Министерство образования и науки Российской Федерации

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)

Радиофизический факультет

Кафедра информационных технологий в исследовании дискретных структур

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
д-р физ.-мат. наук, профессор
_____ В.П. Гермогенов
«20» 06 2017 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТОВ МУТАЦИОННОГО ТЕСТИРОВАНИЯ ДЛЯ ОЦЕНКИ КАЧЕСТВА ПРОГРАММНЫХ РЕАЛИЗАЦИЙ НА ЯЗЫКЕ JAVA

по основной образовательной программе подготовки бакалавров

направление подготовки 03.03.03 – Радиофизика

Донгак Бархас Мергенович

Руководитель ВКР
Ведущий аналитик ООО "ОКейСити"
_____ М. С. Форостьянова
«20» 06 2017 г.
Автор работы
студент группы № 736
_____ Б. М. Донгак

Томск-2017

Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (ТГУ)
Радиофизический факультет
Кафедра информационных технологий в исследовании дискретных структур

УТВЕРЖДАЮ
Руководитель ООП
д-р физ.-мат. наук, профессор
_____ В.П. Гермогенов
« 26 » 09 2016 г.

ЗАДАНИЕ
по подготовке ВКР бакалавра

студенту Донгаку Бархасу Мергеновичу группы № 736

1. Тема ВКР «Использование инструментов мутационного тестирования для оценки качества программных реализаций на языке Java»

2. Срок сдачи студентом выполненной ВКР:
а) на кафедре 15 июня 2017 г.
б) в ГЭК 23 июня 2017 г.

3. Краткое содержание работы: целью работы является использование инструментов мутационного тестирования для оценки качества программных реализаций на языке Java.

4. Календарный график выполнения ВКР:

а) изучение теоретического материала по тестированию программного обеспечения, изучение методов тестирования, обзор инструментов для автоматизации тестирования	сентябрь-октябрь 2016 г.
б) проведены эксперименты по сравнению эффективности двух инструментов для автоматизации тестирования: μJava и Pitest	ноябрь-декабрь 2016 г.
в) тестирование интерфейса инструмента FSMTest 2.0	январь-март 2017 г.
г) тестирование XML – файлов	апрель 2017 г.
д) написание выпускной квалификационной работы	май 2017 г.

5. Дата выдачи задания: 28 сентября 2016г.

Руководитель ВКР:
Ведущий аналитик ООО "ОКейСити" _____ М.С. Форостьянова
Задание принял к исполнению 29 сентября 2016 г. _____

РЕФЕРАТ

Бакалаврская работа 40 с., 4 раздела, 4 рисунка, 8 таблиц, 16 источников.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ, ОШИБКА, ТЕСТИРОВАНИЕ, КОНЕЧНЫЙ АВТОМАТ, МУТАНТЫ, МУТАЦИОННОЕ ТЕСТИРОВАНИЕ, XML, FSMTest

В работе проводились экспериментальные исследования по оценке качества инструмента для автоматизации тестирования FMSTest 2.0, и экспериментальное сравнение инструментов для мутационного тестирования.

Цель работы: экспериментально установить соответствие между ошибками в программных реализациях и ошибками в расширенных автоматах.

В работе получены следующие основные результаты.

1) Изучены основные понятия тестирования программного обеспечения.

2) Проведен краткий обзор методов тестирования программных реализаций, в частности, рассмотрены методы мутационного тестирования, а также тестирования на основе формальных моделей.

3) Проведено исследование инструментов для автоматизации тестирования, в том числе для мутационного тестирования.

4) Проведены эксперименты по сравнению эффективности двух инструментов для автоматизации мутационного тестирования: μJava и Pitest.

5) Исследована структура и принцип работы инструмента FSMTest 2.0.

8) Проведены эксперименты с генерацией тестовых наборов для инструмента FSMTest 2.0, найденные неисправности переданы разработчикам.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире цифровых технологий тестирование стало неотъемлемой частью жизненного цикла любого программного продукта, будь то пользовательский интерфейс или критическая система. Тестирование направлено на выявление неисправностей в коде, а также на проверку соответствия заданной спецификации. Поскольку программирование, как и любая человеческая деятельность невозможна без ошибок, важно правильно выставлять требования, как к продукту, так и к планированию и целям тестирования. Примерами последствий таких ошибок являются: авария ракеты-носителя «Ариан – 5» 4 июня 1996 года; потеря личных данных, хранящихся в «облаке» в результате отключения серверов Amazon в 2013 году; отключение электроэнергии на северо-востоке США в 2003 году из-за ошибки программного обеспечения для мониторинга работы оборудования General Electric Energy.

В этапы разработки программного обеспечения входят:

- Сбор и анализ требований
- Системный анализ
- Разработка системы
- Кодирование
- Тестирование
- Реализация

Спецификация требований к программному продукту (SRS) - это документ или комплект документации, описывающий функции и поведение системы или программного приложения [1].

На основании первых спецификаций пишутся тест–планы, разрабатываются тестовые наборы, а также оценивается необходимость использования автоматизации тестирования.

Как и любой другой сложный процесс, тестирование программного обеспечения состоит из разных этапов, и каждый из них представлен конкретной направленностью действий.

Из этапов тестирования можно выделить следующие:

1. Проектирование тестов — разработка стратегии тестирования, разработка планов тестирования, разработка и документирование тестовых кейсов
2. Выполнение тестового цикла — анализ спецификаций и кода. Кодирование и проведение тестовых кейсов
3. Улучшение тестирования ПО — проведение исследований по результатам выполненного тестирования с получением информации по покрытию тестовыми кейсами исходного кода или функциональности программы. Разработка рекомендаций по улучшению тестирования
4. Улучшение качества программного продукта — выдача рекомендаций по улучшению программного обеспечения в целом или с целью соответствия поставленным требованиям и задачам
5. Оптимизация тестирования ПО — разработка программ, позволяющих в автоматическом режиме обрабатывать исходные коды и делать необходимые заключения по его качеству

Основной целью тестирования является поиск несоответствий в спецификациях, а также поиск фактических ошибок программиста. Также один из важных этапов проверки качества программного продукта, это верификация — процесс оценки промежуточных рабочих продуктов жизненного цикла разработки, для проверки правильно ли создан конечный продукт.

С развитием цифровых технологий также происходит развитие методов тестирования, увеличивается сложность применяемых алгоритмов, что непременно введет к развитию инструментов для автоматизации тестирования.

Одним из таких инструментов является FSMTest 2.0, который разработан в рамках проекта на кафедре информационных технологий в исследовании дискретных структур Томского государственного

университета. Данный инструмент предназначен для автоматизации синтеза тестовых последовательностей на основе конечно-автоматных моделей. Однако при распространении данного программного продукта, как и любого другого, со стороны разработчика, также важно давать оценку полученному инструменту, тщательно тестировать его, и в итоге гарантировать качество и надежность. Поэтому в рамках нашего исследования за экспериментальную основу были поставлены задачи: исследовать существующие методы тестирования, провести сравнительные эксперименты для некоторых популярных инструментов для автоматизации тестирования, и обнаружить все неисправности и уязвимости продукта FSMTest 2.0.

Структура и объем работы. Дипломная работа включает введение, 4 главы, заключение, список используемой литературы и приложение. Отчет содержит 4 рисунка и 6 таблицы. Объем работы составляет 40 страницы, в том числе: титульный лист – одна страница, оглавление – 1 страница, основной текст – 35 страниц, список использованных литературных источников из 17 наименований – 2 страницы.

В первом разделе данной работы вводятся основные понятия и определения. Обознаются основные типы ошибок, такие как дефекты и уязвимости. Во второй части данного исследования составлен обзор существующих методов и подходов к тестированию, составлена классификация, а также раскрыты методы, используемые в следующих разделах. В третьей части работы исследована область инструментов для автоматизации тестирования разных типов программного обеспечения и проведены эксперименты по сравнению двух инструментов для автоматизации мутационного тестирования: μJava и Pitest. В завершающей главе приведены результаты исследования качества программного продукта FSMTest 2.0, приведена статистика обнаруженных неисправностей.

1 Тестирование программного обеспечения

Существует большое количество признаков, по которым можно классифицировать тестирование (рисунок 1).

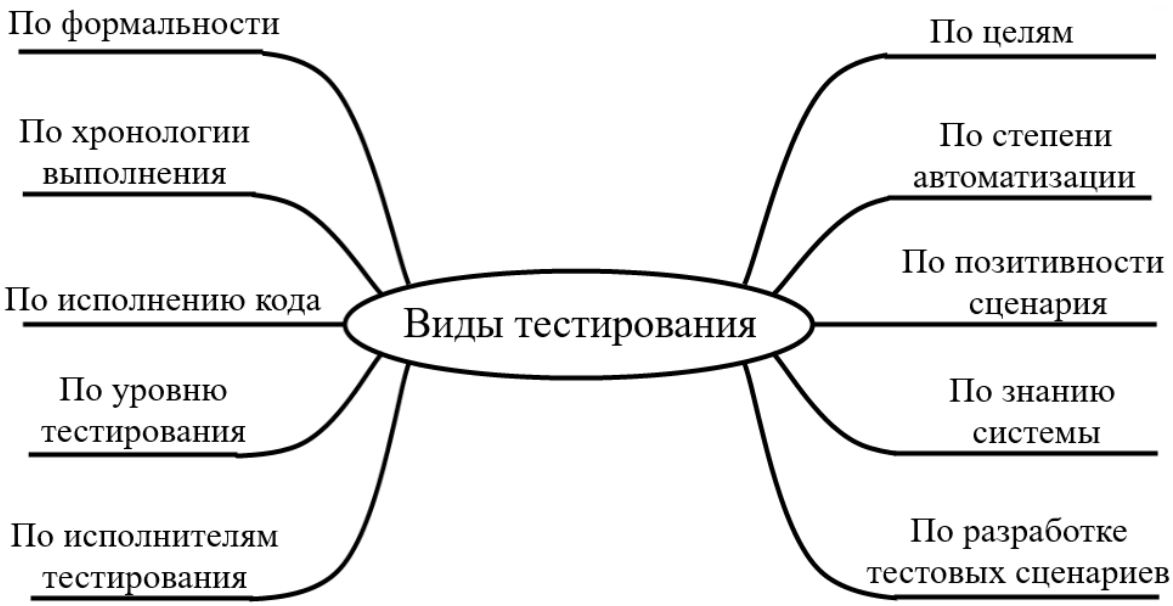

Рисунок 1 – Способы классификации видов тестирования ПО

Виды тестирования программного обеспечения, в зависимости от преследуемых целей, можно условно разделить на следующие группы:

- Функциональные виды тестирования
- Нефункциональные виды тестирования
- Связанные виды тестирования с изменениями

В функциональном тестировании в основном выполняется тестирование функций.

Нефункциональные виды тестирования оценивают работу нефункциональных составляющих программного продукта и описывают характеристики системы. Например: анализ производительности системы при обработке различных объемов данных, проверка стабильности программного продукта и т.п.

Связанные виды тестирования дефектов программного продукта используются для подтверждения того факта, что проблема была действительно решена

Тестирование по доступности программного кода можно разделить на три вида: белый, черный и серый (гибридный) ящики.

Тестирование методом «черного ящика» (Black-box testing) - это метод тестирования программного обеспечения, который анализирует функциональные возможности приложения на основе спецификаций, без знания внутреннего устройства. Обычно тестировщик выполняет этот тип тестирования в течение всего жизненного цикла тестирования программного обеспечения.

Тестирование методом «белого ящика» (White-box testing) - это метод тестирования программного обеспечения, в котором внутренняя структура тестируемого элемента известна инженеру качества. Тестировщик выбирает входы для прохождения путей через код и определяет соответствующие выходы. Тестирование происходит за пределами пользовательского интерфейса.

Тестирование методом «серого ящика» - это метод тестирования, который выполняется с ограниченной информацией о внутренней функциональности системы. Специалисты по качеству имеют доступ к подробным проектно-техническим документам вместе с информацией о требованиях[2].

Используя любой подход тестирования со знанием кода, допустимо сделать оценку покрытия исходного программного кода и сделать заключение отсутствии определенного класса неисправностей.

1.1 Дефекты программного кода

Дефектами программного обеспечения являются условия или входо-выходная реакция в программном продукте, которые не соответствуют требованиям к программному обеспечению или ожиданиям конечных пользователей. Другими словами, дефект - это ошибка в кодировании или логике, которая приводит к сбою в работе программы или к получению неверных и неожиданных результатов.

Дефекты ПО можно классифицировать по [3]:

- Степени серьезности дефекта
    1. Критический: дефект влияет на критическую функциональность или критические данные. У него нет обходного пути
    2. Основной: дефект влияет на основные функции или основные данные. У этого есть обходной путь, но это не очевидно и сложно
    3. Незначительный: дефект влияет на незначительную функциональность или некритические данные. У этого есть легкий обходной путь
    4. Тривиальный: является простым неудобством. Дефект не влияет на функциональность или данные. Не нуждается в обходном пути, также не влияет на производительность или эффективность
- Вероятности обнаружения дефекта
    1. Высокий: обнаружено всеми или почти всеми пользователями функции
    2. Средний: обнаружено примерно 50% пользователей функции
    3. Низкий: очень малое количество пользователей обнаружило дефект или совсем не обнаружило
- Приоритету устранения дефекта
    1. Срочно: должен быть исправлен в следующей сборке
    2. Высокий: может быть исправлен в любой из предстоящих сборок, но должен быть включен в выпуск
    3. Средний: может быть исправлен после выпуска / в следующей версии
    4. Низкий: может быть устранен или вообще не устранен
- Компоненту, в котором обнаружен дефект
- Фазе жизненного цикла разработки ПО, где обнаружен дефект
- Фазе жизненного цикла разработки ПО, где была создана ошибка

Ошибки ПО можно разделить на следующие типы[4]:

- Ошибки пользовательского интерфейса - отсутствие функций или данных, информация, вводящая в заблуждение, неверная информация в тексте справки
- Обработка ошибок – плохая защита от поврежденных данных, игнорирование переполнения данных
- Пограничные ошибки – ошибки в границах цикла, неверное обращение к границам памяти
- Ошибки расчета - неправильное преобразование из одного типа данных в другие, неправильные формулы
- Начальные и последующее состояния - невозможность установить элементы данных в ноль, инициализировать переменную цикла или повторно инициализировать указатель
- Ошибки потока - неверное возвращаемое состояние
- Условия загрузки - требуемые ресурсы недоступны, не возвращает неиспользуемую память
- Аппаратное обеспечение - неправильное устройство, устройство недоступно, недостаточное использование информации об устройстве
- Ошибки тестирования - неспособность заметить / сообщить о проблеме

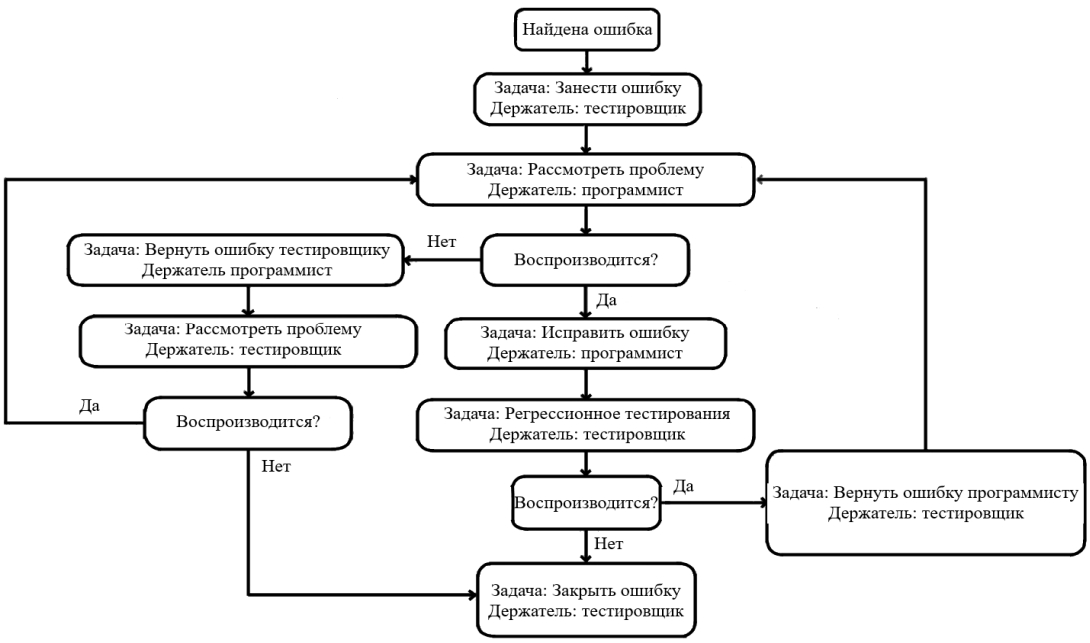

Рисунок 2 – Жизненный цикл дефекта

Среди дефектов самые сложные в обнаружении – не декларированные возможности программного обеспечения. В правильном, с точки зрения и функциональности, и информационной безопасности, коде может присутствовать фрагмент, реализующий функциональность, которая не была задуманная заказчиком. Его для своих целей привнес разработчик.

Обычно не декларированные возможности программного обеспечения разделяют на закладки и секретный вход (back door).

Программная закладка — это внесенные в программное обеспечение функциональные объекты, которые при определенных условиях (входных данных) инициируют выполнение не описанных в документации функций, позволяющих осуществлять несанкционированные воздействия на информацию [4].

Секретный вход – это фрагмент кода, позволяющий программисту получать доступ к программным обеспечениям в обход правил, указанных в техническом задании. Наиболее часто секретными входами наполняют программное обеспечение, которое разрабатывается на заказ, для того, чтобы можно было выполнять удаленную диагностику ошибок при эксплуатации программного приложения заказчиком.

1.2 Уязвимости

Исследовательская группа MITRE, занимающаяся анализом и разрешением критических проблем с безопасностью, разработала следующее определение уязвимости [5]:

Уязвимость — это состояние вычислительной системы (или нескольких систем), которое позволяет:

- исполнять команды от имени другого пользователя
- получать доступ к информации, закрытой от доступа для данного пользователя
- показывать себя как иного пользователя или ресурс
- производить атаку типа «отказ в обслуживании»

Джастин Шу, Джон Данн Макдональд и Марк Дауд в своей работе: «Искусство оценки безопасности программного обеспечения: выявление и предотвращение уязвимостей программного обеспечения» объединили уязвимости в следующие классы [6]:

- Уязвимости в архитектуре. Они возникают вследствие неправильной реализации требований по безопасности на уровне архитектуры будущего программного средства или неправильной идентификации потенциальных угроз
- Уязвимости в реализации. Данные уязвимости возникают вследствие некорректной реализации программного продукта
- Уязвимости на этапе эксплуатации. Эти уязвимости не являются результатом ошибок в коде программы

Уязвимости выявляются на стадии тестирования программного обеспечения с помощью специальных тестов на проникновение. Так же уязвимости выявляют эксперты в области анализа кода, знающие о наличии побочных эффектов у тех или иных языковых конструкций. Однако наиболее эффективный метод их обнаружения – это полуавтоматический статический анализ кода, который выполняется экспертами с применением специальных инструментальных средств.

Обнаружить не декларированные возможности полностью в автоматическом режиме нельзя, так как такой код является полностью правильным. Подобными проблемами занимаются эксперты в области информационной безопасности. Они находят дефекты в программном коде посредством ручного анализа или с помощью программных инструментов, позволяющих обнаруживать в исходном коде шаблоны языковых конструкций, характерных для построения не декларированных возможностей.

Таким образом, ошибкой называют некорректно работающий участок кода, написанный разработчиком. Результатом ошибки является дефект, то есть участок кода, результат работы которой приводит к сбою в работе

программы или к получению неверных и неожиданных результатов. Дефекты в программе, которые позволяют нарушать целостность системы, а также снизить информационную безопасность системы называются уязвимостями

2 Методы построения тестов

Основные методы построения тестов можно разделить на следующие группы.

1. Вероятностные методы

Эти методы основаны на вероятностной генерации тестовых воздействий в соответствии с определенными распределениями. Вероятностные методы хорошо автоматизируются и являются наименее трудоемкими. Однако обеспечиваемая ими полнота тестирования варьируется случайным и плохо предсказуемым заранее образом — в одних случаях оказывается достаточно хорошей, в других очень плохой. С помощью вероятностных методов хорошо находятся случайные ошибки и опечатки, в отличие от других, более сложных ошибок. Используются такие методы, когда о тестируемой системе почти ничего неизвестно, а получение дополнительной информации и построение тестов другими методами не может быть осуществлено в рамках проекта.

2. Методы, нацеленные на полное покрытие

В рамках этих методов тесты строятся целенаправленно таким образом, чтобы обеспечить покрытие выделенных критерием покрытия классов ситуаций. Эти методы автоматизируются плохо, в основном используются при ручной разработке тестов и достаточно трудоемки. Обеспечиваемая ими полнота при адекватном выборе критерия полноты тестирования очень высока. Позволяют находить любые виды ошибок. Используются при наличии практически полной информации о системе, достаточных ресурсов и необходимости провести систематическое и аккуратное тестирование.

3. Комбинаторные методы

Эти методы основаны на разбиении тестовых воздействий на некоторые элементы и составлении различных комбинаций из этих элементов по определенным правилам, с целью получить достаточно систематический перебор тестовых воздействий. Они хорошо автоматизируются, более трудоемки, чем вероятностные, но значительно проще, чем нацеленные на

покрытие методы. Обеспечиваемая ими полнота тестирования возрастает вместе с количеством получаемых тестов и может быть достаточно высокой. Позволяют находить случайные и достаточно простые ошибки. Используются при наличии лишь самой поверхностной информации о работе системы и при ограниченных ресурсах на тестирование.

4. Автоматные методы

Автоматные методы построения тестов используют модели тестируемой системы в виде конечных автоматов и их различных обобщений. Они хорошо автоматизируются, но требуют определенных затрат на выполнение тестов. Обеспечивают очень высокие значения полноты тестирования. Позволяют находить ошибки разных видов, в том числе и очень сложные, практически не обнаруживаемые другими методами. Используются при наличии четкой и полной информации о системе, достаточных ресурсов и повышенных требований к ее надежности и качеству.

5. Алгебраические методы

Эти методы используют алгебраическое описание тестируемой системы. Хорошо автоматизируются, требуют средних затрат ресурсов. Обеспечивают средние показатели полноты тестирования. Обнаруживают различные виды ошибок. На практике почти никогда не используются, потому что требуют описать тестируемую систему в виде алгебраической системы или набора абстрактных типов данных с полным набором аксиом.

2.1 Нагрузочное и стресс- тестирование

Нагрузочное тестирование - это процесс моделирования спроса на программное обеспечение, приложение или веб-сайт таким образом, который проверяет или демонстрирует его поведение в различных условиях.

Чаще всего нагрузочное тестирование выполняется в конце разработки ПО вместе со стресс - тестированием

Стресс-тестирование проводится для определения надежности системы с точки зрения экстремальной нагрузки и помогает администраторам

приложений определить, будет ли система работать в достаточной степени, если текущая нагрузка будет намного выше ожидаемого максимума.

2.2 Модульное тестирование

Модульное тестирование или юнит тестирование — тестирование, проверяющее корректность работы каждой отдельной функции или метода.

На данный момент существует огромное количество библиотек для написания юнит тестов под C++. Наиболее известными среди них являются: CxxTest, QTestLib, Boost::Test, UnitTest++.

Модульное тестирование имеет ряд преимуществ. Среди них можно выделить:

1. Уверенность в работоспособности каждой функции в отдельности
2. Проверка старой функциональности при внесении новой
3. Спецификация по использованию методов основной программы

Среди минусов модульного тестирования можно выделить то, что набор тестов приходится исправлять при каждом изменении функциональности.

Основные принципы модульного тестирования:

1. Отдельные тесты маленькие и быстрые
2. Автоматический запуск набора тестов
3. Независимость тестов друг от друга и от порядка следования
4. Найденная ошибка - повод для написания теста под нее
5. Иерархическая структура тестов
6. Тесты отделены от основного проекта
7. Наличие системы именования тестов

2.3 Методы построения тестов на основе конечного автомата

Гарантированная полнота тестирования возможна только с применением формальных моделей [7]. Среди таких моделей, на основе которых могут синтезироваться тесты для программного обеспечения, выделяют модели с конечным числом переходов - автоматные.

Конечным автоматом называют абстрактную математическую модель, которая позволяет описывать пути изменения конечного числа состояний объекта, в зависимости от входных данных и состояния, в котором объект находится в данный момент.

Под *конечным (инициальным) автоматом S* понимается пятерка $S = (S, I, O, T_s, s_0)$, где S – множество состояний с выделенным начальным состоянием s_0, I – входной алфавит, O – выходной алфавит, T – отношение переходов, $T \subseteq S \times I \times O \times S$.

Конечные автоматы хорошо изучены, в частности, хорошо известны методы построения проверяющих тестов для конечных, в том числе, недетерминированных автоматов.

Наиболее известными методами построения тестов являются W-метод, HSI-метод, Wp-метод. Данные методы основаны на синтезе тестов относительно модели "черного ящика".

2.3.1 W - метод

W- метод назван в честь Василевского, человека, предложившего данный метод в 1973 году. В методе Василевского тест строится на основе множества последовательностей, различающих любую пару состояний эталонного полностью определенного автомата[8].

Эталонный автомат – автомат, описывающий эталонное поведение технической системы.

W – метод предназначен для моделей, удовлетворяющий следующим требованиям:

1. Автомат должен быть *полностью определенным* — для всех состояний должны быть определены переходы по всем последовательностям из входного алфавита.
2. Автомат должен быть *минимальным* — никакой автомат с меньшим числом состояний не должен быть ему эквивалентен

3. Автомат должен быть детерминированным и должен иметь только те состояния, которые достижимы из начального по переходам.
4. В автомате должна быть возможность возвращения реализации в начальное состояние из любой произвольной ситуации

Конечное множество V обозначим как *множество достижимости* автомата S. Для каждого состояния s автомата S в множестве V существует последовательность, которая переводит автомат S из начального состояния s_0 в данное состояние s.

Через W обозначим *множество различимости* приведенного автомата S с числом состояний n. Для каждой пары различных состояний в автомате S во множестве W существует последовательность, реакция на которую в этих состояниях отличаются. Согласно требованию минимальности автомата любая пара состояний в автомате различима при помощи последовательности, длина которого не больше n и, следовательно, такое множество можно построить [9].

Через X^r обозначим множество входных последовательностей, состоящее из всех последовательностей длины r.

Тогда полным набором проверяющих тестов будет называться множество

$$A = VW \cup VXW \cup \ldots \cup VX^{(m-n)}W \cup VX^{(m-n+1)}W$$

Сложность построения теста W- методом экспоненциальна от (m-n).

2.3.2 HSI – метод

HIS – метод является модификацией метода Василевского. Он использует семейство гармонизированных идентификаторов.

Идентификатором состояния s_i называется множество $d(s_i)$ входных последовательностей автомата S, таких, что реакция автомата S/ s_i на последовательности из этого множества отличается он реакции автомата S/ s_j, $s_i \neq s_j$

Совокупность $\{d(s_0),\ldots,d(s_n)\}$ идентификаторов состояний автомата S называется *семейством гармонизированных идентификаторов*, если для любых двух различных состояний s_i и s_j в $d(s_i)$ и $d(s_j)$ существуют последовательности с общим начальным отрезком, реакции на которые автоматов S/s_i и S/s_j различны.

Так как длина идентификатора любого состояния не больше длины множества различимости, то длина теста, построенного методом гармонизированных идентификаторов, не больше длины теста, построенного методом Василевского. В случае, когда идентификатор любого состояния есть подмножество множества достижимости, то длина теста доставляемого HSI - методом, может оказаться меньше, чем длина теста, доставляемого W – методом.

2.4 Мутационное тестирование

Метод мутационного тестирования направлен на улучшение тестового набора. В процессе работы в исходный код программы вносятся небольшие изменения, называемые мутантами, после данный код пропускается через набор тестов. Отсутствие ошибок и неверных результатов при тестировании измененной программы на наборе тестов может означать низкое качество предоставленного набора тестов или же низкую значимость измененной части кода. Для применения мутационного тестирования выбирается набор мутационных операторов, которые включают в себя, как правило, удаление строки кода, замену одного оператора другим, замену переменной на другую переменную того же типа, а также другие характерные для программистов ошибки.

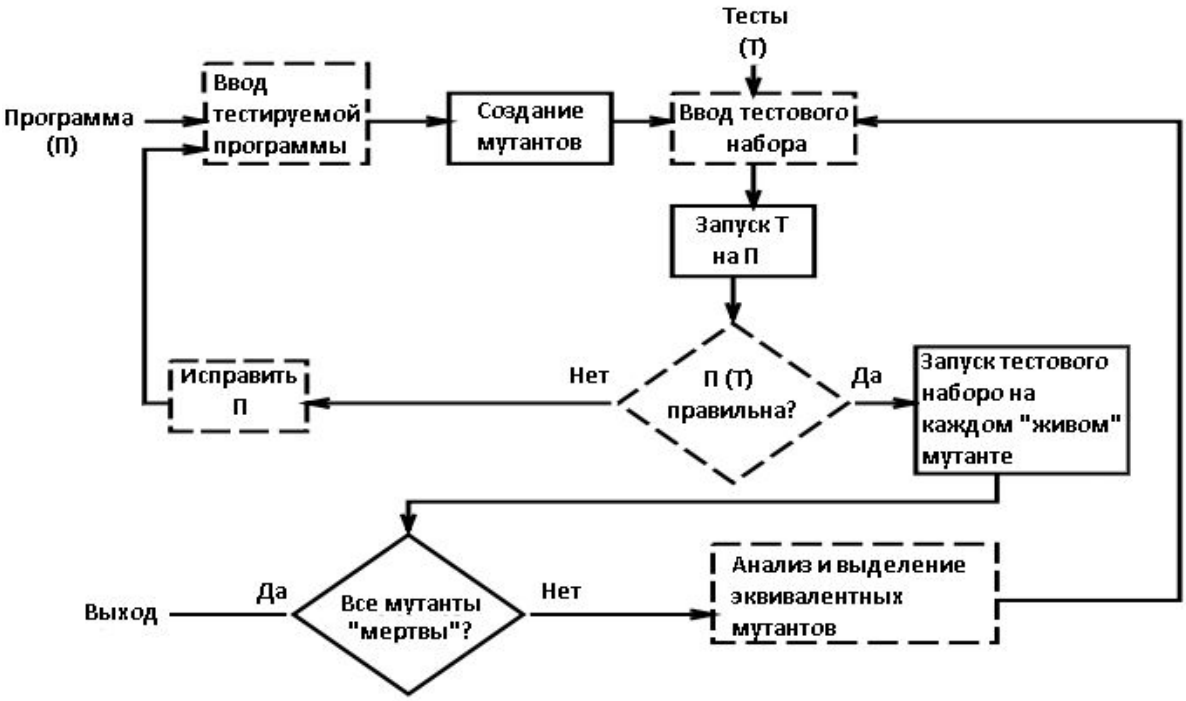

Рисунок 3 – Процесс мутационного анализа

Существует множество инструментов для автоматизации мутационного тестирования для различных языков программирования. Большинство инструментов для языка C++ и C являются платными и предназначены для коммерческого использования. Для языка Java список качественных инструментов не так широк, и нет точно сформулированных плюсов и минусов того или иного продукта. В рамках были определены лидирующие по степени развития два инструмента PitTest и μJava, однако не было проведено работ по сравнению списка операторов мутаций, скорости работы, поддержки платформ разработки и по другим критериям.

3 Автоматизированное тестирование программного обеспечения

Автоматизированное тестирование программного обеспечения - это процесс, в котором программные средства выполняют предварительные сценарии тестирования программного приложения до его выпуска.

Цель автоматизированного тестирования - упростить как можно больше усилий по тестированию с минимальным набором сценариев. Если, например, модульное тестирование потребляет большой процент ресурсов

группы обеспечения качества, тогда этот процесс может быть хорошим кандидатом для автоматизации. Автоматизированные инструменты тестирования способны выполнять тесты, сообщать о результатах и сравнивать результаты с более ранними тестами. Тестирование, выполняемое этими инструментами, может выполняться многократно, в любое время суток.

Между ручным тестированием и автоматическим тестированием существуют определенные различия. Ручное тестирование требует физического времени и усилий для обеспечения того, чтобы программный код выполнял все, что он должен был делать. Кроме того, ручные тестеры должны сделать запись своих выводов. Это связано с проверкой журналов, внешних служб и базы данных на наличие ошибок.

Автоматизированное тестирование отличается от ручного тестирования, где человек несет ответственность за единоличное тестирование функциональности программного обеспечения так, как это делает пользователь. Поскольку автоматическое тестирование выполняется с помощью инструмента автоматизации, в исследовательских тестах требуется меньше времени, и требуется больше времени для поддержки тестовых скриптов, одновременно увеличивая общий охват тестирования.

Преимущество ручного тестирования состоит в том, что оно позволяет человеку получить представление о тесте, которое в противном случае могло бы быть пропущено программой автоматизированного тестирования. Автоматическое тестирование хорошо подходит для крупных проектов, таких, в которых требуется повторное тестирование одних и тех же областей, а также проектах, которые уже прошли первоначальный ручной процесс тестирования.

3.1 Инструменты для автоматизации тестирования программного обеспечения

Atlassian Bamboo – система непрерывной интеграции, используемая для создания, тестирования и выпуска программного обеспечения. Atlassian Bamboo позволяет автоматизировать процесс сборки проекта, организовывать сборку по расписанию, определять конфигурации сборок, а также проводить модульное тестирование и выводить результат [9].

Ranorex – является платформой для автоматизации тестирования графического пользовательского интерфейса. Она позволяет тестировать ПО, а также мобильные и веб-приложения. Ranorex Studio предлагает простые в использовании инструменты автоматизации тестирования для создания надежных проектов [10].

Jenkins – это автономный сервер для автоматизации с открытым исходным кодом. Данный сервер может быть использован для автоматизации любых задач, таких как сборка, тестирование и развертывание программного обеспечения [11].

Cucumber - это инструмент, который запускает автоматические приемочные тесты. Cucumber написан на языке программирования Ruby, но его можно использовать для тестирования кода написанного на других языках, включая, Java, C # и Python [12]

TestNG - это платформа тестирования, основанная на JUnit и NUnit, но предлагающая некоторые новые функции, которые делают ее более мощной и простой в использовании [13].

Основными преимуществами TestNG являются:

- Возможность создавать HTML-отчеты о выполнении
- Тесты могут быть сгруппированы и распределены по приоритетам
- Возможность параллельного тестирования
- Возможность параметризации данных

3.2 Unit тестирование

JUnit во многом упрощает и автоматизирует процесс написания тестов и является оболочкой модульного тестирования. Более того, с помощью таких инструментов, большинство тестов, проверяющих работоспособность программы, не требуют изменения и постоянной поддержки, после любого изменения исходного кода тестируемой программы

Тест в JUnit – это метод, в случае успешного выполнения которого, считается, что тест пройден. Тестирующий класс наследуется от класса TestCase, а все методы для тестирования начинаются с «test». Часто, при написании тестов, используют подход «arrange-act-assert». Данный подход заключается в том, что на начальном этапе пишется подготовительный код для создания тестируемого класса. В последующем этапе происходит выполнение тестируемого метода. И, наконец, в заключительном этапе, тест производит проверку на соответствие между полученным значением и ожидаемым. Проверка соответствия полученного значения ожидаемому происходит в группе методов вида Assert. Подобные проверяющие методы построены таким образом, что в случае несоответствия результата ожидаемому значению, прерывает прохождение теста и выдает исключение, по содержанию которого, разработчик может инкапсулировать и устранить проблему в коде.

В таблице 1 приведены основные методы из класса Assert с кратким описанием.

Таблица 1 – Методы, которые проверяет JUnit

Asserts	Описание
fail(srting)	При достижении данного фрагмента тест считать неудачным, и выводится текстовое сообщение

assertTrue(boolean condition) (assertFalse(boolean condition))	Проверяет, что логическое условие имеет истинное значение (ложное значение)
assertEquals(<тип> expected, <тип> actual)	Проверка на равенство значений двух переменных: ожидаемое и полученное в тесте значение
assertNotNull(Object object) (assertNull(Object object))	Проверяет, что объект не нулевой и содержит какие-то данные, к которым можно обратиться (проверяет, что объект нулевой)
assertSame(Object expected, Object actual)	Проверяет, что обе переменные относятся к одному объекту.

3.3 Тестирование ПО с использованием FSMTest 2.0

FSMTest 2.0 позволяет генерировать классические или синхронизированные конечные автоматы, бинарную композицию конечных автоматов, а также выводить наборы тестов для конечных автоматов.

На рисунке 3 представлен экран FSM Test 2.0. В левой части указаны различные пункты меню, которые позволяют:

- генерировать классические или синхронизированные конечные автоматы (детерминированный или недетерминированный)
- проводить некоторые манипуляции с временными конечными автоматами (получение отличительной последовательности, r-отличительные конечные автоматы, набор тестов для тайм-аутов конечных автоматов)
- получить исчерпывающие тесты для детерминированных конечных автоматов
- получить набор тестов для недетерминированных конечных автоматов
- проводить некоторые манипуляции с композицией конечных автоматов (генерация бинарной параллельной композиции, вывод наборов тестов для

проверки блокировки в реальном времени, получение тестов для компонентов конечного автомата)

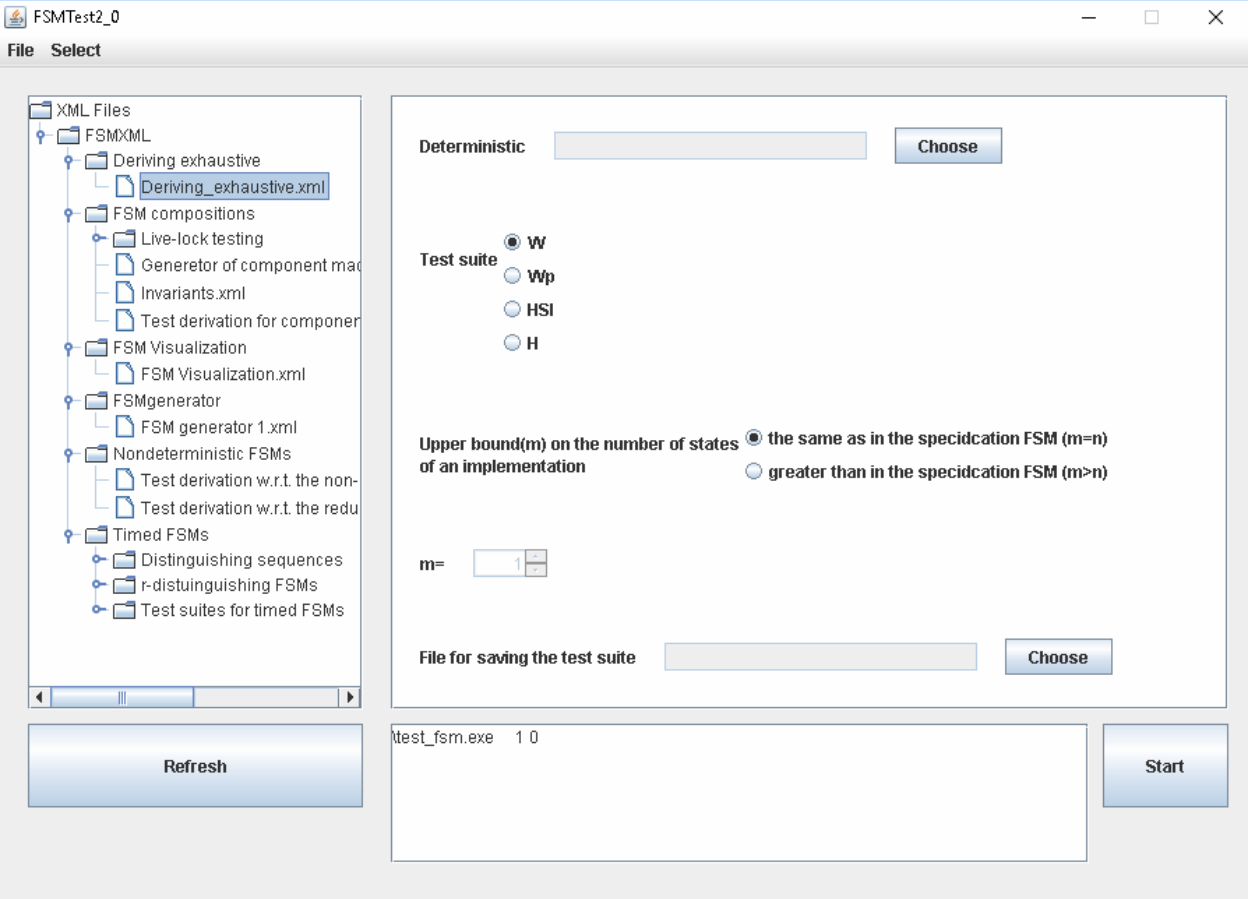

Рисунок 4 – Интерфейс FSMTest 2.0

3.4 Инструменты для мутационного тестирования

3.4.1 μJava

Объектно-ориентированный формат языка Java не позволяет использовать традиционный подход мутационного тестирования из-за более сложной синтаксической структуры, поэтому в 2003 году была выпущена первая версия инструмента μJava [14]. Он автоматически генерирует мутанты, как для традиционного мутационного тестирования, так и для мутационного тестирования на уровне класса, с учетом ошибок наследования и полиморфизма. μJava может тестировать, как отдельные классы, так и пакеты, состоящие из нескольких классов. Тесты представляются

пользователю в виде последовательности вызываемых методов в отдельных классах JUnit.

μJava является результатом сотрудничества между двумя университетами, Корейским передовым институтом науки и технологий (KAIST) в Южной Корее и Университета Джорджа Мейсона в США. Данный инструмент использует два типа операторов мутации, классового уровня и уровневый метод. Операторы классового уровня разработали Ма, Квоном и Оффот, а операторы объектно-ориентированных мутаций разработали Оффот и Александр [14].

μJava может работать на Unix, Linux и Windows платформах. Последняя версия была выложена в 2014 году, которая стала четвертым релизом данного инструмента. В Таблице 3.2 и 3.3 представлены списки традиционных и объектно-ориентированных ошибок соответственно, которые могут быть внесены посредством μJava

Таблица 2 – Традиционные мутанты

Операторы	Описание
AOR	Замена арифметических операторов
AOI	Вставка арифметических операторов
AOD	Удаление арифметических операторов
ROR	Замена операторов отношения
COR	Замена уловных операторов
COI	Вставка условных операторов
COD	Удаление условных операторов
SOR	Замена операторов сдвига
LOR	Замена логических операторов
LOI	Вставка логических операторов
LOD	Удаление логических операторов
ASR	Замена операторов присвоения

Таблица 3 – Объектно-ориентированные мутанты для каждой из парадигм ООП

	Операторы	Описание
Инкапсуляция	AMC	изменение модификатора доступа
Наследование	IHD	Скрытие переменной удаления
	IHI	Скрытие переменной вставки
	IOD	Переопределение метода удаления
	IOP	Переопределение изменения способа вызова позиции
	IOR	Переопределение метода переименования
	ISI	Вставка ключевого слова
	ISD	Удаление ключевого слова
	IPC	Вызов конструктора удаления родителя
Полиморфизм	PNC	Вызов нового метода с наследственным классом
	PMD	Объявление переменной с типом родительского класса
	PPD	Изменение параметра объявления переменной с наследственным типом класса
	PCI	Тип роли оператора вставки
	PCC	Изменение роли типа
	PCD	Тип роли оператора удаления
	PRV	Менять назначения с другими сопоставимыми значениями
	OMR	Перегрузка метода замены
	OMD	Перегрузка метода удаления
	OAC	Перегрузка метода замены аргументами
Специфические особенности Java	JTI	This ключевое слова вставки
	JTD	This ключевое слово удаления
	JSI	Статический модификатор вставки
	JSD	Статический модификатор удаления
	JID	Удаление переменной удаления
	JDC	Java по умолчанию поддерживает конструктор удаления

	EOA	Ссылать присвоения и менять их
	EOC	Ссылать сравнения и менять их
	EAM	Изменение метода выполнения
	EMM	Изменение метода модификатора

3.4.2 Pitest

Pitest является результатом работы разработчика программных обеспечений из Эдинбурга — Генри Колса [15]. В настоящее время, PIT предоставляет собой сборку некоторых встроенных мутантов, большинство из которых активны по умолчанию. Изначальный набор может быть изменен, путем выбора различных операторов. Мутации выполняются на байт-коде и генерируется компилятором. Основная идея данного инструмента, это генерация стабильных мутантов, т.е. таких мутантов, которые не обнаруживаются на этапе компиляции, при этом с большей вероятностью не будут эквивалентными. Те операторы, что не отвечают этим требованиям, не включены по умолчанию.

Последняя версия программы была выпущена 14 мая 2016 года и совместима с версией Java 8.

В Таблице 4 представлен список основных мутантов, которые могут быть внесены посредством Pitest и по умолчанию включены при генерации.

Таблица 4 – Основные операторы мутации Pitest

Название	Описание	Принцип замены	
Conditionals Boundary Mutator	Заменяет граничные операторы на смежные	Было	Стало
		<	<=
		>	>=
		<=	<
		>=	>
		Было	Стало
Negate Conditionals Mutator	Заменяет граничные операторы на смежные Меняет на отрицательные все	"==”	!=
		!=	“==”
		<=	>

	условные операторы	>=	<
		>	<=
		<	>=
Math Mutator	Заменяет двоичные арифметические операции для любых целых чисел другой операцией	Было	Стало
		+	-
		-	+
		*	/
		/	*
		%	*
		&	|
		|	&
		^	&
		<<	>>
		>>	<<
		>>>	<<
Increments Mutator	Меняет инкремент и декременты местами	Пример	
		Было	Стало
		i++	i--
Invert Negatives Mutator	Инвертирует целые числа и числа с плавающей запятой на отрицательные.	Пример	
		Было	Стало
		-i	+i

3.5 Эксперименты по оценке качества инструментов для мутационного тестирования μJava и Pitest

Помимо анализа литературы и сравнения списков операторов, представленных в мануалах программ μJava и Pitest, были проведены эксперименты по мутированию одних и тех же готовых программ. Отметим, что большую сложность составил запуск инструмента μJava, поскольку он совместим только с 6.41 версией Java, которая в свою очередь совместима только с весьма устаревшими версиями среды разработки Eclipse, таким как Eclipse Ver.Mars.1 Release (4.5.1). Также в рамках исследования нас

интересовала возможность автоматического запуска теста и «убийство» мутантов, с выводом итоговой статистики.

Для первоначального запуска и проверки правильности настройки инструментов мутационного тестирования в среде Eclipse была выбрана классическая программа калькулятора, выполняющая основные арифметические операции с целочисленными значениями.

Для программы классического калькулятора, состоящей из 2 функций, в сумме из 9 строк, были получены результаты, представленные в таблице 5.

Таблица 5 – Результат работы инструментов μJava и Pitest

Описание	μJava	Pitest
Общее число мутантов	30	4
Количество традиционных мутантов	30	4
Количество «убитых» мутантов	20	4

Стоит отметить превосходство инструмента μJava, поскольку количество и разнообразность мутаций намного превышает возможности более современного Pitest.

В рамках исследования была реализована программа на языке Java, выполняющая несколько видов сортировок: пузырьком, выбором и пирамидальную, а также набор Junit тестов, который в свою очередь проверял правильность функционирования программ.

В этом разделе приведена сводная таблица результатов, как независимых параметров, таких как год создания, так и результатами экспериментов.

Таблица 6 – Сравнительный анализ инструментов Pitest и μJava

Параметр	Pitest	μJava
Год создания	Июль, 2011г.	2003г.
Дата последнего обновления	Май, 2016г.	Апрель 2015г.

Поддерживаемые платформы	Ant, Maven, Elcipse, Powermock, Jmock, Jmock2, Mockito, JMockit, EasyMock		Eclipse	
Количество операторов мутирования	7 по умолчанию активных операторов мутирования	6 не активных операторов мутирования	15 традиционных операторов мутирования	28 классово-уровневых операторов мутирования
Количество сгенерированных мутантов	Пузырек и сортировка выбором	Пирамидальная	Пузырек и сортировка выбором	Пирамидальная
	25	30	140	191

- Результаты работы инструмента Pitest:

В результате тестирования мутированных классов, содержащих сортировки, использованный набор тестов не обнаружил мутанты, сгенерированных оператором Conditionals Boundary Mutator. Это связанно с тем, что выбранный набор тестов сравнивает только два соседних элемента массива, но не проверяет размер отсортированного массива. Также, в классе содержащем пирамидальную сортировку, был создан бесконечный цикл оператором мутирования Increments Mutator.

- Результаты работы инструмента μJava:

В результате тестирования мутированных классов, содержащих сортировки, использованный набор тестов не обнаружил мутанты, сгенерированных операторами AOI, ROR, AOR. Данные операторы мутируют операторы сравнения, вследствие чего меняется конечная размерность массива. Выбранный набор тестов не смог обнаружить «выживших» мутантов, так как он проверяет только по два соседних

элемента массива на правильность сортировки, но не проверяет размер отсортированного массива.

В результате работы μJava сгенерировал большее количество мутантов, что крайне важно при проверке качества тестовых наборов, особенно, для критических систем. Однако сложность использования инструмента μJava существенно влияет на ее применимость. Для генерации мутантов необходимо для каждого класса указывать местоположение этого класса, а так же местоположение теста, который будет проверять данный класс. Также сложно определить какие мутанты были «убиты», а какие остались не пойманными тестами, так как все сгенерированные мутанты отображаются в одной консоли. При работе μJava производительность центрального процессора резко возрастала, в следствии, чего Eclipse несколько раз прекращал работу, что также существенно влияет на качество продукта. При работе с Pitest таких проблем замечено не было. Pitest активно поддерживается разработчиками, и регулярно обновляется список библиотек для данного инструмента. Для сравнения, со времен последнего обновления μJava было выпущено два обновления Pitest. Также Pitest довольно легко освоить с базовыми навыками использования среды разработки. В нем легко проследить, какие мутанты не покрыты тестами, а какие были «убиты», так как список мутантов отображается в отдельных окнах, а убитые или пропущенные мутанты помечены визуально разными цветами. Также Pitest значительно быстрее генерирует мутанты и тестирует программу. Кроме того, запуск Pitest возможен в целом списке различных сред и платформ.

4 Проверка качества инструмента для автоматизирования тестирования FSMTest 2.0

4.1 XML – ошибки

Поскольку данная работа посвящена тестированию инструмента FSMTest 2.0, который непосредственно использует в работе xml-конфигурации, мы обратились к изучению древовидных строго заданных форматов представления данных. В настоящее время все чаще используется XML [16] (от англ. – eXtensible Markup Language), например, в качестве языка-посредника для определения форматов обмена данными между различными системами, которые взаимодействуют в сети Интернет, либо в качестве документа-конфигурации, позволяющий запускать программы под правильными клиентами, интерфейсами и т.д. XML-документы имеют очень строгую грамматику, которая достаточно компактно описывает поведение приложения; при этом в XML описываются данные, конфигурации и алгоритмы. Именно поэтому спецификации многих технических систем в настоящее время задаются в языке XML. Соответственно, необходимо качественно тестировать системы, описание которых задано в XML, и особенно критические системы, такие как связь, банковские системы, транспорт и др.

В XML-файле инструкции заключаются в угловые скобки, которые называются тегами и они служат для разметки текста в документе. Также теги служат для определения элементов документа, и других конструкций языка. При создании собственного языка разметки имеется возможность придумывать любые названия элементов, соответствующие контексту их использования.

При написании XML-файла очень легко допустить ту или иную ошибку, однако они часто обнаруживаются парсером – программой, которая анализирует и обрабатывает код.

Таблица 7 – Ошибки в XML-файле

Тип ошибки	Описание	Сообщение об ошибке
Отсутствие корневого элемента	XML-файлы работают на основе древовидной структуры, поэтому необходимо наличие корневого элемента	[Fatal Error] FSM%20generator%201.xml: 4:10: The markup in the document following the root element must be well-formed. Directory FSMXML doesn't have of XML files
Незакрытый тег	Теги служат для разметки текста в документе	[Fatal Error] FSM%20generator%201.xml: 24:23: Element type "param" must be followed by either attribute specifications, ">" or "/>". Directory FSMXML doesn't have of XML files
Неинициализированная переменная	Неинициализированная переменная содержит "мусор", вследствие чего ее использование может привести к некорректной работе программы	–

В ходе анализа ошибок в XML-файле было выявлено, что парсер не проверяет документ на использование неинициализированных переменных. Вследствие этого была реализована программа, проверяющая XML файлы на предмет наличия неинициализированных переменных.

4.2 Использование Pitest для тестирования программного интерфейса

Помимо тестирования XML – файлов, также было проведено тестирование программного интерфейса инструмента FSMTest 2.0. Тестирования проводилось с помощью инструмента Pitest в полуавтоматическом режиме. В различных классах автоматически генерировались мутанты, после чего программа запускалась и анализировалась на наличие ошибок.

Таблица 8 – Сгенерированные мутанты Pitest

Название класса	Количество сгенерированных мутантов	Количество мутантов, не влияющих на работу инструмента	Количество мутантов, влияющих на работу инструмента
FSMCheckBox	26	7	19
FSMFileChooser	39	8	31
FSMGroupRB	41	14	27
FSMSpinner	27	6	21
FSMTest_main	1	0	1
FSMTreeFile	49	3	46
DSMTreeXML	32	2	30

В результате работы Pitest сгенерировал 215 мутантов, 40 из которых не влияли на запуск и работу инструмента FSMTest 2.0. Большая часть этих мутантов являлись незначительными дефектами. Они влияли на незначительную функциональность и некритические данные. Также, в результате работы, Pitest сгенерировал 175 мутантов, которые влияли на функциональность программного инструмента. Данные мутанты являлись критическими и основными дефектами.

ЗАКЛЮЧЕНИЕ

В ходе научно-исследовательской работы были рассмотрены различные виды тестирования и изучены методы построения тестов. По результатам исследования был составлен обзор основных методов и типов тестирования, был проведен обзор наиболее известных инструментов для автоматизации тестирования.

Основным результатом работы является покрытие тестами инструмента для автоматизации тестирования систем с использованием методов на основе конечно-автоматных моделей FSMTest 2.0, который был разработан на кафедре информационных технологий в исследовании дискретных структур Томского государственного университета. Для тестирования использовались современные подходы, в том числе с применением инструментов для автоматизации тестирования. Все найденные неисправности переданы отделу разработки, тем самым внесен весомый вклад в получение патента на вторую версию FSMTest и соблюдение стандартов качества программного продукта. Актуальность исследования обоснована в первых двух главах работы, в которых приводится краткий обзор методов и средств тестирования, в том числе мутационного тестирования и методов на основе моделей с конечным числом состояний.

Среди выявленных неисправностей было обнаружено использование неинициализированных переменных в одном из XML – файлов, полученных в результате опечатки при написании кода.

В работе также проведены компьютерные эксперименты с двумя различными инструментами для мутационного тестирования, что также позволит в дальнейшем проводить более тщательное покрытие тестовыми наборами, в частности, различных сервисных систем, протокольных реализаций и сложных алгоритмических вычислений. Для этого были изучены основные понятия и определения мутационного тестирования. Для проведения экспериментов был изучен объектно-ориентированный язык программирования Java и самостоятельно реализованы несколько программ.

Для написания и подачи тестов, были изучены принципы Junit-тестирования. В отчете в четвертой главе представлены результаты по сравнению инструментов μJava и Pitest, с которыми были проведены компьютерные эксперименты, которые показали преимущества использования каждого из них для генерации мутантов и автоматического запуска тестов; также были выявлены недостатки, в том числе, на этапе запуска программ. Количество мутаций, которые можно внести в программу, используя μJava, в несколько раз превышает возможности Pitest, однако стоит отметить, что количество эквивалентных мутантов в этом случае составило: 97. Основная проблема μJava состоит в сложности запуска, и специфичным, устаревшим требованиям к среде запуска и разработки. В свою очередь, последняя версия Pitest была выпущена 31 марта 2017 года и регулярно совершенствуется. Кроме того, последний инструмент имеет интуитивно понятный интерфейс, который не требует глубоких знаний языка Java и сборок программных продуктов.

СПИСОК ЛИТЕРАТУРЫ

1. *Sarah Geagea, S. Z.* Software Requirements/ Sarah Geagea, S. Z.//Швеция, 2010. – 56 с.

2. Software Testing Fundamentals [Электронный ресурс]. URL: http://softwaretestingfundamentals.com/defect/ (дата обращения: 21.3.2017).

3. Software testing help [Электронный ресурс]. URL:http://www.softwaretestinghelp.com/types-of-software-errors/(дата обращения: 5.4.2017).

4. ГОСТ -51275-2006. Защита информации. Объект информатизации. Факторы, воздействующие на информацию. Общие положения. Москва: Изд-во стандартов 2007-7с.

5. MITRE CVE. (1999) [Электронный ресурс]. URL:http://www.cve.mitre.org/about/terminology.html (дата обращения: 15.3.2017).

6. *Джон Данн Макдональд*. Искусство оценки безопасности программного обеспечения: выявление и предотвращение уязвимостей программного обеспечения/Джон Данн Макдональд, Д. Ш. 2006 – 56 с.

7. *Ермаков А.Д., Евтушенко Н.В*. Метод синтеза тестов с гарантированной полнотой модели расширенного автомата. Моделирование и анализ информационных систем// Моделирование и анализ информационных систем Н.В. 2016.- 12 с.

8. *Гилл А.* Введение в теорию конечных автоматов / под ред. П.П. Пархоменко. Москва Наука, Москва, 1966.-272 с.

9. Atlassian [Электронный ресурс]. URL: https://ru.atlassian.com/software/bamboo(дата обращения: 25.10.2016).

10. Ranorex: Test Automation for GUI Testing [Электронный ресурс]. URL: https://www.ranorex.com/(дата обращения: 25.10.2016).

11. Jenkins [Электронный ресурс], URL: https://jenkins.io/ (Дата обращения: 21.11.2016)

12. Cucumber [Электронный ресурс] URL:https://cucumber.io/(Дата обращения: 21.11.2016)

13. TestNG [Электронный ресурс]. URL:http://testng.org/doc/(Дата обращения: 21.11.2016)

14. μJava. [Электронный ресурс]. URL:http:// cs.gmu.edu/~offutt/mujava/(дата обращения: 20.10.2016)

15. Pitest. [Электронный ресурс]. URL:http://pitest.org/ (дата обращения: 15.11.2016)

16. XML [электронный ресурс] // Java course URL:http://javacourse.ru/begin/xml/ (дата обращения 13.4.2017).

ПРИЛОЖЕНИЕ А

Программа-калькулятор для оценки качества синтеза тестов при использовании мутационного тестирования

```
package testc;
public class calc {
public int add (int a, int b){
        return a+b;
}
public int sub (int a,int b){
        return a-b;
}
}
```

Тест к программе-калькулятору для оценки покрытия сгенерированных мутантов

```
package testc;

import junit.framework.TestCase;

public class calcTest extends TestCase {
        calc c =new calc();
        public void testAdd() {
                assertEquals(4,c.add(2,2));
        }
public void testSub(){
        assertEquals(2,c.sub(4, 2));
}
}
```

ПРИЛОЖЕНИЕ Б

Программа с алгоритмом сортировки для оценки количества мутантов, генерируемых инструментами для автоматического мутационного тестирования.

Главный класс:

```
package Sort;
public class Main{
	public static void main(String[] args) {
		int []mas = new int[10];
		for( int i= 0; i < mas.length; i++){
			mas[i] = (int) Math.round(Math.random()*100);
			System.out.print(mas[i] + " ");
		}
		Sorter sort = new Sorter(mas);
		mas = sort.selectSort();
		System.out.println();
		for(int i = 0; i < mas.length; i++)System.out.print(mas[i] + " ");
		Sorter sort1 = new Sorter(mas);
		mas = sort1.bubbleSort();
		System.out.println();
		for(int i = 0; i < mas.length; i++)System.out.print(mas[i] + " ");
		Sorter2 sort2 = new Sorter2(mas);
		mas = sort2.heapSort();
		System.out.println();
		for(int i = 0; i < mas.length; i++)System.out.print(mas[i] + " ");
		}
}
```

Класс, содержащий сортировку пузырьком и сортировку выбором

```
package Sort;
```

```
public class Sorter {
	private int []mas;
	public Sorter(int []mas){
		this.mas = mas;
	}
	private void swap(int []mas, int index){
		int temp = mas[index];
		mas[index] = mas[index+1];
		mas[index+1] = temp;
	}
	public int []bubbleSort(){
		int []resMas = mas;
		for(int i = resMas.length-1; i >= 0 ; i--){
			for(int j = 0; j < i; j++){
				if(resMas[j] > resMas[j+1])swap(resMas,j);
			}
		}
		return resMas;
	}
	public int []selectSort () {
		int []resMas = mas;
		for (int i=0;i<resMas.length-1;i++) {
			int least = i;
			for (int j=i+1;j<resMas.length;j++) {
				if(resMas[j] < resMas[least]) {
					least = j;
				}
			}
```

```
                int temp = resMas[i];
                resMas[i] = resMas[least];
                resMas[least] = temp;
            }
            return resMas;
        }
}
```

Класс, содержащий пирамидальную сортировку

```
package Sort;
public class Sorter2 {
        private static int heapSize;
        private int []mas;
        public Sorter2(int []mas){
            this.mas = mas;
        }
        public int []heapSort() {
            int []resMas = mas;
            buildHeap(resMas);
            while (heapSize > 1) {
                swap(resMas, 0, heapSize - 1);
                heapSize--;
                heapify(resMas, 0);
            }
            return resMas;
        }
        private static void buildHeap(int[] mas) {
            heapSize = mas.length;
            for (int i = mas.length / 2; i >= 0; i--) {
```

```
            heapify(mas, i);
        }
    }
    private static void heapify(int[] mas, int i) {
        int l = left(i);
        int r = right(i);
        int largest = i;
        if (l < heapSize && mas[i] < mas[l]) {
            largest = l;
        }
        if (r < heapSize && mas[largest] < mas[r]) {
            largest = r;
        }
        if (i != largest) {
            swap(mas, i, largest);
            heapify(mas, largest);
        }
    }
    private static int right(int i) {
        return 2 * i + 1;
    }
    private static int left(int i) {
        return 2 * i + 2;
    }
    private static void swap(int[] mas, int i, int j) {
        int temp = mas[i];
        mas[i] = mas[j];
        mas[j] = temp;
```

```
    }
}
```

Класс, содержащий тест на проверку правильности сортировки массива, для оценки качества сгенерированных мутантов.

```
package Sort;
import junit.framework.TestCase;
public class testSort extends TestCase {
    public void testBubble(){
        int []testMas1 = new int[10];
        for( int i= 0; i < testMas1.length; i++){
            testMas1[i] = (int) Math.round(Math.random()*100);}
        Sorter sortTest = new Sorter(testMas1);
        testMas1 = sortTest.bubbleSort();
        for( int j= 0; j < testMas1.length-1; j++){
            assertTrue(testMas1[j]<=testMas1[j+1]);
            }
        }
    public void testSelect(){
        int []testMas2 = new int[10];
        for( int i= 0; i < testMas2.length; i++){
            testMas2[i] = (int) Math.round(Math.random()*100);}
        Sorter sortTest = new Sorter(testMas2);
        testMas2 = sortTest.selectSort();
        for( int j= 0; j < testMas2.length-1; j++){
            assertTrue(testMas2[j]<=testMas2[j+1]);
            }
        }
    public void testHeap(){
```

```
        int []testMas3 = new int[10];
        for( int i= 0; i < testMas3.length; i++){
            testMas3[i] = (int) Math.round(Math.random()*100);}
        Sorter2 sortTest = new Sorter2(testMas3);
        testMas3 = sortTest.heapSort();
        for( int j= 0; j < testMas3.length-1; j++){
            assertTrue(testMas3[j]<=testMas3[j+1]);
            }
        }
}
```

ПРИЛОЖЕНИЕ В

Класс, содержащий тест, проверяющий XML файлы на предмет наличия неинициализированных переменных.

```
package FSMPackege;

import java.io.File;

import java.io.IOException;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;

import org.w3c.dom.Node;

import org.w3c.dom.NodeList;

import org.xml.sax.SAXException;

import junit.framework.TestCase;

public class Test extends TestCase {

        private static final String FILENAME = "Test derivation for component
FSMs.xml";

        public void test(){

      try {

                final File xmlFile = new File(System.getProperty("user.dir")

            + File.separator + FILENAME);

      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

      DocumentBuilder db = dbf.newDocumentBuilder();

      Document doc = db.parse(xmlFile);

      doc.getDocumentElement().normalize();

      NodeList nodeList = doc.getElementsByTagName("var");

      NodeList nodeList2 = doc.getElementsByTagName("val");
```

```
        for (int i = 0; i < nodeList.getLength(); i++) {
          for (int j = 0; j < nodeList.getLength(); j++){
                    Node node = nodeList.item(i);
                    Node node2 = nodeList2.item(j);
                    assertEquals(node, node2);
                    }
        }
        } catch (ParserConfigurationException | SAXException
              | IOException ex) {
           Logger.getLogger(MainXML.class.getName())
                 .log(Level.SEVERE, null, ex);
        }
    }
}
```

Отчет о проверке № 1

дата выгрузки: 20.06.2017 13:43:00
пользователь: barxas.dongack2011@yandex.ru / ID: 4555447
отчет предоставлен сервисом «Антиплагиат»
на сайте http://www.antiplagiat.ru

Информация о документе

№ документа: 7
Имя исходного файла: Диплом.docx
Размер текста: 253 кБ
Тип документа: Не указано
Символов в тексте: 55857
Слов в тексте: 6760
Число предложений: 249

Информация об отчете

Дата: Отчет от 20.06.2017 13:43:00 - Последний готовый отчет
Комментарии: не указано
Оценка оригинальности: 92.97%
Заимствования: 7.03%
Цитирование: 0%

Оригинальность: 92.97%
Заимствования: 7.03%
Цитирование: 0%

Источники

Доля в тексте	Источник	Ссылка	Дата	Найдено в
1.45%	[1] Тестирование программного обеспечения — EDISON	http://edsd.ru	04.01.2016	Модуль поиска Интернет
1.29%	[2] «Северный (Арктический) федеральный университет имени М. В. Ломоносова»	http://rerefat.ru	23.03.2016	Модуль поиска Интернет
0.96%	[3] Приложение для работы с файлами с функциональностью Проводника Windows - Java SE (J2SE) - CyberForum.ru	http://cyberforum.ru	22.11.2016	Модуль поиска Интернет