Detecting behavioural changes when refactoring a web-based legacy system

Peter Spegel

May 21, 2015
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Mikael Rännar
Examiner: Fredrik Georgsson

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

Legacy code suffers from poor readability and testability. This together with ever changing business requirements leads management and development teams to prioritize quick fixes over risky restructuring of working code. It is clear that the technical debt accumulated through inadequate maintenance will create a sinking ship. The dilemma of having to change code to increase test coverage, to guard against the introduction of bugs withstands.

Characterization testing is a form of automated testing where the goal is to detect behaviour changes rather than to ensure program correctness. This master thesis aims to develop a tool which allows for characterization testing without first changing the code under test. Using only URL’s to create test cases for the web application and ensuring that the server response is the same before and after a refactorization.
# Contents

1 Introduction .......................... 1
  1.1 Solidar .................................. 1
  1.2 Solidar system ............................ 1
  1.3 Definitions .............................. 2
  1.4 Outline .................................. 2

2 Problem Description ................. 3
  2.1 Goals .................................... 4
  2.2 Purpose .................................. 4
  2.3 Related Work ......................... 4
    2.3.1 When to automate tests .......... 5
    2.3.2 Automated testing tools ......... 6

3 Refactorization and characterization testing ......... 7
  3.1 Rewrite .................................. 8
  3.2 Refactorization ........................... 8
    3.2.1 Breaking dependencies .......... 9
    3.2.2 What needs to be refactored .... 11
  3.3 Software testing .......................... 14
    3.3.1 Bug definition ...................... 15
    3.3.2 Testing and refactoring .......... 15

4 Accomplishment ....................... 19
  4.1 Preliminaries ............................ 19
  4.2 How the work was done ............... 19
    4.2.1 Literature study .................. 19
    4.2.2 Evaluation ......................... 20
    4.2.3 Implementation .................... 20
    4.2.4 Report writing .................... 20


5 Results

5.1 Existing tools for automated testing

- 5.1.1 Apache Jmeter
- 5.1.2 ASIS
- 5.1.3 Golden master testing
- 5.1.4 Goutte
- 5.1.5 HTMLUnit
- 5.1.6 Node-replay
- 5.1.7 PhantomJS
- 5.1.8 Record and replay
- 5.1.9 Steam
- 5.1.10 Throwback
- 5.1.11 ZombieJS

5.2 Creating a prototype

- 5.2.1 Test database
- 5.2.2 Comparing page content

5.3 Characterization testing with the prototype

6 Conclusions

- 6.1 Limitations
- 6.2 Future work

7 Acknowledgements

References
Chapter 1

Introduction

Any system which grows organically as new business requirements emerge while being developed by different people over time with varying skill sets, will inevitably suffer from an increasing amount of technical debt. In time the system will become costly to maintain if the code base is not constantly refactored to an acceptable standard. Furthermore, developing in an immature language requires more from the developers in terms of structure and good practices.

Newly designed parts of the system may hold a high standard whereas large areas fall under the description legacy code. Legacy code is commonly defined as code inherited from someone else which has taken on more shades of meaning and weight over time. It can also be defined as code that is hard to cover with automatic tests [15].

This poses a problem to the developer since the code should ideally be covered with tests before any refactoring is made. Writing proper tests for a legacy system is often infeasible due to lack of separation between different software layers, uncertainties about the interactions with other components and the sheer amount of code that has to be covered. Feather [15] calls this problem The Legacy Code Dilemma. One possible solution suggested by Feathers is to ensure that the system behaves the same before and after being refactored rather than trying to figure out how it should work, a process called characterization testing.

This master thesis aims to describe a method of performing automated characterization testing in a web based system by recording responses from a web server and comparing these with the responses subsequent to a wide spread refactoring of the system.

1.1 Solidar

Solidar is a corporate group with its main office in Umeå, Sweden. They offer active administration of premium pension, occupational pension, contractual pension and other types of savings/insurances for private persons.

1.2 Solidar system

Solidar AB owns a web-based, administrative system for managing customers, products, portfolios, circulars, etc. Customer service, insurance advisers, after-market and other employees use this system to administrate Solidar’s roughly 100 000 customers. The application is written in PHP with a MySQL database and is being developed using Scrum methodology.
1.3 Definitions

**API** Application Programming Interface, expresses a software component in terms of operations, input, output and underlying types, independently of the implementation details.

**BDD** Behaviour Driven Development, a common language for developers and stakeholders to describe software changes which can be interpreted into automated tests.

**Headless browsing** Simulated browser environment where the content is not rendered on the screen for a human to see.

**IDE** Integrated Development Environment, a software application that supports the developer when editing source code.

**IEEE** Institute of Electrical and Electronics Engineers, a professional association focusing on educational and technological advancements in the field of electrical and electronic engineering, telecommunications, computer engineering and allied disciplines.

**Factored test** Tests executed on a subset of a system after augmenting it by mocking surrounding components or using some other dependency breaking method.

**Scrum** A methodology for improving agile development processes, emphasizing iterative development resulting in potentially shippable increments that are ”done” according to some agreed upon understanding.

1.4 Outline

This is an overview of the chapters in this master thesis.

**Chapter 2** contains the problem description as well as the goals for the master thesis.

**Chapter 3** is an in-depth study of software testing and refactorization.

**Chapter 4** describes the work process of this project and what was accomplished.

**Chapter 5** contains the results of the thesis project.

**Chapter 6** summarizes the conclusions drawn from the project.

**Chapter 7** is the acknowledgements.
Chapter 2

Problem Description

Solidar is a rapidly growing corporation [6] with an ever changing business model as new ways of packaging products or entirely different products emerge. External dependencies including legal changes, customers demanding administration of additional products, new broker deals or insurance companies making policy changes may also affect the business model. These changes will have to be reflected by changes to the administrative system employed by Solidar, simply called Solidar system.

As different people are given the task of implementing features they may not be able to fully understand the existing code due to time restrictions or bad readability. The lack of test coverage may lead the developer to make only the necessary adjustments to meet the requirements since a refactoring process could break the existing logic. Naturally the complexity will increase as the code is given more responsibilities but it will also reduce the readability of the code since new layers and exceptions are added on top of everything else. Knowledge about the original intent of the code may be lost as people switch jobs and it becomes increasingly difficult to trace the requirements the implementation originated from.

To deal with the legacy code Solidar employs a refactoring process outlined in the book *Modernizing Legacy Applications in PHP* [21]. The author describes how a typical legacy PHP application can be transformed to a quality application step by step. It relies heavily on the methods described by Feathers and introduces solutions for common problems in PHP. These steps are small in themselves but each may result in changes in many places throughout the code base.

Adding tests posthumously is a time consuming task, especially when many files are involved and functions cannot be easily decoupled. Since it is a refactoring process, no changes should be made to the behaviour of the system which makes characterization testing (see section 3.3.2) appropriate for the job. If it can be shown that the output of the system is unaffected in enough places, the developer will be more confident that nothing has been broken. However when the changes are scattered throughout the code it is hard to determine where to put the assertions that will detect changes.

At present Solidar has a dedicated tester in the development team who is integrated into the development process and signs off on features using primarily exploratory testing. Automated testing has so far consisted almost exclusively of unit testing with the PHPUnit framework. Some attempts have been made to introduce higher level automated tests but they have been unsuccessful due to the fragility of their nature.
2.1 Goals

This project aims to automate the process of asserting that no changes have been made by recording the actual output of the system in a stable state, recording the output after some refactoring and then compare the results to detect differences. Since Solidar system is a web application most of the output is either HTML or JSON which can be requested with HTTP GET or POST. A tool will be developed that can visit a set of URL:s and record the content or compare the current content with previously recorded data. There will inevitably be some differences that the tool will have to disregard from (i.e. timestamps or session related data).

Making a POST request will in many cases result in database write operations. This presents a problem since tests should be mutually independent, meaning that the run order should not matter and the result is expected to be the same for each run. Consequently the database will have to be restored between usages of the tool, otherwise unexpected results will follow. The developers must also be able to do other work in parallel with the refactoring (i.e. prioritized issues that may arise). This is one of the reasons why a separate test database will have to be used and why the developer must be able to quickly switch between databases.

The tool will not be able to compare other data than pure text (mainly HTML/JSON), it will not provide any guidance to why a change has occurred and it will not evaluate the importance of testing a certain URL. Part of the project will be to determine when it is feasible to use automated characterization testing and when it is better to use another method, such as manual exploratory testing.

2.2 Purpose

With a tool for automated characterization testing the developers at Solidar should be able to quickly detect changes in the behaviour of Solidar system as they are refactoring its legacy code. The main advantage should be spot checking of the whole system, identifying unforeseen dependencies to other parts of the application. In the early stages of the refactoring process outlined in [21] the developers will make a lot of small changes in many files which would be tedious to test manually.

2.3 Related Work

Characterization testing is a rather new concept, introduced by Feathers [15] in 2005. Automated tests are preferably implemented at a low level which may be the reason why most tools focus on unit testing. The main reasons why higher level tests should be avoided is low robustness and maintainability as argued by Agile alliance co-founder Mike Cohn [9] in his blog. Crispin and Gregory [17] have also arrived at the same conclusion in their book Agile Testing: A Practical Guide for Testers and Agile Teams. The amount of tests at each level is illustrated in figure 2.1.

However, even though some early characterization tests may be improved into real unit tests over time, they are wear and tear in their nature and thus it could be argued that characterization testing at any level should be useful. In a study by Karhu, Repo, Taipale and Smolander [22] it is observed that when investing in testing automation, there should be possibilities to reuse the tests over a long period of time to get a return on the investment.
2.3. Related Work

In this project the investment is in the tool and the URL:s under test and those are both reusable whereas the test itself is cheaply generated and thrown away after its use.

In the article *Agile Regression Testing Using Record and Playback* [26] the author describes the troubles with adding low level automated tests to legacy code. The code is not written with testability in mind and has to be refactored to some degree before it can be covered with tests. This process may in itself result in new errors or other unwanted changes. This leads to the conclusion that the process of adding tests before refactoring should be non-invasive.

2.3.1 When to automate tests

Marick [23] lists three considerations that should be taken when automating a test.

1. Automating a test and running it once will cost more than performing it manually. It is interesting to think about how much more.

2. An automated test has a finite lifetime, during which it must recoup that additional cost. Is it possible to anticipate the events that will lead to the test being discarded?

3. How likely is it that the test will find additional bugs if it is re-run later?

In the same paper Marick also mentions several secondary considerations, some of which are listed below.

1. An automated test’s value is mostly unrelated to the specific purpose for which it was written. It is the accidental, untargeted bugs it finds that counts.

2. Humans notices bugs that automation ignores. I.e. a human may notice that a dialog is off-screen where the automated test may not.

3. While humans are good at noticing oddities, they are bad at highly repetitive tasks or reviewing very precise results.

4. Human makes mistakes as they test which may lead them to stumble across bugs as they take different paths through the application.
5. Bugs found by automated tests are often easy to reproduce, which is important when fixing them.

6. It is impractical to test the whole system using manual testing, whereas automated tests can be re-run with each build.

Sommerville [34] divides software into two broad classes; Generic and customized products. Generic products are stand-alone systems that are sold to an open market and customized products are commissioned by a particular customer or written to support a particular business process. Solidar system is most certainly a customized product since it is an in-house application supporting the unique business requirements of the company.

The authors of *Empirical Observations on Software Testing Automation* [22] observe that automated testing systems have to be supervised in case of faults. They claim that some testing tasks are difficult to automate, especially if they require extensive domain knowledge. According to their observations the need for domain knowledge was extra high for customized systems.

### 2.3.2 Automated testing tools

There exists many tools for automated testing oriented towards regression, functional or characterization testing which can be categorized in the following manner.

**Approval test** Lets the developer visually approve output from an implemented test rather than using written assertions.

**BDD test** Interprets a BDD story as user interactions and asserts that the described behaviour is followed.

**Record and replay** Records browser interaction, input and output to a function or screenshots of the application. The interaction can be replayed later to verify that the application behaves the same.

**Navigation scripting** Similar to unit testing, the developer can write tests that manipulates a headless browser and makes some assertions.

Crispin and Gregory [17] recommends focusing on one tool at the time, addressing its greatest area of pain and giving it enough time for a fair trial before moving on to the next one. The option of developing a new tool should only be entertained if the team has sufficient skills, bandwidth and unique testing challenges in front of them.
Chapter 3

Refactorization and characterization testing

Martin uses the following analogy to describe clean code.

The bad news is that writing clean code is a lot like painting a picture. Most of us know when a picture is painted well or badly. But being able to recognize good art from bad does not mean that we know how to paint [24].

No one is deliberately writing bad code, that would be unprofessional and meaningless. At some point after a piece of code has been written someone will have to read it and as in the analogy above, that person would probably recognize that the code was poorly written. A developer spends a lot more time reading code than writing it - more than ten times as much time according to Martin [24]. The Software Productivity Consortium [12] also conclude that code potentially has to be understood many times and written only once.

Code must naturally be read before making changes to it, either to support a new feature or to correct some bug in the system. A developer must also often answer questions about the code and how it behaves in certain situations. Depending on who is asking the question the developer must be able to not only find the related code but also talk about it on different levels.

Saha [31] divides code understanding into two main perspectives: application-level understanding and programming language-level understanding. The first revolves around understanding the business logic and what side effects it may have. The latter is a lower level understanding of the implementation details such as data structures, algorithms, API:s and naming conventions.

Another common reason for reading code is peer reviews, where one or more developers read newly produced code generally created by someone else. Nevertheless, readability is overlooked in many cases in favour of other concerns. Saha [31] argues in a conjectural short paper that while peer code reviews are often present, the reviewers seldom flag for bad readability.

In a Qualcon paper, Relf [30] mentions sources stating that half of a programmer’s time is spent reading code. At Solidar, readability is prioritized over many other concerns and it is one of the criteria to pass a code review before moving on to test. The informal guideline is to never sacrifice readability to get the job done quicker. Saha [31] writes that since readability cannot be standardized as a process, the alternative is to have a culture among
the programmes to write readable code. This is an attitude that Solidar has been trying to embrace lately, creating a quality culture around their collective work.

Solidar has observed that new developers take a long time to get acquainted with the system. Rough estimations made by the developers suggest that it takes up to six months before a person can be productive and work almost independently, given that the person has experience with the programming language and database environment. It may even take a whole year for a new developer to get comfortable enough to be really creative.

This time line originates not only from bad readability but also from the intimate knowledge of the business required to develop Solidar system. Since Solidar follows an agile process, not much of the business rules are documented. The code and unit tests are the documentation and if they are hard to read and navigate in then a new developer will have to resort to asking questions to senior staff.

Even though questions are happily answered, Solidar is concerned that it will reduce the throughput of the Scrum team in the short term. A study by LaToza, Venolia and DeLine [35] conducted at Microsoft shows that interruptions such as these is the second most serious problem to developers.

It can be argued then that making the code easy to understand is an important task. It is however not enough to merely create new, readable code. The majority of the code would still have poor quality which leads to problems for each developer working on it. The developer would be slowed down trying to understand the old code, making compromises to interface with it and maybe even be prevented to get test coverage of new code utilizing it.

### 3.1 Rewrite

As creative beings, it is tempting for developers to solve this problem by suggesting a rewrite. The argument is that the old code could serve as a reference implementation, there would be no obstacles preventing test coverage, unused parts could be removed and the rest restructured. Jones [21] describes the pitfalls of this method, giving examples of companies getting catastrophic results after undertaking a rewrite. Some of the problems with this approach are as follows.

- The developers are overconfident in their knowledge in the old system as well as their ability to write it better than the last time.
- The developers are easily sidetracked by the prospect of solving problems in new ways. They may even be tempted to add new features while they are at it.
- The old code will still have to be maintained while it is rewritten. The productivity will be affected by the limited resources that has to be distributed for both projects.

### 3.2 Refactorization

A more subtle approach is to rewrite or restructure the code incrementally by taking small steps while verifying that the code changes do not affect the behaviour of the system. Feathers [15] calls the act of improving design without changing its behaviour refactoring. He describes the process as a series of small structural modifications, supported by tests, which do not result in any functional changes. Fowler, et al. describes refactoring in the following way.
Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written [14].

Feathers and Fowler both emphasize that refactoring is not a simple code clean up but a structured process where the quality of the code is increased gradually. Nor is it a rewrite in the sense that the changes are made to smaller chunks of code within the same project and they are continuously being worked into the production code. Feathers suggests the following algorithm for making changes to legacy code.

1. Identify change points
2. Find test points
3. Break dependencies
4. Write tests
5. Make changes and refactor

First the developer identifies where the change must be made to support the new feature. In order to make a change, tests must be written, but in legacy code a lot of thought has to go into determining where to add tests as well. Often times, some changes has to be made before it is even possible to write a test. Methods of doing these changes are presented in section 3.2.1.

When tests are in place that specifies the current behaviour of the system, the new functionality can be developed using a test driven approach, altering the existing tests as needed. Since both the existing behaviour and new functionality is covered by tests the developer can refactor the code with confidence.

### 3.2.1 Breaking dependencies

Breaking dependencies is a necessary step before tests can be put in place. There are generally two reasons for this: sensing and separation. The developer may have to break dependencies to sense if calls to the code under test has some effect which cannot be evaluated. Additionally, the developer may need to break dependencies to separate the code under test from its context before it can be covered by tests [15].

**Sensing**

The predominant way of sensing according to Feathers is using fake or mock objects. A fake object mimics the characteristics of the object class used in production, having only the necessary properties and methods required to perform a test. With a fake object unwanted side effects can be avoided by omitting code that is run in production which does not relate to the test at hand. It is also easier to expose internal properties which can be asserted by the test framework to evaluate expected effects of the test.

Mock objects are more powerful than fake objects since they allow internal assertions within the object as part of the test. This allows the developer to assert calls to functions of the object and mock the response to create a desired scenario. Mackinnon, Freeman and Craig [36] express the difficulties of setting up a complex system state needed to test a
specific scenario, where mock objects can be used to emulate this state. They also point out that assertion reuse is simpler when kept internally in the mock objects, rather than being duplicated across multiple test cases. Setting up mock objects before the execution of test cases enables the developer to provide default values for an entire test suite. As Martin [24] points out, mocking may also be needed to omit calls to third party components.

Mackinnon, et al. [36] have found that developing with mock objects encourages a coding style where objects are passed into the code that needs them. Consequently, mock objects can be used both to avert the need for breaking dependencies in the future and to sense where dependency breaking is needed posthumously. When dependencies have been dealt with mock objects can be inserted through dependency injection.

As with any code, mock objects which are set up wrong or not kept up to date can result in multiple problems which do not occur in production [36].

- The system may enter an unexpected state due to the misconfigured state of the mock object. As stated in a paper by Saff and Ernst [11], a factored test introduces assumptions about the implementation details of the functionality being tested and if these assumptions are violated the factored test becomes useless.

- Similarly, the mock object may return unexpected values if its functions are replaced with mocked behaviour.

- When mocking complex components such as a database, the database mock may expect functions to be called in a certain order even though they are being called independently in production. This may cause the test to fail even though it would not affect the overall behaviour in production [11].

- Depending on the test framework used there may be a lacking support of mocking non-public or static functions or properties. This is not uncommonly by design but it may also not generate a warning and the developer can be led to believe that the function or property is mocked where in reality, it is not.

Separation

Separation can be achieved in a number of ways and on different scales. One kind of separation which allows for easier testing is the Command-Query Separation principle, first described by Bertrand Meyer [27] in the book Object Oriented Software Construction. The principle states that a method should be a command or a query, never both. Meyer uses the metaphor of viewing an object as a machine. As a command is given, the machine enters a new state but it cannot be observed directly because that would require opening the machine. On the other hand, a query to the machine can be issued to yield a response but it does not alter the state of the machine in the same way that asking a question does not change the answer.

If a function is both a command and a query it would mean that it has side effects which are encapsulated by the object and thus invisible to a test. This is an example where it would be better to separate the responsibilities of a single function and thus it illustrates separation on a small scale.

There are also methods of separation which minimize the need for altering the code under test. In object oriented languages troublesome functions can be overridden by subclassing the class which contains it. If a global function is called then a new function with the same signature can be created, which in turn calls the global function. It is then possible to create
3.2. Refactorization of Code

A subclass which is only used by the test, overriding the class function by not calling the global function. Feathers [15] calls this method of breaking dependencies an *Object seam*.

Feathers [15] mentions that developers should be careful when creating non-virtual functions or sealed classes. The idea is often to prohibit misuse of a class or library by restricting polymorphism, but it also makes it hard to test since the technique above cannot be adopted. Feathers calls this the *Restricted Override Dilemma*.

On the other end of the scale are large classes with many functions and responsibilities. The separation methods above may be used to encapsulate certain behaviour which is not desirable under test but over time it gets hard to see what is going on inside. Each subclass adds to the amount of code which has to be considered when making changes or creating a new test. Large classes may also have a large amount of instance variables which are used in many different functions within the class.

To deal with large classes, some sacrifices have to be made and invasive techniques used to get them under test. If an unclear function is particularly interesting then individual responsibilities of that function can be identified and moved to separate functions. These new functions should be developed using *Test-Driven Development* (TDD) to cover parts of the original function. This may leave the original function in an odd state, especially if only small portions of it could be extracted. Another downside is that the class will still contain the same amount of code. The key advantage according to Feathers [15] is that a clear separation of new and old code is created and that there is a clean interface between them.

A large class can have many dependencies and be difficult to instantiate. If this is the case then the extracted function can be made static and the necessary instance variables passed as parameters. Martin argues that a good static method does not operate on a single instance, all its data is supplied through arguments, it does not own any objects and it is improbable that it will ever be polymorphic. Martin [24] further states that in general non-static functions should be preferred to static functions. The same recommendation is given by Jones [21] in *Modernizing Legacy Applications in PHP*. Static methods are generally considered bad from a testing perspective since they probably cannot be mocked.

A better alternative in this case is to create a whole new class to contain the function. The parameters can be passed to the constructor of the class and the extracted function developed using TDD. The new class becomes responsible for a single action performed in the original function and thus easier to understand. If possible, a factory object should be passed to the original class responsible for creating instances of the new class [21]. Factory functions are easy to mock or override during testing.

3.2.2 What needs to be refactored

Code is a constant work in progress, as pointed out by Stack Exchange co-founder Jeff Atwood:

> You should be unhappy with code you wrote a year ago. If you aren’t, that means either A) you haven’t learned anything in a year, B) your code can’t be improved, or C) you never revisit old code [2].

Consequently code must always be refactored, however there are time constraints to the work developers do which force people to prioritize. Different programming and scripting languages have different pitfalls which must be avoided. There are common guidelines to how general design considerations should be addressed.
Chapter 3. Refactorization and characterization testing

General guidelines

A class is a central software element in object oriented languages which describes an abstract data type and its (partial) implementation. A class has a list of operations which can be performed on its objects and the properties of these operations [27]. A class should only have a single responsibility according to Martin. Its name should reflect this responsibility which helps when determining if the class has an appropriate size [24]. Both Meyer [27] and McConnell [25] recommend that classes named after verbs should be avoided as behaviour alone is not enough to form a class.

Having a single responsibility does not infer providing a single service. Meyer [27] states that a class should offer a number of features related to the objects it encapsulates. Furthermore it should have procedures which modify these objects and not only queries. A study by Basili, Briand and Melo [37] showed that C++ programs with a higher number of routines per class were associated with higher fault rates, although there were more significant factors.

Related to the number of functions is the number of instance variables, which according to Martin should be kept at a minimum. Both Martin [24] and McConnell [25] argue that each function should manipulate as many of the instance variables as possible, aiming for high cohesion. This will effectively limit the number of public functions and make sure that they support a central purpose. Martin [24] has found that variables and utility functions should have private visibility within the class but it is acceptable to break encapsulation to support testing.

Functions should have intention revealing names according to Martin [24]. McConnell [25] recommends that all outputs and side effects should be described in the function name. He has also found that good function names tend to be longer. Martin [24] stresses the importance of having functions that only does one thing, without side effects. Meyer [27] defines a concrete side effect of a function as either an assignment or creation instruction whose target is an attribute, or a procedure call from within the function. Meyer [27] further argues as Martin [24] does that functions should not produce side effects since it makes it harder to reason about the software.

Opinions differ about the number of arguments a function should take. Martin [24] is an advocate for having as few as possible, preferably three or less. McConnell [25] points to psychological research which suggests that people generally can only keep track of seven pieces of information at any given moment and thus it could serve as an upper limit to the number of arguments. Meyer [27] simply states that the more arguments, the more has to be remembered.

Regarding the arguments of functions, Meyer introduces the Operand principle which states that with some exceptions, a routine should never take an option as an argument. A routine being either a procedure which do not return a value, or a function which does. Meyer [27] argues that options can be provided using separate option-setting procedures without trading argument complexity for call complexity. A possible intermediate solution is the use of optional arguments which are supported in some languages, including PHP.

The subject of code commenting is widely discussed in the developer community. Some, including Martin [24] and the Visual Studio Magazine columnist Peter Vogel [38], claim that comments should be kept at a bare minimum. It is a necessary evil in some cases but the focus should be on writing self-documenting code. In an article for the ACM Queue magazine, Jef Raskin [29] discourages the use of in-line comments in favour of longer comments. Raskin claims that comments are essential since code cannot explain the rationale behind the selection of a specific solution to a problem. Fluri, Würsch and Gall [3] studied the danger of not maintaining comments as changes are made to the related code. They
3.2. Refactorization

deemed it equally counter-productive to have outdated comments as having no comments at all. However the study showed that programmers in the open source projects they reviewed were more disciplined than anticipated.

**PHP specific problems**

The following problems are some of the most common in PHP. They may exist in other languages as well and some of the problems described here are common to all object oriented languages.

In PHP, script files have to be loaded at run-time before functions and classes contained within them can be called. In unfactored script files the list of loaded scripts grows with each dependency to another class. This incurs a small performance penalty since a whole tree of dependencies are loaded even if only some of them are used.

It is not uncommon to find load commands spread out across the class in each function that has some dependency to some other class. This may reduce the set of loaded script files but it makes the dependencies harder to maintain since the load commands are scattered across a perhaps very lengthy script file. If a load command is missing when an object is instantiated or a function called then the application will produce a fatal error.

To overcome the problem of maintaining the numerous dependencies among classes often present in legacy code, an autoloader can be implemented utilizing the `spl_autoload_register()` function. With it, callbacks can be registered to load a script file whenever a class is required at run-time. To predict where the script file is located in the filesystem the structure must conform to one of the PHP Standards Recommendations (PSR-x). This is the first step outlined in *Modernizing Legacy Applications in PHP* [21].

Object orientation has been around in PHP since the late 1990's but it is still not strictly necessary to place functions or variables in a class context [1]. Functions or variables declared outside of a class are global to the application as soon as the script file accommodating them has been loaded. Functions can simply be called by its name and variables can be accessed using the `global` keyword.

Global objects are hard to test because of the way they are being used. Many functions may depend on them in various ways which are hard to find without a powerful IDE and they may even be aliased or overwritten anywhere in the application at run-time. A unit test in PHP can replace global objects with mocks but since multiple functions may use them for different purposes the chain of assertions becomes difficult to understand and maintain.

Any time a new object is created a dependency is created to the class of the object. When covering a class with tests it is best to focus on the class under test alone and leave the testing of the dependencies to their respective testing classes. The test needs to get control of the instantiation to mask the calls which are not desirable to test. This can be achieved through any of the dependency breaking techniques mentioned in section 3.2.1.

The majority of web applications usually communicate with some kind of database to persist information. Solidar employs a MySQL database to store all their non binary data, making calls to it through a global object from most legacy classes. Databases present the same challenges as global objects or third party libraries. The individual calls are hard to mock and if left alone test results can be expected to differ between runs since the test looses its statelessness.

A simple way to group queries and calls to the database in a structured way is to create a gateway class for each entity in the database. Instead of manipulating a global object directly, a gateway can be instantiated through a factory which knows about the global database object. The gateway exposes functions that runs encapsulated queries and
returns the result as objects, providing a sensible abstraction of the persistence layer. Since
the gateway is created using a factory a test can inject a mock gateway which returns
predefined results instead of interacting with the database. Jones [21] mentions a couple of
other benefits of having gateways, including reducing the number of repeated query strings
and the isolation of security flaws.

The presentation layer in a PHP application often consists of page scripts with inter-
mingled HTML and PHP code unless some templating engine is used. It is easy to break
the flow of HTML and insert PHP snippets where needed to present some complex data,
possibly from a database. Jones [21] recommends extracting the domain logic from page
scripts for the same reason database logic should be extracted to gateways. If moved to
classes, the domain logic is more easily tested and reused if necessary. The reverse problem
of having presentation logic in the business logic is also undesirable since it hinders reuse
and possibly introduces side effects to functions.

3.3 Software testing

In the book Guide to Advanced Software Testing the author lists a number of definitions of
the term testing as given in different industry standards. The IEEE Standard Glossary of
Software Engineering Terminology uses the following definition.

The process of operating a system or component under specified conditions,
observing or recording the results, and making an evaluation of some aspect of
the system or component [13].

The definition from the IEEE Test Plan Outline as given in the same glossary is as follows.

The process of analyzing a software item to detect the differences between ex-
sting and required conditions (that is, bugs) and to evaluate the features of the
software items [13].

Jonassen [18] points out that these definitions and the other ones listed in her book seem
to agree that testing is a process where a software in this case is operated, exercised or
analysed to observe results with the objective of either gathering information or evaluate
it against some expectations. In its simplest form, Jonassen [18] concludes that testing
gathers information about the quality of the object under test. Bach and Bolton propose
the following definition of testing in James’ blog.

Testing is the process of evaluating a product by learning about it through
exploration and experimentation, which includes to some degree: questioning,
study, modeling, observation, inference, etc [8].

They are also adamant that the term checking is to be used when evaluating known facts.

Checking is the process of making evaluations by applying algorithmic decision
rules to specific observations of a product [8].

In their mind, testing encompasses checking but not the other way around. Checking can
be performed in its entirety by a tool whereas testing can only be supported by a tool,
it is an human activity. Another distinction from the above definitions is that testing
does not necessarily have to evaluate anything, its sole purpose may be to learn about the
software. It also worth noting that Bach is highly critical of the International Software
Testing Qualifications Board (ISTQB) whose material is the base for Jonassen’s book.
3.3. Software testing

3.3.1 Bug definition

Patton believes it is evident that the goal of a software tester is to find bugs. In addition, a tester should find them as early as possible and make sure they get fixed. According to Patton [28], a software bug occurs when one of the following five rules is true.

1. The software does not do something that the product specification says it should do.
2. The software does something that the product specification says it shouldn't do.
3. The software does something that the product specification does not mention.
4. The software does not do something that the product specification does not mention but should.
5. The software is difficult to understand, hard to use, slow, or in the software tester's eyes will be viewed by the end user as just plain not right.

Patton defines a bug in relation to a product specification, which can range from a verbal understanding or email to a detailed, formalized written document. In terms of refactoring, the specification is the production code being refactored.

3.3.2 Testing and refactoring

In his definition of refactorization, Feather mentions tests as a way of ensuring that no bugs are introduced during the refactorization. Tests can be a safety net for the developer to rely on, supplying confidence enough to make greater changes to the design by giving constant feedback about the functional state of the code. These tests can either be regression tests or characterization tests.

Regression testing

Myers, Badgett and Sandler [16] defines regression testing as the reuse of test cases after changes to other system components to determine whether the change has regressed other aspects of the program. Patton [28] uses a similar definition by simply stating that the process of rerunning your tests is known as regression testing. In the book *How Google Tests Software* [20], the authors express that regression testing can be used as a way of mitigating risk by ensuring that failures cannot be reintroduced without being detected.

Collins, Dias-Neto and de Lucena Jr. [10] argue that regression tests are especially suited for test automation where test cases are executed iteratively and incrementally after code changes. They do however mention a number of problems with introducing test automation in agile teams.

- If the team already has dedicated testers the developers may feel that it is unnecessary for them to test.
- There is a learning curve involved then introducing new tools and practices. An economical investment is also needed to acquire these tools and to train the people who will use them.
- Legacy systems are not designed for testability.
- Test automation requires dedicated resources. It should be introduced in small steps, having maintenance costs in mind, but it cannot be treated as a side project.
The benefits of employing test automation are according to Collins, et al. [10] the low cost of test execution, easy test replication and the possibility of using the regression tests for stress testing. Solidar does a fair bit of regression testing, both for Scrum story items and bug issues. The newly developed or refactored code is covered by automated unit tests through the PHPUnit testing framework. The team has a dedicated tester who employs exploratory testing (which is inherently manual) to test both new functionality and making sure that the rest works as before.

The term Exploratory testing was coined in 1993 by Cem Kramer [4] to distinguish it from other ad hoc testing. No test scripts are involved and the tests are not documented ahead of time in a test plan. The testing can be aided by tools for practical reasons. Whittaker [39] argues that exploratory testing is especially suited for agile teams since the development cycles are so short and informal.

Characterization testing

Regression testing assumes that the correct behaviour of an application is known. This is often the case but it may require finding the original specification or discussing the matter with domain experts. To guard against change the actual behaviour of an application can be recorded while in a stable state and the results used to create automated tests. Feathers [15] calls this approach a Characterization test.

Feathers [15] argues that in legacy system, what the system does is more important than what it is supposed to do. Implementing a test according to what should happen may cause the test to fail the first time it is being run because of a bug. The code must then be corrected before tests can be put in place and that may cause further bugs which may not be detected by the test. If tests can be put in place with minimal changes to the code the correct behaviour can be specified in the tests later after the code has been refactored safely. Feathers [15] proposes the following algorithm for writing characterization tests.

1. Use a piece of code in a test harness.
2. Write an assertion that you know will fail.
3. Let the failure tell you what the behaviour is.
4. Change the test so that it expects the behaviour that the code produces.
5. Repeat.

A characterization test will document the current behaviour of the system while supplying a safety net for the developers who can begin the refactoring process reasonably quick. Feathers [15] has also found that it is easier to figure out what the software is supposed to do when knowledge of the current behaviour has been acquired. An important aspect of characterization testing is that it is not black box testing. Inspecting the code can raise interesting questions about the behaviour which then can be asked through characterization testing to nail down the answers. Additionally, writing tests for a piece of legacy code may give some hint about how it could be refactored.

Feathers [15] points out that bugs will always be found while testing and they should of course be dealt with. Depending on the severity of the bug it can be fixed right away during the refactorization process or it can be formalized as a bug issue. In some cases someone may even depend on the bug being in place, a fact which means that some investigation may have to be conducted before handling it. The general rule at Solidar is to fix the bug immediately if it is in some way related to the issue the team member was working on when
it was found. Critical bugs would be fixed in a separate priority branch under version control and then be released into production after testing. For minor or unrelated bugs the team member would write a bug issue and let the product owner prioritize it in the bug backlog.

Testing a single class in this way is called class characterization. The first thing to do is to find the responsibilities of the class on a high level. Feathers recommends writing tests for simple cases first and then expand the test suite when follow up questions arise. Some of the things to look for are listed below.

- Tangled pieces of logic which are hard to understand.
- Things that can go wrong related to the responsibility of the class.
- Special input categories such as extreme values or falsey values.
- Invariants of the class which should be true during the lifetime of the class.
Chapter 4

Accomplishment

4.1 Preliminaries

The intent of this master thesis was to find a tool for automated characterization testing of Solidar system. If some part of the problem could not be solved with an existing tool, a new one would be developed that could. The tool should be available to all team members to use before and during a system wide refactoring. The work was planned as follows:

**Literature studies** The first two weeks of the project would consist of studying literature and papers in the field of software testing and refactoring.

**Evaluation phase** During the following two weeks existing tools would be evaluated and some URL’s to test would be gathered.

**Implementation phase** Five weeks of test tool implementation and setting up test data. During the last week of this phase I planned to start writing this report.

**Report writing** Another four weeks of report writing and revisiting of the literature, not including the week of writing during the previous phase.

**Presentation** The last four weeks would consist of final adjustments to the report as well as preparing an opposition and an oral presentation.

4.2 How the work was done

The work was conducted at Solidar’s offices in Umeå where I already had a working station in a room with the other members of the development team. During this period I have worked half time at Solidar as a developer in the Scrum team and the other half has been dedicated to the master thesis. Since Solidar follows a Scrum methodology the sprints have been affected since I only have been producing half as many story points with each cycle. I have however been developing a tool which should make the upcoming work more effective, hoping for a return on the investment placed into this thesis project.

4.2.1 Literature study

Much effort has gone in to keeping to the original plan and the literature study was made during the first two weeks. I did read a lot of software testing books and books on the
subject of code style and refactorization. For the testing part I focused on automated testing in agile environments and were able to find both books and papers on the subject. I did however find it hard to find reliable sources about characterization testing, apart from Feathers book, *Working Effectively With Legacy Code*.

Regarding coding style, the most prominent sources were Martin’s book *Clean Code*, *Object Oriented Software Construction* by Meyer and *Modernizing Legacy Applications in PHP* by Jones. During this time I also gathered a long list of possible tool candidates, some of which were evaluated later on.

### 4.2.2 Evaluation

Having gathered a set of promising tools I went about studying them more closely to see if they would fulfill some of the requirements for the project. I discarded the ones which was not free or which could not be interacted with through code in some manner. There are a great many tools available for testing automation and new ones were coming up constantly during the whole project. As the idea of the capabilities of the finalized product started to formalize, many of the tools could be discarded since they were limited to other ways of testing.

The literature recommended testing each tool quite thoroughly before moving on to the next one which in retrospect might have been unfitting to my time plan. This meant that I did not test as many tools as I would have wanted, discarding them at an earlier stage based on the information available about them.

### 4.2.3 Implementation

I had already begun implementing a simple testing tool during the evaluation phase so I began analysing the Solidar system database. I looked at the tables and how they were related to each other to find a way of extracting a representative amount of testing data. Since the tool should be able to test as many pages of the web application as possible to find changes I had to find a way of generating valid URL’s to test with. To provide valid input in the requests I had to interact with the database to look up suitable data.

### 4.2.4 Report writing

Being new to \LaTeX I had some difficulties setting up the editor to work with the provided template. I got it sorted after a couple of days and started writing the introduction and describing the problem I had set out to solve. I had to read a lot more than anticipated since I did not have sufficient notes from the literature study to write in a structured way. In retrospect I should have made some notes about where to find information about different areas. During this phase I doubled the amount of source material to make the necessary points.

I also did more implementation in this phase than I had anticipated, although I tried to keep it at a minimum. It was good that I started the report writing reasonably early but I would probably have extended the length of the implementation phase if time had been given.
Chapter 5

Results

During the first part of the project, a number of tools for automated testing were evaluated according to the criteria specified in chapter 2. It was not necessary to find one tool that could solve the whole problem if several tools could interact to solve different sub problems.

1. The tool must be able to compare regular output from a web application.

2. It must be sufficient for the tool to know a valid URL to test the output of that request.

3. A test of a single URL must be easy to set up. A test script for each URL is considered too heavyweight to be accepted.

4. The developer must be able to switch between different branches in version control and use the tool to compare output in different branches.

5. The tool must be reasonably easy to set up for the developers, otherwise it may be rejected.

5.1 Existing tools for automated testing

The following tools were evaluated according to the concretized criteria listed above.

5.1.1 Apache Jmeter

This is an open source Java application developed to load test functional behaviour in web applications. It has support for many types of requests including web, SOAP, REST and FTP. It does not have JavaScript support but behaves in other ways like a headless browser. Jmeter requires Java to run and is per default a GUI application but it can be run in command line mode. It does however require test plans to be set up before which are quite detailed with thread groups, listeners and assertions. Consequently, Jmeter does not conform to (3).

It does not seem like Jmeter can be used to scrape a web page for external comparison. Duplicating the assertions for each test plan is not viable. This means that (1), (2) and (5) are not fulfilled. Criteria (4) should be satisfied.
5.1.2 ASIS

Asis is an open source project focusing on testing user interface design in legacy applications. It is specifically designed for characterization testing of web applications running in PHP environments. The Asis tool records the function calls, passed arguments and the received output. The output in this case is HTML, JSON or any other serialized object.

To initialize a recording, a logger object has to be instantiated and call a named function with a set of arguments. The output is saved in an text or XML file on disk and it can then be approved through a simple call to the logger object. This implies a number of prerequisites:

- The function under test must be static or an instance of the class must necessarily be naively created without arguments. Furthermore the function must be public, however that is common in testing frameworks.

- The function under test must not depend on the system and database being in a certain state.

- The function under test must be part of a class.

Asis has no built in support to find a set of functions to test. An Aspect-Oriented Programming (AOP) framework can be used to capture calls to functions as the application is being used for actual work. That does not solve the problem of object instantiation and testing of other code than class functions. To avoid committing fully to the framework and deploying it to the production environment a separate branch could be created to log a list of function calls for future use. This implies that a developer would have to imitate an end user working in the application to create log entries which is an unnatural way of discovering use cases.

Asis fulfils criteria (1) to a degree, having support for any kind of output that can be expected. A difference in output does not generate a diff, however that could be solved by wrapping the test with another tool that presents the results differently. Asis takes another approach than specified in (2) and it is unlikely that calling individual functions will expose the type of bugs that emerge from a system wide refactorization. It is probably possible to use Asis in a way where (3) can be fulfilled, with a list of calls to different functions which are automatically approved in the recording phase. Criteria (4) and (5) are partially fulfilled since logging the function calls and determining if they are testable requires a lot of work.

5.1.3 Golden master testing

Golden master testing is a technique for performing characterization tests. It is supposed to be complementary to unit testing and acceptance testing by generating a large number of test cases automatically to do sanity checks between different version control branches. The technique assumes that input can be automatically rendered, which is fine for this project. The output of each run is recorded from a stable branch and then repeated in the branch under test to detect variances. Further, golden master testing requires a human to check if a difference should be considered an unwanted change [19]. The author mentions a few problems related to determining if a difference should be considered a fail.

- Floats cannot be naively compared.

- Timestamps and similar sequences will inevitably vary between runs.

- Data structures may be ordered differently without it being an error.
A rule in golden master testing is that the production code should not be changed [7] and thus it fulfills (4). I have not been able to find any tool which can perform golden master testing on Solidar system. Since no tool could be found, (1), (3) and (5) does not apply. The technique is well suited for (2).

5.1.4 Goutte

Goutte is an open source web crawler for PHP which utilizes the HTTP client Guzzle to navigate web pages and submit forms. Additionally, Goutte is dependent on a number of components of the Symfony framework. The capabilities of Goutte relating to this project is the abstraction of visiting web pages using only a URL and retrieving its content. Since Goutte can fill out forms and submit them it is also easy to login to an application and maintaining a session.

Goutte does not support other HTTP methods other than GET unless a form is available which it can POST to. Posting a form requires some scripting to find the fields and enter the correct values which makes it unsuitable for general POST requests. As such Goutte does not fulfill (1) however it does handle output like a browser which makes it easy to retrieve the output in a suitable way for external comparison. Goutte conforms to (2) and (3) if the comparison can be performed by some other tool and POST is not required. The other criteria can be met since only a URL is required and it is easy to install through composer.

It is important to realize that Goutte only solves a small part of the problem domain.

5.1.5 HTMLUnit

HTMLUnit is an open source headless browser for Java which also includes JavaScript support. It can be used to simulate a browser and performing assertions on the pages visited. It supports all the HTTP methods through submitting forms. The JavaScript support makes it possible to watch for unwanted alert dialogs, events and unreachable resources. It is worth noting that HTMLUnit does not use the same JavaScript engine as any of the popular browsers and that results may differ significantly.

HTMLUnit can be used to scrape web pages in the same manner as Goutte. The JavaScript can be used to check for simple errors but it also makes the tool slower, a consideration which is important in automated testing. HTMLUnit does not fulfil (5) since it requires Java and Java programming skills to make changes to the tool.

5.1.6 Node-replay

Node-REplay is an open source record and replay tool. It allows the user to record the response of a web request and replay it at a later time. It depends on Node libraries to create requests and to assert responses. Responses are saved as text files so they can be replayed multiple times quickly, essentially mocking the web server behaviour. The text files follow a proprietary format from which it is easy to extract the interesting parts including the HTTP verb, URL, response code and content.

It is possible to route specific requests to any server, including the tool if the replay function should be used. The benefits of this functionality are small for this project and thus it can only be used as a rudimentary scraper, which can be achieved by using some other tool built for that purpose.
5.1.7 PhantomJS

PhantomJS is an open source headless browser based on the WebKit engine. It has a JavaScript API which lets the user navigate web pages and attach handlers to the callbacks of these operations. There exists PhantomJS wrappers for PHP making it easy to install through composer and for simple cases no JavaScript is required to run the tool. However, to gain access to more advanced features such as maintaining sessions between calls or asserting JavaScript events, script files has to be used.

PhantomJS does not support any comparison features but it can scrape the contents of a web page so it partially conforms to (1) and (2). If a JavaScript file is used to handle the session it satisfies (3) and (4) as well. If a PHP wrapper is used if fulfils (5) to a great extent.

5.1.8 Record and replay

A blogger at DZone [32, 33] describes a method of characterization testing PHP scripts using superglobal variables and output buffering. The PHP language has some built-in variables which are available in all scopes. These variables contain all the server and execution environment information including \texttt{GET} and \texttt{POST} parameters. This means that all input parameters of a script can be read and manipulated through these super globals.

The output buffering functions \texttt{ob_start()} and \texttt{ob_get_clean()} can be used to capture all output from \texttt{echo} like commands and any bytes located outside the PHP scope. This means that any output that a browser would perceive can be recorded using this technique. Thus it conforms to (1) to the same extent as a scraper tool. Since the technique requires invasive measures to the page scripts of Solidar system it violates (2) and (4) quite badly. The author recommends copying the script files before preparing to test them, however that will affect the maintainability negatively which relates to (4) and (5).

5.1.9 Steam

Steam is an open source project for Ruby driven by the Java integration testing framework HTMLUnit. It is capable of testing web pages including JavaScript and CSS. It uses Cucumber which is a tool for running automated tests written in plain language, more specifically in BDD style. Cucumber has been ported to a number of languages including PHP, however Steam is designed to work with the original Ruby implementation. This means that the developers would be forced to install Ruby and a number of libraries, which is risky considering (5).

Furthermore Cucumber tests to execute Steam would have to be created for each URL which will result in unnecessary duplication of test cases. Cucumber does not have the functionality required to compare large amounts of text in a smart way out of the box. Hence this part would have to be developed in Ruby and thus (1) and (3) are violated. Criteria (4) should not be a problem. It is probably a better idea to use a PHP implementation of Cucumber together with a headless browser. However, some of the same problems would still exist and the team has bad experiences with Cucumber from the past.

5.1.10 Throwback

This is a framework for testing legacy PHP applications that run on an outdated PHP version which does not support PHPUnit. It is a single executable which can be run from the command line to execute a suite of unit tests contained in PHP classes extending a test
5.2. Creating a prototype

Since a tool could not be found that took the same approach to characterization testing, the decision fell on developing a set custom tools to solve different parts of the problem domain. The following problems had to be solved by one or more tools.

1. Test cases are essentially URL’s and these have to be generated to point to pages in the web application and also to contain the parameters required to access these pages.

2. URL’s must be visited and the contents of the response recorded. It must also be possible to login automatically before executing a test case.

3. The output of a test must be compared to the recorded content in an intelligent way and presented to the user.

To achieve this a number of tools were developed in PHP. To gather a list of URL’s to test, the Apache access logs from the production environment was used. It contains information about which pages are frequently visited during actual work by Solidar personnel. The access log also gives a hint to which parameters these pages require. However, since actual values are given which may not exist in the test database, the values had to be replaced by place-holders to be filled in later.

This rather complex task was divided into several sub problems and solved by different tools. The script accessToUrl is able to parse an access log and create a text file containing HTTP verbs and corresponding URL’s. Given an URL file the script urlListToRequests finds unique URL’s disregarding the actual values passed to the page. It replaces each actual value with a placeholder that hints which type of value that is expected.

Since the access log contains many duplicate URL’s, especially if the values are disregarded, the outcome of the mentioned script should be an enormously reduced set of requests. Another script called fillPlaceholders interacts with the test database to guess suitable values to put in the placeholders. After this process the text file produced will contain mostly valid URL’s with new actual values from the test database.

Any scraping tool or headless browser could have been used to visit these URL’s and record the content, however Goutte was chosen because of its simplicity. In retrospect,
cURL would perhaps have been a better choice in the long run since it supports POST. The goutteRunner script takes as input a text file with HTTP verbs and URL's. It logs in to Solidar system using specified credentials and records or compares content depending on the mode it is run in.

In recording mode, the web page content and the HTTP response code is stored on local disk. In diff mode it compares the content of the web page with the previously recorded content and creates a smart diff of these texts. The diff is first created by a diff tool and then filtered to remove unwanted changes that are not considered behaviour changes. Each failing diff is presented to the user after the test suite has finished executing. Any changes to the error log of the local PHP environment during the test run is also reported to the user for further inspection.

5.2.1 Test database

The developers at Solidar each have a copy of the production database locally which they typically restore every few days. By doing this they do not have to execute scripts to keep the database structure up to date and if some error occurs in the production environment, it is often possible to debug the scenario locally since the database is a few days older. Ideally, a test database should be used which does not contain real customers and other sensitive data. With production data, a mishap could cause the developer to send a circular or perform a fund change on a real customer. This is something that has happened to other companies in the past [5].

Another issue with the production database is the unnecessarily large amount of data which affects performance in a testing environment. Some of the entities in the database require documents to exist on local disk to be valid and if they are picked on random, all these files need to be in existence. Consequently, setting up a testing environment can be simplified if a subset of the production data can be used to create a test database.

The backup scripts at Solidar had support for creating a limited dump of the production data. As part of this project these scripts were updated to make the data as slim as possible while maintaining a valid system state. A batch script was also created to load the limited data into a new test database locally. Additionally, a script was created to download the necessary documents present in the test database.

Setting place-holder type

To replace the values in a URL with a place-holder, the type of the value must be guessed from all the instances of that URL. An example can be seen in listing 5.1. The first line shows one of the original URL’s taken from the access log. The second line shows the processed URL where the values have been replaced by place-holders with types.

### Listing 5.1: URL values replaced by place-holders

1. https://domain.se/?p=page&do=showList&id=31&all=true
2. https://domain.se/?p=page&do=showList&id=<int>&all=<false | true>

The chosen classifications of value types are shown below.

**Flag** A value which is not always present.
5.2. Creating a prototype

**Boolean** An integer value which is always '0' or '1'.

**Integer** An integer value.

**String** Any value.

**Boolean as string** A string value which is always 'true' or 'false'.

**Single value string** A string value which is always the same.

**Date** A string value which is a valid date.

The tool `urlListToRequests` tries to pick as narrow place-holder types as possible, given the different values that has appeared in the same URL. I.e. if '1' is encountered, the type will be set to boolean until a '2' comes along, then the type is changed to integer.

### Filling in the place-holders

The place-holder type hints what the value is expected to be. Some of the narrower types allow the tool `fillPlaceholders` to pick one of its discrete values at random. Others must be picked at random from a suitable candidate table in the database.

For each URL, a list of candidate tables is compiled by looking at the query string parameters that direct the application to a page. This list includes all tables in the test database whose name partially matches these parameters. I.e. if the page 'customer' is visited then the tool will look for tables with a name that is similar to 'customer'.

When the tool encounters a complex place-holder it tries to find a column in one of the candidate tables which matches the name of the parameter. If successful, a value from that column will be picked at random. If not, the tool will continue searching for a matching column in any other table which has a foreign key to a candidate table.

If the URL has multiple parameters the tool makes sure the foreign key constraints holds for the chosen values of all place-holders. If a foreign key fails to hold, the tool starts over with another value picked at random until all constraints are satisfied or until it has to give up.

The tool `fillPlaceholders` also has support for specifying known aliases of parameters, parameters to remove entirely and special constraints that apply to specific URL's. An example of one such constraint would be a limitation where only documents with a certain extension may be opened through a certain page. The constraint will then add a condition to the query which fetches a document ID.

### 5.2.2 Comparing page content

Comparing the contents of web pages is a challenge which has not been met by any of the evaluated tools. The responses contain timestamps, session ID's and hashes which has no meaning to the behaviour of the application. A diff tool will include these changes which are not of interest in a characterization test. A diff interpreter was implemented to deal with this issue, taking a diff generated by a diff tool and filtering it according to some rules which ignore meaningless changes.

A diff tool called PHP Text Difference based on the Python package difflib was used to generate the unfiltered diffs. This engine generates diffs on the unified format exemplified in listing 5.2. The first line describes the range of the section where the first pair is in the original file and the second pair in the new file. A line starting with a single '-' describes a removal and a '+' signifies an addition.
Listing 5.2: A sample diff with a single unwanted change

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>@@</td>
<td>−286,7</td>
<td>+286,7 @@</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In listing 5.2 a single line has been changed according to the unified diff. The UID is a number which is randomly generated at login and it is not significant to the characterization test. The rest of the line should still be compared but it should not matter if the UID is different. As can be seen in algorithm 1 the whole range is removed from the diff if the lines in it are considered equal by the diff interpreter.

The function `areConsideredEqual()` in algorithm 1 compares a line with its replacement using regular expressions. Each regular expression specifies a pattern which splits a line in three parts. The first and last part has to match exactly but the middle part is not compared at all since it contains a meaningless change, such as a timestamp.
Algorithm 1: Diff interpreter

Data: Unified diff
Result: Filtered unified diff

begin
  lineIndex ← 0;
  rangeIndex ← −1;
  ranges ← [];
  startLineForRanges ← [];
  while end of diff is not reached do
    Read current line;
    if line contains range information then
      rangeIndex ← rangeIndex + 1;
      startLineForRanges[rangeIndex] ← lineIndex;
    else
      Append line to ranges[rangeIndex];
    end
    lineIndex ← lineIndex + 1;
  end
  foreach range do
    removed ← [];
    added ← [];
    foreach line in range do
      if line starts with ‘-’ then
        Append line to removed;
      else
        Append line to added;
      end
    end
    remove ← true;
    if size(removed) = size(added) then
      for i ← 0 to i ← size(removed) − 1 do
        if areConsideredEqual(removed[i], added[i]) = false then
          remove ← false;
          break;
        end
      end
    else
      remove ← false;
    end
    if remove then
      Remove all lines in current range using startLineForRanges;
    end
  end
end
5.3 Characterization testing with the prototype

At the time of this writing the prototype has been used in one actual refactorization project performed by one of Solidar’s developers. The task was to move one of the largest legacy classes which represents a customer in the system to a namespace and also to rename it. The refactorization process was quicker than expected using the refactorization tools available in the IDE.

A set of URL’s were generated from an access log from the production server. These were converted to a set of unique URL’s with place-holders instead of values. The place-holders were replaced by actual values from the test database. During this test run, close to 90 % of the place-holders were supplemented with data.

The version of the code which was running in the production environment at the time was used as a reference. Out of 244 URL’s in the input file, 168 could be recorded as the correct behaviour. Some errors were reported by PHP during the recording and five URL’s had to be changed or removed manually. After the correction, 165 requests were recorded without errors.

When the developer had performed the refactorization which affected a total of 164 PHP script files, the tool was rerun in diff mode. No changes were detected by the tool and no errors occurred during the run either. The developer and the tester performed some spot checking and found only one minor issue which did not affect behaviour, in a web service which was not tested by the prototype. If the prototype had tested the web service it would most likely have found it too.

The unit tests also passed but it was later discovered that one test suite should not have passed since the IDE had confused two classes during the refactorization. When this test suite was moved to another namespace the tests failed and the problem was corrected. However, the test suite covered a scheduled job which cannot be tested by the prototype and it was only the unit tests that failed, not the job. A summary of the steps taken to test the application are shown in table 5.1.

Table 5.1: Steps taken to test application

<table>
<thead>
<tr>
<th>Step</th>
<th>Tool/Action</th>
<th>Major input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Restore test database</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Switch to master branch</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>accessToUrl</td>
<td>Access log text file</td>
<td>URL’s</td>
</tr>
<tr>
<td>4</td>
<td>urlListToRequests</td>
<td>File from prev. step</td>
<td>URL’s with place-holders</td>
</tr>
<tr>
<td>5</td>
<td>fillPlaceholders</td>
<td>File from prev. step</td>
<td>(Mostly) valid URL’s</td>
</tr>
<tr>
<td>6</td>
<td>goutteRunner (record mode)</td>
<td>File from prev. step</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Switch to test branch</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>goutteRunner (diff mode)</td>
<td>File from step 5</td>
<td>Test results</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

The prototype developed in this project is well suited for navigating to pages and verifying the contents displayed there. If the test database contains enough data to show the page properly without errors there is a good chance that any behavioural change will be detected. If the request performs an action there is a risk of having to manually adjust the parameters if the URL is unintuitive. Also, testing pages which are part of a work-flow may cause errors due to missing parameters. At the moment these have to be tested manually, mimicking the actions of a user. It is possible to create automated tests for these kind of scenarios but they are seldom cost effective since complex work-flows are changed frequently in Solidar system.

Since the tools collectively only support GET requests there is limited possibilities of testing web services. Other HTTP actions will also affect the database which then has to be restored between each test. At present this takes around 16 seconds which is a long time for one test and it will not be done automatically. Hence, most web services and other actions which manipulate the database are easier to test manually.

With the approach taken to characterization testing for this project it will never be possible to test scheduled tasks and other scripts which are not accessible through URL’s. These will have to be tested manually or by developing custom scripts that exercise them. Nevertheless, the prototype was able to visit over 160 pages and verify their intactness in the first real test. These numbers can be improved upon by manually adding new URL’s as the refactorizations are spot checked by testers.

After struggling to get a project idea approved by the client, there was limited time left to evaluate existing tools. If more time had been given, more time would have been spent researching different practical approaches to characterization testing. Goutte should be replaced by a scraping tool which handles POST requests to increase the domain of the tool.

6.1 Limitations

Given the limited time, the test database was never obfuscated as intended. Obfuscation techniques were surveyed during the literature studies but it was never implemented. Solidar system has safety mechanisms in place to ensure that production data is never leaked from test environments and the developers are used to working with sensitive data. The data contained within the local copies of the production data never leaves the building and therefore this task was skipped.
As mentioned the prototype only fully supports \texttt{GET} requests at this time. Many of the tools have some support for adding \texttt{POST} data but it has not been rigorously tested since the scraper does not.

Some of the tools, most notably \texttt{fillPlaceholders} has hard coded rules which are specific to Solidar system. All passwords and SQL connection strings have to be passed as arguments but other data such as URL parameter aliases are contained within the scripts. These should be loaded from a suitable file format like JSON instead.

6.2 Future work

The following list includes some of the improvements which can be made to the prototype.

- The scraper should be replaced by a tool which supports all HTTP verbs.
- The command line arguments of the scripts should be stream lined to support piping. A batch script could be created to execute the tools quicker.
- The list of URL’s needs to be extended and maintained. It can be regenerated to reflect changes to the database and code, however a few fixes that always has to be applied should be integrated into the toolbox.
- To further increase the user friendliness the tools should warn if the testing environment is set up wrong.
- The toolbox can be adapted to also gather URL’s from the local access log. Then the developer would be able to clear his access log, click around in Solidar system and then create characterization tests from the URL’s visited.
Chapter 7

Acknowledgements

I would like to thank Solidar for giving me the opportunity to undertake this thesis project. Special thanks to my external supervisor Simon Mannelqvist who came up with the project idea and has taken the time to discuss testing and refactorization many times. I would also like to thank my internal supervisor Mikael Rännar for reviewing the thesis report as well as Fredrik Georgsson for his swift replies to my e-mails.
References


[29] J. Raskin. Comments are more important than code. ACM Queue, pages 62, 64, 2005.


REFERENCES


