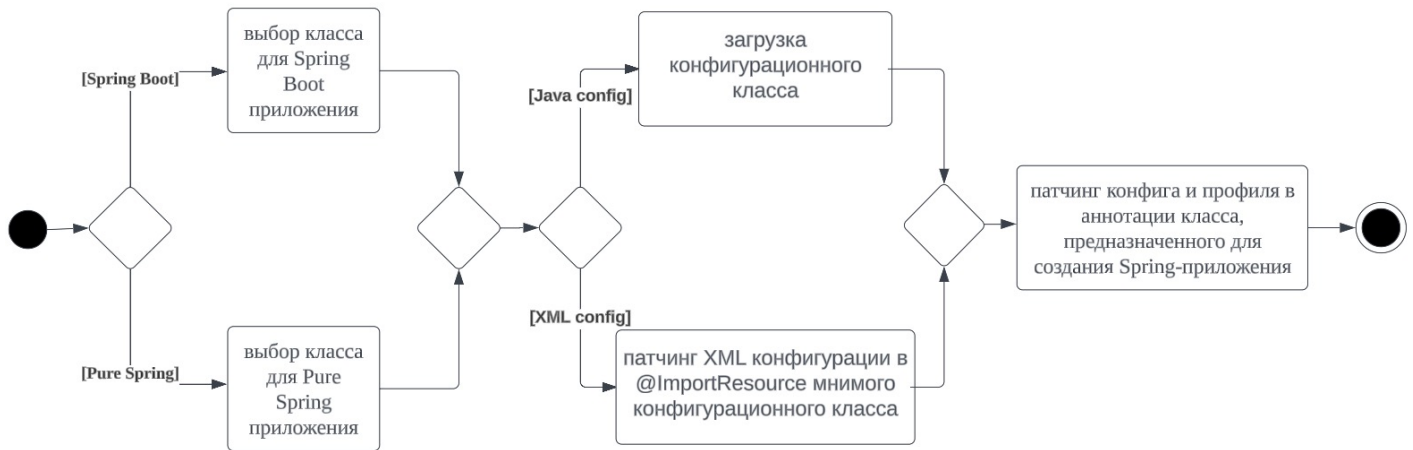


Рис. 2: Схема создания “гибридного” *Spring*-приложения

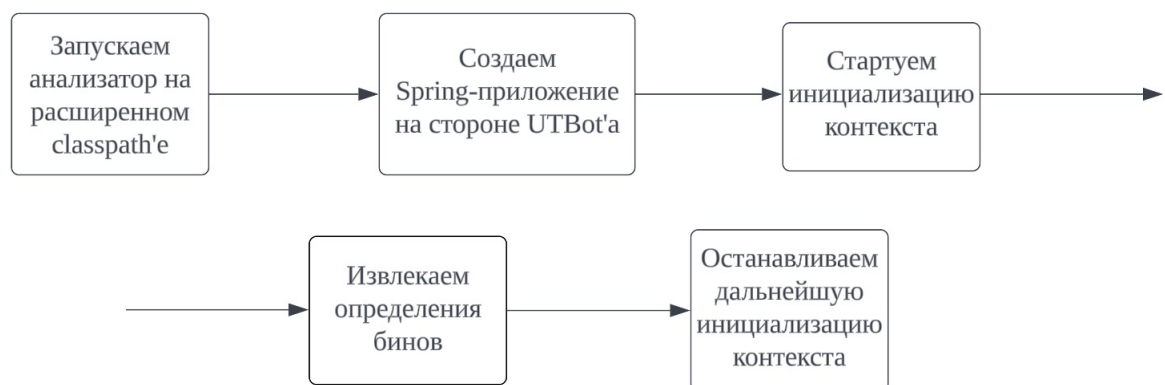


Рис. 3: Схема работы анализатора конфигураций *Spring*-приложения

4.2. Модернизация символьного движка

Суть модернизации символьного движка состоит в изменении механизма выбора типов символьных объектов: если раньше выбирался произвольный тип, который определялся SMT-решателем как удовлетворяющий ограничениям символьного пути, то теперь рассматриваются

только те типы, которые используются в выбранной для генерации тестов конфигурации приложения.

Обратимся к примеру с генерацией тестов для метода `getSpecies` из класса `SpeciesService`, представленного во введении 1. Если раньше при генерации тестов, как удовлетворяющей ограничениям символьного пути, могла быть реализация `Dog` для интерфейса `Animal`, тесты с которой не имеют практического значения, поскольку не тестируют реальное поведение программы, то теперь обязательно будет выбрана реализация `Cat`, как заданная для использования в качестве реализации интерфейса `Animal` в конфигурации приложения.

4.3. Взаимодействие символьного движка и анализатора конфигураций

Запуск анализатора пользовательских конфигураций происходит в отдельном от символьного движка процессе. Это предотвращает возможное падение всей генерации тестов в случае возникновения ошибки во время анализа пользовательских конфигураций: например, запуск на стороне *UnitTestBot Java Spring*-приложения, созданного на основе пользовательского, вполне может привести к crash JVM.

Процессы символьного движка и анализатора пользовательских конфигураций, как и все остальные процессы в *UnitTestBot Java*, взаимодействуют с помощью фреймворка *RD*.

5. Апробация

Модернизированный инструмент способен сгенерировать тест, который проверяет реальное исполнение метода `getSpecies` из класса `SpeciesService` модельного примера, приведенного во введении 1. В этом разделе я также привожу тесты, сгенерированные при помощи модернизированного инструмента, на реальных проектах с открытым исходным кодом *Java Blog Aggregator: Boot*²³ и *Mall*²⁴.

5.1. *Java Blog Aggregator: Boot*

Проект *Java Blog Aggregator: Boot* был выбран, поскольку он упоминается, например, в статье [9] как из рекомендованных к ознакомлению для начинающих *Spring*-разработчиков.

Рассмотрим класс `AllCategoriesService`, который имеет `@Autowired` поле `CategoryService`:

```
1 @Service
2 public class AllCategoriesService {
3
4     @Autowired
5     private CategoryService categoryService;
6
7     public Integer[] getAllCategoryIds() {
8         List<Category> categories
9             = categoryService.findAll();
10        Integer[] result
11            = new Integer[categories.size()];
12
13        for (int i = 0; i < categories.size(); i++) {
14            result[i] = categories.get(i).getId();
15        }
16        return result;
17    }
18 }
```

Как видно из листинга, класс `CategoryService` имеет аннотацию `@Service`, что означает, что данный класс является бином. Сигнатура

²³Java Blog Aggregator: Boot. – URL: <https://github.com/jirkapinkas/java-blog-aggregator-boot>

²⁴Mall. URL: <https://github.com/macrozheng/mall>

класса `CategoryService` имеет вид:

```
1 @Service
2 public class CategoryService
```

В результате генерации теста для метода `getAllCategoryIds` из класса `AllCategoriesService` получаем тест:

```
1 @Test
2 public void testGetAllCategoryIds() throws Exception
3 {
4     AllCategoriesService allCategoriesService =
5         new AllCategoriesService();
6
7     CategoryService categoryService =
8         new CategoryService();
9
10    CategoryRepository categoryRepositoryMock =
11        mock(CategoryRepository.class);
12    (when(categoryRepositoryMock.findAll()))
13        .thenReturn(new ArrayList<>());
14
15    setField(categoryService, "cz.jiripinkas.jba.service.CategoryService",
16        "categoryRepository", categoryRepositoryMock);
17
18    setField(allCategoriesService,
19        "cz.jiripinkas.jba.service.AllCategoriesService", "categoryService",
20        categoryService);
21
22    Integer[] actual =
23        allCategoriesService.getAllCategoryIds();
24
25    assertEquals(0, actual.length);
26    assertTrue(
27        deepEquals(new ArrayList<>(), actual)
28    );
29 }
```

В данном тесте вместо мокирования класса `CategoryService`, используется его конкретная реализация, за счет чего тест стал более выразительным. В частности становится видно, что `CategoryService` взаимодействует с базой данных.

5.2. Mall

Еще одним примером генерации, основанной на конфигурации *Spring*-приложения, может послужить тест, сгенерированный модернизированным инструментом *UnitTestBot Java* для метода одного из классов проекта *Mall*, репозиторий которого имеет около сотни тысяч форков и звезд.

В качестве примера рассмотрим класс `UmsAdminCacheServiceImpl`:

```
1 @Service
2 public class UmsAdminCacheServiceImpl implements UmsAdminCacheService
3 {
4     @Autowired
5     private UmsAdminService adminService;
6
7     @Autowired
8     private RedisService redisService;
9
10    ...
11
12    @Override
13    public void delAdmin(Long adminId) {
14        UmsAdmin admin = adminService.getItem(adminId);
15        if (admin != null) {
16            String key = REDIS_DATABASE + ":" + REDIS_KEY_ADMIN + ":" +
17                admin.getUsername();
18            redisService.del(key);
19        }
20
21        ...
22
23    }
```

У этого класса есть два `@Autowired` поля: `UmsAdminService` и `RedisService`, имеющих бины в конфигурации приложения.

Тест, сгенерированный для метода `UmsAdminCacheServiceImpl`, выглядит следующим образом:

```

1 @Test
2 public void testDelAdmin()
3     throws Exception
4 {
5     UmsAdminCacheServiceImpl umsAdminCacheServiceImpl
6         = new UmsAdminCacheServiceImpl();
7
8     UmsAdminServiceImpl adminService
9         = new UmsAdminServiceImpl();
10
11     UmsAdminMapper adminMapperMock
12         = mock(UmsAdminMapper.class);
13     when(
14         adminMapperMock.selectByPrimaryKey(any())
15     ).thenReturn(null);
16
17     setField(adminService, "com.macro.mall.service.impl.UmsAdminServiceImpl",
18         "adminMapper", adminMapperMock);
19
20     setField(umsAdminCacheServiceImpl,
21         "com.macro.mall.service.impl.UmsAdminCacheServiceImpl",
22         "adminService", adminService);
23
24     umsAdminCacheServiceImpl.delAdmin(null);
25 }

```

В этом тесте вместо мокирования абстрактного типа `UmsAdminService` подставляется его конкретная реализация `UmsAdminServiceImpl`, соответствующая конфигурации приложения. Создание экземпляра второго `@Autowired` поля не произошло, поскольку он не нужен в тестируемой ветви исполнения программы. И хотя в тесте нет asserts по причине того, что метод имеет тип `void`, этот тест представляет существенную практическую ценность. Поскольку тестируемый метод принимает nullable значение, то сценарий, в котором `adminId` принимает значение `null` возможен и является случаем, который может привести к `NullPointerException`. Сгенерированный тест позволяет убедиться, что в данном случае исключения не происходит. При написании тестов вручную подобные сценарии часто упускаются из виду.

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:
 - (а) определять, возможно ли конкретизировать тип
 - (б) конкретизировать тип при необходимости
4. Реализован механизм коммуникации между анализатором конфигурации и символьным движком
5. Протестированы основные сценарии работы анализатора
6. Предложенная функциональность апробирована на реальных проектах

Помимо этого, результаты, полученные в ходе выполнения работы, были представлены на профильных конференциях “Современные технологии в теории и практике программирования”²⁵[10] и SYRCoSE²⁶.

²⁵“Современные технологии в теории и практике программирования”. – URL: https://hsse.spbstu.ru/sovremennue_tehnologii/

²⁶SYRCoSE. – URL: <http://syr cose.ispras.ru>

Список литературы

- [1] C. Cristian, D. Daniel, and E. 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, pp. 209–224.
- [2] “Workshop on Search-Based and Fuzz Testing,” — URL: <https://sbft24.github.io> (дата обращения: 3 июня 2024 г.)
- [3] D. Ivanov et al., “UTBot Java at the SBST2022 Tool Competition,” *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, Pittsburgh, PA, USA, 2022, pp. 39–40.
- [4] D. Ivanov, A. Menshutin, M. Pelevin et al., “UTBot at the SBFT 2023 Java Tool Competition,” *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 2023, pp. 68–69.
- [5] “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: 3 июня 2024 г.)
- [6] M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for REST APIs: no time to rest yet,” *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, Association for Computing Machinery, New York, NY, USA, 2022, pp. 289–301, doi: <https://doi.org/10.1145/3533767.3534401> (дата обращения: 3 июня 2024 г.)
- [7] Accelerate Unit Testing of Spring Applications With Parasoft Jtest & Unit Test Assistant. — URL: https://alm.parasoft.com/hubfs/New_Pages/Whitepaper:%20Accelerate%20Unit%20Testing%20of%20Spring%20Applications%20with%20Parasoft%20Jtest%20and%20Unit%20Test%20Assistant.pdf (дата обращения: 3 июня 2024 г.)

- [8] R. Baldoni, E. Coppa, D. Cono D’elia, C. Demetrescu, and I. Finocchi, “2018. A Survey of Symbolic Execution Techniques,” *ACM Comput. Surv.* 51, 3, Article 50, 2019, 39 pages, doi: <https://doi.org/10.1145/3182657> (дата обращения: 3 июня 2024 г.)
- [9] R. Fadatare, “10+ Free Open Source Projects Using Spring Boot,” — URL: <https://www.javaguides.net/2018/10/free-open-source-projects-using-spring-boot.html> (дата обращения: 3 июня 2024 г.)
- [10] К. А. Шишин, Е. К. Куликов, “Генерация модульных тестов с конкретизацией типов для Spring-специфичных приложений,” *Современные технологии в теории и практике программирования : Сборник материалов научно-практической конференции студентов, аспирантов и молодых ученых*, Санкт-Петербург, 2024, pp. 234–236, edn: <https://elibrary.ru/UGKFDM> (дата обращения: 3 июня 2024 г.)