ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

С.В. Дианов

ВЕРИФИКАЦИЯ И ПРОВЕРКА КАЧЕСТВА ПО

Учебно-методическое пособие
по выполнению практических заданий

для студентов III курса
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2021

УДК 004.05
ББК 6Ф7.3
Д44

Рецензент:
Затучный Д.А. – канд. техн. наук, профессор

Дианов С.В.
Д44 Верификация и проверка качества ПО [Текст] : учебно-методическое пособие по выполнению практических заданий / С.В. Дианов. – М.: ИД Академии Жуковского, 2021. – 36 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Верификация и проверка качества ПО» по учебному плану для студентов III курса направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 26.01.2021 г. и методического совета 26.01.2021 г.

УДК 004.05
ББК 6Ф7.3

В авторской редакции

Подписано в печать 14.05.2021 г.
Формат 60x84/16 Печ. л. 2,25 Усл. печ. л. 2,09
Заказ № 749/0330-УМП16 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

Практическое занятие №1

Формальные методы верификации. Alloy - декларативный язык спецификаций для исследования структурных ограничений и процессов в программных модулях.

Цель и задачи проведения практического занятия.

Получение практических навыков исследования структурных зависимостей в ПО и исследования накладываемых в связи с этим на них ограничений. Поиск потенциальных проблемных мест в ПО.

Методические указания по теме.

Существующие современные методы верификации программного обеспечения (ПО) условно можно разделить на эмпирические (использующие экспертизу), формальные (использующие математический аппарат для верификации программного обеспечения), динамические (анализирующие программу при её практическом запуске) и статические (анализирующие программный код).

Формальные методы – такие методы спецификации, проектирования и анализа свойств систем, которые имеют строгое математическое обоснование. Формальные методы верификации можно рассматривать как одни из самых надёжных и в то же самое время одними из самых дорогостоящих. В связи с этим при тестировании коммерческих программных продуктов применяется крайне редко. Например, в настоящее время известно, что этот подход к тестированию ПО применяется только лишь кампанией Amazon. Причина этого заключается в том, что для организации и реализации такого подхода к тестированию, необходимо иметь персонал, обладающий не только высочайшими навыками в программировании, но и обширными знаниями в высшей математике для построения математических моделей ПО. Вместе с тем следует помнить, что в соответствии с международными отраслевыми промышленными стандартами разработки ПО повышенной надежности КТ-178B(C), DO-178B(C), DO-278 применяемыми в авиастроении, ПО создаваемое для авиационной индустрии должно отвечать высочайшим критериям надёжности, причина этого в высокой стоимости ошибок в этой сфере. Это делает актуальным вопрос использования формальных методов верификации ПО, создаваемого как для бортовых, так и обслуживающих систем.

Вместе с тем, не будет лишним напомнить и всегда иметь ввиду высказывание великого Эйдсгера Вибе Дейкстры (Edsger Wybe Dijkstra) разработчика концепции структурного программирования, исследователя формальной верификации и распределённых вычислений и Тьюринговского лауреата о том, что тестирование показывает

присутствие ошибок, а не их отсутствие, то есть полностью исключить ошибки в ПО невозможно никакими методами.

Язык спецификаций Alloy можно рассматривать, как один из инструментов применения формальных методов анализа ПО. В рамках рассматриваемого подхода спецификация понимается как свойства или требования к ПО в терминах области разработки ПО. В информатике и программной инженерии Alloy - это декларативный язык спецификаций для выражения сложных структурных ограничений и поведения в программной системе. Рассматривать его в качестве языка программирования будет неверно. Alloy можно использовать как простой инструмент структурного моделирования. Alloy нацелен на создание микромоделей, которые затем могут быть автоматически проверены на правильность. На практическом занятии мы будем использовать приложение Alloy Analyzer. Скачать его можно бесплатно с официального сайта https://alloytools.org/download.html

Язык Alloy и Alloy Analyzer разработаны командой во главе с Дэниелом Джексоном в Массачусетском технологическом институте в Соединенных Штатах Америки. Первая версия языка была представлена в 1997 году. За прошедшее время был наработан огромный опыт практического использования этого программного продукта, существует международное сообщество разработчиков и тестировщиков ПО использующих Alloy - https://alloytools.org/community.html

В соответствии с рекомендациями, размещёнными на официальном сайте, Alloy предлагается запускать с использованием среды разработки или IDE (Integrated development environment) Eclipse. Для экономии времени и вычислительных ресурсов кафедральной вычислительной сети на практическом занятии можно установить Alloy как отдельно работающее приложение. Для выполнения задания необходимо так же установить среду выполнения Java (JRE). На практическом занятии мы будем использовать AlloyAnalyzer, который разработан как "облегчённая" версия Alloy, выполняющий анализ в автоматическом режиме. Непосредственно Alloy предназначен для "интерактивного доказывания теорем". AlloyAnalyzer разработан как автоматический анализатор на базе тестировщика моделей, во основе которого используется булевый SAT или B-SAT решатель (солвер) от, термин является производным от англ. Boolean satisfiability.

Alloy Analyzer представляет модель, выраженную в реляционной логике, в соответствующую булеву логическую формулу, к которой применяется готовый SAT-решатель по булевой формуле. Если решатель находит решение, результат преобразуется обратно в соответствующую

привязку констант к переменным в исходной реляционной логической модели. Модели в Alloy представляются следующими типами утверждений:

- факты (fact) - это ограничения, которые, как предполагается, всегда выполняются;

- предикаты (pred) - это параметризованные ограничения, которые могут использоваться для представления операций;

- функции (fun) - это выражения, возвращающие результаты;

- утверждения (assert) - это предположения о модели.

AlloyAnalyzer в свободном доступе расположен здесь https://github.com/beckus/AlloyAnalyzer

На практическом занятии будет рассмотрена модель веб атак на интернет магазин (Store). Эта модель описывает магазин и предоставляет ряд запросов на изменение магазина (INIT, BUY, LOGIN, STUTTER)

Затем показано, как использовать магазин с хорошим пользователем Алисой (Alice) и иметь плохого пользователя Еву (Eve), который не даёт никаких учётных данных. Поэтому Ева никогда не должна иметь возможность войти в систему или что-либо купить.

Модель описывает следующую логику:

- аутентификация, используется идентификатор пользователя для обработки таблицы. Функция digest вычисляет хэш. В этом случае мы используем то же значение для простоты использования, но это должна быть односторонняя хэш-функция, такая как алгоритм SHA-x. Учетные данные определены для Алисы, но не для Евы. То есть у Евы нет доступа к магазину (строки 11-13 листинга программы);

- состояния - это состояние в State. Для удобства это состояние поддерживает состояние как браузера (файлы cookies), так и сервера (акции, генератор токенов и т.д.). Требуется составить «предложения» с порядком. Например, Ева КУПИТЬ товар (строки 15-26 листинга программы);

- след. действий - это последовательность значений состояния. Для перехода из состояния n в состояние n + 1 необходимо, чтобы один из предикатов действия был истинным (строки 5-10 листинга программы);

- предикаты действий, логин - первый предикат действия это логин. Браузер присваивает идентификатор пользователя и пароль. Эти учетные данные должны быть аутентифицированы до успешного входа в систему. В случае успешного входа в систему нам потребуется файл cookie с токеном. Этот токен является возможностью совершать покупки в магазине. Вход не выполняется, если пользователь уже вошел в систему (строки 27-32 листинга программы);

- предикаты действий, покупка - после входа в систему в браузере появляется файл cookie с токеном, который используется для записи продажи в соответствующей корзине (строки 33-37 листинга программы);

- предикаты действий, подвисание - существует, чтобы заполнить пробелы. Если трассировка не может продвигаться, она всегда может обеспечить “подвисание”. Для команд проверки они не важны (строки 38-47 листинга программы);

- клиенты - есть браузер, как пользовательский агент, а так же два пользователя: Алиса и Ева. Алиса – хороший пользователь и получает свой пароль. Ева же плохой пользователь, и она не ограничена в использовании полномочий. На практике это означает, что Alloy будет использовать все возможные учетные данные для Евы (строки 48 - 56 листинга программы);

- трассировка - каждой комбинации всех возможных действий. Действия ограничены теми, которые разрешены предикатами действий Логина, Покупки и Подвисания. Каждый предикат действия используется для ограничения State n → State n + 1 (строки 57 - 67 листинга программы);

- проверка - того, что Ева никогда ничего не может купить и не может войти в систему (строки 74 - 77 листинга программы);

- атаки - пользователи, должны держать свои пароли в секрете (строки 68 - 70 листинга программы);

- атаки - нужно зафиксировать тот факт, что модель может предполагать, что в реальном мире файлы cookies защищены с помощью HTTPS и не могут быть украдены. В нашей модели это означает, что можно «доверять» защите файлов cookies. (строки 71 - 73 листинга программы).

Модель описана следующим кодом :

```
1.  module store
2.  open util/ordering[Token] as tk
3.  open util/ordering[State] as st
4.  some sig Item, Token {}
5.  enum Action {
6.  INIT,
7.  BUY,
8.  LOGIN,
9.  STUTTER
10. }
11. fun digest[password : String] : String { password }
12. pred authenticate[userid, password : String] {
13. userid->digest[password] in ("Alice" -> digest["Bob123"])
14. }
15. sig State {
16. browser : lone Browser,
17. action : Action,
18. bought : lone Item,
```

```
//reflects the browser state
cookies : Browser -> Token,
//reflects the server state
stock : set Item,
cart : Token -> (Item+String),
token : lone Token,
nextToken : Token
}
pred login[s, s' : lone State] {
let browser = s'.browser,
userid = browser.userid,
password = browser.password {
    a. one userid
    b. one password
    c. no s.cart.userid
    d. authenticate[userid, password]
    e. s'.action = LOGIN
    f. s'.nextToken = s.nextToken.next
    g. s'.stock = s.stock
    h. s'.cart = s.cart + (s.nextToken->userid)
    i. s'.token = s.nextToken
    j. s'.bought = none
    k. s'.cookies = s.cookies + (browser->s.nextToken)
}
}
pred buy[s, s' : State] {
let item = s'.bought,
tkn = s'.token {
    a. some item
    b. item in s.stock
    c. one tkn
    d. some s.cart[tkn]
    e. s'.action = BUY
    f. s'.nextToken = s.nextToken
    g. s'.stock = s.stock - item
    h. s'.cart = s.cart + (tkn -> item)
    i. s'.token = tkn
    j. s'.bought = item
    k. s'.cookies = s.cookies
}
}
pred stutter[s, s' : State] {
s'.action = STUTTER
s'.nextToken = s.nextToken
s'.stock = s.stock
s'.cart = s.cart
s'.browser = none
s'.token = none
s'.bought = none
s'.cookies = s.cookies
*}
abstract sig Browser {
userid : String,
password : String
```

```
}
one sig Alice extends Browser {} {
userid = "Alice"
password = "Bob123"
}
one sig Eve extends Browser {}
fact {
st/first.nextToken = tk/first
st/first.stock = Item
no st/first.bought
no st/first.cart
no st/first.browser
no st/first.cookies
st/first.action = INIT
all s' : State - first, s : s'.prev {
    a.  login[s, s']
    b.  or
    c.  buy[s, s']
    d.  or
    e.  stutter[s, s']
}
}
fact NoSharedPassword {
//  all disj b1, b2 : Browser | b1.userid = b2.userid => b1.password != b2.password
}
fact HTTPS {
//  all s : State | one s.token implies s.token = s.cookies[s.browser]
}
assert Evil {
no s : State | s.browser = Eve
}
check Evil for 4
```

Задание на практическое занятие:

1. Установить и запустить AlloyAnalyzer.
2. Открыть проект practise-class1.als, запустить его выполнение.
3. Найти результат Executing "Check Evil for 4".
4. Найти подтверждение в таблице того, что Ева украла учётные данные Алисы.
5. Исследовать визуализацию работы алгоритма в виде графа.

Контрольные вопросы:

1. Каким образом классифицируются современные методы верификации ПО ?
2. Каковы достоинства и недостатки формальных методов верификации ПО ?

3. Почему формальные методы верификации ПО следует применять в авиационной индустрии ?

4. Какие международные стандарты описывают требование к ПО применяемому в авиации ?

5. Что такое Alloy и Alloy Analyzer и в чём между ними разница ?

6. На каком математическом аппарате базируется Alloy ?

7. Какими средствами можно создать абсолютно надёжное ПО ?

8. Поясните смысл термина "спецификация" применительно к языку спецификаций Alloy.

9. Можно ли считать Alloy языком программирования ?

10. Какую информацию можно получить о ПО при выполнении Alloy - проекта ?

Практическое занятие №2

Интеграционное тестирование. Selenide - тестирование веб-приложений.

Цель и задачи проведения практического занятия.

Получить практические навыки в области интеграционного тестирования. Знакомство с системой автоматического тестирования UI (пользовательских интерфейсов) веб-приложений.

Методические указания по теме.

Интеграционное тестирование (англ. Integration testing, иногда называется англ. Integration and Testing, аббревиатура англ. I&T) — одна из фаз тестирования ПО. Этот вид тестирования проводится на этапе, когда тестирование более низкого уровня или модульное тестирование успешно завершено. Следует отметить, что этот вид тестирования полностью базируется на требованиях к ПО, выявленных в самом начале процесса создания ПО. По этой причине, в настоящее время его можно относительно просто формализовать и следовательно автоматизировать. Для этого было создано семейство систем автоматизированного тестирования, одним из представителей которого является Selenide.
Интеграционное тестирование это процесс, при котором отдельные программные модули объединяются и тестируются в группе, состоящей из связанных между собой модулей. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное

тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием метода тестирования «чёрный ящик».

Непосредственно Selenide создан на основе продукта Selenium WebDriver, который предназначен для написания приёмочных/интеграционных тестов для веб-приложений (как видно из название). В отличие от Selenium сам Selenide позволяет автоматизировать процесс тестирования программно оформив многие рутинные процессы чтобы не писать один и тот же код, например, инициализировать браузер вначале, закрыть его в конце, делать скриншоты после каждого упавшего теста или другие рутинные операции, о которых будет сказано ниже. Причём не только сказано, но и указано как выполнить и предложено это сделать в рамках данной практической работы.

Подводя итог вышесказанного, Selenide — это надстройка над Selenium WebDriver, которая позволяет сосредоточится на написании, выполнении и анализе тестов, сосредоточившись на логике работы веб-приложения, требованиях к этому веб-приложению, а не возне с браузером.

Перейдём непосредственно к установке и настройке Selenide. В рамках данной практической работы предложено использовать среду разработки IntelliJ IDEA, поскольку она используется в других практических занятиях. Кроме того она предложена к использованию на официальном сайте https://ru.selenide.org/quick-start.html Так же на этом сайте размещено пошаговое руководство по установке и началу работы с данным приложением. Важно обратить внимание на следующие моменты, необходимые для начала работы. Перед настройкой IntelliJ IDEA и прописыванием необходимых зависимостей, необходимо установить среду выполнения Java (JRE), а так же выбрать фреймворк для автоматизации сборки проектов. Напоминаю, что сборка в данном контексте означает исполняемый откомпилированный файл. Для работы с Selenide подходят фрэймворки Maven, Gradle и Ivy. Фре́ймворк это программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта. То есть это то, что позволяет в рамках среды разработки писать код только для выполняемой логики, и не отвлекаться на то, как это будет выводиться на экран, печатать, отправлять сообщения и т.д. В рамках

данной работы предлагается использовать Maven. Apache Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM (англ. Project Object Model), являющемся подмножеством XML. Соответственно для среды Java (JRE) и Maven требуется прописать переменные окружения, для того чтобы Java и Maven стали частью операционной системы. Установка этих продуктов доступна совершенно бесплатно. В интернете можно найти множество руководств как это сделать, единственное, на что стоит обратить внимание, следует скачивать исходный код с официальных сайтов для Java https://www.java.com/ru/download/manual.jsp и для Maven http://maven.apache.org/download.cgi

Нюансы работы с IntelliJ IDEA рассматриваются на другом практическом занятии, поэтому здесь они пропущены.

После настройки программного обеспечения предлагается выполнить тестирование главной страницы сайта университета МГТУГА http://www.mstuca.ru/ в соответствии с кодом pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <version>1.0-SNAPSHOT</version>
 <name>my-app</name>
 <!-- FIXME change it to the project's website -->
 <url>http://www.example.com</url>
 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
 </properties>
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.11</version>
   <scope>test</scope>
  </dependency>
  <dependency>
   <groupId>com.codeborne</groupId>
   <artifactId>selenide</artifactId>
   <version>5.16.2</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
 <build>
  <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be
moved to parent pom) -->
   <plugins>
    <!-- clean lifecycle, see https://maven.apache.org/ref/current/maven-
core/lifecycles.html#clean_Lifecycle -->
    <plugin>
     <artifactId>maven-clean-plugin</artifactId>
     <version>3.1.0</version>
```

```
            </plugin>
            <!-- default lifecycle, jar packaging: see https://maven.apache.org/ref/current/maven-
core/default-bindings.html#Plugin_bindings_for_jar_packaging -->
            <plugin>
              <artifactId>maven-resources-plugin</artifactId>
              <version>3.0.2</version>
            </plugin>
            <plugin>
              <artifactId>maven-compiler-plugin</artifactId>
              <version>3.8.0</version>
            </plugin>
            <plugin>
              <artifactId>maven-surefire-plugin</artifactId>
              <version>2.22.1</version>
            </plugin>
            <plugin>
              <artifactId>maven-jar-plugin</artifactId>
              <version>3.0.2</version>
            </plugin>
            <plugin>
              <artifactId>maven-install-plugin</artifactId>
              <version>2.5.2</version>
            </plugin>
            <plugin>
              <artifactId>maven-deploy-plugin</artifactId>
              <version>2.8.2</version>
            </plugin>
            <!-- site lifecycle, see https://maven.apache.org/ref/current/maven-
core/lifecycles.html#site_Lifecycle -->
            <plugin>
              <artifactId>maven-site-plugin</artifactId>
              <version>3.7.1</version>
            </plugin>
            <plugin>
              <artifactId>maven-project-info-reports-plugin</artifactId>
              <version>3.0.0</version>
            </plugin>
          </plugins>
        </pluginManagement>
      </build>
    </project>
```

Примеры того, что можно тестировать с помощью Selenide :

- Например, вы можете проверить, что элемент видимый (**visible**). Если пока нет, Selenide подождёт до 4 секунд.
  $(".loading_progress").shouldBe(visible);

  Можно так же проверить, что элемент не существует. Если элемент всё же найден, Selenide предположит, что он вот-вот пропадёт и подождёт до 4 секунд.
  $(By.name("gender")).should(disappear);

  Возможно сделать несколько проверок в одной строке (т.н. «fluent API» и «method chain»), что сделает тесты ещё более лаконичными:
  $("#menu") .shouldHave(text("Hello"), text("John!"))
  .shouldBe(enabled, selected);

- Selenide позволяет закачивать файлы предельно просто:
  $("#cv").uploadFile(new File("cv.doc"));
  Есть возможность закачать несколько файлов разом:
  $("#cv").uploadFile(new File("cv1.doc"), new File("cv2.doc"), new File("cv3.doc"));
  Закачивание файлов тоже крайне просто:File pdf = $(".btn#cv").download();
- проверить, что на странице ровно N таких-то элементов:
  $$(".error").shouldHave(size(3));
  Вот так можно отфильтровать подмножество элементов:
  $$("#employees tbody tr") .filter(visible) .shouldHave(size(4));
  Проверить тексты элементов. В большинстве случаев этого достаточно, чтобы проверить целую таблицу или строку в таблице:
  $$("#employees tbody tr").shouldHave(texts("John Belushi", "Bruce Willis", "John Malkovich"));
- В Selenide каждый метод умеет немножко подождать, если надо. Это называется «умными ожиданиями».
  $("#menu").shouldHave(text("Hello"));
  Selenide проверит, существует ли элемент с ID=«menu». И если нет, Selenide чуть-чуть подождёт, проверит ещё. Потом ещё подождёт. И только когда элемент появится, Selenide проверит, что у него нужный текст.

А так же множество других полезных методов

```
$("div").scrollTo();
$("div").innerText();
$("div").innerHtml();
$("div").exists();
$("select").isImage();
$("select").getSelectedText();
$("select").getSelectedValue();
$("div").doubleClick();
$("div").contextClick();
$("div").hover();
$("div").dragAndDrop()
zoom(2.5);
```

По результатам выполнения выводится отчёт об ошибках. Найденные ошибки иллюстрируются "скринами" экранов, которые сохраняются в проекте.

Задание на практическое занятие:

В соответствии с проектом произвести тестирование главной страницы сайта университета http://www.mstuca.ru/ Ознакомиться с отчётом об ошибках.

Изменить проект так, чтобы он мог произвести поиск отдельных элементов и проверку текстов.

Контрольные вопросы:

1. Что такое интеграционное тестирование ?
2. Для чего предназначен Selenide и как он связан с интеграционным тестированием ?
3. Какие операции автоматизирует Selenide ?
4. Что такое сборка ?
5. Что такое Maven и для чего он нужен ?
6. Что такое фрэймворк ?
7. Какие фрэймворки можно использовать для создания проектов на Selenide ?
8. С какими объектами работает Selenide ?
9. Что может проверить Selenide ?
10. Каким образом выводятся сообщения об ошибках в Selenide ?

Практическое занятие №3

Модульное тестирование. Интегрированная среда разработки программного обеспечения IntelliJ IDEA.

Цель и задачи проведения практического занятия.

Изучить понятие модульного тестирования. Получить практические навыки организации и проведения модульного тестирования в среде программирования IntelliJ IDEA .

Методические указания по теме.

Модульное тестирование, которое так же известно как блочное тестирование или юнит-тестирование (англ. *unit testing*) — это процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. В рамках этого подхода исследуются так же наборы из одного или более программных модулей. Для получения корректных результатов в процессе тестирования используются управляющие данные, и взаимодействующие с исследуемыми модулями функции и методы.

Суть такого подхода заключается в том, чтобы писать тесты для каждой оригинальной функции или метода. Это позволяет достаточно быстро определить, не привёл ли рефакторинг исходного кода к ошибкам в

тех местах, где их раньше не было. Соответственно в результате устранение ошибок и "багов" происходит максимально оперативно.

Модульное тестирование является "рабочей лошадкой" всего процесса верификации и проверки качества ПО. Соответственно этому виду тестирования уделяется наибольшее внимание при планировании и организации работ по верификации ПО.

В рамках настоящей практической работы будет рассмотрена технология организации модульного тестирования в интегрированной среде разработки программного обеспечения IntelliJ IDEA с подключённой библиотекой JUnit. Стоит упомянуть, что существуют и другие библиотеки или среды модульного тестирования, такие как TestNG, Spock и многие другие. Соответственно перед началом работы в этой среде, её надо корректно развернуть и настроить. Среда разработки IntelliJ IDEA является разработкой кампании JetBrains. Для начала необходимо загрузить дистрибутив с официального сайта https://www.jetbrains.com/idea/download. По этому адресу можно найти пакеты для Windows, MacOS, Linux. Кроме того, сама среда доступна в двух версиях - Ultimate (платная с пробным бесплатным периодом) и Community (бесплатная). В данном случае выберем бесплатную версию Community.

После установки и запуска IntelliJ IDEA следует создать новый проект. Для этого на стартовом экране следует выбрать опцию New Project. В качестве сборщика выбираем Maven (подробнее про сборщики и их назначение обсуждаем на другом практическом занятии). Указываем Groupid и Artifactid - соответственно название кампании и имя проекта, следует указать mstuca и своё имя. Далее следуя подсказкам, завершаем создание проекта.

В созданном проекте появился уже знакомый по предыдущей практической работе файл pom.xml, в котором содержится информация о составе проекта. XML (eXtensible Markup Language) — расширяемый язык разметки. Так же как и при настройке проекта для Selenide в pom.xml добавляются "зависимости" - это указатели или идентификаторы библиотек, используемых в проекте. Теперь в папке проекта содержатся папки main и test, в которых в папках java будут расположены коды программы и тестов соответственно.

В папке исходного кода java, той что синего цвета, создаём тестируемый класс. В данном случае их предложено два, чтобы метод main был в отдельном классе. Сделано это исключительно для удобства и упрощения написания теста. Можете создать свой класс или классы или выбрать программу, суммирующую два введённых в консоли числа, к которым прибавляется третье. Код приведён ниже.

```
import java.util.Scanner;
public class summator {
    public static void main(String[] args) {
        // write your code here
        Scanner sc = new Scanner(System.in);
```

```
        int a, b, c;
 System.out.println("Введите первое число");
        a = sc.nextInt();//Считываем первое число
        System.out.println("Введите второе число");
        b = sc.nextInt();//Считываем второе число
        mult Mult = new mult();
        int k = Mult.incr(5);
        c = a + b + k;
        System.out.println("c = " + c);
    }}
```

И второй класс

```
public class mult {
    public int incr(int N) {
    int rez = N+5;
    return rez;
    }}
```

Выше описан тестируемый класс, который нужно создать в папке java, пусть он будет называться summator. Второй класс mult , в котором прописан метод incr. После настройки и согласования версии Java и компилятора Maven запускаем программу на выполнение. В соответствии с логикой работы программа запрашивает в консоли два числа и выводит их сумму, прибавляя некое число, определяемое методом incr.

Следующий шаг это создание теста. В папке mult.java , наведя курсор на имя класса mult, нужно нажать комбинацию Alt + Enter . Далее следуя подсказкам в выпадающем окне создаётся тестирующий класс multTest. В процессе создания нужно определиться с библиотекой Junit4 или Junit5. Разница заключается в закачивании необходимых обновлений. Так же в процессе создания тест-класса можно выбрать опцию проведения тестирования до или после выполнения кода. В папке test-java должен появиться тест multTest. Одновременно с этим в файле pom.xml прописались библиотеки (dependency) Junit4 или Junit5. Теперь multTest содержит каркас теста, который нужно наполнить содержанием. Листинг теста предложен ниже

```
import org.junit.Test;
import static org.junit.Assert.*;
public class multTest {
    @Test
    public void incr() {
        mult Mult = new mult();
        int actual = Mult.incr(6);
        int expected;
        expected = 11;
        assertEquals("Unexpected string value",expected,actual);
     }}
```

Далее запускаем тест на выполнение. Меняя значение expected можно видеть результат прохождения теста. Если результат соответствует ожиданиям, можно увидеть сообщение "Process finished with exit code 0". Если программа завершилась не так, как требовалось, сообщение может выглядеть так

```
java.lang.AssertionError: Unexpected string value
```

```
Expected :12
Actual   :11
<Click to see difference>
```

Задание на практическое занятие:

Создать проект в среде разработки IntelliJ IDEA в соответствии с методическими указаниями. Провести тестирование класса mult проекта summator, изменяя константу expected. Изучить отчёты о положительных и отрицательных результатах прохождения тестов.

Контрольные вопросы:

1. Что такое модульное тестирование ?
2. Место модульного тестирования в общем процессе верификации ПО ?
3. На каком этапе создания ПО применяется модульное тестирование ?
4. Что такое IntelliJ IDEA ?
5. Каким образом IntelliJ IDEA готовится для проведения модульного тестирования ?
6. Что такое Maven и для чего он нужен ?
7. Что такое сборка ?
8. Что такое Junit ?
9. Что содержит файл pom.xml в рамках проекта в IntelliJ IDEA ?
10. Что такое dependency в файле pom.xml ?

Практическое занятие №4

Статический анализ программного кода. PVS-Studio - инструмент статического анализа.

Цель и задачи проведения практического занятия.

Ознакомиться с подходами к организации статического анализа программного кода. Получить практические навыки организации и проведения статического анализа программ, написанных на языках программирования Java, C и C++.

Методические указания по теме. Краткие теоретические сведения.

Статический анализ кода (англ. static code analysis) или ещё можно встретить термин Static Application Security Testing (SAST) — это такой вид

исследования программного обеспечения, который производится (в отличие от динамического анализа) без реального запуска на выполнения исследуемых программ. Как правило статический анализ осуществляется на начальных этапах верификации ПО. В большинстве случаев анализ производится над какой-либо версией исходного кода, написанном на процедурном языке программирования, хотя вместе с вышеуказанным анализироваться может уже откомпилированный код, представленный в виде сборки. Термин обычно применяют к анализу, производимому специальным программным обеспечением (ПО). Как это происходит, увидим в процессе выполнения данной практической работы.

В зависимости от используемого средства автоматического тестирования глубина анализа может варьироваться от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Соответственно перед организатором процесса верификации стоит задача выбора соответствующего инструмента и соответствующих этим инструментам вычислительных мощностей. Кроме того, необходимо организовать работу персонала, владеющего этими инструментами и могущего эффективно их использовать.

Далее встаёт вопрос анализа и использования полученной в ходе статического анализа информации. Способы использования полученной в ходе анализа информации также различны — от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации).

Кроме очевидной задачи выявления ошибок, статический анализ позволяет решать и другие задачи, как например, рекомендовать корректное оформление кода, подсчитывать количество показателей, или метрик, обуславливающих качество программного обеспечения.

Как любой метод статический анализ не является идеальным и обладает своими достоинствами и недостатками. К числу достоинств принято относить:

1. выявление большого количества ошибок на этапе конструирования ПО даже в тех фрагментах кода, которые невозможно протестировать другими методами;
2. независимость от среды и компилятора, что дает возможность обнаруживать скрытые ошибки;
3. быстрое обнаружение опечаток и дублирования символов.

"Ложками дёгтя" применительно к этому методу можно считать:

1. невозможность или большое затруднение выявления ошибок, для обнаружения которых необходимо выполнить программу или ее часть;

2. так называемые ложно-позитивные срабатывания — фрагменты правильного кода, тем не менее, вызывающие у анализатора подозрения.

Основная ценность статического анализа заключается в его регулярном использовании, так что ошибки выявляются и фиксируются на самых ранних этапах. Нет никакого смысла тратить время на поиски ошибки, которую можно было бы найти с помощью статического анализа. Итак, еще раз подчеркнем - основная идея статического анализа заключается не в том, чтобы найти одну скрытую ошибку за день до релиза, а в том, чтобы ежедневно исправлять десятки ошибок.

PVS-Studio-это инструмент для обнаружения ошибок и слабых мест безопасности в исходном коде программ, написанных на языках C, C++, C# и Java. Анализатор кода может работать ночью на сервере и автоматически предупреждать о подозрительных фрагментах кода. В идеале, эти ошибки могут быть обнаружены и исправлены до попадания в систему управления версиями. PVS-Studio может быть автоматически запущен сразу после запуска компилятора для файлов, которые были только что изменены. Он работает в Windows, Linux и macOS.

Основная информация об инструменте PVS-Studio размещена на официальном сайте https://www.viva64.com/ru/pvs-studio/ Он работает в 64-разрядных системах в средах Windows, Linux и macOS и может анализировать исходный код, предназначенный для 32-разрядных, 64-разрядных и встроенных платформ ARM (Advanced RISC Machine).

PVS-Studio выполняет статический анализ кода и формирует отчет, который помогает программисту находить и исправлять ошибки. PVS-Studio выполняет широкий спектр проверок кода, а также полезен при поиске опечаток и ошибок копирования-вставки.

Теперь рассмотрим процесс установки и запуска. Стоит сразу указать, что в общем случае, это программный продукт является платным. Однако, разработчиками любезно предоставлена возможность использовать PVS-Studio бесплатно в учебном процессе, а так же индивидуально для студентов. Этой возможностью мы воспользуемся в рамках данной практической работы.

Итак, скачать дистрибутив следует с официального сайта, указанного выше. Условия бесплатного использования в учебном процессе студентами и преподавателями прописаны на том же сайте здесь https://www.viva64.com/ru/for-students/ , где указано, что есть три способа бесплатного использования PVS-Studio в учебных целях:

1. Добавление специальных комментариев в код;
2. Использование сайта Compiler Explorer;
3. Бесплатная версия для открытых проектов.

Нас интересуют первые два пункта. В рамках первого пункта мы будем использовать уже знакомую среду IntelliJ IDEA. Анализатор PVS-

Studio Java можно использовать в виде плагина к IntelliJ IDEA. В таком случае разбор структуры проекта производится средствами этой IDE, а плагин предоставляет удобный графический интерфейс для работы с анализатором. Плагин PVS-Studio для IDEA можно установить из официального репозитория плагинов JetBrains https://plugins.jetbrains.com/plugin/12263-pvs-studio . Ещё один способ установки (включая ядро анализатора) - через установщик PVS-Studio для Windows, доступный на официальном сайте разработчика. Страница загрузки расположена здесь https://www.viva64.com/ru/pvs-studio-download/. Ниже приведен порядок установки плагина из репозитория.

1) File -> Settings -> Plugins
2) Manage Plugin Repositories
3) Add repository (http://files.viva64.com/java/pvsstudio-idea-plugins/updatePlugins.xml)
4) Install

После этого нужно ввести данные лицензии.

1) Analyze -> PVS-Studio -> Settings
2) Вкладка Registration:

Поскольку мы будем использовать бесплатную версию, предназначенную для использования в образовательном процессе, лицензионный ключ будет выглядеть следующим образом:

Name: PVS-Studio Free
Key: FREE-FREE-FREE-FREE

На вкладке Registration появится сообщение зелёного цвета "valid license". Кроме того, в начале каждого файла с кодом следует вставить следующие комментарии:

// This is a personal academic project. Dear PVS-Studio, please check it.
// PVS-Studio Static Code Analyzer for C, C++, C#, and Java: http://www.viva64.com

Теперь можно запустить анализ текущего проекта.

Другая опция выполнения практической работы это исследование программы, состоящей из одного файла, с помощью сайта Compiler Explorer (godbolt.org). Сайт в online режиме позволяет писать, компилировать и запускать программы на языках Ada, C, C++, D, Fortran, Go, Pascal, Rust и так далее. Особенно удобно, что можно сгенерировать ссылку на получившийся код и отправить его на проверку. Если программа разрабатывается на языке C и C++, то дополнительно её можно сразу проверять с помощью PVS-Studio и знакомиться на практике с работой статического анализатора кода. В процессе проведения анализа кода, экран будет разделён на следующие основные зоны:

1. Окно редактирования текста программы;
2. Получившийся ассемблерный код;
3. Результат работы программы;
4. Предупреждения анализатора PVS-Studio.

Основные возможности PV S-Studio :

1. Автоматический анализ отдельных файлов после их перекомпиляции
2. Онлайновое справочное руководство для всех диагностических правил, которое доступно локально, на нашем веб-сайте и в виде единого документа .файл PDF. Более 600 страниц документации!
3. Сохранение и загрузка результатов анализа позволяют выполнять ночные проверки - в течение ночи анализатор выполняет сканирование и предоставляет вам результаты утром.
4. Вы можете сохранить результаты анализа в виде HTML с полной навигацией по исходному коду.
5. Анализ может быть выполнен из командной строки: это помогает интегрировать PVS-Studio в ночные сборки; свежий журнал будет выпущен утром.
6. Отличная масштабируемость: поддержка многоядерных и многопроцессорных систем с возможностью указания количества используемых ядер; поддержка распределенного анализа IncrediBuild.
7. Интерактивная фильтрация результатов анализа (файла журнала) в окне PVS-Studio: по номеру диагностического правила,имени файла, ключевому слову в тексте диагностики и др.
8. Автоматическая проверка наличия обновлений (внутри IDE и при выполнении ночных сборок).
9. Утилита BlameNotifier. Инструмент позволяет отправлять разработчикам по электронной почте уведомления об ошибках, обнаруженных PVS-Studio во время ночного запуска.
10. Большое количество вариантов интеграции в проекты, которые разрабатываются под Linux и macOS.
11. Пометка как ложная тревога-возможность пометить фрагмент кода для подавления определенной диагностики в этой строке.
12. Массовое подавление-возможность подавления всех существующих сообщений анализатора, вызванных для устаревшего кода, так что анализатор начинает выдавать 0 предупреждений. Вы всегда можете вернуться к подавленным сообщениям позже. Эта функция позволяет легко интегрировать PVS-Studio в процесс разработки и сосредоточиться только на ошибках, обнаруженных в новом коде.
13. Статистику по предупреждениям анализатора можно посмотреть в Excel-это способ отслеживать скорость исправления ошибок, количество найденных ошибок за определенный промежуток времени и так далее.
14. Относительные пути в файлах отчетов для их просмотра на разных машинах.
15. Функция мониторинга компилятора позволяет анализировать проекты, не имеющие файлов Visual Studio (.ФСЛ./vcxproj) без необходимости вручную интегрироваться с системой сборки; ручная интеграция в любую систему сборки возможна, если это необходимо.

16. pvs-studio-analyzer-инструмент для мониторинга компиляторов под Linux.

17. Возможность исключить файлы из анализа по имени, папке или маске; чтобы запустить анализ на файлы, измененные в течение последних N дней.

18. Интеграция с SonarQube-это платформа с открытым исходным кодом, предназначенная для непрерывного анализа и измерения качества кода.

Задание на практическое занятие:

Провести статический анализ кода программы в среде IntelliJ IDEA с помощью плагина PVS-Studio. Для исследования можно выбрать проект, созданный на предыдущем практическом занятии или же создать проект самостоятельно. При использовании созданного проекта, по результату работы PVS-Studio появится сообщение "Congratulations ! PVS-studio has not found any issues in you source code !". Измените программу так, чтобы в результате работы анализатора PVS-Studio появились замечания и предупреждения. Изучите форму отчёта, которую предоставляет PVS-Studio.

Исследуйте программный код с помощью сайта Compiler Explorer (godbolt.org).

Сравните результаты полученные в обоих случаях.

Контрольные вопросы:

1. Что такое статический анализ ПО ?
2. Какие преимущества статического анализа ПО ?
3. Какие недостатки статического анализа ПО ?
4. На каком этапе процесса верификации ПО проводится статический анализ ПО ?
5. Какими средствами производится статический анализ ПО ?
6. Какая информация может быть получена в результате проведения статического анализа ПО ?
7. Для чего предназначена PVS-Studio ?
8. Что является исходными данными для статического анализа ПО ?
9. Может ли плагин PVS-Studio самостоятельно исправлять найденные ошибки ?
10. Какую информацию и в каком виде предоставляет PVS-Studio по результатам выполнения статического анализа кода программы ?

Практическое занятие №5

Системный анализ ПО. Katalon Recorder - инструмент для записи тестовых сценариев веб-приложений.

Цель и задачи проведения практического занятия.

Ознакомиться с подходами к организации и проведения системного тестирования программного кода. Получить практические навыки проведения системного тестирования веб-приложений с помощью средств автоматизированного тестирования Katalon.

Методические указания по теме. Краткие теоретические сведения.

Системное тестирование программного обеспечения — это тестирование ПО, выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям. Как следует из определения, системное тестирование - это завершающий этап верификации ПО, который проводится после интеграционного тестирования. Системное тестирование относится к методам тестирования на основе "чёрного" ящика, и, тем самым, не требует знаний о внутреннем устройстве системы. Следовательно персонал, проводящий системное тестирование, не обязательно должен обладать квалификацией, позволяющей погружаться в глубины написания и организации программного кода. В рамках системного тестирования различают альфа-тестирование и бета-тестирование.

Альфа-тестирование - это имитация реальной работы с системой штатными разработчиками, но это на случай экономии средств. Для ПО повышенной надёжности это должна быть реальная работа с системой потенциальными пользователями или заказчиком. Хотя альфа-тестирование может проводится на ранней стадии разработки продукта, но нас интересует случай, когда оно применяется для законченного продукта в качестве внутреннего приёмочного тестирования. Для исключения влияния ошибок и "багов" в ПО, которые могут привести к существенным материальным и имиджевым потерям, альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки и локализовывать их. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа. Получается, что одни тестировщики передают данные другим тестировщиком. Здесь нет противоречия, это всего лишь значит, что тестировщики, производящие системное тестирование, не обладают квалификацией достаточной даже для того, чтобы локализовать ошибку.

Бета-тестирование — применяется выборочно, в некоторых случаях, предполагает распространение предварительной версии ПО для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Таким образом, бета-тестирование производится даже не тестировщиками, а обычными пользователями. Этот этап можно рассматривать уже не как этап создания ПО, а как начальный шаг продвижения или маркетинга. Так же бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей. В некоторых случаях, бета-тестирование рассматривают как этап исправления ошибок. Однако, это категорически исключено в соответствии с международными отраслевыми промышленными стандартами разработки ПО повышенной надежности КТ-178B(C), DO-178B(C), DO-278 применяемыми в авиастроении. В соответствии с этими документами тестировщику, категорически запрещено вносить какие либо изменения в тестируемый продукт.

Системное тестирование ПО может проводится в ручном режиме, но это в самом примитивном случае. Обычно принято использовать специальные инструменты для записи действий пользователя (тестировщика). То есть в итоге запускается сама программа, но щелканье по кнопкам осуществляется автоматически. Правило такое — на каждый тестовый сценарий пишется по скрипту, который описывает действия пользователя. Если все тестовые сценарии покрыты тестами и тесты проходят, то можно сдавать систему заказчику, подписав акт сдачи-приемки.

В качестве примера системы автоматического системного тестирования рассмотрим Katalon Recorder, который поддерживается последними версиями Firefox и Chrome (также будет поддерживаться их последующими версиями) и может быть добавлен в качестве расширения для этих браузеров. Данный инструмент является отличной и бесплатной альтернативой Selenium IDE, Silk test, TestComplete, Ranorex, Tricentis Tosca, Micro Focus UFT(QTP), QuicktestPro, а также другим подобным фреймворкам с открытым исходным кодом. Katalon является развитием рассмотренной на одном из предыдущих практических занятий системы Selenide. Katalon Recorder – это удобный и мощный инструмент, поддерживающий возможность записи тестовых сценариев, в котором основные функции Selenium IDE были импортированы для Chrome и Firefox. Katalon Recorder поддерживает два дополнительных фреймворка: Robot Framework и Katalon Studio, которые позволяют тестировать веб-интерфейс выбранного проекта в привычной для неподготовленного тестировщика среде. Кроме того, данный инструмент совместим с языком программирования Groovy. Katalon Recorder позволяет записывать произведённые тестировщиком действия, собственно поэтому он и Recorder,

и сохранять их в качестве автотеста. Ниже перечислены основные особенности приложения:

1. С помощью данного инструмента вы можете захватить необходимые элементы веб-приложения и записать выполняемые с ними действия. Вы можете создавать новые тестовые случаи, редактировать существующие, а также воспроизводить автоматические тестовые случаи. Katalon Recorder позволяет выполнять все эти действия легко и быстро.
2. Все поддерживаемые Selenide функции доступны в Katalon Recorder.
3. Тестовые сценарии для Selenium IDE могут быть импортированы в Katalon Recorder (в скриптовом режиме) и экспортированы для приведенных ниже языков программирования и фреймворков.
4. Изменение языка программирования или фреймворка приведет к автоматическому изменению синтаксиса сценария:
5. Katalon Recorder поддерживается последними версиями Firefox и Chrome.
6. Данный инструмент работает на ядре Selenium 3.
7. Несколько тестовых случаев могут быть объединены в один тестовый набор. Такие объединенные тестовые случаи будут выполняться в виде полного тестового набора.
8. Тестовые случаи могут быть импортированы в Katalon Studio – полностью бесплатный инструмент для автоматизации тестирования, поддерживающий key-driven и data-driven подходы, TDD/BDD-тестирование, тестирование API, а также включающий такие возможности как: применение паттерна Page Object Model, управление и запись процесса тестирования, написание расширенных скриптов, параллельное выполнение, выполнение CI-интеграции (continuous integration) и многие другие возможности!
9. Katalon Recorder позволяет записывать и воспроизводить автоматизированные тестовые сценарии не только пользователям Katalon Studio, но и тем, кто пользуется устаревшей версией Selenium IDE и другими популярными фреймворками с открытым исходным кодом.
10. Существует ряд команд, которые на данный момент находятся на стадии разработки. Например: sendKeys является экспериментальной командой. Вероятно, в будущем эта команда будет заменена на другую – typeKeys. Таким образом, есть вероятность того, что при использовании экспериментальных команд придется обновлять некоторые тестовые сценарии.
11. В инструменте отсутствует функция отображения базового URL-адреса. В Selenium IDE такая функция была очень удобна для инициализации тестовых случаев в нескольких разных доменах.
12. Кроме описанных выше недостатков в инструмент также есть и другие ошибки. В разделах «Suggestions» (Предложения) и «Katalon Automation Recorder Bugs» (Журнал ошибок Katalon Automation) вы можете найти информацию о существующих ошибках и способах их устранения.

Скачать приложение можно бесплатно с сайта https://www.katalon.com/ , предварительно зарегистрировавшись. Далее можно установить Katalon Recorder, как расширение к браузеру, или использовать для работы Katalon Studio. Katalon Studio имеет удобный и понятный интерфейс, начало работы сопровождается подсказками и примерами. При первом запуске, в качестве примера автоматически запускается тестирование сайта amazon.com Созданные автотесты сохраняются либо в приложении, если это Katalon Studio, либо в личном кабинете на сайте https://www.katalon.com/ , в случае если это расширение Katalon Recorder. Собственно автотест в данном случае это запись действий тестировщика на тестируемом сайте, которые в случае их сохранения, воспроизводятся автоматически. Результаты выполнения автотестов выводятся в удобной и наглядной форме.

Прописывать пошагово процесс создания автотеста в рамках данного пособия я не вижу смысла по причине того, что во-первых интерфейс приложения чрезвычайно прост и дружелюбен к пользователю, а во-вторых может обновляться, и порядок действия может меняться. Подчеркну лишь ещё раз, что как инструмент системного тестирования Katalon Recorder доступен для персонала самой невысокой квалификации.

Подводя итоги:

1. Данный инструмент можно установить очень легко и быстро.
2. Он полностью бесплатный.
3. Прост в использовании. В разделе «Reference» (Справка) указана полезная информация о том, каким образом работают команды и в каких случаях их необходимо использовать.
4. Инструмент поддерживает несколько языков программирования и фреймворков, что только увеличивает его эффективность.
5. В нем сохранены практически все функции, присутствующие в Selenium IDE.
6. Кроме того, в Automation Recorder было добавлено несколько новых команд, которых нет в Selenium IDE, что делает его еще более эффективным инструментом.
7. Возможность интеграции с платформой отчетов интеллектуального тестирования – Katalon

Задание на практическое занятие:

С помощью приложения Katalon Studio и Katalon Recorder создать, сохранить и запустить автотест сайта, предложенного преподавателем. Проанализировать полученный отчёт о прохождении теста. Изучить интерфейс и возможности приложения Katalon Studio.

Контрольные вопросы:

1. Что такое системное тестирование ?
2. В чём заключается различие системного и интеграционного тестирования ?
3. Исследуемое ПО рассматривается как "чёрный", "белый" или "серый" ящик в случае проведения системного тестирования ?
4. Что такое альфа-тестирование ?
5. Что такое бета-тестирование ?
6. Какие требования предъявляются к персоналу, проводящему системное тестирование ?
7. Для чего применяется Katalon Recorder ?
8. Назовите известные вам фреймворки для тестирования веб-приложений.
9. Какое противоречие может возникнуть при бета-тестировании ПО применяемому в авиационной индустрии ?
10. Перечислите преимущества применения автотестов для тестирования веб-приложений.

Практическое занятие №6

Фаззинг тестирование. MUll Mutation jittester - инструмент для фаззинг тестирования.

Цель и задачи проведения практического занятия.

Ознакомиться с подходами к организации и проведения фаззинг тестирования программного кода. Получить практические навыки проведения фаззинг тестирования программного кода с помощью MUll Mutation jittester.

Методические указания по теме. Краткие теоретические сведения.

Фаззинг (Fuzzing) — это методика, или так же можно сказать способ, тестирования программного обеспечения, суть которой заключается в автоматизированном обнаружении ошибок в программном коде путем отправки заведомо неверных данных и анализе реакции программы на них. Фаззинг следует отнести к методам динамического анализа кода. С помощью фаззинга можно обнаружить такие ошибки в рабочем коде, которые иначе никуда не исчезнут даже по прошествии времени и возможно будут проявляться самым неожиданным образом. Тут следует сразу оговориться, что технологию фаззинга применяют к программным продуктам, которые уже могли пройти основные этапы тестирования. Как правило, в настоящее

время фаззингом завершают процесс разработки, но "фаззить" можно и отдельные функции разрабатываемого продукта в процессе процесса верификации ПО. Фаззинг позволяет повысить устойчивость уже разработанных и введённых в промышленную эксплуатацию приложений, выявить трудно определяемые уязвимости программного кода. Например, известен случай, когда фаззинг позволил обнаружить около пятидесяти CVE (Common Vulnerabilities and Exposures - общеизвестные уязвимости информационной безопасности) в Adobe Reader за 50 дней. Исследователи смогли найти такое количество уязвимостей, не имея доступа к исходному коду.

Основные отличительные преимущества фаззинга перед другими методами тестирования:фаззер можно запустить и забыть о нём на несколько часов или даже дней окончания тестирования, а работать уже с результатами, тестировщик может вечером уйти домой, а утром, придя на работу, обнаружить, что обнаружена очередная уязвимость кода. Таким образом:

1. автоматизированное тестирование в силу естественных причин, за счёт большего покрытия кода, позволяет охватить больший объём кода, напомним, что до недавних пор считалось, что покрытие тестами 50% кода - это очень хороший показатель;
2. позволяет сделать заключение о степени защищенности тестируемого кода.

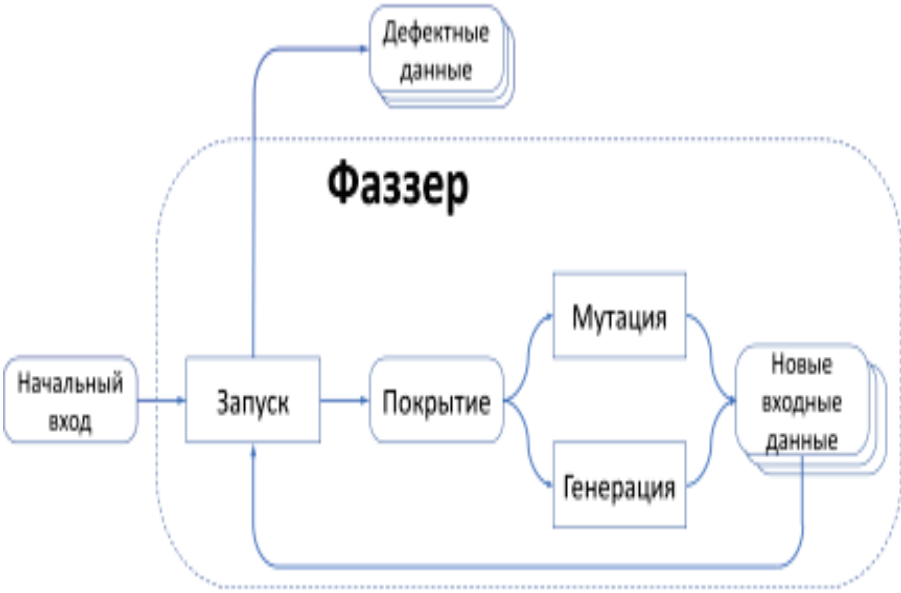
Для организации фаззинг тестирования тестировщику необходимо иметь общее понимание процесса, а так же обладать необходимыми навыками, приобретению которых посвящена данная практическая работа.

Основной отличительной особенностью, "изюминкой" фаззинга является подача случайных или сформированные особым алгоритмом данных вместо ожидаемых входных данных программе. Обычно это либо некорректно составленные данные, либо данные находящиеся в некоторой "серой" зоне. То есть не то чтобы совсем неприемлемые, но несколько отличающиеся от обычных. Например, если у нас задан диапазон изменения некоторой величины $|\mathbf{x}| < \mathbf{5}$, в процессе фаззинга заменяется на $|\mathbf{x}| \leq \mathbf{5}$. Вот такие небольшие "провокации" программного кода позволяют вскрывать "баги" и уязвимости. Таким образом, появляется возможность обнаружить непредусмотренные техническим заданием на ПО входные данные, которые приводят к аварийному завершению программы или к её некорректному поведению. Входные данные могут передаваться через файлы, сетевые сокеты, API, стандартного входа, переменные окружения и т. д. Предметом интереса процедуры фаззинга являются, например, такие случаи как, если программа входит в бесконечный цикл или аварийно завершает работу, это считается нахождением дефекта в программе, который может привести к обнаружению определенной уязвимости. К наличию подобных проблемные

места особенно ПО повышенной надежности, разработанном в соответствии с международными стандартами КТ-178B(C), DO-178B(C), DO-278 применяемыми в авиастроении.

В работе [10] предложена классификация основных этапов фаззинга и приведена принципиальная схема:

1. Определение цели.
2. Определение протокола и типа входных данных. Для старта анализа требуются начальные входные данные.
3. Изменение данных с учетом особенностей протокола передачи и типа данных. Изменение входных данных производится двумя методами — мутацией и генерацией.
4. Исполнение программы с подачей ей на вход некорректных данных.
5. Обнаружение ошибок, с целью определения на каких данных или цепочке данных обнаружилась уязвимость, а также подсчет покрытия по базовым блокам.
6. Исследование уязвимости на возможность её эксплуатации.



Наиболее известными инструментами выполнения фаззинга являются:

- MUll Mutation jittester, с которым мы имеем дело на данном практическом занятии;
- AFL (American fuzzy lop) —который использует обширный список типов инструментации кода для получения информации о покрытии и

множество генетических алгоритмов мутации для автоматического обнаружения различных тестовых примеров, которые вызывают новые внутренние состояния в бинарном коде ПО. Будет рассмотрено на другом практическом занятии;

- LibFuzzer — это часть проекта LLVM (Low Level Virtual Machine — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-подобными инструкциями), позволяющий производить анализ ПО без его перезапуска, осуществляя мутацию напрямую в памяти. Данный инструмент собирает информацию о покрытии кода, и накапливает тестовые примеры, приводящие к его увеличению. Будет рассмотрено на другом практическом занятии;

- Syzkaller — это инструмент для исследования и поиска ошибок и уязвимостей в ядрах ОС.

Отметим инструменты для фаззинга созданы под операционную систему Linyx, соответственно являются проектами с открытым кодом, которые доступны в репозитарии Github.

Перед началом ознакомления с MUll Mutation jittester, приведём определение ещё одного важного понятия, связанного с фазингом - это мутационное тестирование. На этот случай в википедии дана ёмкая и толковая формулировка.

Мутационный анализ или мутация программ — это метод тестирования программного обеспечения, который включает небольшие изменения кода программы. Если набор тестов не в состоянии обнаружить такие изменения, то он рассматривается как недостаточный. Эти изменения называются мутациями и основываются на мутирующих операторах, которые или имитируют типичные ошибки программистов (например, использование неправильной операции или имени переменной) или требуют создания полезных тестов.

Mull - представдяет собой инструмент для тестирования мутаций, основанный на LLVM / Clang с упором на языки C и C ++ . На странице https://mull.readthedocs.io/en/latest/ выложена вся документация. Непосредственно сам проект расположен по адресу https://github.com/mull-project/mull

Mull имеет 3 уровня тестирования:

1. Модульное и интеграционное тестирование на уровне классов C ++
2. Интеграционное тестирование известных реальных проектов, таких как OpenSSL (криптографическая библиотека с открытым исходным кодом)
3. Интеграционное тестирование с использованием LLVM Integrated Tester (в процессе)

Рассмотрим подробнее концепцию мутационного тестирования:

- Ошибки (мутации), которые задаются доступными операторами, автоматически встраиваются в код, а далее тесты запускаются на

выполнение, в рамках этой практической работы это осуществляет MUll Mutation jittester. Если тесты падают, считается, что мутация “убивается”, а если проходят – она выживает.

- Количество убитых мутаций определяет качество тестирования. Чем больше убитых мутаций, тем тест более успешный.
- Качества исследований может определятся метрикой MSI (Mutation Score Indicator) — процентное отношение между убитыми и выжившими мутантами. Чем больше разница между покрытием кода тестами и MSI, тем хуже отражает актуальность наших тестов процент покрытия кода.

Вот пример доступных операторов мутации :

Operator Name	**Operator Semantics**
cxx_add_assign_to_sub_assign	Replaces += with -=
cxx_add_to_sub	Replaces + with -
cxx_and_assign_to_or_assign	Replaces &= with |=
cxx_and_to_or	Replaces & with |
cxx_assign_const	Replaces ‘a = b’ with ‘a = 42’
cxx_bitwise_not_to_noop	Replaces ~x with x
cxx_div_assign_to_mul_assign	Replaces /= with *=
cxx_div_to_mul	Replaces / with *
cxx_eq_to_ne	Replaces == with !=
cxx_ge_to_gt	Replaces >= with >
cxx_ge_to_lt	Replaces >= with <
cxx_gt_to_ge	Replaces > with >=
cxx_gt_to_le	Replaces > with <=
cxx_init_const	Replaces ‘T a = b’ with ‘T a = 42’
cxx_le_to_gt	Replaces <= with >
cxx_le_to_lt	Replaces <= with <
cxx_logical_and_to_or	Replaces && with ||
cxx_logical_or_to_and	Replaces || with &&
cxx_lshift_assign_to_rshift_assign	Replaces <<= with >>=
cxx_lshift_to_rshift	Replaces << with >>

Подробная инструкция по работе с MUll Mutation jittester изложена в документаци по приведённой выше ссылке, здесь приведём кратко основные особенности технологии использования этого продукта:

1. Для применения этой технологии у нас, очевидно, должен быть исходный код (source code), некоторый набор тестов (для простоты будем говорить о модульных — unit tests).
2. После этого можно начинать изменять отдельные части исходного кода и смотреть, как реагируют на это тесты. Одно изменение исходного кода будем называть Мутацией (Mutation). Например, изменение бинарного оператора "+" на бинарный "-" является мутацией кода.
3. Результатом мутации является Мутант (Mutant) — то есть это новый мутированный исходный код.
4. Каждая мутация любого оператора в вашем коде (а их сотни) приводит к новому мутанту, для которого должны быть запущены тесты. Кроме изменения "+" на "-", существует множество других мутационных операторов (Mutation Operator, Mutator) — отрицание условий, изменение возвращаемого значения функции, удаление строк кода и т.д.
5. Итак, мутационное тестирование создает множество мутантов из вашего кода, для каждого из них запускает тесты и проверяет, выполнились они успешно или нет.
6. Если тесты упали — значит всё хорошо, они отреагировали на изменение в коде и поймали ошибку. Такой мутант считается убитым (Killed mutant).
7. Если тесты выполнились успешно после мутирования — это говорит о том, что либо ваш код не покрыт в этом месте тестами вовсе, либо тесты, покрывающие мутированную строку, неэффективны и в недостаточной степени тестируют данный участок кода. Такой мутант называется выжившим (Survived, Escaped Mutant).
8. Это так называемый вероятностный JIT-подход к тестированию компиляторов, Just In Time (JIT), который основан на создании случайных, но синтаксически и семантически правильных программ на виртуальной машине Java HotSpot ™ (JVM).
9. Тестовая программа сначала выполняется в режиме интерпретатора, а затем в режиме компилятора. Если получающиеся состояния будут отличаться, можно сделать вывод, что обнаружена ошибка.
10. Реализация JITTester поддерживает несколько функций, таких как генерация выражений, приведения типов, анализ потока управления, вызовы функций и полная поддержка синтеза иерархий классов. JITtester продемонстрировал высокую эффективность в обнаружении ошибок в виртуальной машине Java HotSpot ™ (JVM).
11. Проект Mull - это попытка создать универсальный инструмент тестирования мутаций, ориентированный на уже скомпилированные языки. Mull построен поверх структуры компилятора LLVM: он

использует IR или битовый код LLVM для выполнения мутаций и механизм JL LLVM для выполнения оригинальных и мутированных программ.

12. Основное внимание Mull уделяется C и C ++, но благодаря своей природе LLVM его можно применять для других языков, таких как Swift, Rust, Objective-C.
13. Mull - это инструмент командной строки.
14. Он принимает файл конфигурации в качестве входных данных и создает базу данных SQLite с результатами в качестве выходных данных.
15. Файл конфигурации содержит список файлов битовых кодов тестируемой программы, набор операторов мутации, тестовую среду и несколько других вещей.
16. База данных SQLite содержит информацию, собранную во время выполнения, такую как тесты, точки мутации, мутанты (убитые или выжившие) и многое другое.

Ниже приведены шаги, которые Mull выполняет во время сеанса:

- ✓ Шаг 1: Mull загружает LLVM Bitcode в память.
- ✓ Шаг 2: Mull вставляет инструментальный код в каждую функцию. Этот код используется для сбора информации о покрытии кода.
- ✓ Шаг 3: Mull компилирует инструментированный битовый код LLVM в машинный код и подготавливает машинный код для выполнения с помощью механизма JL LLVM.
- ✓ Шаг 4: В коде IR LLVM Mull находит тесты в соответствии с тестовой структурой, указанной в файле конфигурации.
- ✓ Шаг 5: Mull запускает каждый тест с использованием движка LLVM JIT и собирает информацию о покрытии кода.
- ✓ Шаг 6: Mull находит мутации в ИК-коде LLVM на основе информации покрытия кода, собранной для каждого теста. Набор точек мутации создан.
- ✓ Шаг 7: Для каждой точки мутации Mull создает мутанта и запускает каждый тест, который может убить мутанта. Для каждого мутанта в машинный код перекомпилируется только часть битового кода.
- ✓ Шаг 8: Вся информация, собранная во время сеанса, записывается в базу данных SQLite. Это последний шаг. На этом Mull заканчивает свое исполнение.
- ✓

Задание на практическое занятие:

На вычислительной машине с предустановленной ОС Ubuntu установить Mull в соответствии с инструкцией, размещённой на сайте https://mull.readthedocs.io/en/latest/Installation.html , далее выполнить пример фаззинг - тестирования изложенный здесь https://mull.readthedocs.io/en/latest/HelloWorld.html

Далее провести тестирования любой программы на C в соответствии со следующей инструкцией.

Инструкция по установке и использовании MUll Mutation jittester

Установка

Этап 1: установка Clang на Ubuntu

Вводим в терминал следующие команды:

sudo apt update

sudo apt install clang

Этап 2: установка Mull

Вводим в терминал следующие команды:

wget https://api.bintray.com/users/bintray/keys/gpg/public.key

sudo apt-key add public.key

echo "deb https://dl.bintray.com/mull-project/ubuntu-18 stable main" | sudo tee -a /etc/apt/sources.list

sudo apt-get update

sudo apt-get install mull

Выполнение тестирования

Этап 1: написание кода программы (C++/C)

Этап 2: создание объектного файла с помощью команды clang

clang -fembed-bitcode -g [компилируемый файл] -o [Название_объектного_файла]

Этап 3: запускаем тестировщик с помощью команды

mull-cxx -test-framework=CustomTest -ide-reporter-show-killed [Название_объектного_файла]

Контрольные вопросы:

1. Что такое фаззинг ?
2. На каких этапах процесса верификации ПО применяется фаззинг ?
3. Какие задачи можно решать с помощью технологии фаззинга ?
4. В чём отличительная особенность фаззинга ?
5. Какие вы знаете фаззеры ?
6. Для какой ОС разработаны основные фаззеры ?
7. Что такое мутант применительно к фаззингу ?
8. В чём разница между убитым и выжившим мутантом применительно к фаззингу ?
9. Какая метрика используется для оценки результатов фаззинга ?
10. Поясните по схеме основные этапы фаззинга.

Рекомендуемая литература и материалы для изучения и выполнения практических заданий :

1. Бурякова Н. А., Чернов А. В. Классификация частично формализованных и формальных моделей и методов верификации программного обеспечения // Инженерный Вестник Дона. 2014. № 4. 129–134 с.
2. Гленфорд Майерс, Том Баджетт, Кори Сандлер.Искусство тестирования программ, 3-е издание– The Art of Software Testing, 3rd Edition. —М.:«Диалектика», 2012. — 272 с. —ISBN 978–5-8459–1796–6
3. Гурин Р. Е.1,*, Рудаков И. В.1, Ребриков А. В. Методы верификации обеспечения УДК 004.3+519.6 Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. № 10. С. 235–251.
4. Гурин Р. Е. Обзор и анализ инструментов, который осуществляют верификацию бинарного кода программы // Новые информационные технологии в автоматизированных системах: материалы 17-го научно-практического семинара. Вып. 17. М.: ИПМ им. М. В. Келдыша, 2014. 514–518. 421 с.
5. Дж. Уиттакер, Дж. Арбон, Дж. Каролло Как тестируют в Google. — СПб.: Питер, 2014. — 320 с.: ил. ISBN 978-5-496-00893-8
6. Канер Сэм и др. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ./Сэм Канер, Джек Фолк, Енг Кек Нгуен. — К.: Издательство «ДиаСофт», 2001. — 544 с.
7. Карпов Ю.Г. MODEL CHECKING. Верификация параллельных и распределенных
программных систем. СПб.: БХВ-Петербург, 2010. 560 с.
8. Куликов, С. С. Тестирование программного обеспечения. Базовый курс / С. С. Куликов. — Минск: Четыре четверти, 2017. — 312 с.
9. Мандрыкин М. У., Мутилин В. С., Новиков Е. М., Хорошилов А. В. Обзор инструментов статической верификации С программ в применении к драйверам устройств операционной системы Linux // Сборник трудов Института системного программирования РАН. Т. 22. М.: ИСП РАН, 2012. С. 293–294. DOI: 10.15014/ISPRAS-2012–22–17. 345 с.
10. Мишечкин, М. В. Обзор различных средств фаззинга как инструментов динамического анализа программного обеспечения / М. В. Мишечкин. — Текст : непосредственный // Молодой ученый. — 2017. — № 52

(186). — С. 28-31. — URL: https://moluch.ru/archive/186/47575/ (дата обращения: 07.01.2021).

11. Рудаков И. В., Гурин Р. Е., Ребриков А. В. Верификация программного обеспечения: обзор методов и характеристик // Национальная ассоциация ученых (НАУ). Ежемесячный журнал. 2014. № 3, ч. 2. 22–26 с.
12. Элфрид Дастин, Джефф Рэшка, Джон Пол Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация - Лори, 2003. - 312 с
13. Cem Kaner, James Bach, Bret Pettichord. Lessons Learned in Software Testing
    A Context-Driven Approach Published by John Wiley & Sons, Inc., New York, 2002 - 316 с.
14. IEEE 1012-2004 Standard for Software Verification and Validation. IEEE, 2005.