UNIVERSITY OF OSLO
Department of Informatics

Is using images to test web pages the solution to a Sisyphean task?

Tarjei Huse

April 30, 2008

Summary

This thesis describes a system, Sysifos, that automates capturing and segmentation of screen dumps of web pages. The system builds a model of the spatial structure of a page based on the segmentation. The model is used when comparing two screen dumps. The system uses image analysis techniques to segment the page. The model is then compared to a model generated earlier. The model comparison is packaged in a form so that it may be used as a test oracle in standard Java testing frameworks, for example Junit or TestnG. The motivation for the development of Sysifos is that there are currently no established ways to automate testing of browser rendering, although browser related bugs are important. In this thesis one operation running 56 high volume websites was investigated, and it was found that browser related bugs represented around 13% of all bugs needing developer attention. Sysifos was evaluated using a test set containing known errors. It found 100% of the errors it was expected to find, but reported one false positive. Test results are visualized using SVG. This thesis show that using a test oracle may be beneficial when testing browser rendering of web pages. Currently, the image capturing service of Sysifos is not satisfactory according to speed and reliability, but the results of the evaluation indicate that Sysifos has the potential to become a valuable tool if the image capturing service can be improved.

Acknowledgements

First and foremost I would like to thank my main advisor, professor Anne Schistad Solberg, for providing timely inputs and keeping me afloat at times when I wanted to sink. She has always had enough time to provide me with the advice I've needed to continue writing.

Thanks are also due to my other advisor, associate professor Bente Cecilie Anda, for providing a helpful critique of the thesis.

The guys and girls at Apressen Interaktiv also deserve a sincere thanks, both for letting me use your resources as well as access the bug database. You rock!

Whiskey and thanks will be delivered to my good friend Knut Olav Nortun who took on the ungrateful task of correcting my English into something readable.

Last but not the least I would like to thank my family and my lovely wife Kaja for support and understanding during the last hard months of this thesis.

Contents

1 Introduction **13**

1.1 Is it possible to use images to test web page rendering? 13

1.2 Can pattern recognition techniques bring something to the table for testers? 14

1.3 The differences between a test, a test case and an assertion 15

1.4 Focus of this thesis 16

1.4.1 Create a test framework for capturing screen dumps of web pages 18

1.4.2 Explore different assertion methods 18

1.4.3 Evaluation of the framework's usability in web development 18

1.5 Why Sysifos? 19

2 Background **21**

2.1 Escaping fear using "Test driven development". 21

2.2 Web programming: The complexity is in the browser 22

2.2.1 Existing methods for testing web pages 25

2.2.2 General advice on GUI testing: don't. 25

2.2.3 Test oracles for GUI programming . . . 26
2.2.4 Use of image analysis techniques on webpages . . . 27

3 System overview and design 29

3.1 A brief overview of the system components . . . 30
3.1.1 Sysifos: An image assertion framework . . . 31
3.1.2 Test framework: JUnit or TestNG . . . 31
3.1.3 Browser agent: Selenium Remote Control (Selenium RC) . . . 32
3.1.4 Image processing library: ImageJ . . . 34
3.1.5 Result visualization . . . 34
3.2 Image storage container . . . 35
3.3 Extending Selenium RC to get complete images of the webpage. . . . 36
3.3.1 Merging algorithm . . . 37
3.3.2 Selenium startup . . . 38
3.3.3 Problems with Selenium RC . . . 38
3.3.4 Supported browsers and operating systems . . . 39
3.4 API description . . . 40
3.4.1 Test acquisition . . . 40
3.4.2 How do you assert image values? . . . 40
3.4.3 Singel image asserts . . . 42
3.4.4 Comparisons with the original . . . 43
3.4.5 Build a model of the webpage . . . 43

3.5 Page model XML format 44

4 Creating a high level model of an image of a webpage 49

4.1 What to expect in a screen dump of a web page 49

Image size 51

Operating system and browser 52

Color settings 52

Object shapes found in typical images 53

Borders, margins and space 53

Page background 54

4.2 Process overview 55

4.3 Finding pictures in the image 55

4.3.1 Choosing the right threshold and cross size 60

4.3.2 Gradients 60

4.3.3 Cleanup and labeling of the pictures 61

4.4 Segmenting on color 61

4.5 Region filtering 63

4.5.1 Separating out text regions 65

4.5.2 Filtering out block regions 66

4.5.3 Defining region relations 68

4.5.4 Grouping text regions 68

4.6 Building a region model 69

4.6.1 Border detection 71
4.6.2 Segmentation results 72
4.7 Page model representation 72
4.8 Looking for model deviations 76
4.8.1 Comparing page graphs 76
4.8.2 Node comparison algorithm 77
4.8.3 Finding the real error - not the sideeffects. 78
4.9 Defining and representing different variations 79
4.9.1 Dimension variations 79
4.9.2 Text and image variations 80
4.9.3 Model visualization and error representation 81

5 Experimental results **83**
5.1 How important are visual and browser related bugs in web development? 83
5.2 Using Sysifos for continuous monitoring 86
5.2.1 Test setup 87
5.2.2 Results 88
5.2.3 Discussion 90
5.3 Using the page model to find errors 91
5.3.1 Test setup 92
5.3.2 Evolving the pagemodel 95
5.3.3 Results 97

5.3.4 Cross browser testing 97
5.3.5 Discussion 101

6 Discussion and concluding remarks **103**

6.1 Future work 106

A Appendix **109**

A.1 Source code for the template page experiment. 109
A.2 Package versions 113
References 117

Chapter 1

Introduction

This thesis investigates two questions. The questions relate to each other and were developed in parallell. They are:

1. Is it possible to use images to test web page rendering?
2. Can pattern recognition or other computer vision techniques bring something to computer testing?

Answering these questions fully is beyond the scope of this thesis. The goal is to be able to give guarded answers to the above questions. Let us explore them in detail.

1.1 Is it possible to use images to test web page rendering?

This thesis is motivated by a very specific need, namely: How can you verify that your combination of CSS and HTML/XML markup creates a document that looks the way it should in different web browsers?

A web page may consist of a large amount of different files that are downloaded, parsed and interpreted by a browser, in the end creating a visual image that is viewed by humans.

While the there are some fairly well developed practices and tools [2] for automated functional testing of web applications there are none that test a browsers rendering of a web page.

Because of the complexity of the web pages there is little use of automated tools for creating CSS and markup for larger sites, this is done by developers by hand starting with an image of how a designer wants the page to look. This provides ample room for creating subtle errors and problems that do not show up at once.

A major change from earlier GUI environments is that it is the users who decide which program they want to use to view a page. Both the browser and the operating system markets are fragmenting, and the result is that it is no longer commercially viable to just target one browser.

Web browsers are very complex beasts. Theoretically evaluating what a browser will do with your code and how a page will be rendered is not likely to be feasible.

Thus what is needed is an approach that is based on what each browser model actually renders. The problem is how to verify that the rendering is correct.

1.2 Can pattern recognition techniques bring something to the table for testers?

“An image says more than a thousand words”

What can then image processing and analysis techniques provide to the domain of testing? Mainly the ability to find patterns that are not directly discernible by looking at raw markup or by doing a bit by bit comparison of two images. By segmenting the image we can extract information that can be used by a test oracle.

A test oracle can be described as somewhere your test may go to find out if the outcome from a test case is correct. If the inputs are complex and changing, then it may be possible to use an external generator of expected outcomes instead of trying to anticipate all test outputs in the test.

The idea in this thesis is to use an “original” image of a web page to be the oracle and then try to extract out information from this image that may be used to find unwanted changes in the new web

page. The original may be created by a designer or be created using a screen dump of the page.

One thing that makes this especially hard is that for the tool to be useful, it must be able to catch changes down to the level of pixel changes, for example a border moving one pixel to the right.

1.3 The differences between a test, a test case and an assertion

There are some terms that should be explained before we continue.

In the article that defined modern unit testing, “Simple Smalltalk Testing: With patterns”[7],Kent Beck defines some terms related to testing which are worth mentioning.

A **test case** is usually a class containing a set of methods where each of the methods may or may not be a test.

Testcases are aggregated into a **test suite**.

To run the tests a **test runner** is used. It executes each test case, collecting the results into a **testreporter**. The test reporter will then report the testresults back to the user. The amount of information in the test report may vary.

The totality of these functions can be referred to as a test framework. The definitions above were created in a programmatic setting, which is why the terms class and method is used above.

The term **test** or test case is defined by the IEEE Standard 610 (1990) as:

> (1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. (2) (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

In this text we mainly focus on the first item above. Our focus will mainly be on the “expected results developed” part of the definition. When a test is run, the results produced must be evaluated and found to contain the values we want. These checks will be called **Assertions**. The term will both be

used to define the act of checking some values after a test and also program libraries used to help the programmer perform these checks.

Also note that the term **test case** is often used for a set of tests that are closely related (but not connected).

A simple example might help here. A small test may look like this:

```
1 public void testMultiplication() {
2   int result = 2 * 3;
3   assertEquals(6, result);
4 }
```

The test above is in Java and written for the JUnit[18] library. The program we run here is very small, just the 2 * 3 part of the test. It produces an output recorded in the integer result. The integer is then checked or asserted on the next line using the helper function assertEquals. Sometimes this text will refer to these helper functions as an assertion as well. An assertion method may then be a way to do a check, for example to assert that the value is not null or to assert that two pointers point to the same object.

The term **browser** or web browser is used to refer to a device used for interpreting a combination of HTML/XML markup, images and CSS that together make up a web page. Figure 1.1 shows a screenshot of a browser and marks out the **viewport**, the visible part of a web page in a browser. Modern browsers provide scrollbars if the web page is longer or wider than the viewport, but when to do so is not clearly defined.

1.4 Focus of this thesis

The focus of this study is divided into three parts:

1. Create a framework for acquiring images of web page rendering from different browsers.
2. Explore different assertion methods that may be applied to images.
3. Evaluate the usability of the framework as a tool in web development.

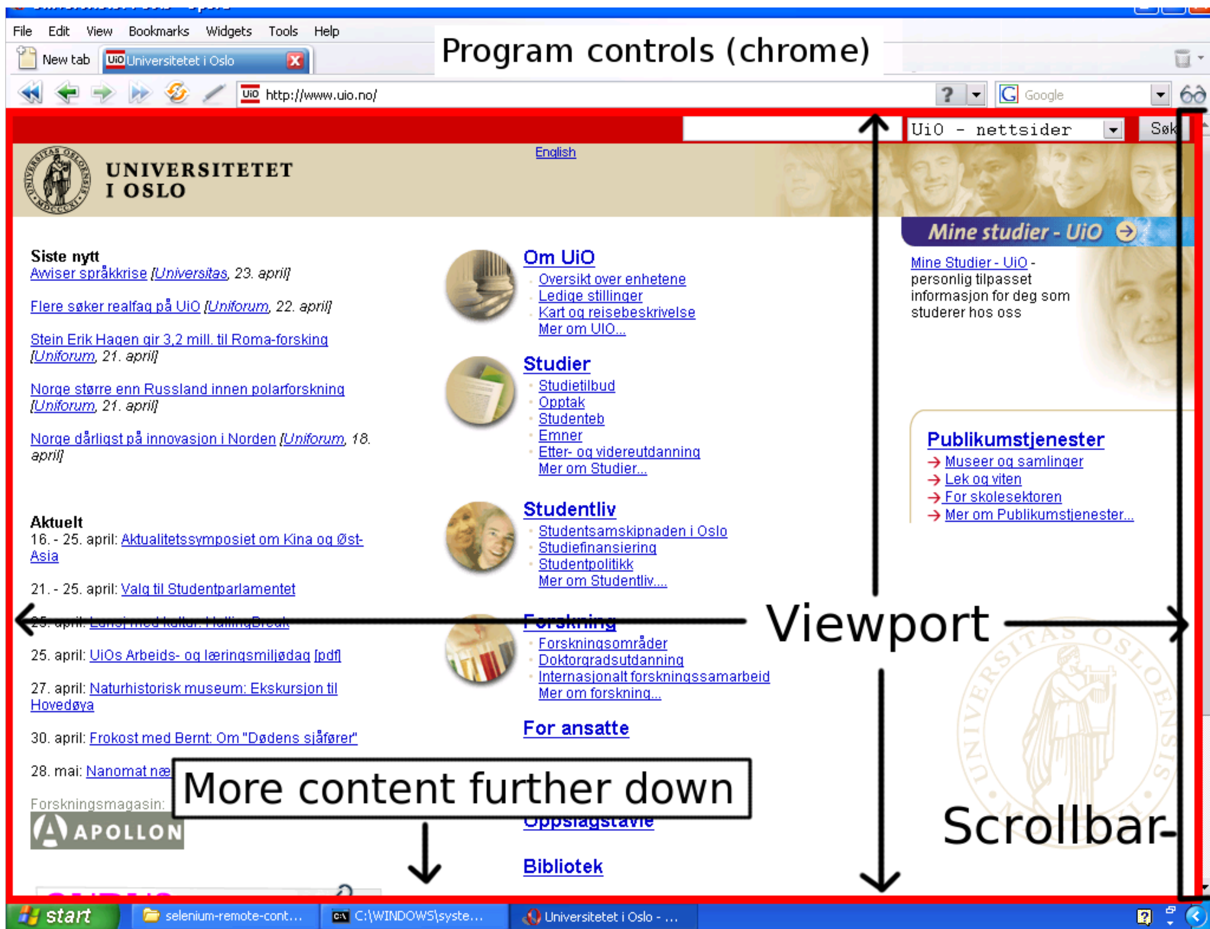


Figure 1.1: Browser terms. The viewport is the part of the browser that shows you the web page. The program controls are often called chrome. Also notice that there is more content on the page if you scroll.

1.4.1 Create a test framework for capturing screen dumps of web pages

To be able to run tests one needs a framework to run the tests within. This study will tries to create a framework for both image acquisition and test generation. It is not the goal of this study to write this framework from the bottom up, but rather to add the missing components to frameworks that already exist.

1.4.2 Explore different assertion methods

When the output of a test is an image, the number of output values is just too high check using most normal assertion methods, more specialized assertions are needed, especially as the image isn't the same from test to test.

This thesis will explore both simple pixel by pixel assertions and using image segmentation techniques to extract information from the image that may be used in the comparisons. One special case here is an idea of an ”original“, an image that is defined as a correct view of the site. Possible comparison methods will be explored.

1.4.3 Evaluation of the framework's usability in web development

Based on the results of the two above sections, we will try to evaluate the framework's usability for web development. The evaluation will be based on some simple usage tests of the framework.

The study will focus on traditional websites serving articles as well as videos and images such as the web pages for newspapers, TV-stations or some NGOs. These sites share some common traits with regard to layout and how different types of content is visually linked in the page.

The target for this thesis is a tool for web developers, not developers of image processing applications. This constrains the universe of possible tools to use.

The main result of this constraint is that tool developed should not rely on special image processing packages not easily available (i.e. Matlab) or on other closed source packages.

1.5 Why Sysifos?

The system created by this thesis is called Sysifos. This comes from Sisuphos son of Aeolus and founder and king of Corinth according to Greek mythology. For his disrespect to Zeus, he was condemned to eternally push a heavy rock to the top of a steep hill, where it would always roll down again.

Sometimes making sure that a webpage looks perfect across all browsers seems like a Sisyphean task. This thesis tries to at least automate most of the work.

Chapter 2

Background

This chapter will delve into the reasons for this thesis and look into prior work. It will look into the current trends in test automation for web development and then continue on to look at how computer vision (CV) techniques have been applied to segmenting web pages.

2.1 Escaping fear using "Test driven development".

Test driven development(TDD) has risen to prominence during the last ten years. Partly as a result of the rise of TDD, IEEE Software published a feature edition dedicated to TDD and related techniques with Ron Jeffries and Grigori Melnik as guest editors.

In their introduction article, Jeffries and Melnik[22] writes that TDD first appeared "in Kent Beck's Extreme Programming Explained, which came out in 1999. In 2002, Beck released Test Driven Development: By Example".

The basic tenet behind test driven development is that the developer writes tests for the code he creates as the code is created[1][8]. This ensures that when the developer later goes back to change something in the code he can be fairly sure that his tests will tell him if the change has a side effect.

Because the developer knows that the tests will tell him if something is wrong, he can make changes

[1]What comes first, tests or code, is a matter of debate, but will not be discussed here.

more quickly than if he didn't because he does not have to think through every possible consequence of his modifications. This increases productivity according to evangelists of TDD.

In "TDD: The art of fearless programming", Melnik and Jeffries examine a long list of studies looking at the effects of TDD on productivity and quality The authors write, "All researchers seem to agree that TDD encourages better task focus and test coverage. The mere fact of more tests doesn't necessarily mean that software quality will be better, but the increased programmer attention to test design is nevertheless encouraging.".

The studies Melnik and Jeffries summarize either report that an increased amount of effort as a result of TDD or that it had no effect. They also report that either the number of bugs was reduced or that there was no effect. A point that Melnik and Jeffries do not make is that if you only look at the studies of projects lasting for more than 6 months, then they all report improved quality and reduced productivity.

It is clear that TDD has increased the enthusiasm for automating tests in the developer community. Although the initial focus of TDD was on unit tests, many developers have moved on to look at automating other types of testing.

This has resulted in new testing frameworks like Fitnesse[2], TestNG and Selenium[9, 21]. These frameworks go further in creating developer centric test automation solutions for different kinds of test problems. Fitness is a tool to let nondevelopers create acceptance tests that are translated into real unit tests for the programmer to run. TestNG and Selenium are covered other places in this text.

2.2 Web programming: The complexity is in the browser

Before the web came along most GUI (Graphical User Interface) programming was done in different APIs based on normal, eventbased programming. Now a lot of the GUI is defined by markup languages like XML or HTML. The programmer used to have complete pixel by pixel control over the interface, but also had to stay within the bounds of what was possible to do using the APIs available. Usually another constraint was that the user interface should have a lot in common with other programs running on the same computer. Most vendors of graphical user interfaces like Apple, Microsoft

[2] http://fitnesse.org/

or Gnome issue human interface guidelines explaining how GUIs should be written so programs running the same operating system share a common look and feel.

Web programming throws a lot of these old paradigms up in the air. Usually the application GUI is built using a graphical design that is application specific. Also, the developer has given up complete control over how the product is rendered and how the user interacts with the product. Still, the user often expects the pages to render properly regardless of browser choice.

The new paradigm creates new costs. As presented in detail in section 5.1, around 13% of the bugs reported against a web application are browser related. Some 17% of the bugs were related to visual characteristics in the page. Note that the numbers are just from one site, but we have not managed to find any other studies of these issues.

The very openness of the web standards is a part of the problem. Every standard contains points that may be interpreted differently. Then you have to consider how many standards there are.

A webpage may contain up to ten different formats and standards that interact with each other, some delivered as separate files to the browsers.

As an example, I checked on March 17th 2008 what is needed to view the frontpage of Dagbladet(http://www.db.n a Norwegian newspaper. To render the page, a total of 202 http requests are required. 135 of these are images. You got 2 stylesheet files and 19 javascript files as well as 8 different flash objects. You also have 19 iframes with external content that may influence the rendering of the page and may again contain unknown extra requests as well. 17 of the requests where just redirects.

In April 2008, the website WebSiteOptimization.com published an analysis of how the size of web pages have evolved the last 5 years. They found that the average size of the 1000 most popular webpages had increased by 223% since 2003. The number of external objects in the pages increased at an annual rate of 14,5% in 2007.

Figure 2.2 shows some of the standards most modern browsers implement. Add this together in 3 -4 competing implementations as it is obvious that there will be differences in rendering. Differences that developers are told should not show up on their sites. At the same time the browser market has fragmented.

Browser usage statistics vary with the period and sites you choose to monitor but the trend is clearly

- HTML 4.01
- XHTML 1.0
- XHTML 1.1
- CSS 1.0
- CSS 2.1
- DOM Level 2
- DOM Level 1
- ECMAScript 3
- PNG
- JPEG
- GIF
- SVG

Source: `http://www.webdevout.net/browser-support-summary`. Links to the complete standard descriptions as well as more information may be found there.

Figure 2.1: Some of the standards implemented by a modern browser.

pointing towards a future where there is not one dominant browser.

The Counter [15] sells a statistics tool for webpages. They also provide global browser usage statistics based on the data they gather for their customers. In the period from May to September 2003, their usage statistics gave Microsoft's Internet Explorer versions 6 and 5.5 a total of 93% of the browser market, with Netscape 4 and 5 having 1% and 2% of the market respectively. For January 2008 the same numbers show a much larger fragmentation. Internet Explorer 6 is still on top with 42% of the market share, but then comes Internet Explorer 7 with 38% and Firefox with 14%. Safari has 3%. Opera captures around 1% of the market for both periods.

In a few months time Internet Explorer 8 will also come on the market. Then developers will have 3 different versions of the popular browser to work with, as well as Firefox, Opera and Safari. Staying on top of the different browser versions will require tools for automating testing in each browser.

What a web page should look like is defined according to the HTML/xHTML standards, the CSS standard and the different standards defining how images should be rendered. Even the image standards are not always implemented equally[30, 19, 10]. Internet Explorer 6 has problems rendering

PNG images.

Another emerging complexity is that more and more application work is moved from the server and the browser. Technologies like AJAX (Asynchronous Javascript and XML) have changed the way applications work, making them more asynchronous and harder to test.

This complexity has spawned new and novel testing methods.

2.2.1 Existing methods for testing web pages

The most basic way to test a web page is to check if a certain request actually responds with the correct HTTP response code. This approach is usually used to monitor system uptime and is sometimes regarded as a simple integration test because it verifies that the different subsystems needed to create the page works.

Having established that there is in fact a web page there, one important subclass of checks is to test that the different linked components actually exist on the page.

Another class of tests try to run a series of different requests representing a user session. For example they may log in, try to change an item and verify that this is saved to the database. This is often refereed to as web testing. jWebunit[3] for Java is one example of such a suite. Still this test can only be used to verify that the server side part of the transaction is working.

A method for testing what goes on in the browser is to use Selenium. Selenium is a tool to ”drive” a browser through a set of injected javascript commands to verify that the site works in different settings.

None of these methods test the actual rendering of the web page.

2.2.2 General advice on GUI testing: don’t.

It is worth noting that the web page is not the first graphical user interface that has been produced. Quite a few tools have been developed earlier to automate testing of other applications.

[3] http://jwebunit.sourceforge.net/

Cem Kaner [24] describes some of the lessons learned from this work. One of them is this quote:

> ”We don’t use screen shots "at all" because they are a waste of time.” - Cem Kaner.

Tests that are too connected to the user interface (like images) tend to become problematic as you have to revise all your tests after just a minor change in the user interface. This is a major problem related to image based testing.

Kaner does however note that testing just parts of the image may be useful. The sentiment above is based on testtools that only do bit by bit image comparisons instead of more advanced approaches.

So, has anyone tried to do something more advanced?

2.2.3 Test oracles for GUI programming

A test oracle is a program that a test framework may consult to check if the outputs are correct based on the set of inputs delivered. How this oracle work depends on its design.

Hoffmann [20] talks about five categories of Oracle, starting with not having one at all. The next category is the consistent oracle that just compares the data from the test to the data of the last testrun, thus protecting from regressions. Instead of just comparing results the sampling oracle selects a specific collection of inputs or results to work with. Having sampled some values the heuristic oracle also verifies some of the values. At last, the true oracle generates all results independently and can therefore verify all of them.

Memon, Banerjee and Nagarajan[28] examined the effectiveness of different test oracles in testing event driven GUIs. They define different oracles based on how much information the oracle gets from the application under test (AUT). This ranges from only the information related to the single widget associated with the event tested to complete information about all widgets in all windows associated with the application under test. They find that increasing the amount of information presented to the oracle increases its ability to detect bugs but that it also increases the storage and processing capacity needed to run the tests.

Most of the work done regarding test oracles use formal languages to describe different program

states and possible transition points. Peters and Parnas[29] describe a test oracle based on system documentation written in a formal language.

The most common way to create GUI oracles is to use a capture and replay method[27] although formal specifications are also used[28].

None of the approaches to creating test oracles that we have found use images. Most get information about widgets, windows and values from the AUT. The tests do not focus on rendering of the application, they test GUI interaction. For web applications, this may be automated using the tools described in section 2.2.1.

2.2.4 Use of image analysis techniques on webpages

We have not managed to find anyone who has tried to use CV techniques to find errors in webpages, but we have found some work on segmenting webpages based on visual cues as well as some other interesting approaches. There is a large body of literature related to document image analysis. It is beyond the scope of this thesis to review these methods in detail.

Most of the work done on image analysis has been focused on semantic analysis of a web page. VIPS(Vision-based Page Segmentation Algorithm)[13] is a much cited work from 2003. VIPS works by extracting information from the browsers Document Object Model(DOM) about where different nodes are placed and uses a complex heuristic to define which nodes are grouped together. VIPS does not do any image processing to do this work. Later work uses the VIPS application to organize images by looking for links between images and the content close to the images[12].

Roast[31] created GIST, a system for "automatically collecting screen images of inter-connected web pages and visualizing the collection so that pages with distinct visual qualities could easily be identified"([31], page 4). The purpose of the system is to help designers and developers of webpages. GIST uses average values for different color parameters such as hue, saturation and brightness from different regions of the image as inputs to a machine learning algorithm (which is not specified) that then groups images together.

Song[32] looks at web page complexity by comparing complexity measured by human participants to an algorithm for defining common blocks in a webpage. The algorithm uses morphological closings

and linefinding algorithms to find different blocks in the page.

Baluja [6] uses a method close to the one used by VIPS to decide which regions in a webpage to group together when displaying the page on a small screen device like a PDA or a mobile phone.

Cao, Luo and Mao[23] uses actual screenshots. The images are acquired by using browser specific APIs. The authors use an iterated dividing and shrinking strategy on a binary image to segment the page. The authors manage to get quite good segmentation of letters but do not look further into what they may find in the image. They also point to Fu, Wenyin and Deng[17] who generate signatures of images generated from webpages. The signatures are used to detect web pages that are used in phishing attacks.

Karatzas and Antonacopoulos[26] look at text contained in images embedded in webpages. This is a common method to get more advanced texteffects - for example a gradient color. They try to extract the text information out of these images because it may contain information of interest to search engines. This is done by converting the image to the hue, saturation and lightness (HLS) color space. They do not try to apply OCR techniques to the resulting image.

None of the approaches above look at differences between browsers or try to define formats for saving information about the structure. They use the segmentation techniques to extract special attributes from the webpages or to try to assign different semantical values to different parts of the text.

Chapter 3

System overview and design

How do you test an image of a webpage?

As stated in the introduction, part of this project has been to evaluate different approaches to how one can test web page rendering by looking at screen dumps of browsers rendering these pages.

The original idea was to compare a screen dump of the webpage with a known good original - for example a draft made by a graphical artist on how the page should look or an earlier screenshot of the site. It is thought that this image tells more about how the site should look than it is possible to contain within normal tests, because it is too time consuming to try to formulate each requirement as a test.

The trivial possibility is the pixel by pixel comparison between the two image maybe constrained to a Region of Interest(RoI). That this method has some shortcomings, as it leaves very little room for variations in the image - for example due to changing content or images from different browsers.

An extension is to check some characteristics within the RoI, for example the main color for an area. It may also be possible to define that the RoI should contain text with a special size or if it should contain an image.

This thesis thinks that variations can be tracked better if we try to abstract out a model of the web page from the image. The model we define is a hierarchy of regions where different regions get different characteristics, for example if we think that this region is an image or a block of text.

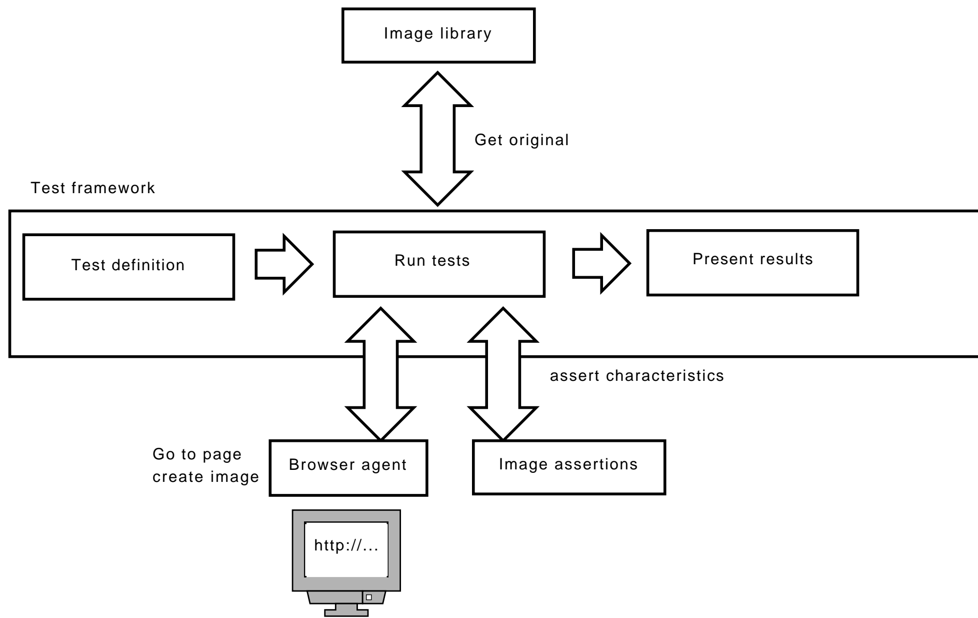


Figure 3.1: System overview. The test runner creates the test instance that runs the test. The test instance fetches an image and compares it to the original. If errors are found they are reported back to the test runner.

It will then be possible to track changing regions and try to find out what the exact differences are. This makes it possible to define region by region allowed variations.

This chapter gives an overview of the system created as well as a description of the APIs and fileformats exposed to a developer using the system.

3.1 A brief overview of the system components

This section contains a brief description of the different components in the system.

The basic inputs to the test are an URL of a page we want to test, the operating system and the browser we want to use, as well as an optional image model generated initially. Figure 3.1 shows the different components.

To organize the different tests we want to run in different environments we need a framework to run

the tests, collect results and present the results. Large parts of this framework come from standard off the shelf components. The parts of the system which have been developed specifically for thesis have been named Sysifos.

3.1.1 Sysifos: An image assertion framework

The different image assertions lie at the heart of this project and are therefore mentioned first. An image assert is a method you may use to see if an image meets an expectation, for example if an area of the image has changed when comparing the image to another image.

The different ways this is done is explained in section 3.4. The main point here is that if the assertion fails, it emits an AssertionException. This is the same way that assertions in both TestNG and Junit work.

Although the assertions are the main product, the name Sysifos is often used about the complete system as well.

3.1.2 Test framework: JUnit or TestNG

A test framework collects the tests defined, run them and provide a summary of the test results. JUnit [1] is the original Java implementation by Kent Beck of a unit test framework. Although JUnit provides the core services of a test framework, it is very focused on unit testing and does not provide for more complex test scenarios. To supplement this, TestNG[2] was created by Cedric Beust. In "Next Generation Java Testing" [9] he outlines the main reasons for creating TestNG, mainly the ability to group tests, define dependencies and configure test cases from different data providers.

TestNG is the preferred testrunner for running Sysifos although JUnit may also be used.

A test definition is just a method in a normal Java class. There are two ways a method may fail, either it emits an exception or it emits an AssertionException. The difference is that the exception comes from a program failure while an assertionException is the result of a test check that has failed. If no

[1] http://www.junit.org
[2] http://testng.org

```
1   @Test
2   public void testNovap() throws Exception {
3     String url = "http://www.uio.no/";
4     test = library.getTest("uio-frontpage");
5     factory.requestImageFromUrl(url, "*firefox", "osx",4444, test);
6         Selenium selenium = new DefaultSelenium(host, port, browser, url);
7     selenium.start();
8     selenium.initScreenshots();
9     selenium.open(url);
10    test.setImage(selenium.getFullScreenshot("png"));
11    selenium.stop();
12    Roi roi = new Roi(0,0, 200, 300);
13    Asserts.assertNoChanges(test.getImage(), test.getOrig(),roi);
14  }
```

Figure 3.2: A simple example of a test making sure that the top of www.uio.no doesn't change.

exceptions are emitted, the test is considered a success.

TestNG binds together the other components that we are interested in. Figure 3.2 shows what a simple test may look like. This is a very simple test. It starts Selenium, fetches an image of the url and compares a part of the image to the original image. TestNG's job is mainly to run the tests and keep tally of the results. TestNG has not been modified during this thesis.

3.1.3 Browser agent: Selenium Remote Control (Selenium RC)

Selenium RC[3] is a test tool that provides an API for remotely controlling a browser. Figure 3.3 shows how the testrunner connects to different agents, each running Selenium RC.

Selenium RC comes in two parts, a server that launches the browser and runs a set of commands and a client API for controlling the server. The commands are injected into the browser by launching all pages within a javascript session that runs within the browser.

Selenium has client APIs for Java, Python, PHP, Perl and Ruby, but Sysifos only works with Java (and probably Groovy).

Selenium RC has rudimentary support for taking screenshots through the `captureScreenshot`

[3]http://selenium-rc.openqa.org

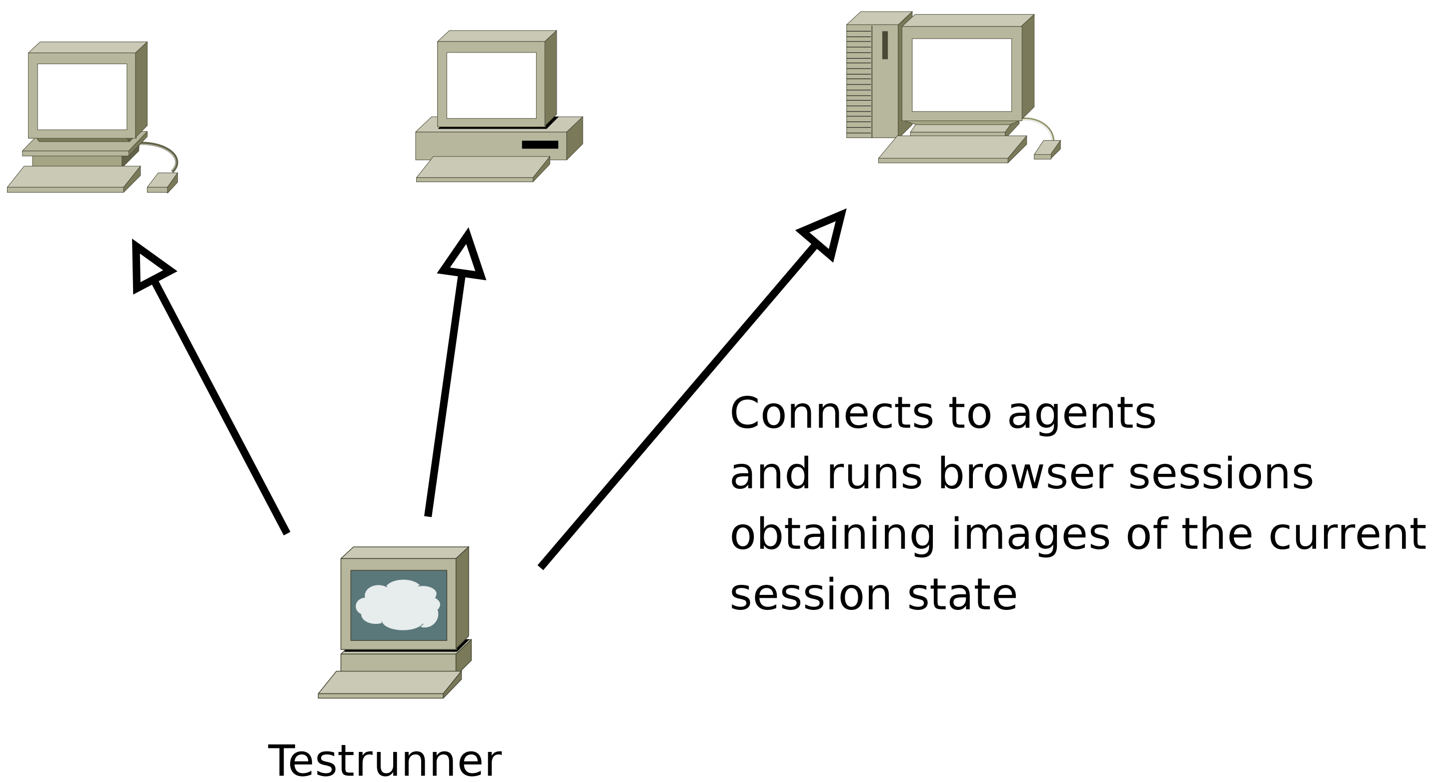


Figure 3.3: Agent overview.

command. This command will save a screenshot of the browser to a file on the machine running the agent - the image is not obtainable through the Selenium RC API. Robert Zimmerman has extended this command with a patch that sends the image back to the user[4].

These commands both provide images that contain the whole screen, including operating system buttons, scrollbars from the browser etc. To make the images usable for testing, they must only be of the webpage - not all the chrome surrounding the page. Also the image should contain the non-visible parts of the page, those you find only when scrolling down.

3.1.4 Image processing library: ImageJ

Although Java has the JAI(Java Advanced Imaging) libraries, ImageJ[4] was chosen as the image manipulation library to use. ImageJ is a Public Domain image manipulation package originally written by Wayne Rasband at the Research Services Branch, National Institute of Mental Health, Bethesda, Maryland, USA.

ImageJ comes with many different filters and functions, but most of them are only useful from within the ImageJ GUI application. However the API is very workable and there is an excellent introduction to it in the book Digital Image Processing by Wilhelm Burger and Mark J. Burge[11]. Also there is a load of plugins that implement different image operations that may be modified to work in a stand alone context.

3.1.5 Result visualization

Both TestNG or JUnit can create testreports in different formats. TestNG has support for adding extra information into it's XML test report. This report forms the basis for the other report types and thus makes it possible to include links to images in HTML reports for example. JUnit does not provide opportunities for manipulating the test reports.

As detailed further out in this chapter, the more advanced tests use SVG to display both the image-model and any errors identified.

[4]http://jira.openqa.org/browse/SRC-395

```
1 ImageLibrary lib = new ImageLibrary("/path/to/lib");
2 /* get a ITestImage object related to browser and OS. */
3 TestImage images = lib.getTest('TestId', 'windows', 'firefox');
4 /* get the original */
5 BufferedImage original = images.getOrig();
6 /* .. then you run a test and get another image, and save a new version */
7 images.setImage(newImage);
```

Figure 3.4: An example of how the image storage API is used.

3.2 Image storage container

To be able to go back in time or to look at what caused a test to fail or just to document the results there needs to be a way to store the test outputs - i.e. the images.

Neither TestNG nor JUnit contains functionality to link large blobs to testresults. A simple API to store images was implemented. The API relates each image to operating system, browser and (optionally) test run.

The issue of relating a test to an image is more complex than may be thought because the name of the test or the url may change. Also you want to be able to compare images from different browsers or reuse the model from one test in another.

To facilitate this the API has two basic classes, ImageLibrary, a container, and TestImage containing the images and models tied to a specific id. ImageLibrary also has the option to generate test identificators from URLs. Figure 3.4 shows how the library is used. Each time a new image is generated and setImage is called an image is saved into the image library. Depending on setup, the new version either overwrites an existing image or is saved as a new version.

Often you want to connect the test images to a special test run, but most test frameworks do not contain a concept of testruns as something you want to go back to. A continuous build system must then be deployed around the test runner to provide this functionality. Continuous build systems are used to automate testruns so they run at a certain time or based on triggers like every time code has been changed.

3.3 Extending Selenium RC to get complete images of the webpage.

Selenium RC provides a good framework for instructing a browser to navigate pages, click on links and provide information about the page it is on. What it doesn't provide is an image of the complete page. A patch for this was created. The patch adds three new commands to the Selenium RC API: initScreenshots, setScreenshotRoi and getFullscreenshot.

The function signatures are:

```
1 public void initScreenshots() throws Exception;
2 public BufferedImage initScreenshots(String returnFormat) throws Exception;
3 public String setScreenshotRoi(int x, int y, int width, int height);
4 public BufferedImage getFullScreenshot(String fileType) throws Exception;
```

Initscreenshots must be called before you try to get a full screenshot. It opens up a browser session but paints the whole page in one color using javascript. After this, it takes a screenshot. If the method was supplied with a format argument, it will then send back the image to the client in the format the client asked for. The client may then process the image to define the area that contains the browser viewport. This may be sent back to the Selenium RC server using setScreenshotRoi().

If the client is not interested in computing the ROI itself, then Selenium RC will try to decide the coordinates of the browser viewport itself. It will do this by creating histograms of the main color in the image and filtering them to find the image edges.

The reason for this dual approach was to avoid adding an extra dependency to Selenium RC. The client side code for finding the ROI uses ImageJ to compute which color has the highest frequency in the image. This is a more robust approach than the one taken in the Selenium RC patch where this color is hardcoded. Modifying the Selenium RC patch to use a similar approach would not completely negate the need for clients to set ROI as you often want to reuse the ROI to reduce the time it takes to run tests.

Errors will occur if the browser window is resized during the subsequent browsing. Then initScreenshots must be called again.

To secure consistent results, the page that is created for these images is created with scrollbars. This

reduces the width of the captured images with around 20 pixels for webpages that do not use scrollbars. This may also generate problems with pages using layouts with relative widths so they fill 100% of the screenwidth.

Then the outer areas of the images without scrollbars will not be included. Very few large sites are designed this way because it is hard to position content within the page (one major exception is Slashdot). A solution to this problem is to use the calibration algorithm only to select the upper coordinates for the viewport and then use javascript to extract the viewport width and height. This has not been tested.

3.3.1 Merging algorithm

After the agent has been calibrated it is possible to generate an image of the webpage by calling `getfullScreenshot()`.

`getfullScreenshot()` captures a screenshot of the webpage and then crops it using the information gained in the initialization phase. It then scrolls the page down and captures another image that overlaps the first one. This is done until the complete page has been captured. The the different screenshots are merged to one complete image.

To merge the images into one consistent image a merging algorithm was used that compares a 15x100 pixel box on both images to determine if it fits. A pixel by pixel comparison between each image is done and a vote given for each match that is found. Then the merge coordinates gets computed using the line that has the highest score.

This works well for sites with fairly large variations on the left side of the site, but may fail if the site for some reason does not have changing content or happens to have an animation inside the area used for comparison. To protect against the first effect, the box that is used to compare the two images is placed in the middle of the image where it is expected that the highest variation in content will be.

A better strategy may be to compute the amount of entropy in the area and move or extend the area if the entropy is too low.

The current implementation does not scroll sideways. That is because this would add a lot of complexity to the implementation, without providing higher value because most users only scroll down.

Horizontal scrolling has been one of the big taboos of webdesign for some years, and got a third place in Jacop Nielsen's "Top Ten Web-Design Mistakes of 2002"[5]. As a result, most larger sites do not design for horizontal scrolling. Therefore supporting wide sites isn't as important as supporting long sites.

It should be noted that for very modern browsers it might be possible to use only javascript to find the coordinates of the viewport. This approach would be faster than having a separate calibration step, but would not work with Internet Explorer. Only using javascript has not been tested.

Section 5.2 describes a test done of this part of Sysifos. The main experience is that trying to capture screenshots only using javascript and Java's robot class is harder and more error prone than creating browser specific capture commands that provide complete images.

3.3.2 Selenium startup

Selenium must be started with the -multiWindow commandline option so that the javascript command window is in a separate window from the one used to get the screenshots.

It is also important to note that the screensaver on the Selenium RC server must be turned off. Also you must be sure that the screen is actually showing something and has not been turned off to save electricity. The best way to do this on Windows is to maintain a RDP (Rdesktop) session to the computer.

3.3.3 Problems with Selenium RC

During the development of the system, we experienced a number of ways that the Selenium RC server may hang that it is worth to comment on.

One problem we had was that the current merge method demands that the client has more than 300 MB of RAM. This becomes a constraint on the types of client you can check (say, a mobile phone or a low powered machine). This should not be a large problem as Selenium does not support smaller

[5]Top ten Web design mistakes of 2002: `http://www.useit.com/alertbox/20021223.html`, Nielsen also reminded of the problem in 2005: `http://www.useit.com/alertbox/20050711.html`

clients either.

Selenium uses HTTP for communication between the server and the client. If the client exits without sending a stop command to the server, the server will continue to keep the browser window open and wait for more commands. This can cause the client to have a lot of browserwindows open, something that takes resources and may cause other tests to fail.

A bigger problem is that modern operating systems and browsers like to ask users questions using popups in front of the current window. This happened with mesages from Windows about different status changes on the test computer as well as with an error message from Opera signaling that it wanted the user to set up email.

Although it should be possible to get rid of these problems by careful setup of the browser agent and its underlying operating system, the end result is that the test environment requires more handholding than is usual to have the tests running smoothly.

The main weakness with the system is that it uses screenshots to capture the images instead of using browser specific APIs. The result is that the image capture process is slow and prone to errors. Future improvements of this part of the system should focus on creating browser specific capture methods.

3.3.4 Supported browsers and operating systems

Firefox 2, Opera 9 and IE6 have been tested with good results. IE7 has proven to be a big problem due to problems with the scrollTop and scrollHeight attributes. The result is that it is not possible to determine if a page has been scrolled, thus you do not know when to stop to take screenshots.

Safari does have support for these properties so this should not be a problem with that browser[6] but it has not been tested.

Selenium RC supports the major browsers for Windows, Mac Os X and Linux. The patched version has only been tested it on Linux and Windows.

---

[6] `http://www.quirksmode.org/dom/w3c_cssom.html` contains a complete table of browser support for scrollTop and related properties.

3.4 API description

This chapter describes the API of the libraries developed as part of this thesis. It also explains how the simpler ones have been implemented. Chapter 4 details the implementation of the advanced parts, those that build an image model.

The general workflow of setting up and running tests using Sysifos is shown in figure 3.5. When setting up the test an initial image is set (either from a designers sketch or from the webpage) and the initial model is generated. The model is then used when running the test to find errors.

3.4.1 Test acquisition

To acquire a test, you need to have both a Selenium client and server version that is patched to provide complete images of a webpage. The image acquisition is a two step process. First you calibrate the browser by calling initScreenshots(). Then you ask for a screenshot with getFullScreenshot(). You then have an image to use for testing.

The API is designed to run so that each browser/operating system combination is run as a separate test case. This creates a possibility for confusion as the the test unit in the image library may contain more than one image coming from a set of separate tests. Defining a test unit containing separate images was done to make it possible to do cross browser tests.

The best way to test multiple browsers and operating systems is to use TestNG's dataprovider functionality. An example of this is found in the appendix detailing the code used in one of the experiments.

3.4.2 How do you assert image values?

There are roughly three approaches to inspecting images that have been implemented:

- Single image checks
- Pixel by pixel comparisons with the original

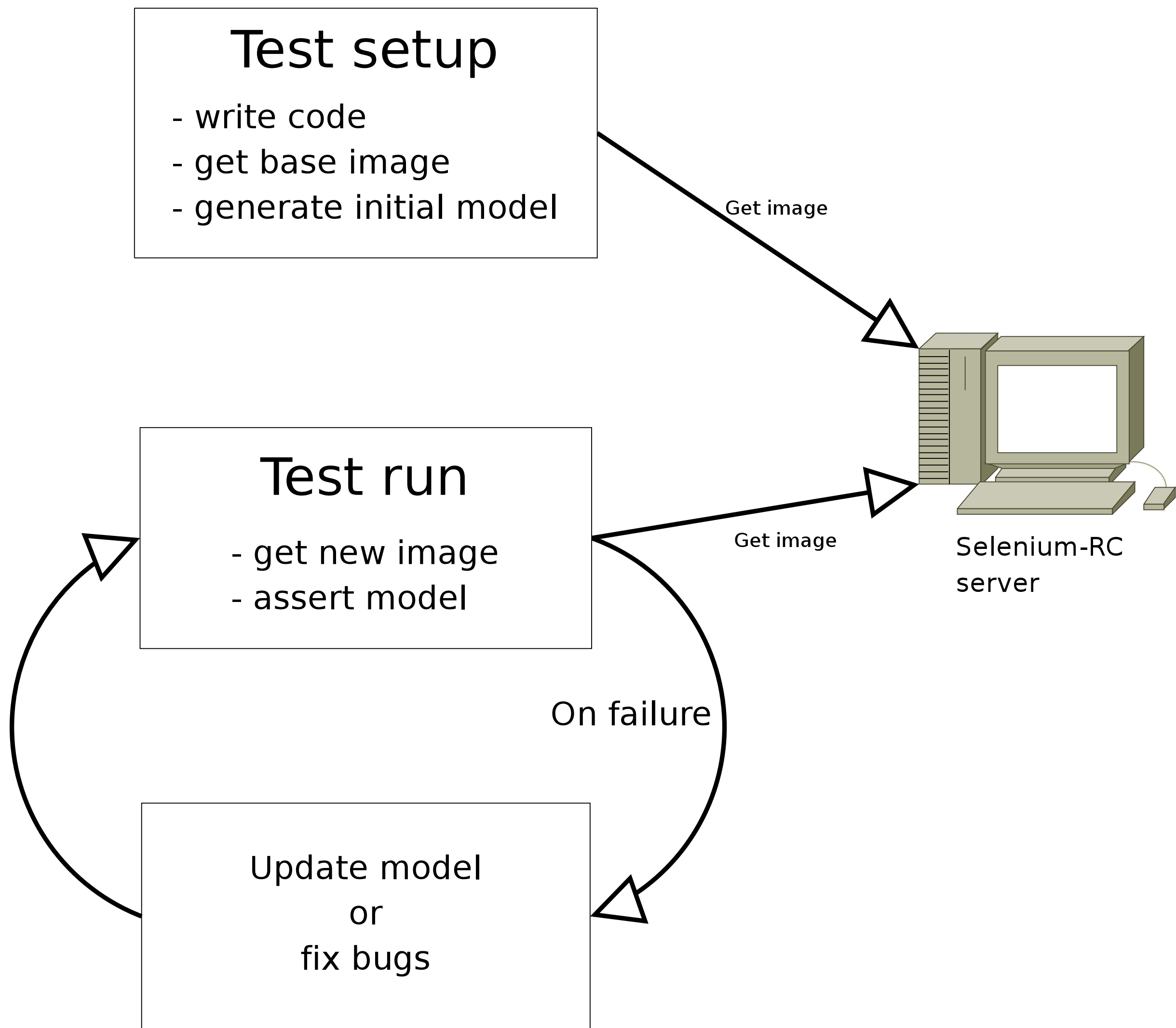


Figure 3.5: Standard test workflow. Apart from writing code you also need to get the initial image and generate the imagemodel.

```
1 void assertMainColor(ImagePlus img, Roi roi, String color)
2 void assertMainColors(ImagePlus img, Roi roi, String[] colors)
3 void assertMainColor(ImagePlus img, Roi roi, int color)
4 void assertImageSizeAtLeast(ImagePlus img, int width, int height)
```

Figure 3.6: Signatures for the single image assert methods.

- Image model checks

3.4.3 Singel image asserts

Single image asserts try to extract simple characteristics from the image and verify their values. The characteristics can be applied globally or to a defined region of interest(ROI). We tested two variants of this.

assertImageSizeAtLeast This method just asserts that an image is of a certain size. This may quickly catch bugs like halted rendering or errors in the acquisition system. The method only inspects the dimensions of the image at test to be sure that they are not too small. Figure 3.4.3 shows the signature for the method.

assertMainColor This method may be used to make sure that the main color of an area is of a certain value. Note that due to the instability of color rendering (see section 4.1) a 5% variation in each color channel(RGB) is allowed. This may for example be used to verify that the background has not been changed.

This method computes the frequency of all colors inside the region of interest and then returns true if the color asked for is close to the main color found.

This method may be used to catch changes in the background of a webpage or to make sure that block boundaries are honored. The test may be invoked with different parameters figure 3.4.3 shows the different method signatures.

The main difference between the signatures is if you want to define the color as a string or int and if you want to check a RoI or the complete image.

```
1 ImageModel imageModel = new ImageModel(test.getImage(), test.getBase());
2 Region root = imageModel.build(true);
3 imageModel.saveModel(test.getModelPath(true), root);
```

Figure 3.7: This listing shows how a new model is created and saved.

3.4.4 Comparisons with the original

The second approach is to compare the two images and look for differences. This method is very unstable when comparing images created from two different operating systems where the rendering is different or between browsers and may then report a lot of false errors. The function signature is:

```
1 void assertNoChanges(ImagePlus orig, ImagePlus new, ROI roi)
```

The ROI value may be null. The assert does a simple pixel by pixel comparison and reports an error if two pixels differ. It may be useful to ensure that a logo is correctly placed or to ensure that boundaries between blocks are not crossed.

3.4.5 Build a model of the webpage

The other approach was to try to create an abstract representation, a model, of the page. This may then be used to assert that the page model has not broken down or been changed. Chapter 4 describes the details of this method.

The API for models is a bit more involved as you might want to specify a separate model for different tests and also that it needs to accommodate for saving models. To create a model you create an ImageModel object and ask it to build a model for a specific image. Figure 3.4.5 shows how a new model for an image is created and saved. The image is obtained from the image storage library and the model is saved in the same directory as the image is stored.

Apart from the ImagePlus object, the imageModel object also takes a string to where you want any byproducts of the process saved. This is where the imagemodel will store any files made while generating the model. On line 3 the image storage library provides the path to where the model is stored. The testimage class contains support for storing the model XML files together with the generated images.

After storing the model (and maybe modifying it) you can use it to verify another model by calling the assertModel() function. The function signature looks like this:

```
void assertModel(VaringRegion orig, VaringRegion result,
        ImagePlus img, String savePath);
```

Just as with the imageModel, assertModel requires that the user provides a savePath where the resulting visualizations of the errors found may be placed. How the error visualizations are created is described in the next chapter, but to keep the readers interest, we have included an image from the image model test described in chapter 5. Figure 3.8 shows the webpage with different errors marked as red and green lines in the page. This is done by using SVG to paint on top of the image specified when calling assertModel.

The green lines mark where the original model defined the regions to be while the red ones mark the new place for the same lines.

Most test tools only report the first error that appears with the assumption that a good test only tests one thing and also so that it can give a precise line number for where the error occurred. Sysifos differs in this regard that it may report more than one error from one test.

The reasoning behind this is that an image test tests more than one item and is a fairly slow test to run. The user therefore wants to know of all errors found in the image rather than having to run the test multiple times to find each error.

The total number of errors found is emitted in the text of the AssertionError that is thrown by the assertModel function. The test framework will only report one failure - but the failures will be found on the image marking the errors.

3.5 Page model XML format

Figure 3.5 shows a minimal page model document for a completely empty page. The root element tag is imageModel and it contains one child, the root tag. The reasoning for this is that we might want to add metadata tags that should not be part of the node tree.

The `root` node may have one or more box tags as its children.

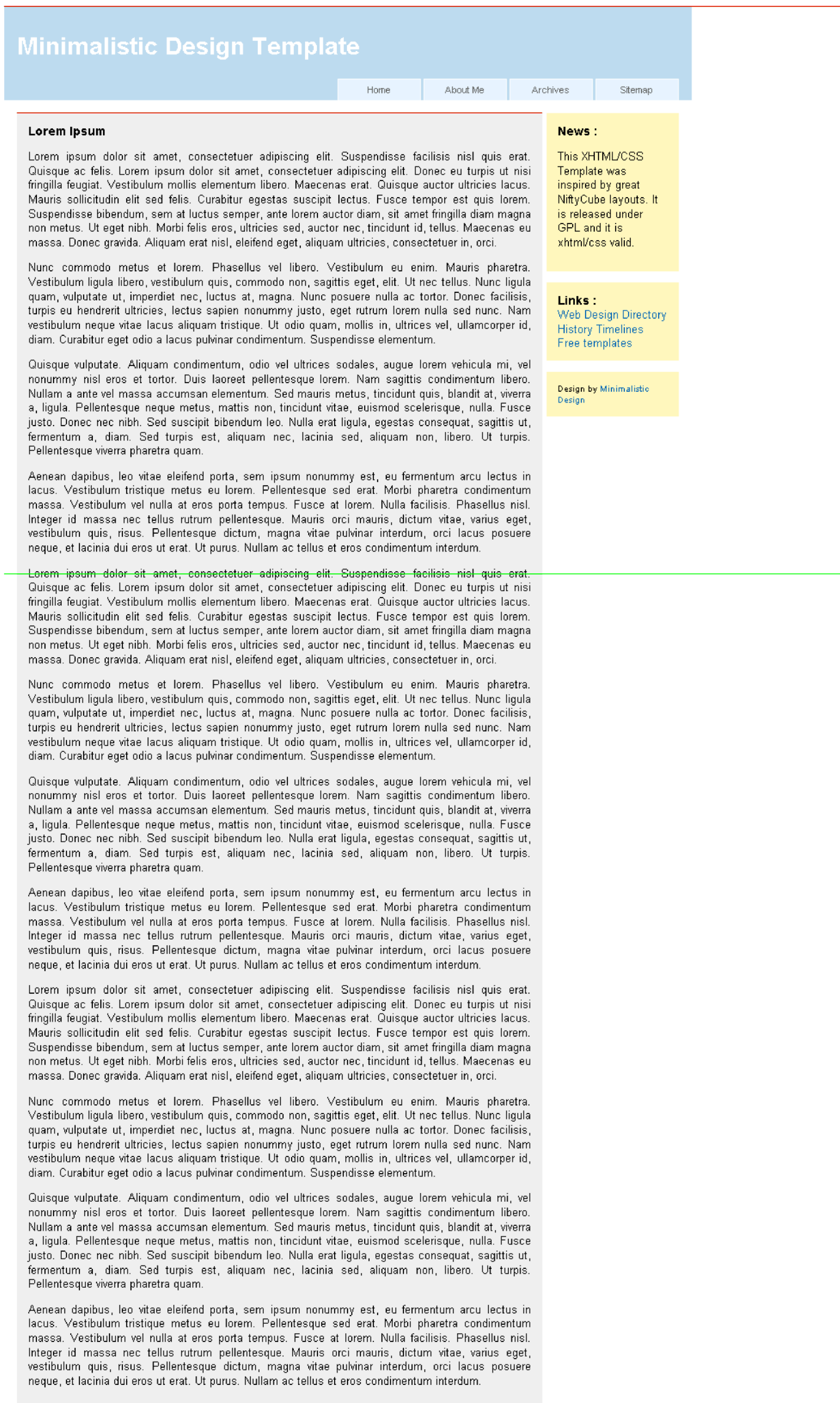


Figure 3.8: Sample output from assertModel showing which areas have changed. Red lines show where the blocks were found in the new image while green lines show where the corresponding blocks were in the original image. Brown lines are places where the two conflicting regions overlap.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <imageModel>
3   <root>
4   </root>
5 </imageModel>
```

Figure 3.9: An empty model representation.

The `box` node describes a block containing text and/or images. It has the the following attributes; height, width, area, color (if known) and the coordinates for the blocks top left corner. The same attributes are given for the other node types as well.

A `box` may contain 5 types of children, `image`, `text`, `variations`, `border` or `box`. The variations node is used for encoding different variations that apply to the current box node or its children. Other nodes may also contain the variations tag, but it is only checked when inspecting a `box` or `image` node.

The possible variations that may be encoded into a block or image region are:

- topx
- minx
- topy
- miny
- maxWidth
- minWidth
- maxHeight
- minHeight
- minNumText*
- minNumImages*
- minTextWidth*
- maxTextWidth*
- maxChildren*
- minChildren*

Those marked with an “*” may only be used in box tags.

A `border` node defines the border that the box may have. `text` describes an area inside the box that contains text. The last type of node is `image`. It describes a picture found inside the web page.

Section 5.3 gives an example of how the page model file is modified so a web page may evolve.

Chapter 4

Creating a high level model of an image of a webpage

This chapter explains the algorithm that is used to create a model of the web page. It also describes how this model is represented and how two models are compared.

To be able to know how these regions may be found and defined as well as the reasoning behind the algorithm we describe later, a closer look at the shapes, colors and other characteristics of the image is in order.

4.1 What to expect in a screen dump of a web page

To help the reader, let's start with the top of the Norwegian website www.siste.no, a newssite created by the newsagency ANB. The image is shown in figure 4.1.

The image of the webpage can be characterized by the following items:

- Image size and type
- Its origin with regard to operating system and browser
- Which colors are present

Figure 4.1: The top of www.siste.no

- What shapes to expect
- Different types of delimitiers between objects
- How the background is used

The list isn't comprehensive, but it tries to point out items that are important in the later discussion.

Image size

The width of the image is defined by the screensize of the computer where the image is taken. Usually this means that the image width is between 800 and 1900 pixels.

Extremes may happen if the system is used to test mobile applications or dual screen systems, but this isn't as interesting as most sites adapt their layout to the most normal screensizes.

We checked the screen size distribution for one of the larger producers of online newspapers in Norway, API, and found that for February 2008 they had the following distribution of screensizes:

$Screensize$	%
$1024x768$	$31,9$
$1280x800$	$22,0$
$1280x1024$	$18,50$
$1440x900$	$5,8$
$1680x1050$	$4,0$
$1152x864$	$2,8$
$1360x768$	$2,2$
$1400x1050$	$2,0$
$1280x768$	$1,5$
$1920x1200$	$1,4$

Usually larger sites target screens that are either 800 or 1024 pixels wide as they reach around 98% of the audience. Screenwith is increasing so this is a moving target.

When it comes to the height of the pages, this depends a lot on the type of page. The test images used later in this chapter differ between 910 and 6275 pixels.

Operating system and browser

The patched version of Selenium RC was mainly tested using Windows XP running on a VmWare image running either Internet Explorer or Firefox. However some effort was spent testing the sensors using other operating systems and browsers.

On Feb. 16th 2008 the system was tested on Mac OS X. One of the problems on the platform proved to be that the window was not placed in the foreground when the browser is started and also that java.Robot could not produce an image of just the browser, but had to make an image of the whole window.

Later the application was also tested on Linux with better results, but it was decided that this thesis would not focus on making the sensor work flawlessly on different operating systems.

Different operating systems provide different APIs to the browsers for displaying the webpages. The differences both in APIs and implementations of graphic libraries may result in different color rendering. Another effect is that in browser buttons and controls may differ between operating systems and also browsers. Browsers on Apple OS X show the buttons differently than Microsoft Windows.

Color settings

While working on asserting the main color (see section 3.4.3) of an area, it became apparent that colors were not represented by the same values in the screenshot as they were in CSS, different images or in the markup. The reason is that different combinations of hardware and operating systems end up showing colors differently - not only on the physical screen, but also in the screen dumps.

There has been some work to try to standardize color display across different systems. Microsoft and HP did some early work[5] on providing color profiles with images and making sure that colors were shown equally on different screens. This resulted in the color profile SRGB[3]. SRGB is a reduced RGB colorspace that was defined to work on commodity hardware around the time the standard was created.

This work partly lapsed with only Internet Explorer 4.0 and newer versions of Safari implementing support for SRGB[16]. SRGB is mainly used to ensure that the colors in pictures in the webpage are shown correctly.

Tests done while working on this thesis show that IE7, Opera and Firefox seems to render colors the same way when running on the same hardware / operating system platform but that you cannot expect the same colors coming out as where specified in CSS or in HTML markup.

Object shapes found in typical images

It is hard to define complex shapes like circles or triangels in CSS or HTML. As a result, there is little use of text flowing around circles or other uses of masks or complex shapes in the webpages relevant for this thesis. The logical areas in the page may be described as a rectangle.

The main exception to this rule is pictures and images included into the page. Images come in roughtly two forms, computer generated ones and images made by cameras - the distinction being that the images made by cameras does not contain contigous areas of one color.

With images it is possible to create areas that are oddly shaped by using the same background color as the area.

Text is the other major exception. Text has the special characteristic that it is usualy just one color with butmany objects in an area that logicaly belong together.

Borders, margins and space

Equally interesting as to what an object is, is the amount of space around the object and any borders that surrounds it. Spacing and borders often define an objects relation to other objects. For example if an image follows with an article or if it is just a general part of the sites layout.

Often space may be a bit hard to define as it may belong to different objects. Figure 4.2 shows an example where there is no border around the menu and a listing on a webpage. Which should “own” the whitespace is not given.

Figure 4.3: Do the images belong to each other or to the text below? The spacing differs with one pixel.

It may also be hard to define which objects belong together, something that is easy a human observer. The next figure, 4.3, shows two images with text below. The spacing between the images is 8 pixels while the spacing between text and image is 9 pixels. However, the logical grouping is text to image, not the images as a separate object from the text.

Page background

Most of what is written above expects a fairly simple background, i.e. not a complex one containing an image. This is true for most high volume websites focusing on content, but more arty sites may use more complex backgrounds. This thesis will

Figure 4.2: Is it the menu to the left - or the article listing to the right that “owns” the whitespace between the two?

not try to adapt to these sites.

4.2 Process overview

Figure 4.4 shows the different steps that lead up to creating the complete hierarchy of regions.

Three webpages were used during the development process for testing and tuning of the different steps. They were chosen both because the sites were known to the author and because they represent variations of page layout and different ways to show structure to the reader. Figure 4.5 shows the URL and the size of the images generated from the sites as well as the date the images were collected.

The images are too large to display completely in this report, but scaled versions are shown in figure 4.6

4.3 Finding pictures in the image

In this section, the term image may be used in two ways. The main image is the screen dump of the webpage that we are analyzing. The webpage may contain some pictures. To separate these images from the main screen dump, we will call them pictures.

We want to identify these pictures separately and remove them from further segmentation. This makes the rest of the job a lot easier.

A picture in a webpage is defined by an area that both contains more colors or greyvalues than most of the image and also where the color values vary a lot more than in the rest of the area. This is because the picture is generated from the analog world where colors or greylevels vary. Figure 4.7 shows the image we used for testing as well as an image where the areas that ideally should be defined as pictures have been marked.

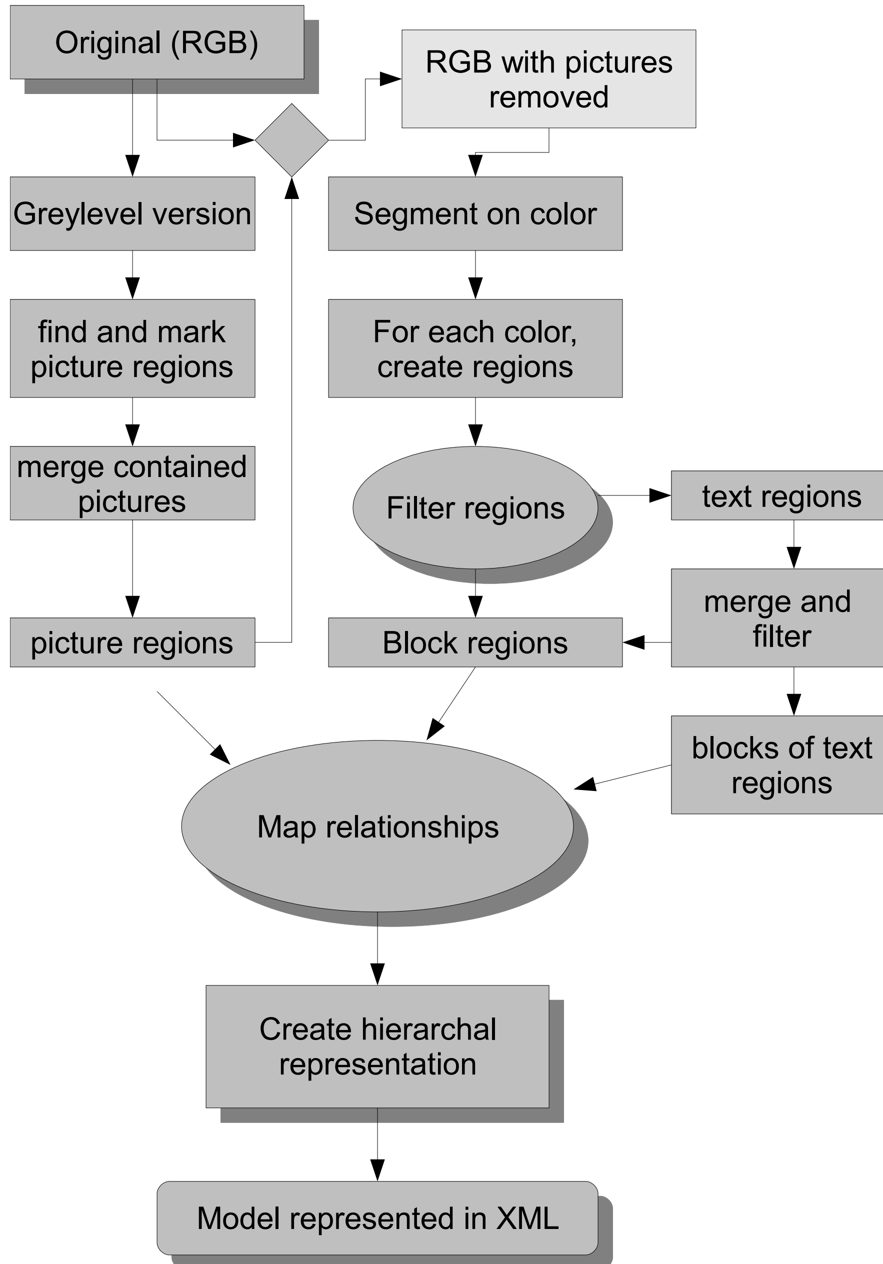


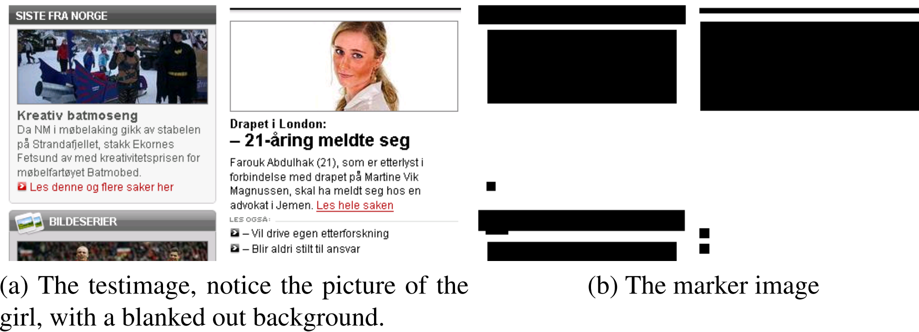
Figure 4.4: Overview of the segmentation and region relation algorithm used for generating the page model.

Url	Date acquired	Size
http://www.novap.no	26.02.08	1000 x 888
http://www.siste.no	23.03.08	983 x 6275
http://www.agderposten.no	26.02.08	983 x 1338

Figure 4.5: The URLs the test images were captured from.

(a) www.novap.no (b) www.agderposten.no (c) www.siste.no

Figure 4.6: The different images used while developing the image segmentation and labeling algorithm. Note that only a third of the image is shown for the largest one.



(a) The testimage, notice the picture of the girl, with a blanked out background.

(b) The marker image

Figure 4.7: Testimage and the manually made marker image

Note that the marker image marks all the gradients as pictures. Gradients should be handled separately.

Pictures may produce artifacts in later region finding algorithms because parts of the object show up when segmenting the different colors. Also, they are regions in their own right and should be identified.

The first thing we did was convert the original 24 bit RGB image to an 8 bit grayscale image. This was done by taking the average of the RGB values. Then it must be determined which pixels belong to a picture and which do not. This is done by examining the local neighborhoods of the image.

The first approach to examining the area around each pixel was to divide the image into a grid. For each block in the grid, we computed a histogram over the number of greylevels in each row and column. To decide if a pixel was in the foreground (i.e. part of a picture) or in the background, we applied a threshold to the histogram.

Figure 4.2 shows how this works for one block on the grid. The first square (marked a)) shows the pixel values within that grid. The values marked “x” and “y“ on the sides are the histogram values. They are generated by counting *the number* of greylevels in the respective row or column.

After the histograms have been computed, every pixel is linked to two values, one for each histogram. For example the top right pixel of the example grid has the associated values $(x = 3, y = 1)$ from the respective histograms.

We then apply thresholds to the histogram to decide if a pixel is part of an picture or not. The threshold

	a) Original				b) Sum threshold				c) Separate thresholds			
y:	3	2	1	1								
x:												
3	3	4	3	2	1	1	1	1	1	1	0	0
3	4	4	3	2	1	1	1	1	1	1	0	0
4	5	4	3	2	1	1	1	1	1	1	0	0
3	1	3	3	2	1	1	1	1	1	1	0	0

Table 4.2: Example of applying the algorithm to an 4 x 4 window with a threshold of 2. To the right the result images with two different methods for applying the threshold.

5	1	2	4	9	3
5	4	3	5	8	5
5	5	**5**	9	7	6
5	3	5	5	6	7
1	5	2	5	4	8
3	5	7	8	1	9

Figure 4.8: The pixels within the plus are the pixels we are segmenting. The bold pixel is the center of the plus-shaped window.

defines how many greylevels must be within the window of a pixel if it should be defined as part of a picture.

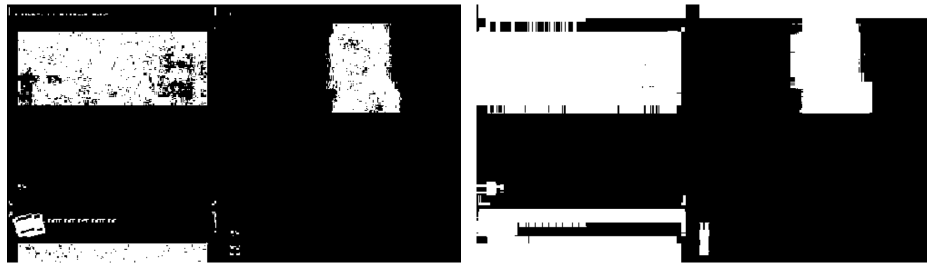
Figure 4.2 b) and c) show the results of applying the threshold in different ways. Figure 4.2 b) shows what happens if you apply the threshold to the average of the two histograms. Then all the pixels in the example image are marked as foreground. Figure 4.2 c) shows what happens if the threshold is applied separately to each of the two histograms.

The example above describes how this is done on grids. A weakness with grids is that they decide the faith of a pixel only based on the information within the grid. This seemed to work well until it was noticed that often one or more of the edges of a picture was mislabeled. This was because the edge was a member of a grid that had mainly been outside the window.

To counter this, a new version was written that used a plus-shaped moving window instead of a grid. Figure 4.8 shows how pixels are counted. The new approach also generates two numbers, either counting the number of value changes or the number of different greylevels. Using a moving window removed the problems generated by grid placement.

As noted above, the new version tried a different method to determine the amount of variation between pixels. Instead of counting the number of greylevels in the area, it counted the number of greylevel changes. The idea was to use a simple measure of entropy instead of counting greylevels. This proved to be worse because it picked up noise from letters and other elements that were only of one greylevel.

Therefore we went back to using the number of greylevels in the window instead of the number of changes in greylevel.



(a) Result image when using a gridsize of 3 and a threshold of 2

(b) Result when using a threshold of 3 and a grid size of 15.

Figure 4.9: Two images showing the results of different threshold/grid size combinations.

4.3.1 Choosing the right threshold and cross size

How can the best combination of threshold and cross size be selected? While still working with grids instead moving windows, we tested grid sizes as high as 100 pixels and as small as 3 with different thresholds. The general result was that too large grids made it possible to bridge gaps between two pictures and that small girds didn't find enough pixels.

Figure 4.9 shows the results of applying the algorithm to the testimage using different grid sizes. Notice that there are gaps even inside the larger picture area. Figure 4.9 b) also shows bridging effects between the picture and the headline above.

The picture of the girl on the right hand side has been manipulated before it was published so that the background has been removed. This is often done to reduce the size of the images shown. That is the reason why only the head and shoulders are marked as pictures.

After some testing we found that a window size of 10 and a threshold of 3 gives good results in detecting pictures in the test images we have used.

4.3.2 Gradients

The picture finding algorithm responds badly to gradients because they usually only vary in one direction (apart from text placed on top of the gradient). A separate gradient detection step would produce better results for websites using gradients. Gradients are very popular with the current crop

of Web 2.0 designs so this is a weakness.

4.3.3 Cleanup and labeling of the pictures

After generating a binary image where only pixels thought to belong to pictures are marked, the image is run through a morphological closing with a structuring element of 5x5 pixels. As noted above, the main picture detection step does not mark all pixels inside a picture. By applying a morphological closing, we merge close pixels into one region which simplyfies the labeling step. Figure 4.10 shows the example image after the closing.

Then the image is labeled using 8-connected labeling.

Having generated the new labels, the resulting regions are run through a stage where regions that are side by side get merged and then a stage where regions contained within other regions are removed.

Working with region relationships instead of pixels proved to be a very efficient method to segment out the images and it is might be possible to drop the morphological closing and only work on the labeled regions to merge the images. It is still an open question if this is faster.

It is the region bounding boxes that are used to define which pixels are pictures. This is because all pictures included in a webpage are rectangular.

To help the further discussion the image with the pictures removed will be called the pictureless image. There are two version of this image, one where all the pixels defined to be within pictures have been set to 0 (i.e. black) and a mask image where all the pixels within the pictures have been set to 0xff (it's a 8-bit image). The last image is used when segmenting black from the image. Figure 4.11 shows parts of the pictureless image for one of the test images. Please note that the pictureless image is an RGB image.

4.4 Segmenting on color

Note that this section works on the pictureless image, which is a RGB image. We are now on the right side of figure 4.4.

Figure 4.10: The segmented image before labeling the different picture regions. The white pixels belong to pictures.

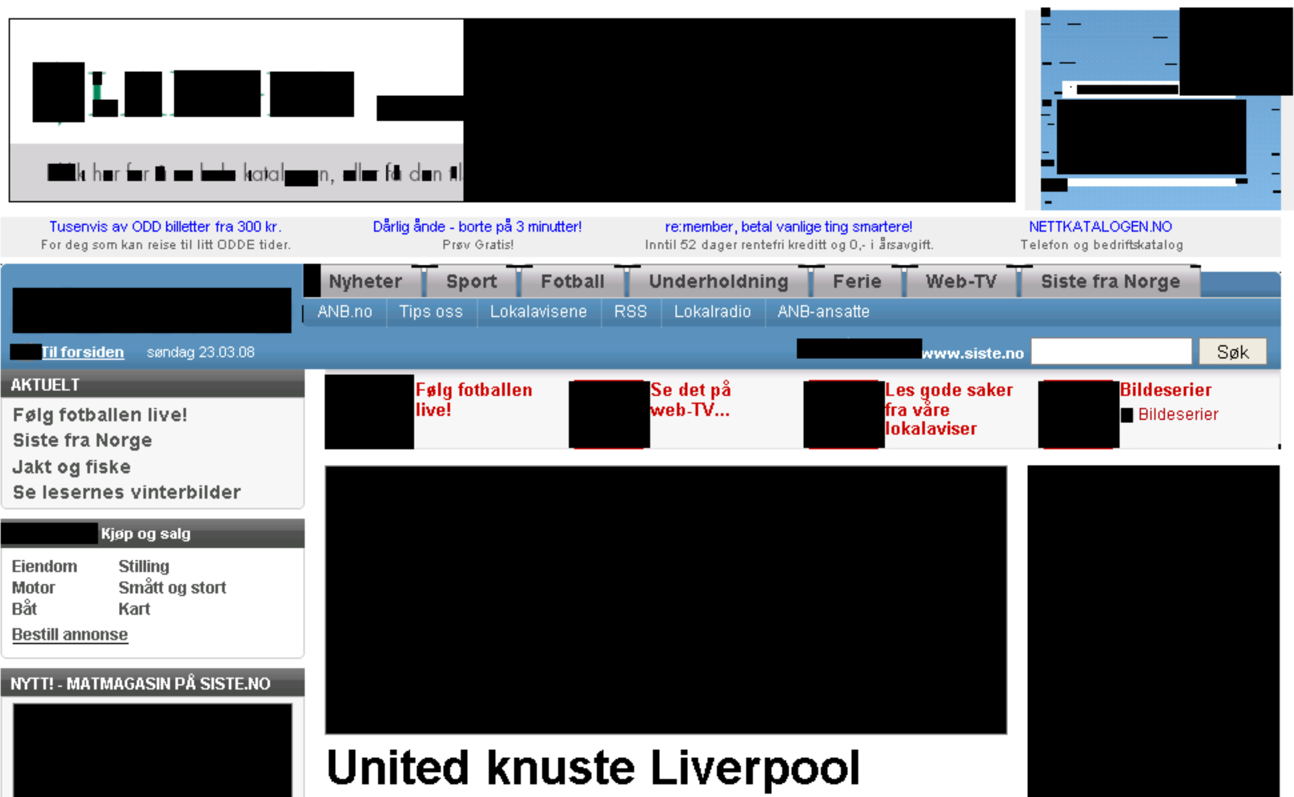


Figure 4.11: The original image with the picture regions removed. This is the pictureless image that is used when segmenting based on color.

After removing the pictures, the next move is to segment the pictureless image into separate layers based on RGB values. The general idea is that elements with the same color belong together and that they will mark out different regions in the image fairly well. Note that the nominal number of colors on a webpage without pictures is low.

To decide which colors that were going to be used, a frequency histogram of the different RGB values in the pictureless image is created. We then select all colors that mark more than 0.004% of the image. For each color, we create a binary layer where all the pixels in the original image with the corresponding RGB value are marked. The layers are then labeled using 8-connected labeling.

The pixels which have other RGB values than the ones selected to will be placed in a special ”remainder” image that is also labeled. Figure 4.14 shows how many color layers were created for each of the different example images.

The value 0.004% was chosen so that if there is a block of 50 x 100 pixels in an image with a special color, then this color should be segmented specially. This should be low enough so that the foreground and background of a specific area is not both placed in the “remainder” image with the result being that the content of the block is lost.

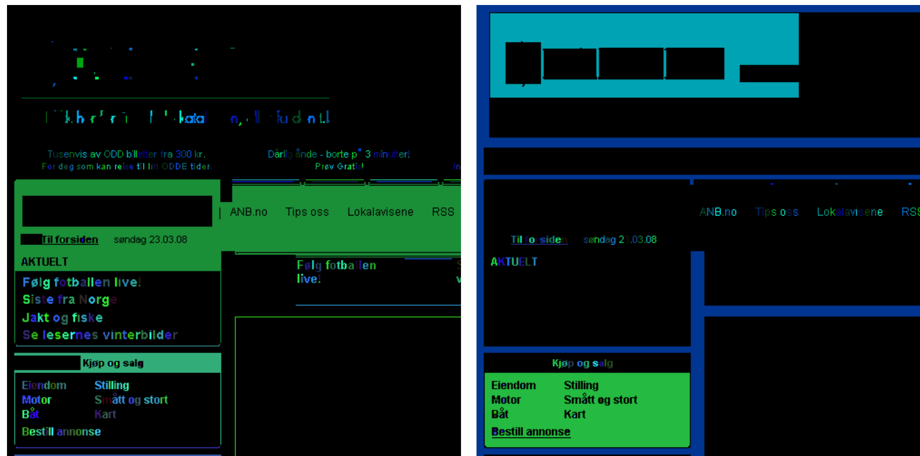
Note that black is treated separately because all the pixels that were defined as pictures were also set to black. To segment the black pixels we first segmented the original image and then removed the areas belonging to pictures.

This ends the segmentation step. Figure 4.12 shows two of the images that are generated in the segmentation phase. Figure 4.12 a) shows the labeled version of the “reminder” image. Figure 4.12 b) shows the labeled version of the pixels that had the RGB value 0xFFFFFF in the pictureless image.

4.5 Region filtering

The next step is to filter the regions created from the color layers so that large regions, representing logical structures, are separated from smaller regions that usually represent text.

Region filtering was done on the basis of a few different parameters and in several stages. We were interested in finding regions that were part of the text and larger regions that might be useful for



(a) The remainder image. Notice that most of the larger areas were actually gradients.

(b) The labeled version of the white part of the example image.

(c) The original image.

Figure 4.12: Two of the images generated from the color segmentation. Both have also been labeled.

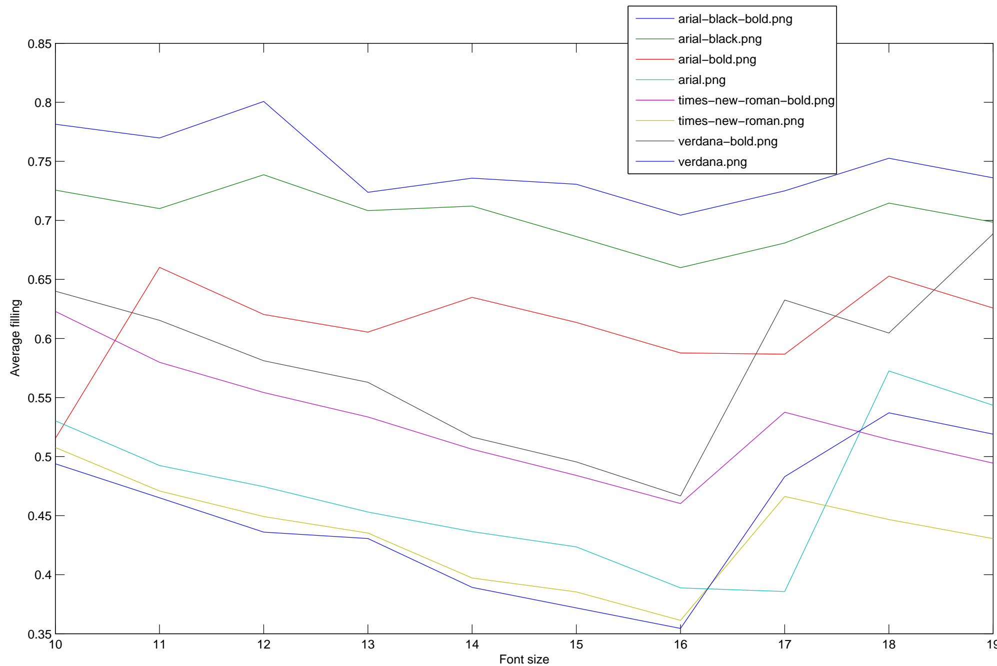


Figure 4.13: Average font coverage for the most used fonts on the web.

defining the page structure, the latter often called boxes or blocks in this text.

4.5.1 Separating out text regions

Some research was done into characteristics of most web fonts, especially looking at the average coverage ratio for Arial Black, Arial, Verdana, Times New Roman and Sans Serif. The main findings were that the coverage ratio varied with font type and size and that it varied between 0.35 and 0.9. Figure 4.13 shows a graph of the average font coverage for the fonts mentioned above. Note that some fonts tend to turn bold at a point, explaining the sudden jumps in their coverage ratio.

With the crude selection criteria used in this thesis these statistics are not very important, but they may be a starting point for adding font detection to the image model.

There is an obvious scope here to use more advanced Bayesian pattern matching methods like Support Vector Machines to separate the regions, but so far, simple criteria have worked well.

4.5.2 Filtering out block regions

The first filtering phase filtered the regions into two categories, block and text, using fairly simple criteria. The most advanced criteria were for larger regions. Before describing the criteria we would like to define a few terms that are used when describing them.

The area of a region is the number of pixels that have been marked to be owned by this region. The boundary area (ba) of the region is area of the regions bounding box. The ratio $coverage = area/ba$ is the ratio of pixels marked as foreground pixels within the regions bounding box. Maxdim is the size of the largest dimension of the region, either height or width.

The block regions were chosen based on the following criteria:

1. If maxdim > 100, then it is a block
2. If region area > 7500 then it is a block
3. If maxdim < 100 and coverage > 0.95 and height/2 > width then it is not a block
4. If maxdim > 20 and coverage > 0.9 then it is a block
5. If height > 10 and area / height > height*2 then it is a block

Item number 3 makes sure that the letter “I” in large fonts is not placed in the wrong category. The last two criteria catches smaller blocks and lines.

The regions not defined as blocks were placed in a separate container and their pixels marked as foreground in a binary image which was then labeled using 8-connected labeling. In some experiments a morphological closing operation with a 5 x 5 rectangle was performed on this image before labeling, but this approach was later removed in favor of using the distance between regions to determine if they had a connection. By merging all the regions into one image we lost information about the color the region had originally. It is possible that the whole merging step should be removed. Since all backgrounds have been removed, (they are larger than the criteria to be considered text), the resulting labeling mainly merges the regions into one container.

Figure 4.14 shows how many of the different region types that resulted from using the algorithm on the test images. Figure 4.15 shows a small part of one of the test images with the different region types marked in different colors. Notice how different region types may stand shoulder to shoulder or be contained within each other. The image has been created before the region merge steps.

Image	***Size***	***T. regions***	***T. boxes***	***Pictures***	***Boxes***	***Sum***	***Colors***
novap.no	983 x 910	2343	48	29	42	2414	5
siste.no	983 x 6275	10733	289	264	281	11278	7
agerposten.no	983 x 1338	3331	83	83	113	3527	19

Figure 4.14: Number of regions generated from three testimages. Text regions were merged to text boxes. The last column is the number of colors used in the segmentation algorithm.

The next step is an iteration over all combinations of picture and text regions. If two regions are connected, i.e. that the text region borders the picture region, then the boundaries of the picture is expanded to also encompass the text region. This catches cases where a few pixels related to the picture region have been left outside it.

The text regions that were not merged with picture regions were then filtered and some regions are categorized as either block regions or a “unknown” category. Instead of trying to describe the criteria, here is the function that handles them:

```
1  private boolean regionMayBeText(Region region) {
2    float boundingArea = region.getHeight() * region.getWidth();
3    float expArea = (boundingArea == 0) ? 0 :  region.area / boundingArea;
4    if (region.getHeight() < 5) return false;
5    if (region.area > 70 && expArea > 0.4) return true;
6    if (region.getHeight() < 4 && region.area > 4) return false;
7    // catch i's.
8    if (region.area / expArea > 0.95 &&
9      region.getHeight()/ region.getWidth() > 5) return true;
10   if (expArea > 0.9 ) {
11     if (region.getHeight()/2 > region.getWidth()
12       && region.getHeight() < 30) return true;
13
14     if (region.area > 6) return false;
15   }
16   return true;
17 }
```

Listing 4.1: The method checking if a regions should be marked as text

There is scope for improving these criteria.

Figure 4.15: The different regions before merging. The blue regions are image regions, red are boxes and green are text regions.

After this step, all the text regions are iterated against each other and regions that overlap are merged.

4.5.3 Defining region relations

The region merge operations described in the last section point to a significant work done when defining the regions: looking at the relationships between them.

A lot of time was devoted to being able to check different region relations. This payed off by making it possible to build a hierarchical model from the regions. Figure 4.16 shows some of the different region relationships we can check for. Apart from those listed, we were also able to measure the distance between region centroids and the shortest distance between two region boundaries.

4.5.4 Grouping text regions

Because of the amount of text regions, it was decided to create a higher grouping for them.

This was done by measuring the distance between two region borders and merging the two regions if they were closer than the height of the highest region. It should not be hard to expand this algorithm to also check directions to create an extra level of grouping that groups together regions that form a

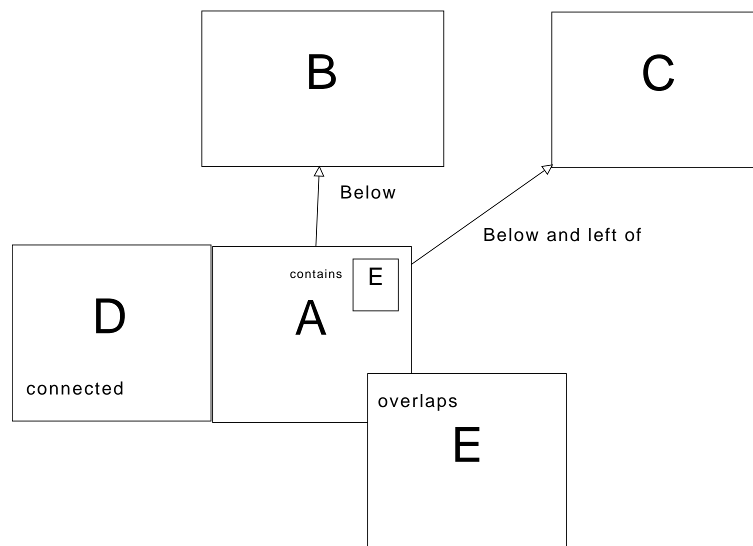


Figure 4.16: Different ways two regions may be related.

line in the page and then group lines together in to form blocks. This would make it possible to check attributes like “ the heading should be in a 12 pixel font”.

After this step we had a manageable amount of regions. It was time to build the model.

4.6 Building a region model

The inputs from the segmentation algorithm are a set of box, text and picture regions.

From the previous step we had the knowledge of which regions were contained in other regions, but we didn’t know if the relationship should work through another region. The first thing that was done was to build a directed tree of nodes describing the relationships between the box regions.

The first step is to map each box region to every other box region. This is done by the following for-loops:

```
1    for (int i = 0 ; i < regions.length ; i++) {
2      for (int j = i+1; j < regions.length; j++) {
3        if (regions[i].contains(regions[j].relateTo(regions[i]))) {
4          regions[i].addRelation(j, 0);
```

```
5          } else if (regions[j].contains(regions[i].relateTo(regions[j]))) {
6            regions[j].addRelation(i, 0);
7          }
8        }
9        root.addRelation(i, 0);
10     }
```

The last step (root.addRelation()) adds every region to a root region that serves as the root of the tree.

Figure 4.17 shows the different region relationships after this step. As the figure shows, we needed to decide which path was the longest path from the root region and to each node. This was done using a version of Dijkstra's shortest path algorithm where we maximize the weight of each node instead of minimizing it. Again a bit of code might help:

```
1   // in the calling function:
2   for (int childregion : root.relations.keySet()) {
3       computeRelations(childregion, 1, -1);
4   }
5   private void computeRelations(int regionId, int weight, int parentId) {
6     if (regions[regionId].weigth < weight) {
7       regions[regionId].parentId = parentId;
8       regions[regionId].weigth = weight;
9
10      if (regions[regionId].relations == null) return;
11      weight++;
12      for (int child : regions[regionId].relations.keySet()) {
13        computeRelations(child, weight, regionId);
14      }
15    }
16  }
```

Having computed the parents, a cleanup step was done which resulted in another iteration. The end result is a complete ordering of the blocks based on which blocks are contained within other blocks. Note that no blocks overlap with each other, but pictures may overlap blocks.

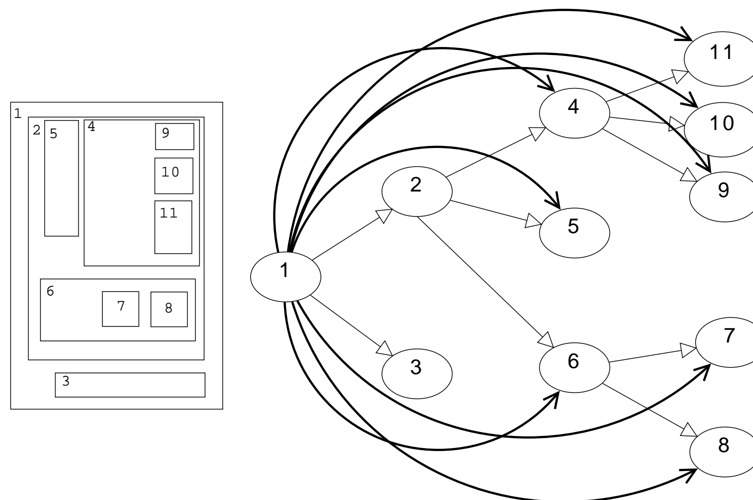


Figure 4.17: Example of a set of regions and the graph generated. The thicker lines are the ones we want to remove so that we end up with a simple tree structure.

4.6.1 Border detection

A very common way to define an area on a webpage is to use a border to separate it from other areas. With the segmentation method described above, this will create two regions, one for the area inside the border and one representing the border. In a lot of applications, this doubles the number of regions that exist in the image. To handle this situation, a special check for bordering regions was implemented.

The check works by inspecting the borders of the two regions. It examines the difference between the pairs of corner points for the region. If this difference is the same for all corners, then the outer region is the border of the inner region. The inner region then moves it's children and elements to the parent region and turns into a border type region, at the same time the color information of the two regions is swapped.

Detecting borders reduces the number of boxes and makes the resulting model closer to the mental model a user has of a web page.

After removing borders, the last step is to add picture and text regions to the resulting image. This is done by doing a recursive iteration of the box nodes from the root and outwards using a depth first recursive function. This ensures that it is the deepest node that is mapped to the picture or the

text node. For each node, the function then checks all the picture and text regions and adds the ones contained in the node to its list of elements and removing them from their respective lists.

4.6.2 Segmentation results

Figures 4.18, 4.19 and 4.20 show how the segmentation algorithm worked on the different test images. Notice that there is still work to be done both on better heuristics for merging text regions and also in joining image regions.

Figure 4.18 shows that there should be a step after the node tree is created that splits a block if the text and pictures inside it are in separate columns or rows. Notice that there is one large block that contains the text and images for all three columns in the content area. A better version of the algorithm should generate three separate blocks inside the large block that would contain the text regions.

The algorithm is not good enough in detecting images containing areas that are completely uniform - for example advertisements. Notice the advertisement at the top of figure 4.20. It contains some areas that are marked as pictures but the complete advertisement is not marked as a picture.

4.7 Page model representation

With a complete page model, we are ready to write out the model to a representation format to keep between test runs. XML was chosen as the format because this is the most standard hierarchical representation format and because there are well defined APIs for reading, writing and querying the results.

Each object type is represented with its own tag with a common set of attributes defining the objects shape and other characteristics. Figure 4.21 shows a sample model for a very simple page.

This is not the complete page, but shows how each objecttype is represented. The amount of information to save about each object is extensible. It was chosen not to try to expand the amount of information outside object position, height, weight, area, border detection and sometimes color.

Figure 4.18: The final regions for novap.no. The red regions are boxes, the blue are pictures and the yellow are text boxes.

Figure 4.19: The final regions for agderposten.no. The red regions are boxes, the blue are pictures and the yellow are text boxes.

Figure 4.20: The final regions for siste.no. The red regions are boxes, the blue are pictures and the yellow are text boxes. Note that the image has been cropped. The complete image is 1 meter long.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <imageModel>
3    <root x="0" y="0" width="1" height="1" area="0">
4      <box x="0" y="0" width="1400" height="765" area="533560"
5        color="#ffffff">
6        <box x="253" y="107" width="554" height="522" area="2148"
7           color="#ffffff">
8          <border width="554" color="#ffffff" x="253" y="107" height="522"
9                  area="215969" />
10           <text x="297" y="177" width="470" height="88" area="47873" />
11           <text x="297" y="177" width="470" height="88" area="56888" />
12       </box>
13   </box>
14   </root>
15 </imageModel>
```

Figure 4.21: A fragment of the model representation that is written to disk.

4.8 Looking for model deviations

Trying to define a model from the webpage is the most ambitious part of the project as it tries to do a large amount of work itself instead of relying on the user to define how the page should look.

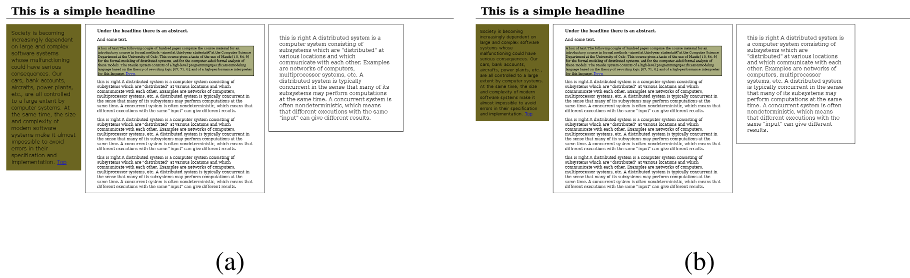
There is an endless amount of options available when it comes to properties that can be used to compare two images. To test the concept, we focused on region dimensions.

4.8.1 Comparing page graphs

A problem with comparing two node graphs is to find a reasonable way to describe differences and also to decide if those variations are within the bounds defined. There are some existing algorithms for finding the differences between different XML documents.

A tool that was investigated was JXyDiff[1]. It implements the Fast match/Edit script change detection algorithm originally created by Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom[14]. A problem with JXyDiff is that it was only possible to download the compiled version of the program.

[1] JXyDiff is based on XmlDiff originally developed by Gregory Cobena, Serge Abiteboul and Amelie Marian. http://potiron.loria.fr/projects/jxydiff



(a) (b)

Figure 4.22: Two very simple testimages. Each image generated a model representation of 20 to 25 nodes.

The fast match / edit script algorithm computes the minimum amount of changes that has to be done to convert one hierarchy of nodes to another one. It's output is a list of inserts, deletions, moves and updates that will transform the original tree into the new one.

DiffXML[2] is written by Adrian Mounat as his CS4 dissertation and implements the fast match / edit script.

Xmldiff[3] by Logilab is a python implementation of the fast match/edit script algorithm. It's man page states: "xmldiff uses an algorithm with a (too) high algorithmical complexity, which makes it unsuitable to process large XML documents. If your document has more than about 100 nodes, you should probably look for an alternative solution.".

These three tools were tested on a very simple page where two error were injected as seen on figure 4.22. On these simple images, the difference XML files generated by the tools were generally larger than the representations they compared and proved to be a bad starting point for trying to define variations.

4.8.2 Node comparison algorithm

As the two node trees are fairly close in structure (or else it's a failure) and we have prior knowledge about the relations between different types of nodes in the trees, we chose a very simple recursive algorithm where we compared the two trees node by node. Figure 4.23 shows some pseudocode of

[2] `http://diffxml.sf.net`
[3] `http://www.logilab.org/859/`

```
1
2  public void compareNodes(node1, node2) {
3    if (node1 is larger/smaller than node2) {
4      reportError();
5      return;
6    }
7    if (!checkChildAggergateValues()) {
8      reportError();
9      return;
10   }
11
12   for (child1 in node1.getChildren(),child2 in node2.getChildren()) {
13     compareNodes(child1, child2);
14   }
15
16   for (pictureChild1 in node1.getPictures(),pictureChild2 in node2.getPictures()
         ) {
17     compareNodes(child1, child2);
18   }
19 }
```

Figure 4.23: Pseudocode for the recursive function used to compare two nodes

the algorithm.

It is important to note that when the trees are written to disk and also when they are read, the child nodes and the child elements (images, text) gets sorted based on which node is closest to the top left corner of the image as well as node type. This ensures that when comparing children of a node chances are high that the two nodes compared represents visual objects having the same role in the two images.

4.8.3 Finding the real error - not the sideeffects.

Often text or images are defined to flow within a region. If the regions size increases then the objects inside may also change shape. If every check gets done on every level in the hierarchy, then one error may be reported multiple times as it percolates down to the childnodes. To stop this, errorchecking is not done for children of a node containing errors.

```
1  <box x="820" y="108" width="414" height="330" area="126217" color="#ffffff">
2          <variations maxWidth="100" minWidth="100" minx="800" maxx="820"
3   maxHeight="500" minHeight="200" miny="100" maxy="100"
4   maxTextWidth="500" minTextWidth="400" minNumText="1" minNumImages="10">
5              <variation ... />
6       <variation ... />
7          </variations>
8  </box>
```

Figure 4.24: Adding variations to the page model makes it more flexible.

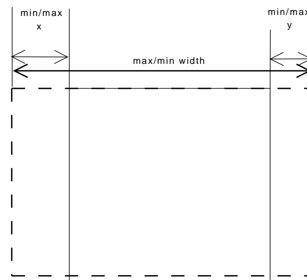


Figure 4.25: Two boxes, and the possible ways to define how they differ in the horizontal (x) dimension.

4.9 Defining and representing different variations

The idea is to try to express the allowed variations as part of the representation format and try to make sure that each subtree of boxes conforms with a set of permitted variations.

For each node in the image model, we now add a possible subnode “variations” that contains information on how the parent may vary. Special variations may be encoded in children of the node. For now that is unused. Figure 4.24 shows a box node with some variations added.

4.9.1 Dimension variations

The first type of variations we check for is variations in height and width. Figure 4.25 shows how we can define different allowed variations in one direction. Section 3.5 lists all the possible variations a node tag may have.

This list is not complete (it is missing maxNumImages for example) in terms of possible variations we might want to allow for, but it is fine balance between having enough flexibility to test the models and the time it took to test each attribute implementation.

If an attribute is not defined, then it is expected that the two objects should have an exact match. This is also the case if the parent is lacking the variations child.

One difficult item is when you want a node to be able to vary in a way that affects the parent node. A typical example for this is when you want to let the text be longer than in the original template. This will then affect the height of all the parent nodes.

The solution is that when a variation that may affect the parent is set to a node, then this variation is also updated for the parent nodes. There are weaknesses in this approach. For example, if two nodes both have maxWidth then the maxWidth of the parent should be the sum of this value. This may be better approach but was not implemented due to time constraints.

4.9.2 Text and image variations

Text and images may vary in different ways than normal boxes, for example we may want the text to be of varying size. Therefore a set of variations are defined on the node level that allow for variations in text or image aggregates:

- The minimum number of text items.
- Minimum and maximum widths of the text areas.
- A minimum number of images.

This list could also have included a maximum number of text areas and min and max height of the complete text areas as well as a defined place for the text areas to start. These were excluded due to time constraints.

Pictures are not treated as text but instead checked individually to open up for more precise placement. An interesting case is if you want pictures to alternate between two positions, for example in an article

listing. In the current implementation, it is not possible to make sure that the picture then is not in the middle of the text instead of along one of the edges.

A more complete implementation should probably also contain the option to do special checks for the n'th item of either text or images, for example checking that the first textblock is aligned with the first image. This could be done using xpath expressions.

4.9.3 Model visualization and error representation

To visualize the model, SVG was used to paint the different regions on top of the image used to generate them. This was also used to visualize errors that were found.

To represent errors, the original edge is painted in green while the new edge is painted in red. This makes it possible to discern errors caused by height or weight.

It was harder to decide on how to visualize errors coming from a reduced number of text regions or missing images. To represent missing images, the area where the image was located in the original was marked. The same way was used for missing text.

Chapter 5

Experimental results

This chapter looks at the results of two experiments as well as some research into the importance of rendering errors in web pages.

5.1 How important are visual and browser related bugs in web development?

Very little research has been dedicated to this issue although more and more programmers move from more traditional programming and into web programing.

A search on the internet finds a lot of websites[1] dedicated to helping developers solve problems related to different browsers that render webpages differently. It is not hard to find long rants against a specific browser, but it is harder to find studies on the cost of these bugs or how much developer time is used making workarounds for browser bugs.

To study this in more detail, we got access to the bug database from Apressen Interaktiv AS(API). API produces web sites for 56 different newspapers in Norway. They deliver all the services newspapers need to have a modern professional web page. This is one of the 3 largest operations of this kind in Norway.

[1] http://alistapart.com, http://csszengarden.com and http://quirksmode.org are good examples.

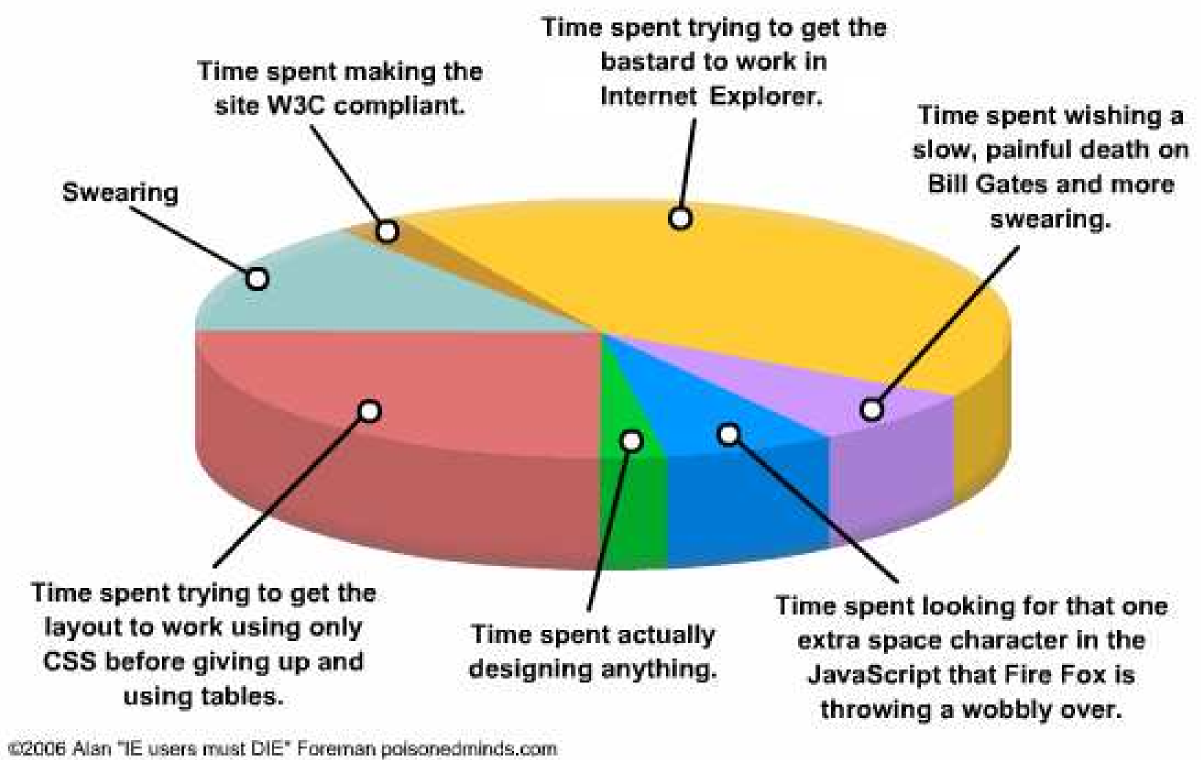


Figure 5.1: A lighthearted look at the frustrations of web development.

The bug database only contained bugs that needed developer attention. Most bugs had been entered by customer support. There were very few duplicates or unconfirmed bugs, but some feature requests. The general quality of the bugreports was high, but it must be noted that not all bugs reported or fixed were registered in the system. This may happen when a developer is notified of a bug and fixes it right away or if the bug is reported using email or other means of communication.

This is a general problem with bug databases. They are not maintained for statistical purposes, but to make sure that important tasks are not forgotten. Bug reports are filed by humans and thus the descriptions and the usage of the system may vary between users and over time.

The bugs were categorized by reading though each bugreport and it's comments and adding a set of categories to the bugreport. The categories were defined as follows:

1. Visual bugs were bugs that somehow affected the rendering of the webpage.
2. Backend bugs were bugs that were related to backend services like databases, caches or the CMS.
3. Browser bugs were related to one of several browsers.
4. Internal are bugs that were related to different internal development projects.

We tried to make sure that visual bugs were errors in how the page was rendered and not feature requests regarding the current layout.

Category	**Nr. of bugs**	%
Browser related	73	12.5%
Visual	101	17.3%
Backend	76	13.0%
Internal projects	429	73.6%
Total nr of bugs	583	

Figure 5.2: Bug database categorization results. Internal projects refer to bugs that were directly related to specific internal projects.

The results are summarized in figure 5.1. Note that a bug could be placed in several categories. While the "visual" category is a bit hard to define and prone to researcher bias, the category related

to browser bugs is thought to contain very few mislabeled bugs as the bug descriptions would contain which browser the bug was related to and the difference compared to other browsers.

As the results show, around 13% of all bugs reported were related to differing browser behavior. Around 30% (32 out of 101) of the visual bugs were also categorized as browser bugs. A categorization of bugs related to different browsers was also done and is shown in figure 5.1. As there were some bugs that were related to multiple browsers, the results do not add up to the same amount of bugs as in figure 5.1.

Browser	**Nr. bugs**	%
Firefox	11	13%
IE6	39	46%
IE7	28	33%
Safari	5	6%
Opera	1	1%
Sum	84	

Figure 5.3: Number of browserbugs related to different browsers.

It was not the object of this thesis to find out why bugs occur or which browsers are better than other browsers. Our main point is that visual bugs and bugs related to a specific browser both form significant portion the number of errors that escape into released versions of the software.

The results come from only one site that is highly focused on high volume content delivery. It would be interesting to compare this bug database to one from team that works on a more traditional webapplication to see if there are any differences.

5.2 Using Sysifos for continuous monitoring

One interesting way to use of Sysifos is to set it up to check a web site every hour to detect changes in rendering. This may detect errors coming from third party contributors such as comments, advertisements, externally sourced material or related to new changes being rolled out.

This experiment tests using Sysifos in a fully automated setting. It looks at the quality of the acquisition method and also to see how well different errors are detected.

To run the test we used a Continuous integration tool called Apache Continuum[2]. Continuous integration systems are usually used to run automatic tests of the code under development. In this test it was set up against live websites.

5.2.1 Test setup

A simple continuous build setup was created on a Dell poweredge 860 with 4Gb of memory and a Quad Core Xeon processor running Ubuntu 7.10. On the box, VMware Server was installed with a fully patched Windows XP running on it. The Windows XP image contained three browsers, Internet Explorer 7.0, Firefox 2.0 and Opera 9.

Apache Continuum was used run the tests. Initially the tests were executed every 4 hours, but later the rate was increased to every hour for a total of 200 test runs.

The pages used were selected using the following criteria:

1. Errors detected by others would be reported back to the authors of this thesis
2. They should contain a high rate of change
3. They should have different layouts

Item 3) is included to see how robust the testmethod is with regard to to false positives.

The initial URLs selected were:

- http://www.viddi.no
- http://news.portalfabrikken.no/Ibsen
- http://agderposten.viddi.no/arendal
- http://www.ba.no

[2]http://continuum.apache.org/

- http://www.nordlys.no
- http://www.ta.no
- http://www.siste.no
- http://www.db.no

The test cases was set up so that it would use assertNoChange on areas of the page that were thought to seldom change.

5.2.2 Results

After 184 testruns, the image test code had reported errors in 136 runs, of which 7 contained errors reported by the assertions described in section 3.4. A total of 86 runs (46,7%) can be viewed as successfull in the sense that they produced images that where usable for further processing.

The other failures were related to three main problems:

- Timeouts: The pages tested where so large that for the page to be rendered completely, a requiremen for Selenium, it takes more than 50 seconds. This was mainly the test for “www.db.no” so this test was disabled after testrun 22. Later random timeouts for different pages occurred.
- Selenium error: This happened due to a bug where Selenium didn’t close the browser after finishing the test. Eventually the test machine had to be restartedbecause it was overloaded.
- Unknown: The errors that have been coded as unknown here fall in two categories, either code bugs (the early errors) or lost connections (the late errors).

After filtering away test runs that didn’t take any time due to connection errors, a complete test run took a mean time of 9 (8.96) minutes.

The system did not manage to find any errors caused by changed ways of rendering. This was mainly because the number of false errors reported was too high to make it possible to notice any real errors. The errors that were reported due to asserts failing were mainly because the images returned from Selenium RC were corrupt. This could happen when the browser window used when defining the

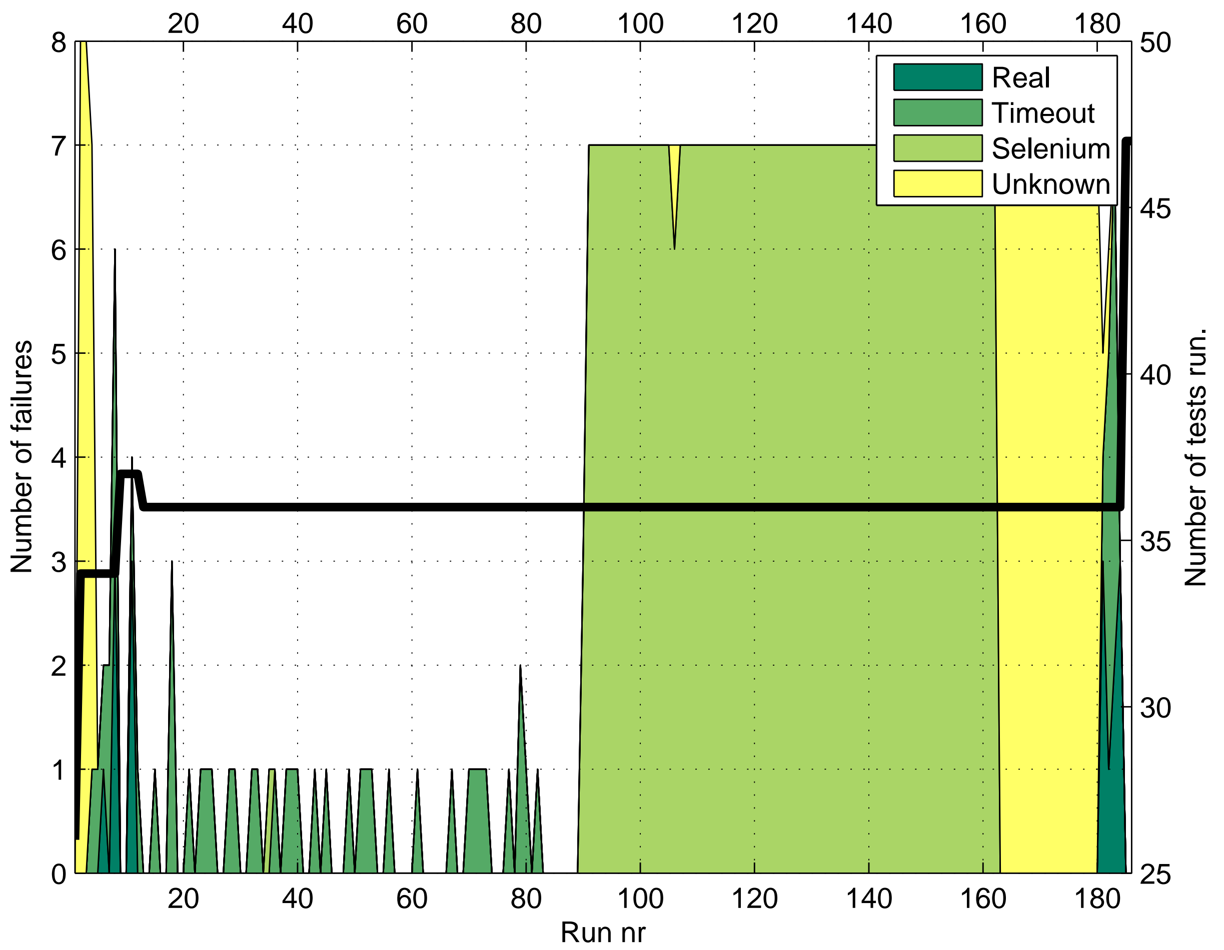


Figure 5.4: Number and type of errors reported in the first experiment. The right hand scale is the number of test cases for each test run.

viewport area of the browser was placed behind another window so the viewport coordinates were wrong.

5.2.3 Discussion

This simple experiment provided valuable input with regard to several different parts of the design of the application as well as highlighting some usability issues:

- Each test for different browsers must be separated out from each other as separate tests so it is possible to find out which browser or operating system that failed, not just which test that failed.
- It is important to reduce the amount of failures due to connection or timeout problems.
- Using the most basic assertions like assertNoChange does not produce tests that are likely to find many errors.
- It is important to be able to connect the image data saved with the testrun that generated the image. If not, it is not possible to generate a reasonable user interface for viewing testresults.

The last point was noticed when we were analyzing the trial results and couldn't match an image to a test run. This made it hard to find out why some test runs had failed and we ended up matching the time of the test run to when the file had been created. The image storage library was later modified so that it is possible to set the versions of an image from the test runner.

The stability problems in capturing the images were so large that they overshadowed the focus of the test - to see if any errors were found. Overcoming these problems is important to make the system reach production stability. It is probably wise to use browser specific APIs to capture the screenshots instead of relying on Javas slow Robot class.

The experiment also showed that it is hard to use the simple image assertions to detect changes in rendering because they can only work on areas that rarely change. Assuming that areas with little change have a lower risk of generating errors, then the simple assertions will not be very good at finding errors.

The last assumption may very well be wrong because even areas with a low rate of change (say the logo of the website) may depend on the other items in the code that generates the webpage. For

example, if there is an error in the CSS defining where the content area should be, it may end up overlapping the logo.

5.3 Using the page model to find errors

The point of this experiment was to look at three different perspectives related to Sysifos. We were interested in seeing:

- How the system performed when testing against a set of known bugs.
- How it worked to evolve a page model based on the initial representation.
- If Sysifos works in a cross browser context.

The first item looks at the systems technical capabilities, while the second deals with the usability of the system.

The test will highlight:

- What errors are found
- How different variations are dealt with in the model
- Problems with inner objects expanding the outer ones
- How errors are reported

To test how a page model could be used to find and report errors we needed a simple webpage that we could modify. Instead of creating a custom page for the test, the template Simple3color[3] was downloaded from the website Open Source Web Design (`http://www.oswd.org`).

One change was done to the template. The body element was given a specific width of 800 pixels. This was done because the template was dynamic with regard to screensize and would therefore trigger scrollbar problems if this value was not set.

Two images downloaded from flickr.com were also used[4].

[3] `http://www.oswd.org/design/information/id/3533`
[4] The images were downloaded on Saturday April 19 2008 from http://flickr.com/photos/nuomi/77156374/in/photostream/

Figure 5.5: The original page

Simple3color was chosen based on two criteria, 1) that it was listed on the first page of OSWDs “Our favorite designs” page and 2) that it didn’t contain gradients. Figure 5.5 shows the original.

Using a template instead of a live page was chosen as it would contain fewer items and a more simple structure.

5.3.1 Test setup

The template was then modified into different versions in the following ways:

1. A version with more text, so that there were differences in the amount of text in the content area between this image and the new one (test1).
2. A version containing two images (test2).
3. A version with an extra menu item (test3).

4. A version where the content area has become much wider, squeezing the right hand menu below it (test4).
5. A version with different image placements, one of the images is placed on the right side of the article (test5).
6. A version where some of the boxes had changed color (test6).
7. A version where an image overlaps the content area and the news listing on the right hand side (test7).
8. A version with no content in the content area (test8).

A set of rules for the pages was defined before the test code was written. The rules may be looked at as a “user story” defining what variations may happen to the page.

The rules define a page that may change content but where the layout is set. The page rules are:

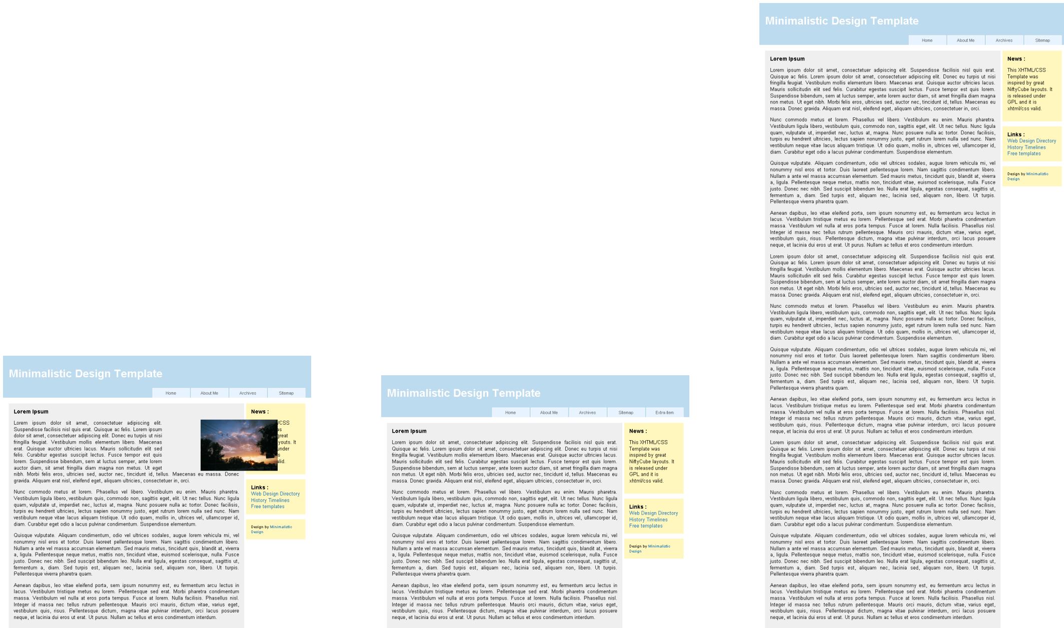
- The general layout should not change, except that the content area may be expanded to contain more text.
- The content area may contain images, but they must not be wider than 200 pixels and stay on the left side of the area.
- The content area should always contain content.
- The menu may contain an extra item.

Based on these rules, the first three versions of the page should not report errors while the last four should.

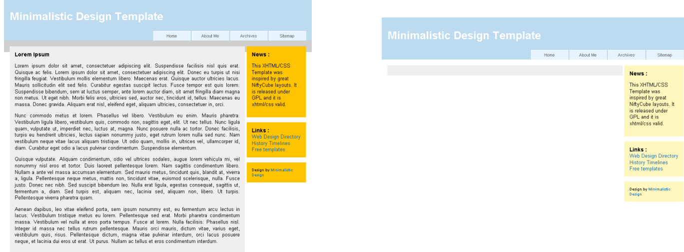
A test was created using the patched version of Selenium RC and the Sysifos framework. The java code used is found in appendix A.1.

The test had 9 test cases with each test case testing one of the test images shown in figure 5.6 or the original image.

The test was run once to generate the images and page models. Figure 5.6 shows the different test images. The pagemodels for the test images were discarded, only the model for the original was kept. All the tests used the same page model file as the original file to compare with.



(a) Test image 7 (b) Test image 3 (c) Test image 1



(d) Test image 6 (e) Test image 8

(f) Test image 2 (g) Test image 4 (h) Test image 5

Figure 5.6: The different test images. Each number corresponds to the number in the test id.

Then a new test run was done with all images comparing to the model created by the original image. This was defined as T0. Then we started to apply the different rules described bellow and rerunning the test after each change. This is defined as runs T1 to T4.

Running one test through all the test cases including image capture took about 1 minute and 30 seconds on a 2GHz Intel Core 2 Duo processor having 2GB of ram. The Selenium RC server was run using VMware on the same machine using the same Windows XP image as in section 5.2.

5.3.2 Evolving the pagemodel

In this section we go through each of the rules above and show how they modified the page model markup. This both shows how the test was modified as well as it shows the amount of effort that is put into the model.

Rule nr 1: The content area may grow. The main consequence of this rule was that we should detect elements that cross box boundaries, thus catching the change in test nr 7 where the image is placed across two regions. Also we change the maxheight for the content area so that it may grow. Figure 5.7 shows the changes.

```
1 <box x="17" y="106" width="759" height="475" area="329123" color="#efefef">
2     <variations maxHeight="2000"  ></variations>
3 </box>
```

Figure 5.7: Adding maxHeight to the box defining the content area.

Rule nr 2: The first image in an article must not be wider than 200 pixels and stay on the right side. To represent this rule, we had to change the box that represented the content area and add an extra image definition to it containing the settings for image size and where the image may be placed. Note that if we wanted to have this rule for many images, then it would become cumbersome. Figure 5.8 shows the changes.

The minX attribute makes sure the image is placed to the left. The minY attribute says that it should not be above the text.

```
1 <box x="17" y="106" width="759" height="475" area="329123" color="#efefef">
2     <variations maxHeight="2000" ></variations>
3      <image x="41" y="178" width="500" height="333" area="157891" >
4     <variations  maxWidth="200" minHeight="50" maxHeight="500" minX="543"
5            minY="124" topY="500" ></variations>
6      <image>
7 </box>
```

Figure 5.8: Adding an extra image object to the markup to define where the image may be placed, and its size.

Rule nr 3: The content area should always contain content. This rule is modeled by setting the minTextHeight higher than 0. Figure 5.9 shows the changes.

```
1 <box x="17" y="106" width="759" height="475" area="329123" color="#efefef">
2     <variations minTextHeight="5" maxHeight="1500"  ></variations>
3 </box>
```

Figure 5.9: By adding minTextHeight we are demanding that there should always be text in the content area.

Rule nr 4: The menu may contain an extra item. To model this change, we allowed the number of child elements to the top box to increase by setting the maxChildren variation attribute.

The problem then became that the extra menu element pushed the other elements to the left, with the result that each element reported that it had been moved, causing the test to fail.

To fix this, we had to allow the child elements to move sideways. To do this, we set the minX value for each. It would probably not hurt to add an extra option for this in the parent but this was not done at the time of the experiment. Figure 5.10 displays the changes.

Adding minX values to the elements added 4 extra lines to the model. The area changed became:

The resulting model after all the changes is shown in figure 5.3.2.

```
1  <box x="0" y="0" width="800" height="109" area="73859" color="#bddbef">
2    <variations maxChildren="5"></variations>
3    <box x="389" y="85" width="95" height="24" area="2209" color="#e7f3ff">
4      <variations minX="100" />
5      <text x="423" y="93" width="26" height="8" area="92" />
6    </box>
7    <box x="489" y="85" width="95" height="24" area="2161" color="#e7f3ff">
8      <variations minX="100" />
9      <text x="513" y="93" width="45" height="8" area="147" />
10   </box>
11   <box x="589" y="85" width="95" height="24" area="2180" color="#e7f3ff">
12     <variations minX="100" />
13     <text x="614" y="93" width="43" height="8" area="129" />
14   </box>
15   <box x="689" y="85" width="95" height="24" area="2172" color="#e7f3ff">
16     <variations minX="100" />
17     <text x="717" y="93" width="37" height="10" area="274" />
18   </box>
19   <text x="17" y="36" width="396" height="26" area="18248" />
20 </box>
```

Figure 5.10: Added minX values to the menu element so that they may be pushed to the left by another element.

5.3.3 Results

The tests were run 5 times. Once with the original mode, then once after each of the above changes had been added to the model to make sure that there were no unintended consequences of each addition.

4 of the tests were created to fail and 4 to succeed. Figure 5.3.3 show the results after each run. T0 is the before any modifications while T1 - T4 is after each of the modifications described above.

As described in section 3.4.3, a test may report more than one error when it fails. It is the number of errors reported by this reporter that is the number of errors reported by the `assertModel()` call. Figure 5.3.3 shows the results of the different test runs.

The image model detects most of the errors. The reason the error in test nr.7 is not detected is because we are lacking options to define max number of images in a region and a way to allow or deny objects to overlap it's siblings like the image does.

Figure 5.13 shows how errors are marked using SVG to paint on top of the new image. The image marks errors in T4 for test 6. The green lines show where the original box was, while the red lines display the new dimensions. Brown lines are where the two boxes have the same dimensions.

The test shown actually has a new box due to changing colors of some of the boxes. That is the one shown in red. The green lines show where a box with the same place in the node tree was in the original. The error messages show that it isn't the color changes that have been detected, it is the new box that popped up because it suddenly got a different color than the background.

5.3.4 Cross browser testing

```xml
<?xml version="1.0" encoding="UTF-8"?>
<imageModel>
  <root x="0" y="0" width="1" height="1" area="0">
    <box x="0" y="0" width="983" height="660" area="184306" color="#ffffff">
      <box x="0" y="0" width="800" height="109" area="73859" color="#bddbef">
        <variations maxChildren="5"></variations>
        <box x="389" y="85" width="95" height="24" area="2209" color="#e7f3ff">
            <variations minX="100" />
          <text x="423" y="93" width="26" height="8" area="92" />
        </box>
        <box x="489" y="85" width="95" height="24" area="2161" color="#e7f3ff">
            <variations minX="100" />
          <text x="513" y="93" width="45" height="8" area="147" />
        </box>
        <box x="589" y="85" width="95" height="24" area="2180" color="#e7f3ff">
            <variations minX="100" />
          <text x="614" y="93" width="43" height="8" area="129" />
        </box>
        <box x="689" y="85" width="95" height="24" area="2172" color="#e7f3ff">
            <variations minX="100" />
          <text x="717" y="93" width="37" height="10" area="274" />
        </box>
        <text x="17" y="36" width="396" height="26" area="18248" />
      </box>
      <box x="15" y="124" width="611" height="536" area="296094" color="#efefef"
          >
        <variations minTextHeight="5" maxHeight="2000"  ></variations>
        <image x="400" y="178" width="200" height="333" area="157891" >
             <variations maxWidth="200" minHeight="50" maxHeight="500" minX="543
                " minY="124" topY="500"  ></variations>
        </image>
        <text x="29" y="140" width="88" height="13" area="1076" />
        <text x="28" y="169" width="584" height="113" area="98638" />
        <text x="28" y="298" width="584" height="97" area="199741" />
        <text x="27" y="411" width="585" height="113" area="147764" />
        <text x="28" y="541" width="584" height="96" area="63764" />
      </box>
      <box x="631" y="124" width="154" height="184" area="26609" color="#fff7bd"
          >
        <text x="645" y="140" width="36" height="10" area="176" />
        <text x="644" y="169" width="101" height="110" area="4084" />
        <text x="755" y="219" width="5" height="10" area="22" />
      </box>
      <box x="631" y="321" width="154" height="91" area="13094" color="#fff7bd">
        <text x="644" y="337" width="126" height="62" area="9493" />
      </box>
      <box x="631" y="425" width="154" height="52" area="7714" color="#fff7bd">
        <text x="645" y="441" width="30" height="22" area="178" />
        <text x="680" y="441" width="70" height="9" area="508" />
      </box>
    </box>
  </root>
</imageModel>
```

Figure 5.11: The complete model after adding the needed changes to it.

Test nr:	**Test image description**	**T0**	**T1**	**T2**	**T3**	**T4**	**Expected Result**
		No. of errors					
Original	The original image	0	0	0	0	0	Pass
1	More text	**2**	0	0	0	0	Pass
2	Two images	**2**	0	0	0	0	Pass
3	extra menuitem	**5**	**5**	**5**	0	0	Pass
4	Wide content area	5	4	4	4	4	Fail
5	Image to the left	2	9	1	1	1	Fail
6	Changed color	5	5	5	5	5	Fail
7	Breaking region boundraries	2	**0**	**0**	**0**	**0**	Fail
8	No content	2	2	2	2	2	Fail

Nr of reporting errors:	3	2	2	1	1
Nr of errors reported as pass	0	1	1	1	1
Nr of failed ok images	3	1	1	0	0

Figure 5.12: Results from running the tests. T0 is before modifying the model, while T1 - 4 is after each of the steps above. To pass a test must have 0 errors.

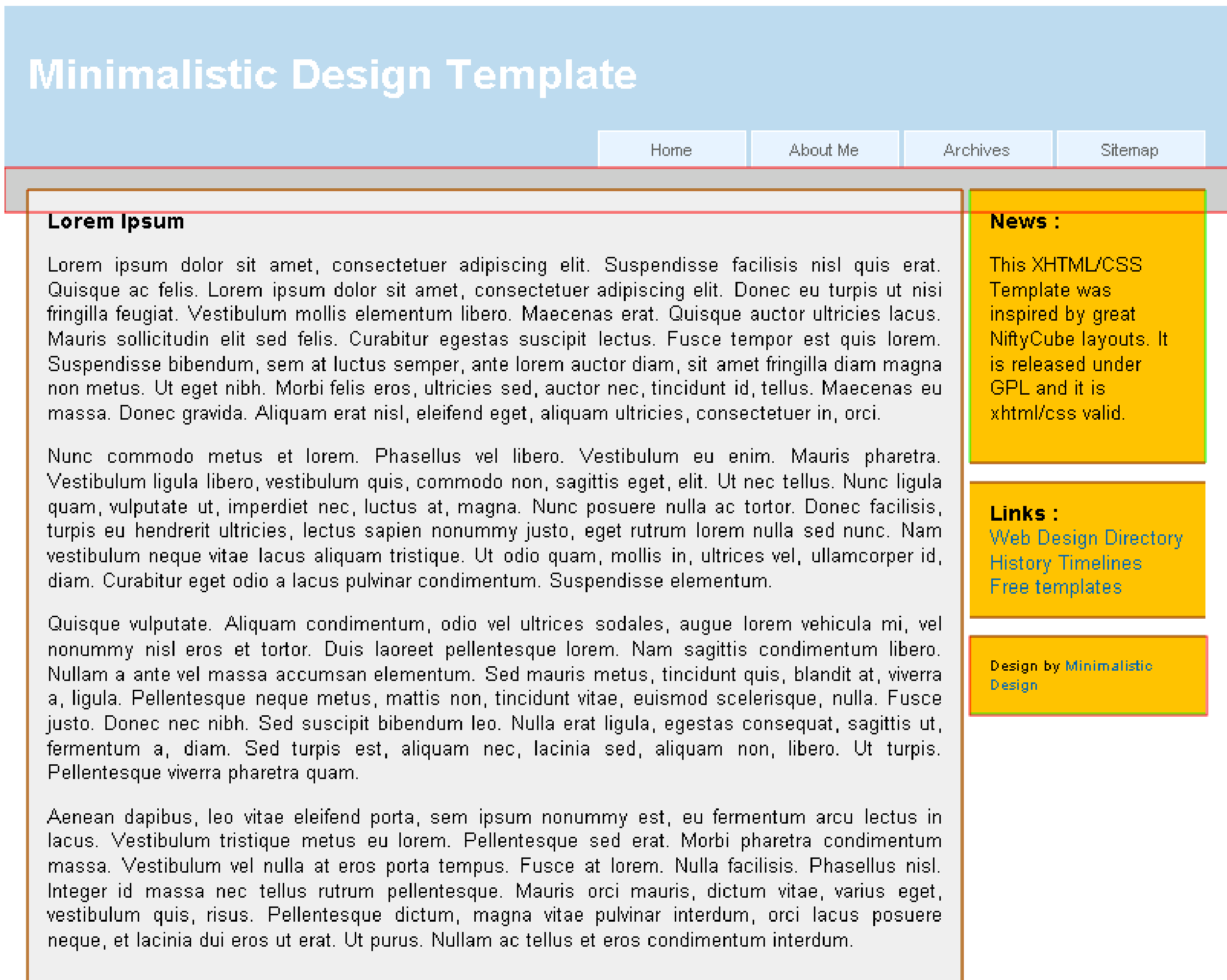


Figure 5.13: A screenshot with errors marked. The green lines show where the original box was. The red lines show where the box is in the new image. Brown lines are where the two boxes share the same placement.

(a) The original image in Firefox... (b) and in Opera

Figure 5.14: The original page in Firefox and Opera. When compared they generate errors.

This simple test showed that this is a tool that may be useful in detecting browser differences in rendering pages. Modifying the model to allow for the differences was not done.

5.3.5 Discussion

From statistics we have the terms type I and type II errors. A type I error is a false positive. It happens when a test passes that should fail. A type II error occurs when a test fails but shouldn't. Both types of errors should be minimized.

The results in figure 5.3.3 show that the end result does not report type I errors, but it contains a type II error. As such, the test does deliver fairy correct results.

The error that it doesn't report, test nr.7, is interesting. The reason it doesn't report an error here is because the system does not contain a way to specify that objects on the same level in the node tree should not overlap. Adding a test for this is fairly trivial but was not done before the experiment was run. The initial expectation was that the test would catch the error.

A question is how this should be treated when the initial model is created. It would make sense that when the model is generated, it would assume that regions should not overlap and that regions that do overlap should have this noted in the variations section. A problem with this approach is that it may increase the amount of code in the model file.

The amount of effort to make a test and to evolve the test is far lower than it would be if we had to write tests that inspect different files to find the same errors. The latter approach would also be less flexible with regard to variations. Another thing is that modifying the model while evolving the test

Hypothesis	Reject	Accept
Real answer:		
True	Type 1 error	
False		Type 2 error

Figure 5.15: Type 1 and type 2 errors.

is easy and fast to do. In this respect the system works.

A valid critique against this, is that the model is quickly approaching a parallel to the original page with regard to the size of the two documents. What if you end up having to keep two different documents in shape? Another problem will be to navigate a large model file containing only abstract "box" items.

The point of Sysifos is to catch errors that come from the interaction of different components. That is hard to do by inspecting each component separately.

A simple way to reduce the amount of time used to modify the model, is to define one browser as the one where the rendering is correct and then only look for differences across browsers. Then there is no maintenance of the model since it is generated every time the test is done, but this requires that the pages look almost completely the same.

Another alternative would be to be able to keep the variations in a separate file. This would make it easier to use the same variations for the different test cases but at the same time let the model files evolve more. This would also make it easier to allow for some browsers differences.

Chapter 6

Discussion and concluding remarks

"Only solutions that produce partial results when partially implemented can succeed" - Clay Shirky

The introduction divided this thesis into three parts:

1. Create a framework for acquiring images of web page rendering from different browsers.
2. Explore different assertion methods that may be applied to images.
3. Evaluate the usability of the framework as a tool in web development.

To do this we created Sysifos, a framework for testing images of web pages. To capture complete images of web pages, we extended Selenium RC. The images are captured by taking multiple screenshots of the browser and scrolling the page between each screenshot. The images are then cropped and then merged to a complete image of the webpage. Testing of the framework showed that using screenshots is prone to errors, timeouts and other externalities. Instead, browser specific methods should be developed.

The problems encountered in the screen capture phase reduced the volume of tests we could do, but still provided valuable inputs in showing how long the tests would take, how to organize tests and other ways to simplify the capturing process. Modifying the design would have required too much work to be done within the limits of this thesis.

The results we got from exploring different assertion methods are more encouraging. Our results show that using simple pixel by pixel comparisons do not work well, but creating a more advanced test oracle gave results.

To create the test oracle, we created an algorithm that creates a model of the web page using a screen dump of the page. The algorithm works by first segmenting out pictures in the image and then finding larger blocks, as well as text characters. It arguments the segmentation step with a series of merge and sort operations working on labels generated in the segmentation process. Then it builds a tree where the blocks found are branches. Borders, pictures and text regions form the leaves of each branch. To compare two models, we explored existing algorithms before designing a simple node by node comparison algorithm.

We also tested Sysifos ability to find a set of known errors on a simple test site. The test was done by taking the original site and making 8 different changes to the site. 3 of the changes were within the rules for how the site may change, for example we added more text to the content area. 5 changes introduced errors into the site, for example an image that was placed on the wrong side of the text. Sysifos found 80% of the errors and did not report any false errors.

The test also showed that using SVG images to visualize the errors is both easy and provides a lot of information into what is wrong in the web page. This is consistent with Kaner[1] who promotes the idea that the goal of a test case is information, not only to expose a defect.

To create a perfect implementation of Sysifos was not possible within the time bounds of this thesis. By having a partial implementation of every step we have proved that the approach can work.

It is now time to answer the third item on the list above, to evaluate the frameworks usability for web development. To do so we will use the following criteria:

1. How long time does it take set up a test?
2. How long does it take to run a test suite?
3. How many real bugs are found?
4. How many false bugs are found?
5. How many new bugs are uncovered compared to a simpler system?

The first criteria depends on how much infrastructure is needed. The first test you set up with Sysifos requires a lot of effort compared to setting up a unit test. You need to set up a Selenium server and you also need to install the framework.

When this is done, then the equation changes a bit. Section 5.3 shows how a simple test setup may test for a large amount of variations without requiring a large amount of coding. This happens because the initial page model generates a lot of code, but also because the system creates a model that may be close to what the system designer wants to express.

With short setup time, the systems Achilles heel is the long and error prone process it uses to capture images. If the process is rewritten to use browser specific capture techniques, then it is possible that the system will be fast enough to be workable.

It is undeniable that Sysifos has the ability to spot bugs, but it is still an open question how well the system will work on more complex pages. The size of the model document grows with the page. If the page is large, then the model document will be harder to work with - thus reducing the usability of the system in a situation it should handle well.

It is also an open question how many false bugs the system will find in a real page changing under real conditions. As section 4.6.2 discusses, there is still ample scope for improving the segmentation algorithm.

The last item in the above list is harder to answer. What would a simpler system be? This thesis tries to automate manual testing of web page rendering so the simpler system would be to manually inspect the pages. If Sysifos is not improved, it is quite obvious that manual testing of rendering will find more bugs, but even if this is true it does not make Sysifos worthless.

In the book “Lessons learned in software testing” [25], the question of automated versus manual testing is discussed. One of the important points made there is that you should not expect automated tests to find most of the bugs. On page 100, the authors write “don’t compare manual tests to automated tests. Instead look at automated tests as making it possible for you to extend your reach”.

In other words, it might not matter that Sysifos doesn’t catch all bugs that a manual tester would. The sweet spot for a system like Sysifos may well be in ensuring that a web page gets tested in a browser the developer seldom uses - not as the main tool for finding rendering bugs in a system.

The introduction raised two questions that have been partially answered by this thesis:

Is it possible to use images to test webpage rendering?

The results of this project show that it is possible to automate testing of webpage rendering using a test oracle that uses a page model created from an initial image to verify its results. More work is needed to see if the method is good enough to be used in production.

Can pattern recognition or other computer vision techniques bring something to computer testing?

This question was also raised in the introduction, and in many ways it is the question that led to this thesis. A rephrasing may be in order: Is it possible to make a test system that manages to find bugs in outputs where the expectations are not explicitly stated?

The answer is no, that is not possible. But, it is possible to create a test oracle that extracts it's requirements from information rich sources, like for example an image, instead of relying on a test engineer to explicitly state them. This is what Sysifos does, and there may be other sources of requirements that other types of machine learning tools may be able to generate tests from.

6.1 Future work

There are many parts of Sysifos that may benefit from more work. The most obvious place to start is to change the image capturing mechanism from using Javas awt.robot class to use browser specific capture systems.

The page segmentation algorithm needs to be expanded to better separate text columns within a block where whitespace is used as separators. This is probably best done by inspecting the text and image regions connected to a node after the page model has been built. The algorithm also needs work when it comes to detecting gradients.

Most other methods for segmenting webpages actually combine knowledge exposed from the DOM with visual cues. This may be an interesting approach to gain a deeper insight into what elements are causing errors and also how elements should relate to each other. It may also be used to provide naming of elements in the image model.

Also there is ample room to expand on the variations available in the image model specifications as well as expand the files functionality in terms of both variables and include functionality. It would be interesting to expand the SVG images of the models so that they may form the basis of a simple graphical interface for the user to use when modifying the image model.

Appendix A

Appendix

A.1 Source code for the template page experiment.

```
1  package experiments;
2
3  import java.io.File;
4  import java.net.URL;
5
6  import no.sysifos.imageLibrary.Asserts;
7  import no.sysifos.imageLibrary.FileImageLibrary;
8  import no.sysifos.imageLibrary.ITestImage;
9  import no.sysifos.imageLibrary.SeleniumImageFactory;
10 import no.sysifos.imageModel.ImageModel;
11 import no.sysifos.imageModel.Region;
12 import no.sysifos.imageModel.RegionUtils;
13
14 import org.testng.annotations.BeforeTest;
15 import org.testng.annotations.DataProvider;
16 import org.testng.annotations.Test;
17
18 public class SimpleImageExperimentTest {
19   URL testRoot = this.getClass().getResource("/SimpleImageExperimentTest");
20   public SeleniumImageFactory seleniumImageFactory;
21   protected FileImageLibrary library = new FileImageLibrary("/tmp/test");
```

```
22    String modelPath = null;
23    String browser = "*firefox";
24    private int oneTest = -1;
25    private boolean reRequest = false;
26    private boolean saveModel = false;
27    @BeforeTest
28    protected void setUp() throws Exception {
29
30      if (System.getProperty("modelPath") != null && System.getProperty("modelPath
          ").equals("${modelPath}") == false) {
31        modelPath = System.getProperty("modelPath");
32      }
33      if (System.getProperty("oneTest") != null && System.getProperty("oneTest").
          equals("${oneTest}") == false) {
34        System.out.println("Running onetest: " + System.getProperty("oneTest"));
35        oneTest = Integer.parseInt(System.getProperty("oneTest"));
36      }
37
38      if (System.getProperty("rerequest") != null && System.getProperty("rerequest
          ").equals("${rerequest}") == false) {
39        System.out.println("Rerequesting Images:" + System.getProperty("rerequest"
            ));
40        reRequest = true;
41      }
42      if (System.getProperty("saveModel") != null && System.getProperty("saveModel
          ").equals("${saveModel}") == false) {
43        System.out.println("Saving the image models");
44        saveModel = true;
45      }
46      if (System.getProperty("browser") != null && System.getProperty("browser").
          equals("${browser}") == false) {
47        browser = System.getProperty("browser");
48      }
49
50      if (reRequest) {
51        seleniumImageFactory = new SeleniumImageFactory(library);
52        seleniumImageFactory.calibrate(browser, "wintest", 4444);
53        seleniumImageFactory.setTimeout("20000");
54      }
```

```
55    }
56
57    @DataProvider
58    public Object[][] urls() {
59      Object [][] ret = new Object[8][2];
60      ret = new Object[][] {
61          {"original" , "The orignal"},
62          { "test1", "More text"  },
63          {"test2", "two images"  },
64          { "test3", "extra menuitem"  },
65          { "test4", "wide content area"  },
66          { "test5", "image placement change"  },
67          { "test6", "changed color"  },
68          { "test7", "image overlaps boxes"  },
69          { "test8", "No content" }
70
71      };
72      if (oneTest > 0) {
73
74        Object[][] nret = new Object[][] { {ret[oneTest][0], ret[oneTest][1]} };
75        return nret;
76      }
77      //ret = new Object[][] { {"test2", "two images"  }};
78      return ret;
79    }
80
81    @Test(dataProvider="urls")
82    public void Simple3Colors(String testurl , String description) throws Exception
          {
83
84      ITestImage test , orig;
85      String url = "http://172.16.180.1/master/testpage/" + testurl + ".php";
86
87      System.out.println("Doing test: " + testurl + " ( " + description + " ) ");
88      if (reRequest) {
89        test = seleniumImageFactory.requestImageFromUrl(url , browser , "wintest",
            4444);
90      } else {
91        test = library.getTestFromUrl(url , browser , "wintest");
```

```
92        }
93        if (saveModel) {
94          ImageModel imageModel = new ImageModel(test.getImage(), test.getBase());
95          Region root = imageModel.build(true);
96          // we only want to save the model when we create new a new model for a new
                browser
97          imageModel.saveModel(test.getModelPath(true), root);
98
99        }
100       /**
101        * Either use the modelPath supplied to run the tests or use the model
              defined by the original image.
102        *
103        * This variation is not needed for normal test runs.
104        */
105       if (modelPath != null) {
106         orig = library.getTestFromUrl(url, browser, "wintest");
107         orig.setModelPath(modelPath);
108         System.out.println("Using modelPath: " + modelPath);
109       } else {
110         orig = test;
111         test.setModelPath(library.getTestFromUrl("http://172.16.180.1/master/
               testpage/original.php", browser, "wintest").getModelPath(true));
112       }
113       Asserts.assertModel(orig.getModel(false), test.getModel(true),orig.getImage
            (), orig.getBase());
114     }
115
116       /**
117        * This test compares the differences between the models generated by images
              from two different browsers.
118        */
119     @Test(dataProvider="urls", enabled=false)
120     public void CrossBrowserTest (String testurl, String description) throws
            Exception {
121       ITestImage test1, test2;
122       String url = "http://172.16.180.1/master/testpage/" + testurl + ".php";
123       test1 = library.getTestFromUrl(url, "firefox", "wintest");
124       test2 = library.getTestFromUrl(url, "opera", "wintest");
```

```
125     Asserts.assertModel(test1.getModel(true),
126                         test2.getModel(true),
127                         test1.getImage(),
128                         "/tmp/browserResults/" + testurl);
129   }
130 }
```

A.2 Package versions

The following external packages have been used:

Program	Version
TestNG	5.7
Selenium RC	0.9.3
ImageJ	1.38
XStream	1.2.2
CruiseControl	2.7.1
Apache Continuum	1.1
Apache Ant	1.7
Apache Maven	2.0.8

Bibliography

[1] *"What is a good test case?"*, 2003.

[2] Selenium. Web page, Jul 2007. Site accessed on 19. July 2007.

[3] Wikipedia: The srgb colorspace. On Wikipedia, last checked 26.3.2008, 2008.

[4] M.D. Abramoff, P.J. Magelhaes, and S.J Ram. Image processing with imagej. *Biophotonics International*, 11:36–42, 2004.

[5] Matthew Anderson, Srinivasan Chandrasekar, Ricardo Motta, and Michael Stokes. A standard default color space for the internet: srgb. Technical report, International Color Consortium, 1996.

[6] Shumeet Baluja. Browsing on small screens: recasting web-page segmentation into an efficient machine learning framework. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 33–42, New York, NY, USA, 2006. ACM.

[7] Kent Beck. Simple smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18, 1994.

[8] Kent Beck. *Test Driven Development: By Example (Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2002.

[9] Cedric Beust and Hani Suleiman. *Next Generation Java Testing*. Addison Wesley, 2007.

[10] Bert Bos, Tantek Aelik, Ian Hickson, and Haakon Wium Lie. Cascading style sheets, level 2 revision 1 css 2.1 specification w3c working draft 06 november 2006. Technical report, W3C, 2006.

[11] Wilhelm Burger and Mark J. Burge. *Digital Image Processing, an algorithmic introduction using java*. Springer Science + Business Media, LLC, 2008.

[12] Deng Cai, Xiaofei He, Wei-Ying Ma, Ji-Rong Wen, and Hongjiang Zhang. Organizing www images based on the analysis of page layout and web link structure. *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, 1:113–116 Vol.1, 27-30 June 2004.

[13] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wen-Ying Ma. Vips: a vision-based page segmentation algorithm. Technical report, Microsoft Research, Microsoft Corporation, November 2003.

[14] Sudarshan S. Chawathe, Anand Rajaraman, Hector G. Molina, and Jennifer Widom. Change detection in hierarchically structured information. In H. V. Jagadish and Inderpal S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 493–504. ACM Press, 1996.

[15] The Counter. Checked 17th March 2008, 2008.

[16] Jeffrey Friedl. History of color mis-management. Authors blog, October 3rd, 2006. Last checked 26.3.2008., 2006.

[17] Anthony Y. Fu, Liu Wenyin, and Xiaotie Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *Dependable and Secure Computing, IEEE Transactions on*, 3(4):301–311, 2006.

[18] Eric Gamma and Kent Beck. Junit. Website, July 2007.

[19] W3C HTML Working Group. Xhtml 1.0 the extensible hypertext markup language (second edition). Technical report, W3C, 2002.

[20] Douglas Hoffman. Heuristic test oracles. Software Testing & Quality Engineering, March 1999.

[21] A. Holmes and M. Kellogg. Automating functional tests using selenium. *Agile Conference, 2006*, pages 6 pp.–, 2006.

[22] Ron Jeffries and Grigori Melnik. Guest editors' introduction: Tdd–the art of fearless programming. *Software, IEEE*, 24(3):24–30, 2007.

[23] Cao Jiuxin, Mao Bo, and Luo Junzhou. A web page segmentation algorithm based on iterated dividing and shrinking. *Network and Parallel Computing Workshops,18-21 Sept 2007. NPC Workshops. IFIP International Conference on*, pages 701–705, 2007.

[24] Cem Kaner. Improving the maintainability of automated test suites. Paper Presented at Quality Week 1997, 1997. http://www.kaner.com/lawst1.htm.

[25] Cem Kaner, James Bach, and Bret Pettichord. *Lessons learned in software testing*. John Wiley and Sons, 2002.

[26] D. Karatzas and A. Antonacopoulos. Colour text segmentation in web images based on human perception. *Image and Vision Computing*, 25:564 – 577, 1 May 2007.

[27] Laurence R. Kepple. The black art of GUI testing. 19(2):40–?? (or 42–??), February 1994.

[28] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should i use for effective gui testing? *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 164–173, 6-10 Oct. 2003.

[29] D. Peters and D. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[30] Dave Raggett. Html 4.01 specification. Technical report, W3C, 1999.

[31] Chris Roast. Viewing visual web site design in context. In *DUX '05: Proceedings of the 2005 conference on Designing for User eXperience*, page 15, New York, NY, USA, 2005. AIGA: American Institute of Graphic Arts.

[32] Guangfeng Song. Analysis of web page complexity through visual segmentation. In Julie A. Jacko, editor, *HCI (4)*, volume 4553 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2007.

List of Figures

1.1 Browser terms. The viewport is the part of the browser that shows you the web page. The program controls are often called chrome. Also notice that there is more content on the page if you scroll. 17

2.1 Some of the standards implemented by a modern browser. 24

3.1 System overview. The test runner creates the test instance that runs the test. The test instance fetches an image and compares it to the original. If errors are found they are reported back to the test runner. 30

3.2 A simple example of a test making sure that the top of www.uio.no doesn't change. . 32

3.3 Agent overview. 33

3.4 An example of how the image storage API is used. 35

3.5 Standard test workflow. Apart from writing code you also need to get the initial image and generate the imagemodel. 41

3.6 Signatures for the single image assert methods. 42

3.7 This listing shows how a new model is created and saved. 43

3.8 Sample output from assertModel showing which areas have changed. Red lines show where the blocks were found in the new image while green lines show where the corresponding blocks were in the original image. Brown lines are places where the two conflicting regions overlap. 45

3.9 An empty model representation. 46

4.1 The top of www.siste.no 50

4.3 Do the images belong to each other or to the text below? The spacing differs with one pixel. 54

4.2 Is it the menu to the left - or the article listing to the right that “owns” the whitespace between the two? 54

4.4 Overview of the segmentation and region relation algorithm used for generating the page model. 56

4.5 The URLs the test images were captured from. 57

4.6 The different images used while developing the image segmentation and labeling algorithm. Note that only a third of the image is shown for the largest one. 57

4.7 Testimage and the manually made marker image 58

4.8 The pixels within the plus are the pixels we are segmenting. The bold pixel is the center of the plus-shaped window. 59

4.9 Two images showing the results of different threshold/grid size combinations. 60

4.10 The segmented image before labeling the different picture regions. The white pixels belong to pictures. 62

4.11 The original image with the picture regions removed. This is the pictureless image that is used when segmenting based on color. 62

4.12 Two of the images generated from the color segmentation. Both have also been labeled. 64

4.13 Average font coverage for the most used fonts on the web. 65

4.14 Number of regions generated from three testimages. Text regions were merged to text boxes. The last column is the number of colors used in the segmentation algorithm. . 67

4.15 The different regions before merging. The blue regions are image regions, red are boxes and green are text regions. 68

4.16 Different ways two regions may be related. 69

4.17 Example of a set of regions and the graph generated. The thicker lines are the ones we want to remove so that we end up with a simple tree structure. 71

4.18 The final regions for novap.no. The red regions are boxes, the blue are pictures and the yellow are text boxes. 73

4.19 The final regions for agderposten.no. The red regions are boxes, the blue are pictures and the yellow are text boxes. 74

4.20 The final regions for siste.no. The red regions are boxes, the blue are pictures and the yellow are text boxes. Note that the image has been cropped. The complete image is 1 meter long. 75

4.21 A fragment of the model representation that is written to disk. 76

4.22 Two very simple testimages. Each image generated a model representation of 20 to 25 nodes. 77

4.23 Pseudocode for the recursive function used to compare two nodes 78

4.24 Adding variations to the page model makes it more flexible. 79

4.25 Two boxes, and the possible ways to define how they differ in the horizontal (x) dimension. 79

5.1 A lighthearted look at the frustrations of web development. 84

5.2 Bug database categorization results. Internal projects refer to bugs that were directly related to specific internal projects. 85

5.3 Number of browserbugs related to different browsers. 86

5.4 Number and type of errors reported in the first experiment. The right hand scale is the number of test cases for each test run. 89

5.5 The original page 92

5.6 The different test images. Each number corresponds to the number in the test id. . . . 94

5.7 Adding maxHeight to the box defining the content area. 95

5.8 Adding an extra image object to the markup to define where the image may be placed, and its size. 96

5.9 By adding minTextHeight we are demanding that there should always be text in the content area. 96

5.10 Added minX values to the menu element so that they may be pushed to the left by another element. 97

5.11 The complete model after adding the needed changes to it. 98

5.12 Results from running the tests. T0 is before modifying the model, while T1 - 4 is after each of the steps above. To pass a test must have 0 errors. 99

5.13 A screenshot with errors marked. The green lines show where the original box was. The red lines show where the box is in the new image. Brown lines are where the two boxes share the same placement. 100

5.14 The original page in Firefox and Opera. When compared they generate errors. 101

5.15 Type 1 and type 2 errors. 102