Оглавление

[Введение 2](#_Toc422309552)

[1 Исследовательский раздел 6](#_Toc422309553)

[1.1 Обзор средств автоматизации визуального тестирования 6](#_Toc422309554)

[1.1.2 Решения для автоматизации тестирования Web-приложений на базе Selenium 7](#_Toc422309555)

[1.1.3 Альтернативы Selenium 12](#_Toc422309556)

[1.2 Обзор методов анализа программного кода тестов web-приложения 13](#_Toc422309557)

[1.3 Обзор методик анализа покрытия функционала приложения тестами 20](#_Toc422309558)

[1.4 Анализ существующих решений 24](#_Toc422309559)

[2 СПЕЦИАЛЬНЫЙ РАЗДЕЛ 27](#_Toc422309560)

[2.1 Формализованная модель представления визуальных элементов веб приложения. 27](#_Toc422309561)

[2.2. Формализованное представление тестов. 29](#_Toc422309562)

[2.3 Математическое описание метрик программного кода тестов интерфейса web-приложения 30](#_Toc422309563)

Введение

Еще совсем недавно тестирование программ проводилось вручную либо самими программистами, либо пользователями, что вряд ли можно было назвать системным подходом и к тому же не позволяло оценивать качество кода. Чуть позже тестирование выделилось в отдельную область знаний в составе разработки программного обеспечения, но быстро пришло понимание того, что тестирование вручную неэффективно, поскольку требует больших трудовых ресурсов и много времени. Первые средства автоматизации тестирования практически представляли собой библиотеки, которые можно было использовать для написания тестов, что требовало от тестировщика умения программировать на уровне разработчика. Современные средства автоматизированного тестирования позволяют создавать автоматизированные тесты с минимальным участием человека.

Если провести поверхностную классификацию, то средства автоматизации тестирования можно поделить на две группы: инструменты функционального и инструменты нагрузочного тестирования. К первой группе относятся инструменты, предназначенные для проверки соответствия приложения предъявляемым бизнес-требованиям, а вторую группу образуют инструменты для проверки и оценки производительности приложений.

На рынке средств функционального тестирования сегодня представлены главным образом продукты следующих компаний: HP (QuickTest Professional, WinRunner), IBM (Robot, Functional Tester), Borland (SilkTest) и AutomatedQA (TestComplete), представляющие собой средства разработки приложений. Причем часть из них использует «промышленные» языки программирования (например, QTP используется в качестве языка разработки скриптов VB, а Functional Tester реализован в среде Eclipse и позволяет создавать скрипты на Java), а часть применяет «диалекты» или свои собственные специальные языки (например, Robot использует язык SQABasic, а TestComplete – язык 4Test). Большинство инструментов ориентировано на работу с Web-приложениями либо с обычными приложениями, написанными на .Net или Java. При этом поддержку «старых» платформ, таких как Centura или PowerBuilder, обеспечивают в основном крупнейшие разработчики средств тестирования, HP и IBM.

Разработчики средств автоматизированного функционального тестирования достаточно оперативно реагируют на появление новых механизмов и платформ разработки программного обеспечения. И все же, несмотря на то что на рынке существует огромное разнообразие различных продуктов для автоматизированного тестирования, некоторые компании–разработчики программного обеспечения предпочитают создавать собственные инструменты, приспособленные для тестирования разрабатываемых ими приложений. Причин для этого как минимум две: высокая стоимость средств автоматизированного тестирования и уникальность тестируемого программного обеспечения, которая не позволяет использовать стандартные средства автоматизации тестирования.

Инструменты нагрузочного тестирования являются более сложными – они фактически «перехватывают» трафик между тестируемым приложением и сервером и представляют его в виде, удобном для работы. Практически все производители продуктов для автоматизированного тестирования предлагают средства нагрузочного тестирования, поскольку в современном виде функциональное тестирование в чистом виде мало кого интересует. Лидерами на рынке средств автоматизированного нагрузочного тестирования являются HP с продуктом LoadRunner и IBM с продуктами Robot

и Performance Tester, поддерживающими множество протоколов (включая терминальные протоколы). Большинство средств нагрузочного тестирования работают лишь с Web-приложениями.

Помимо собственно средств тестирования существуют так называемые средства поддержки процесса тестирования, позволяющие вести учет требования и тест-кейсов, проводить анализы покрытия требования тестами, управлять ходом выполнения тестирования, вести учет обнаруженных дефектов и т.п. Лидирует в данной области Web-приложение HP Quality Center – единый инструмент управления процессом тестирования (включая управление требованиями и дефект-менеджмент), интегрируемый со средствами функционального и нагрузочного тестирования HP QuickTest Professional и LoadRunner. С данным инструментом конкурирует продукт Rational Quality Manager (RQM) от IBM, представляющий собой Web-приложение на платформе Jazz.

Исторически сложилось так, что Россия отстает от остального компьютерного мира по применению средств автоматизации тестирования, и для этого, на мой взгляд, есть несколько причин. Во-первых, неправильное отношение к тестированию как таковому – многие руководители до сих пор считают, что разработчик может написать программу, которая не содержит ошибок. Во-вторых, высокая стоимость инструментов автоматизации тестирования. В-третьих, желание сэкономить на квалифицированных кадрах – работа специалиста по средствам автоматизации тестирования обходится дороже, чем труд обычного тестировщика, поэтому многие компании предпочитают пригласить трех студентов, которые будут вручную тестировать программное обеспечение (в большинстве случае к тому же бессистемно), чем нанять одного специалиста-автоматизатора. В-четвертых, ограничение по срокам – автоматизация тестирования требует значительных временных затрат и имеет смысл только в том случае, если проект как минимум среднесрочный (от полугода и более). И наконец, в-пятых, неудачный опыт применения таких средств и ожидание мгновенного эффекта от их внедрения. А ведь результаты внедрения таких продуктов заметны далеко не сразу и затраты на автоматизацию окупаются за длительный период времени, что не позволяет полностью отказаться от тестирования вручную.

Автоматизация не сокращает этап подготовки к тестированию, а, наоборот, увеличивает его, что может напугать неискушенного руководителя проекта, однако, как только будет выпущена первая стабильная версия приложения и станет возможным проведение регрессионного тестирования, преимущества автотестирования станут очевидны. На моем опыте средства автоматизации тестирования позволили сократить сроки проверки версии с пяти рабочих дней до двух – автотесты прогонялись ночью, а на следующий день анализировался лог и выполнялось ручное тестирование функционала, проверка которого по ряду причин не автоматизировалась.

Автоматизация тестирования позволяет если не избежать, то значительно уменьшить синдром «замыленного глаза», когда тестировщик перестает замечать ошибки при выходе новых версий. Благодаря автоматизации можно не просто ускорить процесс тестирования, но и увеличить тестовое покрытие за счет большего количества перебираемых комбинаций входных данных, что в свою очередь позволяет снизить требования к квалификации разработчиков – с большей вероятностью их ошибки будут обнаружены на этапе тестирования. Если раньше для того, чтобы гарантировать, что с вероятностью 99% в программе не будет критических ошибок, мы должны были использовать команду из 10 высококлассных разработчиков, то теперь мы обходимся командой из 10 разработчиков, среди которых только 2-3 высококлассных специалиста.

Продукты автоматизации тестирования могут успешно применяться не только в компаниях-разработчиках, но и в организациях, использующих готовое ПО, – для них особенно актуальны средства автоматизации нагрузочного тестирования, позволяющие делать прогнозы (например, как долго сможет функционировать система на имеющемся оборудовании при запланированном росте бизнеса?), оптимизировать конфигурацию (настройка серверов для повышения производительности), находить ошибки функциональности, связанные с работой в многопользовательском режиме (подобные ошибки трудно обнаружить на этапе функционального тестирования).

1. Исследовательский раздел

# Обзор средств автоматизации визуального тестирования

Для визуального тестирования важна поддержка конкретной среды разработки, возможность построения отчетов о тестировании, автоматическая регистрация обнаруженных дефектов, наличие сценариев восстановления (recovery-сценариев).

Важную роль при выборе инструментов тестирования играет наличие документации и линии технической поддержки – современные инструменты тестирования не менее сложны, чем средства разработки. Следует обратить внимание и на наличие специализированных форумов, посвященных средствам тестирования, – существование форума по конкретному инструменту и, главное, многочисленной группы активных пользователей говорит о широкой распространенности данного инструмента, что в дальнейшем поможет быстрее получить ответ на интересующий вопрос. Кроме того, нужно обращать внимание на возможность интеграции инструментов тестирования с программным обеспечением, которое используется в компании. К примеру, если в компании уже выстроен процесс разработки программного обеспечения и в качестве средств автоматизации используются продукты IBM, то выбирать в качестве инструмента тестирования TestComplete, возможно, не самая удачная идея.

Следует руководствоваться также стоимостью инструментов тестирования – если вы планируете одноразовое тестирование, то покупать дорогостоящие инструменты нецелесообразно. В качестве альтернативы приобретению лицензий на инструментальные средства тестирования является аренда лицензий (покупка временных лицензий), что обычно значительно дешевле.

* + 1. **Решения для автоматизации с помощью скриншотов**

В каких случаях это выгодно?

* Отсутствует доступ к свойствам элементов
* Свойства есть, но они постоянно меняются
* Приложение больше не обновляется
* Недостаток времени и отсутствие опыта у тестировщика

Один из продуктов, реализующий такой подход **- eggPlant:**

Из его достоинств можно отметить:

* Создан как инструмент тестирования
* Сильный модуль распознавания текста
* Высокая стабильность работы
* Хорошая система отчетов

К недостаткам можно отнести:

* Работает только с удаленной машиной
* Платный

Еще один схожий продукт – **Sikuli:**

В отличии от конкурента можно отметить в лучшую сторону:

* Бесплатный инструмент
* Прост в развертывании и использовании
* Не требует дополнительной машины
* Есть API для Java
* Возможность использовать Jython

Недостатки:

* Cлабый модуль распознования текста
* Отсутствие подробного отчета

## 1.1.2 Решения для автоматизации тестирования Web-приложений на базе Selenium

**Selenium *-*** это инструмент для автоматизированного управления браузерами***.*** Наиболее популярной областью применения Selenium является автоматизация тестирования веб-приложений. Однако при помощи Selenium можно автоматизировать любые другие рутинные действия, выполняемые через браузер.

Разработка Selenium поддерживается производителями популярных браузеров. Они адаптируют браузеры для более тесной интеграции с Selenium, а иногда даже реализуют встроенную поддержку Selenium в браузере. Selenium является центральным компонентом целого ряда других инструментов и фреймворков автоматизации.

Selenium поддерживает десктопные и мобильные браузеры. Selenium позволяет разрабатывать сценарии автоматизации практически на любом языке программирования. С помощью Selenium можно организовывать распределённые стенды, состоящие из сотен машин с разными операционными системами и браузерами, и даже выполнять сценарии в облаках.

**Selenium IDE: возможности применения без использования тяжеловесных решений**

Selenium, как инструмент для автоматизированного тестирования был бы менее эффективен и востребован, если бы не имел средств для быстрой записи/воспроизведения тестовых сценариев. Такой инструмент есть, и те кто уже имел дело с Selenium скорее всего уже знакомы с ним – это Selenium IDE Plugin к Firefox, который позволяет**:**

* Записывать тесты непосредственно из Firefox
* Воспроизводить загруженный тест в Firefox
* Воспроизводить загруженный тест в Firefox через SeleniumTestRunner
* Экспортировать записанный тест в один из поддерживаемых языков (java, ruby, php, c#, python…)

Достоинства:

* Прост в использовании, не требует много ресурсов, не требует специальной подготовки сотрудников.
* Позволяет автоматизировать простые тестовые сценарии/операции

Недостатки:

* Не позволяет использовать логические условия, циклы и т.п. что ограничивает его применимость линейными тестами
* Нет возможности запуска случаев, а не отдельных тестов
* Нет возможности параллельного запуска (только в разных экземплярах Firefox)

Конечно, как самостоятельный инструмент автоматизации, Selenium IDE весьма ограничен. Однако, если у вас есть много небольших или средних проектов, например на аутсорсинг, с фиксированным бюджетом, которые не предусматривают комплексную автоматизацию, данная IDE позволит автоматизировать рутинные операции по проверке функциональных и/или приемочных требований с минимальными затратами на внедрение и использование.

Соответственно Selenium IDE может быть использован как простой фреймворк для автоматизации тестирования, в тех случаях, когда нет времени, денег и необходимости все усложнять.

**Cubic Test: Eclipse + GEF + Selenium = визуальное управление тестами**

Современные проприентарные приложения для автоматизации тестирования хороши, на мой взгляд, прежде всего тем, что предоставляют решение по организации и управлению автоматическими тестами. В случае с open source решениями ситуация хуже, так как тот же Selenium предоставляет скорее основу для формирования фреймворка для каждого конкретного случая, а не готовое решение. Cubic test предлагает вариант решения, сравнимый с мощными средами от ведущих производителей.

Его возможности:

* Интегрируется в EclipseIDEкак отдельная Perspective.
* Имеет инструменты Record/Playback.
* Использует визуальное моделирование и управление тестами (на базе Graphical Test Editor, GEF, также интегрируемого в Eclipse).
* Позволяет выносить общие сценарии в подтесты и подключать их по мере необходимости.
* Позволяет объединять тесты в наборы (случаи), также используя визуальное представление.
* Позволяет экспортировать графическое представление тестов в HTML Prototype или Watir (в том числе допускает написание собственных экспортеров).

Достоинства:

* Оригинальная и простая для понимания концепция визуального управления тестами, основанная на распространенных и доступных инструментах с открытым исходным кодом.
* Прост в использовании и не требует специализированных навыков программирования на том или ином языке (java, ruby, c#…).
* Встроенные средства записи и воспроизведения.

Недостатки:

* Отсутствие возможности параллельного воспроизведения тестовых наборов.
* Некорректная работа с кирилицей.

Итого, Cubic test можно рассматривать как решение для проектов средней сложности, где есть возможность целенаправленно заниматься автоматизацией. При этом радует, что инструмент вполне доступен для использования начинающими тестировщиками. Проблема с кирилицей пока решается только ручной правкой тест-файлов (которые хранятся в текстовом формате).

**Selenium grid: распределенная среда для тестирования web приложений**

Один из недостатков Selenium (он же его достоинство) – это запуск экземпляра браузера при выполнении тестов. Достоинства понятны – мы можем тестировать не просто web-приложение, а web-приложение в реальном окружении – в IE, Safari, Firefox на различных платформах. Недостатки тоже на поверхности – нам нужна графическая оболочка (windows, kde или gnome в linux и т.п.) для выполнения тестов и, что весьма немаловажно, само выполнение тестов в условиях, приближенных к боевым занимает продолжительное время. Эту задачу берутся решить разработчики Selenium–grid. Они предоставляют решение, позволяющее с минимальными затратами получить распределенную и легко масштабируемую среду для параллельного выполнения большого числа тестов.

Возможности:

* Самое главное – быстрое и простое распараллеливание выполнения тестов. В основе данной возможности лежит инструмент TestNG (а не jUnit как у «классических» selenium-тестов).
* Возможность построения распределенной и масштабируемой среды для выполнения тестов.

Достоинства:

* Многократное уменьшение времени выполнения при большом количестве тестовых сценариев.
* Возможность использования ранее написанных тестов (на java, ruby python…).
* Простой способ построения распределенной среды для выполнения тестов.

Недостатки:

* Нет средств Record/Playback.
* Требует более высокой квалификации от сотрудников.
* Нет поддержки selence test cases.
* Имеет недоработки

**Selenium on Rails**.

Простой способ интеграции Selenium с Ruby on Rail приложениями:

* Selenium Core не «замусоривают» /public.
* Не нужно создавать suite-файлы, они генерируются на лету — один suite на директорию в /test/selenium.
* Partial test cases. Позволяет выносить часто используемые последовательности действий в отдельные «кусочные» кейсы и подключать их по мере надобности.
* Загрузка фикстур и имитация сессий.
* Возможность использования нескольких более удобных форматов для TestCases, помимо HTML

Эквивалентный список достоинств можно отметить и для других языков: C#, Java, Pyton, Php

**Bromine**.

Интегрированная тестовая среда на базе Selenium, предоставляет возможности:

* Многофункциональный QA инструмент
* Позволяет создавать «проекты»
* Привязывать требования к проектам
* Привязывать тесты к требованиям
* Предоставляет простой способ управления и запуска тестов
* Позволяет анализировать результаты запуска тестов
* Позволяет заводить дефекты
* Также имеется облегченная light версия только для запуска тестов и анализа результатов

Из недостатков в первую очередь стоит отметить “сырость” продукта

## 1.1.3 Альтернативы Selenium

**WebTest Canoo**

Что касается Canoo, то он с браузером не работает, а работает с библиотекой htmlUnit, которая в свою очередь эмулирует браузер (то есть, эмулирует его DOM и умеет работать с javaScript), и работает с вёб-сервером напрямую. Canoo использует свой язык, основанный на XML. Flow control (if - else), loops (for, while, do, loop...) не поддерживаются.

Недостатки:

* Писать тесты приходится "на ощупь" - с браузером html unit не работает
* htmlUnit несмотря на всю свою мощь, браузером не является, поэтому тот факт, что что-то работает в htmlUnit ничего не говорит о том, будет ли то же самое работать в конкретном браузере и наоборот.
* Отсутствие гибкости

[**Phantomjs**](http://phantomjs.org/)

Проект на основе движка рендеринга веб-страниц WebKit (QT-порт WebKit), который не требует графического сервера для работы. Phantomjs управляется с помощью скриптов на JavaScript/CoffeeScript, позволяя загружать страницы, делать скриншоты, конвертировать и сохранять страницы в pdf-формате. Скрипт может выполнять код в контексте открытой страницы браузера, позволяя взаимодействовать с DOM-элементами страницы.

Начиная с версии 1.8 Phantomjs включает компонент GhostDriver, который реализует WebDriver API. Таким образом, появляется возможность работать с Phantomjs непосредственно с помощью существующих средств работы с Selenium WebDriver.

[**Wight**](https://metacpan.org/pod/Wight)

Perl-модульдля тестирования веб-приложений, который был создан специально для работы с Phantomjs, используя существующий там интерфейс WebDriver поверх WebSocket.

[**WWW::WebKit**](https://metacpan.org/pod/WWW::WebKit)

Еще один Perl-модуль, построенный на основе Gtk3::WebKit — обвязке к порту WebKitGTK+, используемого в проекте Gnome. Применение встроенного механизма отрисовки веб-страниц дает несколько преимуществ:

* Меньший объем используемой памяти
* Нет необходимости в дополнительных программных средствах и заботе об их управлении.
* Упрощается параллельный запуск тестов

Недостатки:

* Требуется виртуальный графический сервер Xvfb.
* Отображение веб-страниц проверяется только на одном из существующих движков — WebKit.

# 1.2 Обзор методов анализа программного кода тестов web-приложения

Отрасль информационных технологий является одной из наиболее интенсивно развивающихся областей в современной жизни. Быт людей в настоящее время устроен таким образом, что даже далекий от информационных технологий человек ежедневно использует их для достижения целей. Более того, компьютерные программы на сегодняшний день используются даже в тех областях, где цена ошибки очень высока. Безусловно, использование автоматизированных систем является очень удобной заменой повторяющихся операций, выполняемых человеком. Это позволяет увеличить эффективность работы и избежать случайных ошибок в процессе работы. Однако, заменяя ручной труд человека программным кодом, появляется риск возникновения ошибки в программе. Дефекты в программном коде могу иметь разную природу и последствия. Одни ошибки возникают по причине низкого качества программного кода и могут доставить пользователю определенные неудобства, другие - иметь случайный характер. Иногда ценой ошибки может быть человеческая жизнь. Как правило, подобный род ошибок фиксируется на этапе разработки и эксплуатации и зачастую постепенно исключаются из рабочего кода. Тем не менее, существуют такие дефекты системы, которые проявляются не сразу, а в процессе поддержки и развития программной системы. Постепенно ошибки, которые казались незначительными, могут становиться серьезными дефектами, которые мешают дальнейшему развитию продукта. Обычно такие дефекты связаны с архитектурными изъянами и некорректным построением программного кода. Ошибки подобного рода несут в себе серьезную проблему, которая проявляется только после длительного срока работы над программной системой. Из-за этой проблемы могут прекращаться работы над, казалось бы, успешными проектами. В лучшем случае программный продукт может быть разработан заново, без использования кода, содержащего подобного рода дефекты.

Мировое сообщество серьезно озабочено возможностью появления различного рода дефектов в программном коде. Подтверждением тому может быть факт создания международных стандартов на разработку программных систем (ISO/IEC, СММ/СММI, др.). Более того, за последние не­ сколько десятилетий было создано большое количество различных методологий и подходов к разработке различных типов программных систем. Каждая такая методология стремится к получению работоспособного продукта после завершения его разработки. Некоторые из методологий делают акцент на последовательность процессов разработки, некоторые - делают процесс максимально простым и гибким. Тем не менее, каждая из таких методологий стремится к созданию качественного продукта.

Однако, как известно, мы не можем контролировать то, что не можем измерить. Это же утверждение можно отнести и качеству кода. Трудно принимать какие-либо проектные решения при отсутствии количественных измерений характеристик. Поэтому проблема измерения качества кода на сегодняшний день является актуальной.

**Качество программного обеспечения и качество кода**

Понятие «качество» является сложным и многогранным. Обычно под качеством понимается соответствие объекта каким-то предъявляемым требованиям. В программной инженерии этот термин можно трактовать по-разному. Можно трактовать качество программного продукта как соответствие его характеристикам, определенных в требованиях к продукту. Это означает, что конечный продукт решает поставленные перед ним задачи. Другая интерпретация может состоять в том, что качество продукта - это отсутствие дефектов и ошибок, связанных с аварийной работой продукта. Также можно трактовать понятие качества как способность к легкому изменению программного кода и возможность легкого добавления дополнительных функциональных возможностей к продукту. Термин «качество» может использоваться на различных уровнях и использоваться в разных контекстах. Поэтому необходимо различать два главных понятия - «качество программного продукта» и «качество программного кода».

Качество программного продукта определяется тем, насколько он решает задачи конечных пользователей. Серьезное влияние на качество продукта оказывает корректность поставленных требований к продукту на этапе анализа. Если аналитики перед началом работ не смогли корректно определить задачи и проблемы пользователей, то такой программный продукт вряд ли сможет по­ мочь пользователям.

Качество программного кода подразумевает грамотно выдержанный архитектурный стиль программного кода, четкое разделение кода на функциональные блоки и строгая структуризация и т. д. Таким образом, качественный программный код - это код, который легко поддерживать, вносить в него дополнительную функциональность, изменять существующие алгоритмы и т. д. Для получения качественного программного кода на сегодняшний день разработчики пользуются различными методологиями и практиками, применяют шаблоны проектирования, для типовых задач используют готовые библиотеки и алгоритмы.

Таким образом, видна существенная разница между понятиями «качество продукта» и «качество кода». Более того, продукт может иметь качественный код и при этом не решать проблем конечных пользователь. Такой программный продукт считается некачественным. И наоборот, качественный продукт, который успешно справляется с задачами пользователей, может состоять из некачественного кода. Естественно, цель каждой разрабатываемой программной системы - решить задачи пользователя. Зачем же тогда бороться за качество кода? Ответ простой - практически каждая система имеет тенденции к дальнейшему развитию и модификации. При этом важными показателями являются количество дефектов в программе и стоимость модификации кода. Если при добавлении новой функциональности возникает неприемлемое число ошибок, то продукт уже не может удовлетворять потребности пользователя. Аналогично, если стоимость добавления новой функциональности в программный продукт слишком высока, то это также негативно сказывается на пользователе. Из этого следует вывод, что качество программного кода - это нефункциональный, но очень важный показатель, который впоследствии оказывает опосредованное влияние на качество конечного программного продукта.

Поскольку качество кода оказывает существенное влияние на качество самого программного продукта, то в данном разделе будут рассматриваться проблемы анализа качества программного кода.

Исходя из приведённых ранее доводов, встает задача контроля качества программного кода и удержания его на необходимом уровне. Однако для того, чтобы контролировать качественные показатели, необходимо иметь численные характеристики этих показателей. Другими словами, для контроля качества программного кода необходимо уметь измерять это качество.

Для получения формальных оценок качества программного обеспечения существуют метрики программного кода. Суть этого механизма заключается в том, что на основе анализа исходного кода программной системы можно получить различные числовые характеристики. Обычно вычисление таких показателей строится на основе анализа графа программного потока или структуры программного кода. На сегодняшний день существует большое количество метрик кода, анализирующих различные аспекты программного кода. К наиболее типичным метрикам можно отнести показатели количества строк кода в системе, сложности графа программы, глубины наследования классов и т. д. Преимущество метрик программного кода состоит в том, что в процессе их вычисления не участвует человеческий фактор, а все вычисления производит компьютер. Это гарантирует факт точности и повторяемости таких измерений для каждой из метрик. Более того, метрики можно вычислять автоматизировано и строить на их основе различные аналитические отчеты. Однако, на сегодняшний день метрики сильно дискредитировали себя в глазах большого количества разработчиков по всему миру. Это произошло из-за того, что каждый разработчик использует сильно огра­ниченный, не связанный набор метрик, и таким образом, получает однобокое представление о системе.

Для построения объектного представления о программном коде сложно использовать связанный набор метрик, которые будут отображать целостное представление о качестве программного кода. Как правило, качественный код представляется как программный код, не наделенный избыточной сложностью и связностью частей системы, хорошо структурированный и имеющий адекватные пропорции для объема (функциональные блоки не должны быть слишком большими или маленькими, что отражает качество декомпозиции в системе). Для того чтобы проанализировать качество программного кода в этой плоскости, предлагается ис­пользовать связанную систему метрик, характеризующую четыре этих аспекта качества программного кода. Таким образом, требуется измерить:

* Сложность программного кода;
* Связность программного кода;
* структурированность программного кода;
* объёмные характеристики программного кода.

Для получения объективной информации в каждой группе необходимо использовать набор метрик. После вычисления каждой метрики в группе необходимо объединить эти показатели и получить единый результат для каждой группы. Для этого могут быть использованы различные математические методы и подходы. Одним из наиболее эффективных подходов является комплексная мера П. Кокола. Эта мера берет за основу одну метрику, а также учитывает другие метрики, которые должны оказывать влияние на конечный результат.

На сегодняшний день проблема контроля и качества кода является актуальной для большого числа разработчиков. При решении этой задачи требуется наличие инструмента, который позволяет это качество оценить. Для этого требуется наличие математической модели и соответствующих алгоритмов. При этом на сегодняшний день уже существует несколько сотен различных метрик кода, которые отражают те или иные аспекты программного кода. Однако использование необработанных значений этих метрик не всегда может быть эффективным. Поэтому требуется наличие системы метрик, которая объективно показывала различные свойства программного кода.

**Каким должно быть качество тестов?**

Одним из главных аргументов против автоматизированного тестирования является увеличение объема кода, который придется поддерживать. И это действительно так, ведь в зависимости от задачи, юнит-тесты вполне могут увеличить суммарный размер кодовой базы на 50 и более процентов. Такой объем кода сказывается на производительности команды, особенно учитывая более пренебрежительное отношение к качеству кода тестов.

Как и в вопросах сложности, у хрупкости тестов есть неотъемлемая (essential) и привнесенная или случайная (accidental) составляющие. С одной стороны, тесты должны и будут изменяться при изменении требований; поскольку тест говорит о том, "что" должен делать тестируемый код, то при изменении предполагаемого поведения тест будет изменен (это часть неотъемлемой хрупкости тестов).

На практике же тесты ломаются гораздо чаще. Хотя считается, что тесты должны быть страховочным тросом вовремя рефакторинга, на практике практически любой рефакторинг приводит к изменению огромного количества тестов. Изменение реализации тоже часто приводит к поломке тестов, хотя по идее этого быть не должно. Добавление функционала (типа добавление еще одной зависимости) приводит к веерным обновлениям большого количества тестов.

Хрупкость тестов определяется не только тем, как написан сам тест; она также связана с дизайном системы. Если объект сложно отвязать от его окружения, поскольку зависимостей слишком много или они скрыты, тест будет падать при изменении отдаленных частей системы. При этом становится сложным оценить «эффект домино» при изменении кода. Поэтому хрупкость тестов можно использовать в качестве ценной обратной связи о качестве дизайна системы.

Причин у такого положения как минимум две. Во-первых, это невысокое качество тестируемого кода (как сказано в цитате выше), а во-вторых, отсутствие отношение к тестам, как к конечному коду. Первая проблема более сложная, и выходит за рамки данного обсуждения (хотя неоднократно обсуждается в книге), поэтому давайте остановимся на второй составляющей.

Для повышения качества тестов стоит проверять "что", а не "как". Юнит-тесты не должны тестировать if statement, они должны тестировать соответствующий аспект класса. [Не нужно тестировать закрытые методы](http://sergeyteplyakov.blogspot.co.uk/2013/05/blog-post.html), нужно относится к тестируемому коду, как к черному (максимум полупрозрачному) ящику. Чем абстрактнее тест, тем он более устойчивый, поскольку не подрывает инкапсуляции тестируемого кода.

Проводить рефакторинг кода тестов. Стоит выделять базовые классы для тестирования схожего кода, выделять утилитные классы и т.п. В общем, следует использовать те же подходы, что и для поддержания качества конечного кода и изменять его по мере получения новых знаний о тестируемом коде.

Читабельность тестов и сообщений об ошибках. Как и любой хороший метод, тест должен оперировать на одном уровне абстракции и быть максимально декларативным. Вместо вызова нескольких методов для инициализации тестируемого кода можно сделать фабричный метод, а вместо пяти утверждений можно вызвать метод VerifyThatClassContainsRequiredData(), особенно если такая проверка нужна в нескольких местах.

В книге "Growing Object-Oriented Software Guided by Tests" этой теме уделяется очень серьезное внимание. Даже конечный код стоит писать так, чтобы он читался как книга, так что уже говорить за тесты, которые вполне могут выступать в форме спецификации.

Анализируя применимость всего вышесказанного к тестированию визуальных компонентов web­ – приложения таких как кнопки, поля ввода, картинки и т. д. стоит ввести дополнительные метрики кода автоматических тестов, которые будут отображать:

* количество проверок в каждом тесте
* количество строк в каждом тесте
* количество возможных ветвлений в тесте
* тестовой покрытия на основе визуальных элементах
* сами элементы, на которые отсутствуют тесты

1.3 Обзор методик анализа покрытия функционала приложения тестами

**Тестовое Покрытие (Test Coverage)**

Тестовое Покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Если рассматривать тестирование как "проверку соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов", то именно этот конечный набор тестов и будет определять тестовое покрытие:

Чем выше требуемый уровень тестового покрытия, тем больше тестов будет выбрано, для проверки тестируемых требований или исполняемого кода.

Сложность современного программного обеспечения и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего высокий уровень покрытия можно использовать специальные инструменты либо техники тест дизайна.

**Существуют следящие подходы к оценке и измерению тестового покрытия:**

Покрытие требований (Requirements Coverage) - оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).

Покрытие кода (Code Coverage) - оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения.

Тестовое покрытие на базе анализа потока управления - оценка покрытия, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

**Различия:**

Метод покрытия требований сосредоточен на проверке соответствия набора проводимых тестов требованиям к продукту, в то время как анализ покрытия кода - на полноте проверки тестами, разработанной части продукта (исходного кода), а анализ потока управления - на прохождении путей в графе или модели выполнения тестируемых функций (Control Flow Graph).

**Ограничения:**

Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом.

Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

**Покрытие требований (Requirements Coverage)**

Расчет тестового покрытия относительно требований проводится по формуле:

Tcov = (Lcov/Ltotal) \* 100%

где:

Tcov - тестовое покрытие

Lcov - количество требований, проверяемых тест кейсами

Ltotal - общее количество требований

Для измерения покрытия требований, необходимо проанализировать требования к продукту и разбить их на пункты. Опционально каждый пункт связывается с тест кейсами, проверяющими его. Совокупность этих связей - и является матрицей трассировки. Проследив связи, можно понять какие именно требования проверяет тестовый случай.

Тесты, не связанные с требованиями, не имеют смысла. Требования, не связанные с тестами - это "белые пятна", т.е. выполнив все созданные тест кейсы, нельзя дать ответ реализовано данное требование в продукте или нет.

Для оптимизации тестового покрытия при тестировании на основании требований, наилучшим способом будет использование стандартных техник тест дизайна. Пример разработки тестовых случаев по имеющимся требованиям рассмотрен в разделе: "[Практическое применение техник тест дизайна при разработке тест кейсов](http://www.protesting.ru/testing/testdesign_practice.html)"

Покрытие кода (Code Coverage)

Расчет тестового покрытия относительно исполняемого кода программного обеспечения проводится по формуле:

Tcov = (Ltc/Lcode) \* 100%

где:

Tcov - тестовое покрытие

Ltc - кол-ва строк кода, покрытых тестами

Lcode - общее кол-во строк кода.

В настоящее время существует инструментарий (например: [Clover](http://www.atlassian.com/software/clover/)), позволяющий проанализировать в какие строки были вхождения во время проведения тестирования, благодаря чему можно значительно увеличить покрытие, добавив новые тесты для конкретных случаев, а также избавиться от дублирующих тестов. Проведение такого анализа кода и последующая оптимизация покрытия достаточно легко реализуются в рамках тестирования белого ящика (white-box testing) при модульном, интеграционном и системном тестировании; при тестировании же черного ящика (black-box testing) задача становится довольно дорогостоящей, так как требует много времени и ресурсов на установку, конфигурацию и анализ результатов работы, как со стороны тестировщиков, так и разработчиков.

**Тестовое покрытие на базе анализа потока управления**

Тестирование потоков управления (Control Flow Testing) - это одна из техник тестирования белого ящика, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей. [1]

Фундаментом для тестирования потоков управления является построение графов потоков управления (Control Flow Graph), основными блоками которых являются:

* блок процесса - одна точка входа и одна точка выхода
* точка альтернативы - одна точка входа, две и более точки выхода
* точка соединения - две и более точек входа, одна точка выхода

Для тестирования потоков управления определены разные уровни тестового покрытия:

Таблица 1. Уровни тестового покрытия

|  |  |  |
| --- | --- | --- |
| **Уровень** | **Название** | **Краткое описание** |
| Уровень 0 | -- | “Тестируй все что протестируешь, пользователи протестируют остальное” |
| Уровень 1 | Покрытие операторов | Каждый оператор должен быть выполнен как минимум один раз. |
| Уровень 2 | Покрытие альтернатив / Покрытие ветвей | Каждый узел с ветвлением (альтернатива) выполнен как минимум один раз. |
| Уровень 3 | Покрытие условий | Каждое условие, имеющее TRUE и FALSE на выходе, выполнено как минимум один раз. |
| Уровень 4 | Покрытие условий альтернатив | Тестовые случаи создаются для каждого условия и альтернативы |
| Уровень 5 | Покрытие множественных условий | Достигается покрытие альтернатив, условий и условий альтернатив (Уровни 2, 3 и 4) |
| Уровень 6 | “Покрытие бесконечного числа путей” | Если, в случае зацикливания, количество путей становится бесконечным, допускается существенное их сокращение, ограничивая количество циклов выполнения, для уменьшения числа тестовых случаев. |
| Уровень 7 | Покрытие путей | Все пути должны быть проверены |

Основываясь на данных этой таблицы, вы сможете спланировать необходимый уровень тестового покрытия, а также оценить уже имеющийся.

# 1.4 Анализ существующих решений

**Автоматизированные инструменты для сбора метрик программного кода.**

В настоящее время основными инструментами для сбора метрик кода и анализа тестового покрытия являются интегрированные среды разработки или специализированные текстовые редакторы, которые осуществляют поддержку наиболее популярных языков программирования и метрик.

Самые популярные из них:

* Visual Studio
* Eclipse
* QT
* PyCharm
* SublimeText

Так же были проанализированы большинство из сторонних инструментов.

Сравнительный анализ представлен в таблице 2:

Таблица 2. Сравнительный анализ code coverage инструментов

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Инструмент | Поддерживаемые языки | Поддерживаемые уровни покрытия | | | Поддержка графического интрефейса | Отчеты |
| Оператор | Метод /функция | Класс |
| JavaCodeCoverage | Java | × | × |  | × | × |
| JFeature | Java |  | × |  | × | × |
| JCover | Java | × | × | × | × | × |
| Cobertura | Java | × |  |  | × | × |
| Emma | Java |  | × | × | × | × |
| Clover | Java | × | × | × | × | × |
| Quilt | Java | × |  |  | No | Нет |
| Code Cover | Java | × |  |  | × | × |
| Jester | Java |  |  | × | × |  |
| GroboCodeCoverage | Java |  |  | × | Нет | Прост. |
| Hansel | Java | × |  |  | Нет | Нет |
| Gretel | Java |  | × |  | Нет | Таблич. |
| BullseyeCoverage | C++ | × | × |  | × | × |
| NCover | C# |  |  | × | × | × |
| Testwell CTC++ | C++ | × |  |  |  | × |
| Coverage.py | Python |  | × |  | × | × |
| eXVantage | C++ | × |  |  | × | × |
| OCCF | C++ | × |  |  | Нет | Нет |
| JAZZ | Java | × |  |  | Нет | Нет |

Подведя итоги, можно смело сказать, что ни одно решение на данный момент не осуществляет сбор специализированных метрик для кода автоматических тестов и осуществляют анализ только модульного тестового покрытия

1. СПЕЦИАЛЬНЫЙ РАЗДЕЛ

# 2.1 Формализованная модель представления визуальных элементов веб приложения.

**XML Schema** — язык описания структуры [XML](https://ru.wikipedia.org/wiki/XML)-документа. Спецификация XML Schema является рекомендацией [W3C](https://ru.wikipedia.org/wiki/W3C).

Как большинство языков описания XML, XML Schema была задумана для определения правил, которым должен подчиняться документ. Но, в отличие от других языков, XML Schema была разработана так, чтобы её можно было использовать в создании программного обеспечения для обработки документов XML.

После проверки документа на соответствие XML Schema читающая программа может создать модель данных документа, которая включает:

* словарь (названия элементов и атрибутов);
* модель содержания (отношения между элементами и атрибутами и их структура);
* типы данных.

Каждый элемент в этой модели ассоциируется с определённым типом данных, позволяя строить в памяти объект, соответствующий структуре XML-документа. Языкам [объектно-ориентированного программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) гораздо легче иметь дело с таким объектом, чем с текстовым файлом.

Другим удобством XML Schema является то, что один словарь может ссылаться на другой, и, таким образом, разработчик может использовать уже существующие словари и легче устанавливать и распространять стандарты XML структуры для определённых задач (например, словарь протокола [SOAP](https://ru.wikipedia.org/wiki/SOAP)).

Далее будет приведена часть спецификации для визуальных элементов веб приложения:

<xs:simpleType name="InputType">

<xs:restriction base="xs:token">

<xs:enumeration value="text"/>

<xs:enumeration value="password"/>

<xs:enumeration value="checkbox"/>

<xs:enumeration value="radio"/>

<xs:enumeration value="submit"/>

<xs:enumeration value="reset"/>

<xs:enumeration value="file"/>

<xs:enumeration value="hidden"/>

<xs:enumeration value="image"/>

<xs:enumeration value="button"/>

</xs:restriction>

</xs:simpleType>

<xs:element name="input">

<xs:annotation>

<xs:documentation>

form control

</xs:documentation>

</xs:annotation>

<xs:complexType>

<xs:attributeGroup ref="attrs"/>

<xs:attributeGroup ref="focus"/>

<xs:attribute name="type" default="text" type="InputType"/>

<xs:attribute name="name">

<xs:annotation>

<xs:documentation>

the name attribute is required for all but submit &amp; reset

</xs:documentation>

</xs:annotation>

</xs:attribute>

<xs:attribute name="value"/>

<xs:attribute name="checked">

<xs:simpleType>

<xs:restriction base="xs:token">

<xs:enumeration value="checked"/>

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<xs:attribute name="disabled">

<xs:simpleType>

<xs:restriction base="xs:token">

<xs:enumeration value="disabled"/>

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<xs:attribute name="readonly">

<xs:simpleType>

<xs:restriction base="xs:token">

<xs:enumeration value="readonly"/>

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<xs:attribute name="size"/>

<xs:attribute name="maxlength" type="Number"/>

<xs:attribute name="src" type="URI"/>

<xs:attribute name="alt"/>

<xs:attribute name="usemap" type="URI"/>

<xs:attribute name="onselect" type="Script"/>

<xs:attribute name="onchange" type="Script"/>

<xs:attribute name="accept" type="ContentTypes"/>

</xs:complexType>

</xs:element>

# 2.2. Формализованное представление тестов.

Описание инструмента для автоматизированного тестирования интерфейса веб приложения можно описать в виде конечного автомата Миля – , где S – множество внутренних состояний (внутренний алфавит);  – множество входных сигналов (входной алфавит);  – множество выходных сигналов (выходной алфавит);  – функция переходов;  – функция выходов; S0 – начальное состояние.

Описание приведено в виде диаграммы состояний, изображенной на Рисунке 5, и в дополнение приведена таблица состояний при поиске web-клиента, поясняющая диаграмму состояний.

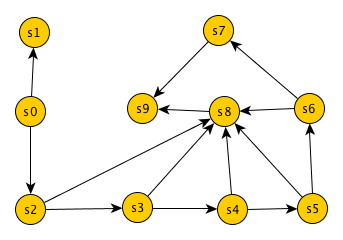


Рис1. Диаграмма состояний инструмента selenium webdriver

|  |  |  |  |
| --- | --- | --- | --- |
|  | Di | Pi | Переход Si |
| S0 | Создается webdriver, открывает ся Chrome | Chrome установлен | S2 |
| Chrome не установлен | S1 |
| S2 | Открывается тестируемое приложение | Соединение установлено | S3 |
| Соединение не установлено | S8 |
| S3 | Нахождение элемента на странице | Элемент найден | S4 |
| Элемент не найден | S8 |
| S4 | Клик по элементу | Успешно | S5 |
| Ошибка | S8 |
| S5 | Совокупность действий над элементом | Успешно | S6 |
| Ошибка | S8 |
| S6 | Проверка в тесте (assert) | Проверка прошла | S7 |
| Проверка не прошла | S8 |
| S7 | Отчет об успешности теста |  | S9 |
| S8 | Ошибка о провале теста |  | S9 |
| S9 | удаляется webdriver, закрывается Chrome |  |  |

2.3 Математическое описание метрик программного кода тестов интерфейса web-приложения  
**Покрытие визуальных компонентов web-приложения**

Расчет тестового покрытия относительно визуальных компонентов проводится по формуле:

Tcov = (Lcov/Ltotal) \* 100%  
где:  
Tcov - тестовое покрытие  
Lcov - количество компонентов, проверяемых тест кейсами  
Ltotal - общее количество визуальных компонентов в web-приложении

Для измерения покрытия, необходимо проанализировать с учетом классов эквивалентности все элементы найденные в web-приложении и тестах.

Элементы вне тестового покрытия расчитываются как разность двух множеств:

C = A / B , где:

С – множество элементов вне тестового покрытия

A – множество элементов, найденных на странице

В – множество проверяемых элементов, найденных в тестах

Так же в качестве метрик расчитываются такие значения, как:

Pe – количество элементов найденных в web-приложении

Te – количество элементов найденных в тестах

T – общее количество тестов

Spt - расчитывается как количество строк для каждого теста (важно, что бы тесты были максимально простыми)

Apt - расчитывается как количество проверок для каждого теста(большое количество проверок усложняет тесты и затрудняет поиск ошибки, обнаруженной тестом)

Ept – количество элементов, проверяемых в каждом тесте (участие сразу нескольких элементов сильно снижается эффективность теста)

Bpt – количество всех возможных вариантов исполнения теста (эффективный тест должен иметь строго один сценарий исполнения)

Для удобства восприятия были также введенны следующие метрики:

* AvgSpt = , где AvgSpt – среднее количество строк в каждом тесте, а - общая сумма строк для всех тестов
* AvgApt = , где AvgApt – среднее проверок строк в каждом тесте, а - сумма всех проверок всех тестов
* AvgEpt = , где AvgEpt – среднее количество элементов в каждом тесте, а - общая сумма всех элементов, затрагиваемых в тестах
* AvgBpt = , где AvgBpt – среднее количество ветвлений в каждом тесте, а - общая сумма всех ветвлений в тестах

Для максимальной эффективности тестов все выше перечисленные отношениядолжны стремиться к единице.

# 2.4 Формализованное описание процедуры анализа покрытия тестами визуальных компонентов веб приложения

Алгоритм нахожденя визуаэльных элементов на странице web-приложения можно представить в виде автомата на рисунке 2. Стоит учесть, прежде чем подавать страницу на этот автомат, она должна пройти XSD валидацию, по принципу описанному выше.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Di | Pi | Переход Si |
| S0 | На вход подается строка | Найден href | S1 |
| Найден input | S2 |
| Ничего не найдено | S0 |
| S1 | Идет поиск идентификатора | Найден id | T0 |
| Найден class | T1 |
| Ничего не найдено | S2 |
| S2 | Идет поиск идентификатора | Найден id | T2 |
| Найден class | T3 |
| Ничего не найдено | S2 |
| T0 | Запись элемента |  | S0 |
| T1 | Запись элемента |  | S0 |
| T2 | Запись элемента |  | S0 |
| T3 | Запись элемента |  | S0 |

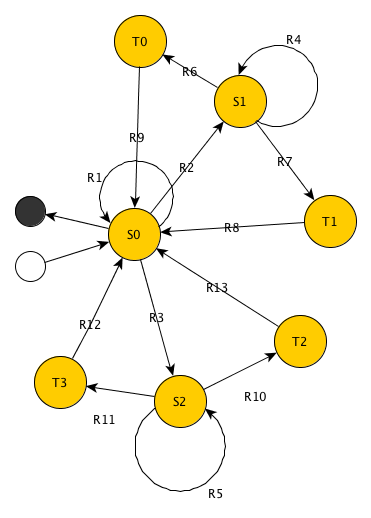


Рисунок 2. Диаграма состояний алгоритма поиска элементов на странице

Алгоритм поиска визуальных элементов в тестах так же можно представить в виде автомата на рисунке 3. Перед подачей тестов на вход, они предварительно проходят валидацию компилятором языка.

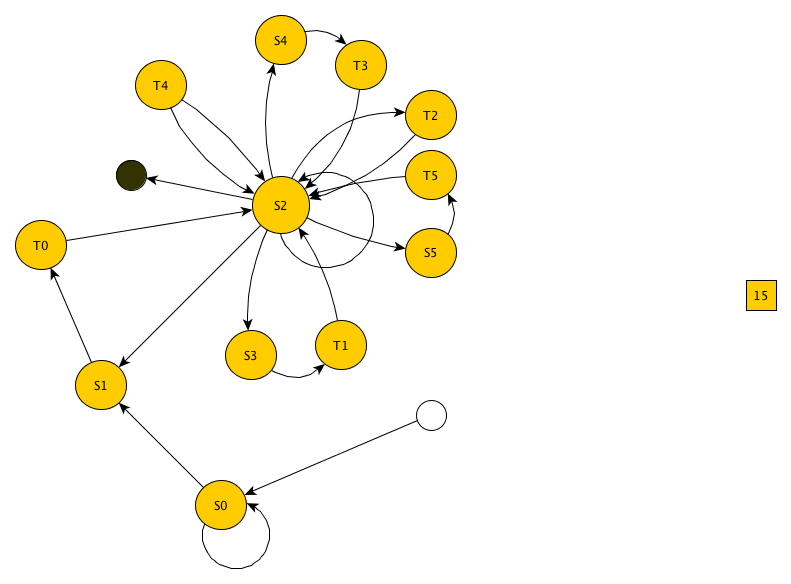


Рисунок 3. Диаграма состояний алгоритма поиска элементов в тестах

|  |  |  |  |
| --- | --- | --- | --- |
|  | Di | Pi | Переход Si |
| S0 | Подается строка | Найден тест | S1 |
| Тест не найден | S0 |
| S1 | Поиск названия теста в строке |  | T0 |
| S2 | Состояние чтения строк с «открытым» тестом | Найдено подключение | S4 |
| Найден локатор элемента | S3 |
| Найдена проверка | T4 |
| Найдена непустая строка | T2 |
| Найдено начало ветвления | S5 |
| Найден тест | S1 |
| S3 | Поиск уникального идентификатора |  | T1 |
| S4 | Поиск адреса |  | T3 |
| S5 | Поиск оператора |  | T5 |
| T0 | Состояние «открытия» теста |  | S2 |
| T1 | Состояние записи элемента |  | S2 |
| T2 | Инкрементация счетчика строк |  | S2 |
| T3 | Запись адреса |  | S2 |
| T4 | Инкрементация счетчика проверок |  | S2 |
| T5 | Запись ветвления. |  | S2 |

Процедуру анализа тестового покрытия визуальных элементов и вычисления метрик программного кода тестов можно разбить на 5 основных блоков (рисунок 4):

* Ввод данных, где пользователь должен ввести адрес web-приложения и указать файл с программным кодом тестов.
* Анализ веб-приложения (основной алгоритм представлен в виде автомата на рисунке 2). В данном блоке происходит синтаксический анализ веб приложения, с целью составления массива визуаэльных элементов
* Анализ кода тестов (основной алгоритм представлен в виде автомата на рисунке 3). В данном блоке происходит синтаксический анализ программного кода тестов и запись необходимой информации в массив, для последуюещго сравнения и вычисления метрик
* С результатов. В это блоке происходит анализ тестового покрытия и вычисления метрик кода тестов.
* Отчет. Данный блок осуществляет вывод всей информации о тестовом покрытии и метриках пользователю

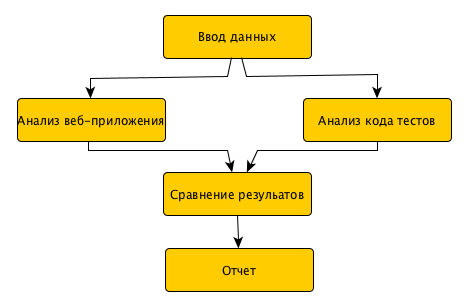


Рисунок 4. Схема процедуры анализа

1. Технологический раздел

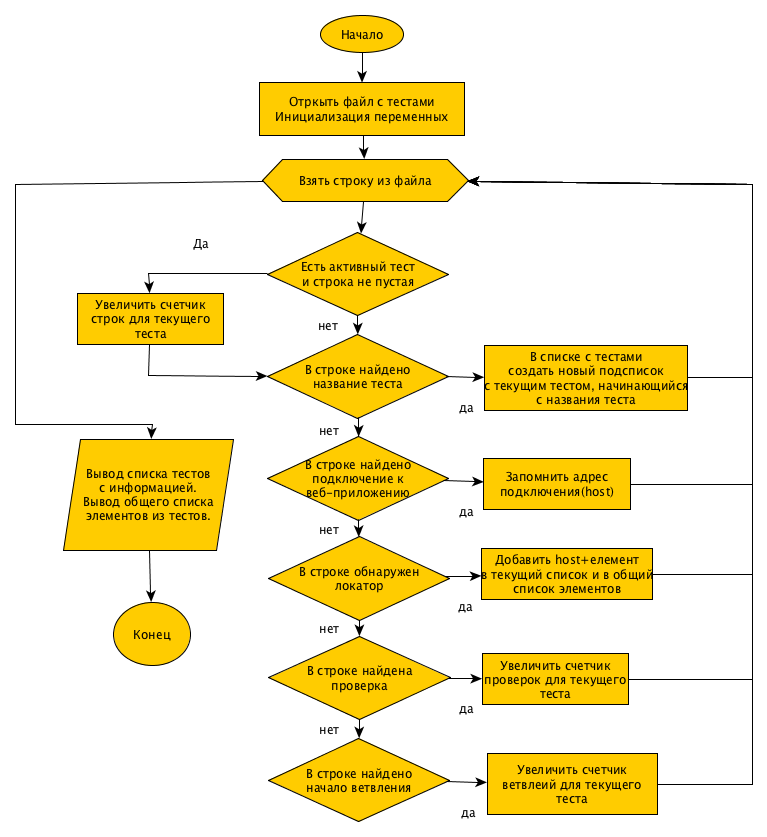
# 3.1 Функциональная схема программного комплекса(модули и движение данных между ними)

# 3.2 Алгоритм расчета метрик программного кода тестов визуального интерфейса

Алгоритм анализа и расчета метрик программного кода тестов веб-приложеня наглядно демонстрируется на следующей блок-схеме (рисунок 5).

Производится синтаксический анализ файла с тестами и формируется выходной удобночитаемый список, содержащий проанализированную информацию о каждом тесте, а именно:

* имена самих тестов;
* элементы, которые проверяют эти тесты;
* количество строк, проверок и ветвлений в каждом тесте;

Рисунок 5. Блок-схема алгоритма расчета метрик тестов

# 3.3 Алгоритм расчета покрытия тестами визуальных компонентов веб приложения

Алгоритм расчет покрытия состоит из нескольких частей:

* поиск элементов, содержащихся в веб-приложении. Блок-схема алгоритма изображена на рисунке 6.
* поиск элементов, проверяемых тестами этого веб-приложения. Блок-схема алгоритма изображена на рисунке 5.
* нахождение разницы между двумя списками, полученных, с помощью предыдущих алгоритмов.

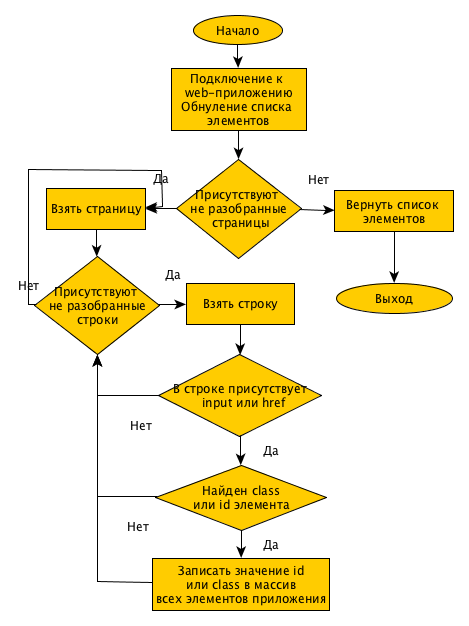


Рисунок 6. Блок-схема алгоритма анализа веб-приложения

# 3.4 Тестирование программного обеспечения на модельных примерах

# 3.5 Нагрузочное тестирование и анализ результатов

Для проведение нагрузочного тестирования была развернута виртуальная среда с конфигурацией:

* MemTotal: 501776 kB
* Cpu MHz: 2593.7848
* Cpu cores: 1
* OS: Ubuntu 14.04.2 LTS

В качестве входных данных приложению на вход были переданны:

* Html-страница, состоящая из 18923 строк, содержащая 462 визуальных элемента
* Файл с selenium webdriver тестами, написанными на языке python, состоящий из 6574 строк, содержащий 500 тестов, которые проверяют 20 элементов веб-приложения

Из результатов работы приложения (рисунок 7) видно, что тестовое покрытие визуальных элементов составило 4.33%. Время работы составило 17.79c, что является адекватным результатом

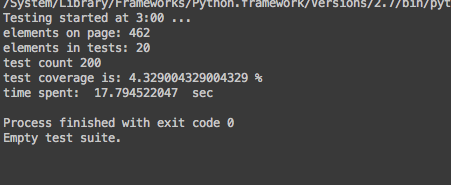


Рисунок 7. Результат нагрузочного тестирования