**Санкт-Петербургский Государственный Университет**

**Математико-механический факультет**

Кафедра системного программирования

# Автоматическое тестирование верстки web-интерфейсов

Курсовая работа студента 445 группы

Шувалова Иннокентия Петровича

Научный руководитель — А.М.Ерошенко,

инженер по тестированию в компании «ООО Яндекс»

Санкт-Петербург — 2012

# Оглавление

## Введение

В современном программировании одним из важных или даже необходимых этапов является тестирование. В больших компаниях существуют отделы, полностью ему посвященные, которые могут составлять довольно большую часть персонала.

В компаниях, занимающихся разработкой веб-интерфейсов тестирование можно разделить на два типа: ручное и автоматическое, которое представляет из себя в основном написание «автотестов» – программ, заточенных на выполнение заложенных в них изначально действий – выполнение тестовых сценариев – и проверку каких-то конкретных признаков поведения интерфейса, протестировать которые, собственно, задачей и является. Например сценарием может быть фраза: «Зайти на такой-то url, нажать такие-то кнопки и проверить, что появился блок с надписью “Вы успешно зарегистрированы”».

Что можно сказать, сравнивая эти два подхода к тестированию? На начальном этапе автоматическое тестирование может быть более затратно, но со временем оно начинает заметную приносить выгоду по сравнению с ручным потому, что, в случае, если тесты написаны грамотно, оно позволяет не задумываться об уже покрытой тестами функциональности и моментально получать уведомления в случае поломок совершенно бесплатно. Но, очевидно, его главный минус заключается в том, что это касается только простых сценариев и базовых проверок. Пока не существует эффективных способов определения «ошибок вообще», в то время как ручной тестировщик способен по одному взгляду на страницу выявить множество ошибок или с уверенностью заявить, что их нет.

Глобальная идея заключается в том, чтобы постараться автоматизировать как можно больше из тех алгоритмов, которые использует ручной тестировщик интуитивно. Представим себе, что мы имеем хотя бы такую программу, которая хоть определяет не 100% и не даже не 60% ошибок, но которая работает абсолютно сама, без помощи оператора. При наличии достаточных мощностей (которые по текущим прогнозам не должны быть хоть сколько-то значимы в масштабах компании) мы сможем значительно сократить объем ручного тестирования без существенной потери производительности всего процесса разработки, а это уже непосредственно сказывается на прибыли компании.

В моем представлении такой гипотетический универсальный тул, способный определять многие типы ошибок самостоятельно, логично делить на различные

независимые составляющие, одной из которых является тестирование верстки страницы, которому и посвящена эта работа.

Насколько мне известно, до сих пор не существует аналогов или прототипов таких программ, поэтому даже для составления конкретных требований необходимо детально изучить предметную область, оценить возможности построения такой программы в принципе и трудоёмкости подзадач.

В конечном итоге, достижение которого лежит за пределами данной курсовой работы, задача состоит в том, чтобы получить программу, которая, получая на вход url страницы, будет сообщать обо всех ошибках верстки, определение которых в принципе поддаётся детерминированному алгоритму. В результате своей работы программа должна создавать визуализированный отчет об ошибках, который будет легко читаем и удобен для анализа.

Целью же данной работы является сбор и структуризация сведений о распространенных ошибках верстки, анализ возможных подходов к построению алгоритмов их нахождения и создание прототипа описанной выше программы и оценка эффективности его применения. Этот прототип дожен определять основные типы ошибок, которые появляются при изменении размера окна браузера и заключаются неправильном расположении блоков страницы относительно друг друга.

Стоит заметить, что невозможно составить универсальные признаки «неправильности», потому как веб-разработка является на текущий момент довольно слабо формализованной отраслью (в рунете – в особенности) и различные программисты могут придерживаться разных способов верстания страницы, ограниченных, грубо говоря, только их фантазией. Поэтому в качестве таких критериев я буду рассматривать наиболее общепринятые и основные правила построения страницы.

Также важно понимать, что ввиду многообразия способов создания верстки и отсутствия стандартов эта область является как довольно частым подспорьем для возникновения багов с одной стороны, так и нетривиальным объектом для тестирования – с другой.

# 1. Ошибки верстки и их обнаружение

В этом разделе я постараюсь описать и систематизировать известные мне встречающиеся на страницах ошибки и привести описания алгоритмов, позволяющих их обнаружать. Важно понимать, что все нижеописанные ошибки могут встречаться как на статичных, так и на динамических страницах. Во втором случае они могут появляться в результате каких-либо действий с объектами страницы, а анализ заложенной в страницу логики является большой проблемой, о которой будет сказно позднее. В любом случае не стоит смешивать непосредственно критерии определения и условия появления ошибок.

Забегая вперед, скажу, что практика применения прототипа показала, что на страницах зачастую встречается очень много мелких ошибок, неразличимых для человека и потому совершенно несущественных. Робот должен уметь отсекать такие ошибки, для чего для каждого из типов необходимо описать критерии определения значимости этой ошибки среди других ошибок такого же типа, где это возможно.

Забудем пока о прозрачности блоков, о слоях (я имею в виду CSS-параметр z-index, позволяющий располагать блоки “ближе” к экрану и “дальше“ от него). Будем смотерть на страницу как на кусок плоскости с расположенными на нём прямоугольными блоками, каждый из которых соответствует одному тегу дерева html. Поведение каждого из них при изменении окна браузера определяется в первую очередь типом тега, а также множеством CSS-параметров как самого блока, так и его родителей в дереве.

**1) Расположение детей относительно родителей**

Наверное, это самая часто встречающая ошибка, когда блок-дитя выезжает за границы блока-родителя. Притом последствия могут буть весьма серьезны: например, какой-либо блок может закрыть собой важную кнопку на странице.

Бывает и так, что такое поведение блоков не является ошибкой, это происходит в следующих двух случаях:

1) CSS-параметр position у блока-дитя выставлен в значение fixed или absolute, что дает возможность позиционировать его на странице не влияя на расположение других блоков.

2) CSS-параметр overflow у блока-родителя выставлен в значение, отличное, от visible (умолчание), т.е. одно из значений: scroll, auto и hidden[1]. В случае hidden всё, что выезжает за границы этого блока, будь то текст или дочерние блоки, не отображается никогда, а в случае scroll блок получает полосы прокрутки по краям. Параметр auto аналогичен параметру scroll за исключением того, что полосы прокрутки добавлятся только в случае необходимости.

Понятно, что для обнаружения этого типа ошибок необходимо пройти по всему дереву html-тегов, начиная от тега <body> и для каждой пары «дитя-родитель» проверить вложенность одного в другой покоординатно. В процессе обхода, конечно, не нужно учитывать приведенные выше исключения, а также невидимые блоки.

Довольно очевидно, что весьма удобной мерой значимости будет то, насколько сильно дитя выехало за дозволенные границы. Можно рассматривать площадь отсеченной фигуры, сумму отклонений по каждой из координат и т.д. Мне кажется наиболее удобной мерой максимальное значение отклонения из всех 4-х (справа-слева, сверху-снизу).

**2) Расположение блоков одного уровня дерева**

К сожалению, в русском языке нету такого удобного слова как «siblings» в английском, обозначающего всех детей одного(-их) родителя(-ей), потому что именно о них пойдёт речь.

Первый тип ошибок с ними связанный – это наползание одного из детей на другого. Естественно, как и в предыдущем случае, не стоит рассматривать здесь блоки с position: absolute или fixed. Однако параметр overflow никак не должен влиять на наши проверки, за исключением того случая, когда он выставлен в значение hidden, и один из двух детей, которых мы проверяем на текущем шаге, полностью находится за границами родителя и, как следствие, не виден пользователю.

---

[1] Значение inherit рассматривать не стоит, так оно свидетельствует об унаследовании этого параметра от родителя.

Возникает вопрос: почему я не рассматриваю наползание друг на друга блоков из разных ветвей дерева, а рассматриваю только соседние вершины? Дело в том, что в случае когда перекрываются два блока, не являющиеся непосредственными детьми одного и того же узла, один из них будет вынужден выехать за границы своего родителя. А поскольку это мы уже проверяем, то нет смысла рассматривать в этом тесте все возможные пары блоков на странице, что существенно уменьшает время работы программы.

Вопрос выбора меры значимости также, как и в предыдущем случае, не представляет труда; на этот раз я предпочел взять площадь пересечения. Сравним эту меру, например, с мерой, выбирающей большую сторону прямоугольника, образованного пересечением, в случае, когда один блок наехал на другой по всей своей высоте, но всего на пару пикселов. Понятно, что такой баг некритичен, и мера по площади даст нам маленькое значение, в то время как мера по максимальной стороне даст весьма немалое значение.

Перейдём ко второму типу ошибок связанных с блоками-соседями. Будем отслеживать их взаимное расположение при разных разрешениях браузера или до и после других действий на странице. Диагностировать ошибку будем в том случае, если несколько блоков, изначально находившихся на одной координате по вертикали или горизонтали, после вышеописанных действий потеряли это свойство: т.е. некотороые из блоков перестали лежать на той же линии, что остальные.

Это тоже весьма распространенный тип ошибок, и по своему происхождению он схож с первым типом, описанным мной в разделе “расположение детей относительно родителей”. Дело в том, что в зависимости от CSS-стилей и типов блоков при изменении размеров блока-родителя его дети либо пытаются подстроиться под его новые размеры и нередко “съезжают” друг оносительно друга, либо же не пытаются подстраиваться и остаются на том же месте, где и были, сохраняя свои размеры, что и порождает выход за границы родителя.

Что может здесь служить мерой? Пусть у нас имеется $n$ блоков, имевших общую вертикальную координату до изменения размера окна – $x$. Пусть после изменения $m < {}^{n}/_{2}$ блоков у нас изменили свою вертикальную координату на значения $x_1, \dots x_m$ соответственно, а остальные блоки остались на высоте $x$. Тогда вес этой ошибки положим равным сумме квадратов отклонений по всем $m$ блокам. Почему это удобно? При такой

мере у нас учитывается, в первую очередь, отклонение самого сильно съехавшего блока, но также, во вторую очередь, учитываются и все остальные отклонившиеся блоки.

**3) Различимость текста**

Конечно, редко, можно встретить, чтобы программист случайно установил одинаковые цвет текста и цвет фона какого-либо блока, но даже это случается. Более вероятна ситуация, когда цвет текста случайно совпадает с фоном не непосредственного родителя, а ближайшего родителя с установленным фоном. Этот родитель может находится довольно далеко в дереве, поэтому такую ошибку сложнее заметить на стадии создания страницы.

Алгоритм детектирования таких ошибок довольно прозрачен: для всех тегов с внутренним текстом будем подниматься по дереву до тех пор, пока не встретим тег с установленным фоном, после чего сравним эти два цвета. Но сравнивать будем «умным» способом: ведь простого сравнения «совпал – не совпал» тут не достаточно потому, что мы имеем дело с цветами, которые могут быть близки, а потом плохо различимы.

За наиболее лучшим решением стоит обратиться к специалистам по компьютерной графике и обработке изображений, но как довольно эффективный, хоть и очевидный вариант могу предложить следующую оценку «похожести» цветов, которая одновременно и будет весом найденной ошибки.

Пусть у нас есть два цвета в пространстве RGB, один из них имеет показатели цветности $(r_1, g_1, b_1)$, а другой – $(r_2, g_2, b_2)$. Тогда различием этих цветов назовем величину, равную нормированому расстоянию между этими двумя точками в трехмерном пространстве:

$$difference = \frac{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}{255\sqrt[3]{2}}$$

$255\sqrt[3]{2}$ – это длина максимальной диагонали куба с длинной стороны 255, т.е. рсстояние между наиболее сильно различаюсимися цветами. Соответственно схожесть будет равна:

$$similarity = 1 - \frac{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}{255\sqrt[3]{2}}$$

Однако скорость работы этого алгоритма можно сильно улучшить следующим очень простым усовершенствованием: мы будем обходить дерево блоков только один раз сверху

вниз на каждом шаге запомниная фон текущего блока и сравнивая цвет его текста с фоном, запомненным на предыдущем шаге алгоритма. Кроме того, мы сможем сразу отбрасывать те ветви дерева, где текста нет ни в одном из блоков.

**4) Картинки**

При описании предыдущего подхода я никак не затронул тот момент, что в процессе обхода мы можем встретить либо картинку, либо тег с фоновой картинкой. Допустим мы обнаружили, что текст расположен поверх картинки. Тогда сначала необходимо определить, на какую именно её часть он накладывается. В целом это не является сложной задачей с точки зрения программирования, а подробно разбирать все комбинации значений параметров, влияющих на расположение текста в теге и расположение картинки я не считаю ни интересным, ни необходимым.

Задача определения читаемости текста на картинке, на мой взгляд, довольно сложна, и в этой работе я не буду останавливаться на этой проблеме, так как она относится скорее к сфере обработки изображений, нежели к web-технологиям.

**5) Прозрачность**

Ни для кого не секрет, что блоки могут быть полностью или полу-прозрачными, и что разработчики очень часто пользуются этой возможностью. Продолжим разговор про читаемость текста. Не умоляя общности рассмотрим ситуацию, когда блок с текстом полупрозрачен, обладает собственным фоном и имеет родителя с другим цветом фона. Нужно рассчитать новый цвет фона на основе двух данных цветов и коэфициента прозрачности, сравнить цвета фона и текста по описанному выше алгоритму, повысив при этом планку допустимых значений для результата сравнения в зависимости от коэфициента прозрачности. Ситуация с полупрозрачным текстом на фоне картинки решается теми же методами.

**6) Исчезание блоков**

Стоит упомянуть также ситацию, когда при изменении размеров окна блок оказывается за границами страницы и имеет при этом имеет абсолютную или фиксированную позицию, либо, в случае если у его родителя установлен overflow: hidden, пропадает с экрана, оказавшись вне родителя. Понятно, что выбор меры и алгоритм обнаружения ошибки достаточно очевидны в этом случае.

**7) Слои**

Блоки страницы могут быть удалены от экрана на задний план или наоборот выдвинуты на передний. Этим, естественно, довольно часто пользуются при разработке. Нередко при этом возникает ситуация, когда один из выдвинутых на передний план блоков пересекается с другим таким блоком. Эти блоки при этом могут принадлежать к разным частям дерева, которые, может быть, даже были написаны разными програмистами, которые не согласовали поведение блоков в такой ситуации. Как следствие любой из двух блоков может оказаться наверху.

Как пример, нередко выпадающее меню оказывается под баннером рекламы.

Сложность определения этого типа ошибок в том, что нет формальных признаков, заложенных в верстку страницы, по которым можно судить о том, какой из перекрывающихся блоков должен быть сверху. Это можно понять только по логике действий на странице, о понимании которой будет сказано позже. В качестве довольно простого приближения можно считать, что тот из блоков, который появился на странице или стал видимым позже – тот и должен находиться сверху.

**8) Прочее**

Кроме того есть немало ошибок, определение которых не требует существенных усилий, но тем не менее немаловажно. Это такие простые проверки как определение уникальносьти значений аттрибутов id среди всех элементов, валидация кода страницы с точки зрения html как языка разметки: все необходимые теги должны быть закрыты и древовидная структура не должна быть нарушена.

К этому разделу я также отнесу проверку соответствия количества ячеек в разных строках таблицы: если не выставлены или выставлены неправильно специальные аттрибуты colspan или rowspan, позволяющие объединять несколько ячеек в одну, то таблица может потерять свой «табличный» вид и ровную структуру, что, конечно, никакому дизайнеру не захочется видеть у себя на сайте.

На самом деле, многие из перечисленных выше ошибок иногда не составляет труда определеить исключительно по коду страницы, но такой подход имеет два минуса. Во-первых, он не настолько универсален, как эмуляция работы со страницей реального человека. Работая исключительно с кодом страницы мы  не сможем найти все возможные ошибки хотя бы потому, что большинство логики работы с элементами часто заложено в

код javascript. И во-вторых, это метод не позволяет на выходе программы выдать результаты, понятные человеку, не разбирающемуся в построении верстки во всех деталях.

## 2. Реализация

На данный момент мной реализован прототип программы на java с использованием технологии Selenium Webdriver [1], которая позволяет вызывать из java-кода методы api практически всех популярных браузеров. Selenium бладает достаточно богатым инструментарием для работы с браузером и элементами страницы. Насколько мне известно, это лучший инструмент для работы с веб-интерфейсами напрямую из java.

В рамках имеющийся программы реализованы пункты «1) Расположение детей относительно родителей» и «2) Расположение блоков одного уровня дерева» из списка рассмотренных ошибок, поскольку эти типы ошибок наиболее распространены по сравнению с остальными.

Программа работает на JUnit 4 [2] и запускается с помощью Apache Maven [3].

**Схема алгоритма работы программы**

Входные параметры подаются через конфигурационный файл. Программа начинает работу с извлечения элементов со страницы для представления в виде дерева объектов в памяти программы. Далее по очереди запускаются следующие модули проверок:

1) проверка расположения детей
2) проверка наложения блоков-соседей друг на друга
3) проверка сохранения блоками-соседями общей координаты

Каждый из них сначала, совершаяя обход этого дерева, составляет списки тестовых сценариев – заданий на проверку конкретных наборов элементов со страницы, по описанным в разделе 1 алгоритмам, после чего выполняет все сгенерированные проверки.

Каждый модуль  при обнаружении ошибки сохраняет скриншот страницы, отмечает на нем проблемные элементы и добавляет соответствующую запись в хранящийся в памяти программы отчет.

После завершения работы модулей, отчет записывается на диск в .xml-файл с помощью технологии JAXB [4], поле чего на его основе через xslt [5] генерируется html-страница с отчетом, содержащим скриншоты вместе с описанием деталей ошибок.

## 3. Дальнейшее развитие и проблемы

### Кластеризация ошибок в отчете

Часто найденные ошибки на странице можно эффективно объединить во множества аналогичных ошибок и значительно улучшить читаемость отчета, но это является довольно непростой задачей.

Приведу пример: допустим, разработчик во многих местах на странице использует один и тот же шаблон, который содержит в себе ошибку – например один из элементов этого шаблона выходит за его границы на 10 пикселов. Пусть на странице присутствует ещё 2-3 аналогичных ошибки, где элементы также выходят за границы на те же 10 пикселов, но эти элементы не относятся к этому шаблону.

Если рассматривать xpath [5] в качестве локаторов элементов, то ошибки в шаблоне от остальных ошибок будет отличать одинаковое окончание их локаторов, но проблема в том, что совпадающий постфикс может быть и не очень длинным, он может состоять вообще из одного-двух тегов, а в таком случае объединять ошибки в одну, конечно, не стоит. Думаю, стоит экпериментально определить оптимальную длинну постфикса, начиная с которой есть смысл делать предположение о кластеризации.

Также можно попробовать дополнить xpath сведениями об html-классах, id элементов или об их тексте – это скорее всего улучшит результаты, но немного увеличит продолжительность работы программы.

### Проверка корректности ссылок

Ссылкой я назваю тег <a> на странице; её корректностью я называю тот факт, что она «ведёт» на реально существующую страницу. Мне кажется неправильно эту задачу относить непосредственно к проверке верстик, потому что всё-таки она, во-первых, работает со многими url'ами, и во вторых опирается на совершенно другие технологии: тут необходимо знание сетвых протоколов пользовательского уровня, таких как http и ftp и других технологий с ними связанных, в то время как проверка верстки работает с html, css и, возможно, в будущем – с javascript.

Мной уже также написан прототип такого робота, но там существуют определенные не решенные пока проблемы, так что к внедрению в использование и к сращиванию с роботом-тестироващиком верстки он пока не готов.

**Динамические страницы**

Как мы все знаем, подавляющее большинство страниц в интернете содержат в себе немало логики, реализованной с помощью javascript. В процессе действий на странице одни элементы могут появляться, другие – исчезать, а третьи – менять свое местоположени или родителей в дереве.

К сожалению, эта область не только совершенно не исследована, но и, насколько мне известно, этой проблемой практически никто не занимается. В идеале хотелось бы действовать примерно так: распознать все возможные действия на странице, разбить их на эквивалентные классы, после чего начать выполнение тех действий, которые мы хотим проверить, проверяя верстку на каждом шаге.

Но это – в идеале, в действительности же очень большой проблемой является уже выявление заложенной в страницу логики – составление списка возможных действий. Мы с коллегами много разработали немало подходов к решению этой проблемы, реализовали некоторые из них, но существенных результатов добиться пока так и не удалось.

Вторая проблема не столь существенна, но всё же заслуживает внимание. На одной странице построенный прототип работает около двух минут, что немного. Но в случае, если мы будем запускать проверку после каждого выполненного действия, время работы может вырасти в десятки раз.

К сожалению, существенно улучшить быстродействие, придерживаясь использования java вместе с Selenium'ом невозможно. Но, в принципе, это не так важно, так как время работы не является самым ключевым требованием к программе такого типа, пока оно остается в разумных пределах.

**Конфигурирование запусков. Искючения**

Понятно, что использование описанных проверок имеет большой недостаток: в стремлении универсализировать проверки приходится жертвовать некоторыми способами определения багов, которые уместны не всегда, но часто. Что необходимо проверять на одной странице, на другой является частью логики разработчика. Поэтому вместе с

увеличением базы проверок необходимо работать и над возможностями гибкой подстройки робота под конкретную страницу или сайт. Такую подстройку должен выполнять оператор, потому что только человек знает, что на данной странице **должно** быть, и как оно **должно** работать согласно задумке разработчика.

Необходимо так продумать интерфейс и сам процесс подстройки, чтобы возложить на человека минимум работы. Как вариант, можно после первого прогона робота предлагать оператору с помощью нескольких кликов указать, что он ошибками не считает, после чего уже перестроить текущий или составить новый, итоговый отчет.

**Обучение на ошибках**

Реализовав систему обратной связи и внедрив робота в довольно регулярное использование, можно начать задумываться об автоматическом его обучении. Обучении чему? В первую очередь тому, какие ошибки важны, какие – нет, а какие редко считаются ошибками вообще. Это не представляется сложно задачей, достаточно просто собрать большую статистику запусков.

Куда более интересной и перспективной выглядит другая идея: предсказание результатов проверок после статического анализа кода страницы. Поясню, что я имею в виду: можно попробовать разработать такую нейронную сеть (или использовать другой подходящий для задач такого типа объект теории искуственного интеллекта), которая будет получать на вход необходимую информацию о странице и на выходе прогнозировать какие типы ошибок на ней возможны, в каких узлах они могут произойти и, главное, насколько они значимы для этой страницы.

Конечно, это является очень непростой задачей, и я, не будучи экспертом в данной области, не берусь утверждать, что результат окупит затраты на построение такой системы.

## Заключение

Даже небольшое количество запусков показало, что этот прототип успешно находит много ошибок, подавляющее число из которых – несущественные, как и можно было ожидать.

Тем не менее используя систему присваивания веса каждой ошибке можно весьма эффективно выделять наиболее существенные поломки и включять их в приоритетную часть отчета, либо вообще не показывать все остальные ошибки – в зависимости от требований заказчика. Таким образом на выходе программы мы будем иметь удобный и информативный отчет о главных поломках на странице.

Не вызывает сомнений как возможность реализации большинства проверок других описанных типов, так и их эффективность.

Что касается способов применения данной программы, то стоит рассказать о двух известных мне способах её применения: первый из них предназначен для самих разработчиков интерфейсов. Можно настроить программу на url, который находится в процессе разработки в данный момент, автоматически запускать при выпуске каждой новой версии и присылать отчет в случае поломок.

Второй способ применения заключается в том, чтобы использовать какой-нибудь имеющийся crawler – обходчик страниц по ссылкам – и запускать проверки верстки на каждой достигнутой странице. На первый взгляд кажется, что это очень долгий процесс, но это не совсем так: как я уже упомянал, на одной странице робот работает пару минут, так что даже для большого сайта возможно за час получить читаемый отчет, не затратив никаких человеческих, а соответственно – и денежных ресурсов.

В виду изложенных достоинств описанного в этой работе подхода я полагаю, что уже в ближайшем будущем эта технология займёт свое место в производственном процессе. Конечно, для повышения её эффективности ещё предстоит проделать немало работы, но я уверен в том, что эти затраты быстро себя оправдают.

## Ссылки


1. Selenium Webdriver // Официальный сайт проекта
   URL: http://seleniumhq.org/projects/webdriver/
2. Junit // Официальный сайт проекта
   URL: http://www.junit.org/
3. Apache Maven // Официальный сайт проекта
   URL: http://maven.apache.org/
4. JAXB // Официальный сайт проекта
   URL: http://jaxb.java.net/
5. xsl // Официальный сайт проекта
   URL: http://www.w3.org/Style/XSL/