Improving the testing of Profit Software's insurance policy database system

Kristian Nordman

Helsinki November 13, 2016

MSc Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author Kristian Nordman
Työn nimi — Arbetets titel — Title Improving the testing of Profit Software's insurance policy database system
Oppiaine — Läroämne — Subject Computer Science

Työn laji — Arbetets art — Level MSc Thesis	Aika — Datum — Month and year November 13, 2016	Sivumäärä — Sidoantal — Number of pages 68 pages + 1 appendices

Tiivistelmä — Referat — Abstract

Profit Software's *Profit Life and Pension* (PLP) is an investment insurance management system. This means that PLP handles investment insurances from the moment they are sold to when they eventually expire. For a system that handles money, it is important that it can be trusted. Therefore, testing is a required part of PLP's development.

This thesis is an investigation into PLP's testing strategy. In this thesis we analyse PLP's current testing strategy to find flaws and impediments. We then offer improvement suggestions to the identified problem areas as well as suggest additions which we found could be beneficial.

ACM Computing Classification System (CCS):
General and reference → Evaluation
Software and its engineering → Software testing and debugging
Social and professional topics → Quality assurance

Avainsanat — Nyckelord — Keywords
testing, test strategy, automated testing

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

The testing strategy defines how a product is tested. It explains when, where, how and how much testing is to be done. In this thesis, we intend to evaluate and offer improvement suggestions for the testing strategy used in Profit Software Ltd.

Profit Software is a software company established in 1992. It provides insurers, such as banks or insurance companies, software to manage their insurance sales and existing insurance policies. Profit Software's main product is the Profit Life and Pension (PLP), an investment insurance sales and management system: it allows its users — the insurance company employees — to create new investment insurance policies and manage old ones. An investment insurance policy generally lasts for at least five years. For simplicity, we split them into two groups: savings and pension insurances. Unlike a risk-insurance — where the insurance is taken in case of death, injury or theft — savings and pension insurances are taken in case the insured lives long enough to redeem possible profits. Such insurances are usually provided by banks and insurance companies. The insurances themselves are actually wrappers containing multiple funds or stocks into which the insurance money is invested in. However, the insurance wrapper means the policies follow a different legislation compared to investing directly into those funds or stocks.

The Finnish law has its own chapter on insurance policies. For this reason an insurance policy has more rules to follow than standard contracts. For example, a pension policy is a special insurance policy in which the payments may be mandatory but the payout taxation is lower within certain limits. Creating a system which manages these policies will therefore require knowledge of insurance law and insurance mathematics in addition to the technical knowledge.

An insurance policy database system is a system which can manage insurance policies and store them in a database. The system becomes increasingly complex as companies using the system invent new products to sell within the legal framework. This is because new products may operate very differently to previous ones in terms of mathematics, investment targets or even client types. While the Finnish insurance law forbids discrimination, it allows some manoeuvrability. This means that such a system needs customisation, i.e. it is tailored to each insurers needs.

We call a system like this a database-centric system: a system where the database plays a significant role and in which redesigning the database requires using an automatic tool to move the data from the old database to the new database as doing so manually would take too long or would cost too much. This definition is ours and while other definitions of a database-centric system may exist, they will likely differ from ours. PLP's database may contain records on the order of 10^8. A database-centric system can be split into two parts: the database and the application. The application contains all logic, user interfaces and any other pieces of software involved in the system's usage. PLP, the system whose testing strategy we are evaluating, is a database-centric system.

We have divided our goal of evaluating and offering improvement suggestions for Profit Software's testing strategy into these three research questions:

Q1 Does the current testing strategy contain problem areas?

Q2 How could we improve those areas?

Q3 How could we add to the current testing strategy to make it more comprehensive and more effective?

To achieve our goals and answer our research questions, we start by analysing the current testing strategy to find factors impeding the testing process. We then proceed, with the help of scientific literature, to offer possible solutions to the impediments and additional methods, methodologies, metrics and processes which have been shown to be beneficial but are not yet included in Profit Software's testing strategy. Out of all the possible solutions and additions, we have selected the ones that were viable in terms of costs and would, therefore, be more likely to be implemented. The general guideline for acceptable cost is that an improvement should not require discarding the current tests but should rather build on them or improve them.

In our analysis, we describe how testing is currently being done in Profit Software. This includes all four types of testing currently being done: unit testing, integration testing, manual end-to-end testing and automated end-to-end testing. We go through them all but will focus on automated end-to-end testing as it is the main driving force behind this thesis. Also, we analyse the most common tools being used by all those involved in testing.

The knowledge of the target system, PLP, comes from our own experience as automation testers and interviews of persons from these four groups: developers, business analysts, manual testers and managers. During our nearly two years of experience we have worked on several long term projects within the company and have gained an understanding on how testing is done and how it differs on a project to project basis.

Our research methods are interviews and a literature study. We did two types of interviews: scheduled ones with preprepared questions and occasional hallway discussions. For the scheduled interviews we selected one person from each of the aforementioned groups, excluding manual testers. Each scheduled interviewee had worked at the company for at least six months and were able to point out weaknesses in the testing strategy.

Our goal is to evaluate the testing strategy and to provide useful feedback and improvement suggestions across the board. We achieve this by providing a set of improvements and providing a reasoning for them either by referencing a study or by providing a motivation based on how the improvement has worked in practice within Profit Software. The most important feature of our suggestions is that they somehow improve the testing process. This is followed closely by their cost effectiveness. I.e. their cost of implementation should be within reason compared to the benefit they provide.

The structure of this thesis is as follows: in section two, we will give background information on the target system and the type of data it handles — investment insurances. In section three, we will explain testing from a general point of view. In section four, we will describe the four testing levels on which testing is being done in Profit. In section five, we will describe a few metrics used in testing and introduce a goal-driven method for selecting additional metrics. In section six, we go through the current state of testing in Profit and point out any problem areas we detect. In section seven, we will suggest improvements on the detected problem areas and offer additional methods which would improve testing. Finally, section eight outlines future work and concludes.

2 Background

To better understand our case study, PLP, we will briefly explain some concepts behind the system. We will describe two basic types of investment insurances and some of their behaviour. We will also define the term *database-centric system*, which describes PLP accurately enough to follow the eventual description of its current testing tools and methods, and the improvements we intend to suggest to it.

We will begin by briefly going through the two main types of investment insurances: savings and pension. We will then define — what we call — a database-centric system, which is a group of systems that includes PLP.

2.1 Investment insurances

To fully appreciate the complexity that goes into making an insurance sales and management system such as the PLP, one should have a basic understanding of the laws and rules governing over the insurance sales. The Finnish law contains a special section devoted entirely to insurances. It dictates how and to whom they may be sold, as well as a framework within which insurance providers must operate. Since insurance legislation is only in the role of background information, we will only consider insurances from the point of view of the Finnish law.

Finanssivalvonta (FIVA) [17] watches over various companies, banks and others offering investment, insurance, pension and credit services. It was founded in the beginning of 2009 and is a conglomerate of Rahoitustarkastus and Vakuutusvalvontavirasto. In short, its goal is to make sure the aforementioned services are not abused and are generally safe. What this means is that an insurance company shouldn't be allowed to give out insurances which it cannot reasonably cover. FIVA's rights are defined in the Finnish law. Basically, for any institution providing aforementioned services, FIVA may inspect any documents related to the operation of the institution, issue warnings and notifications, and forbid a person from acting within a company for a maximum of five years.

An insurance agreement is a special contract between the insurance company and the policyholder. It is sometimes referred to as a policy and we will use *policy* and *agreement* interchangeably.

2.1.1 Insurances in general

Insurances can be split into two categories: *mandatory* and *voluntary* insurances. The focus of this thesis will be on the latter. Mandatory insurances are insurances like car insurance. While you do not have to get a car insurance if you only own a car, you are not allowed to drive it without one. Voluntary insurances are insurances like *savings insurances* and a supplementary *pension insurances.* These kinds of insurances are not required by the law but can be used to improve one's financial status. Essentially they are both investments but they have a few crucial differences.

Simply put, a savings insurance is in fact an investment and is taxed as such, whereas a supplementary pension insurance is more complicated and is governed by laws that have changed over the years. This is reflected in taxation so that money paid into the insurance is taxed based on the regulations which applied at the time of the payment; these are known as *tax bases.* This adds to the complexity of calculating optimal insurance payouts. For example, a pension insurance policy may dictate that its holder is paid €500 monthly. However, the policy contains savings in multiple tax bases, say, in two. One of the tax bases, tax base A, allows payments up to €200 tax free, while the other one, tax base B, allows unlimited tax free payments. To avoid paying taxes, the optimal plan is to take €200 from A and the remaining €300 from B until one of the tax bases is depleted. In reality this may become even more complicated as more tax bases may be available and the total sum as well as the outpayment amount can change throughout the outpayment period.

To make a distinction between payments made to the policy and to the policy's beneficiary, the former is called a *premium* and the latter a payment.

During the lifetime of an insurance, a policy can enter various states. How it transitions between these states are defined in the insurance policy. For example, if enough premiums are paid, the policy becomes a *paid-up policy* which is still in effect, cannot be cancelled by the insurance company and for which no additional payments have to be made. Another example would be cancellation which can be done by the customer at any time but as the insurance company is considered to be the stronger party, it will have a harder time cancelling a policy; having no money left in the policy is a good enough reason, for example.

Each insurance policy is a risk for the insurance company and, in fact, the business model is built heavily on risk management and mitigation. For example, the insured person may die and might end up costing money for the insurance company and the investments themselves may behave unpredictably and can sometimes fall rapidly, especially in the short term. To mitigate health risks the insurance company may require a health declaration which may affect the agreement negatively or may cause the agreement to be revoked altogether. In the absence of a crystal ball, the policies tend to span many years as it mitigates the negative effects of sudden sharp drops in investment rates, however, the mathematics and science behind these methods are beyond the scope of this thesis.

2.1.2 Savings insurances

Savings insurances are essentially investments but which act like insurances from a legal standpoint. For example, they generally have an insured person. Rather than making a direct investment to specific funds or stocks, a savings insurance acts as a wrapper around those funds and stocks. Ultimately, the money paid as premiums into a savings insurance will actually end up invested in stocks or funds but since the money is split between multiple targets, the investment isn't as heavily affected if a single target loses value.

Payments made for savings insurances are not tax deductible but if the policyholder and the insured are the same person, only the payment yield is taxed, i.e. the payment is tax free up to how much the premiums were. Additionally, the contract may hold any number of fees and clauses affecting the amount paid and to whom. Some common clauses are what happens in the event of the insured person dying or becoming unable to pay premiums. When this happens, the premiums may be taken from collected profits, the policy may end or it may be renegotiated.

To put it simply, a savings insurance is an insurance agreement between two parties and as such can have any number of clauses and exceptions. However, since investment companies can have thousands of clients, the agreements they offer are predetermined and give the policy taker limited room for negotiation. This brings some order into the otherwise chaotic insurance agreement space.

2.1.3 Pension insurances

The purpose of pension insurances is to supplement the normal pension. It allows a policy taker to invest money for a fairly low profit, low risk investment instead of simply saving it. These insurances, as supplementary pension, have special laws that apply to them. For example, usually the pension holder or beneficiary will not be able to get the money, invested into the policy, before the pension time begins; with very few exceptions such as in case the insured person dies. The pension time is defined in the policy and it has a minimum age which is determined by the law and the insurance company. In addition, the premiums are mandatory. In return for this, pension payments are tax free up to a certain threshold and pension premiums are tax deductible up to a sum determined by the law.

The way pension time works is that pension payments are made out of what we will call *pools* of money. Each pool has an associated tax base which determines the tax free threshold. During the premium period, when a premium is paid, the money goes to a pool associated with the current tax base, i.e. the tax base determined by current legislation. When a new tax base is legislated, the old pool closes and new payments will be made to the pool associated with the new tax base. When pension time arrives, the beneficiary may choose from which tax bases they wish to take the money from. For example, if the determined pension is €500 and the policy has money in tax bases **A**, **B** and **C** with tax free thresholds at €50, €100 and unlimited, respectively; the beneficiary may choose to take €50 from **A**, €100 from **B** and €350 from **C** for a total of €500 and thus avoiding taxes.

Mandatory, tax deductible premiums and tax bases are the biggest differences between an insurance savings policy and a pension insurance policy. However, beyond that they are very similar and equally modular yet often predetermined.

2.2 Database-centric Systems

We define the term *Database-centric System*(DCS) as a system where the database plays a significant role and cannot be redesigned without using an automated tool to move the data from the old database to the new one. In other words, the data within the database cannot be discarded if the database is redesigned and the database contains enough records that it would be infeasible to manually copy it. Other examples of such systems would be Facebook [15], Twitter [54] and Reddit [27]. A common feature for all the aforementioned systems is that they all provide multiple interfaces to a single large database. Our definition of a DCS is not exact and one must use discretion when deciding if a system contains database in such an important role. However, it is accurate enough for our purposes.

While various architectures exist, usually a database-centric system has at least three parts: the interface layer, business layer and the database itself as shown in Figure 1. The interface layer may consist of multiple user interfaces, usually tailored for specific devices such as tablets, phones or desktop computers. They may also be targeted towards different audiences. For example, a bank's system may provide a desktop interface for its employees to create applications and transfer money, and a phone interface for clients where they can check their balance or investments. Both types of interfaces connect through the business layer into the database but the client interface has restrictions to prevent theft and to stop clients from accidentally breaking the system.

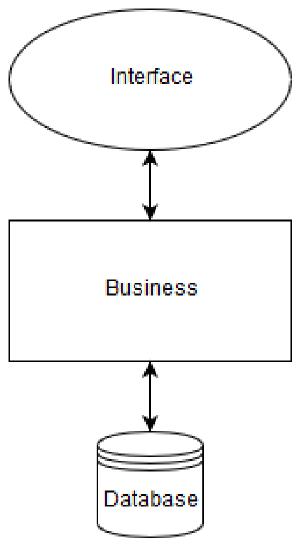

Figure 1: A simple database-centric system

The business layer contains most of the program logic and does most of the work. For example, if an employee wishes to create a loan application for a client, the business layer routines gather relevant information, such as the client's credit rating, and in some cases might even make the decision automatically, e.g. the client has not been paying back previous loans: in such situations the bank usually re-negotiates the loan and a higher level user might be required to accept such a loan. The business layer connects the user interfaces into the database through APIs. The UI API may be split into multiple APIs for each user interface. The database API serves to merely provide easier access to the database and to allow switching the database without touching the business layer code. An example of an architecture for a database API is the Data Access Object (DAO) [41]. A DAO wraps queries and responses into objects which are usually easier to manage than raw strings or tables.

The database layer is the heart of database-centric systems and is shared by all users of the system. By database we don't necessarily mean a single computer running software such as Oracle [42] or IBM DB2 [24], but rather that the layer provides a service, which from the point of view of the business layer, looks like a single database — even though, in reality, it may be distributed.

3 Testing in general

In profit-driven software engineering, testing is a tool used to mitigate the amount of development work needed by detecting bugs early. Testing often manifests itself as unit tests, integration tests, end-to-end tests, etc.; which help identifying bugs by detecting failures in the program execution. Unfortunately, completely bug free software is generally thought of as a myth rather than reality and testing can only tell so much about a piece of software, but even so, testing can help software development by detecting bugs early in the development cycle, hopefully before production.

In this chapter we will provide a motivation for a company to invest in testing as it has been shown to improve quality and in doing so, saving money in the long run. We will then introduce the concept of testing strategy — what it means and how it manifests itself. Then, we will introduce the concept of *continuous integration* along with a study showing its benefits. After that, we explain white-box and black-box testing. Next, we will provide motivation for test automation in general as opposed to pure manual testing. Finally, we will introduce controlled natural languages, which provides a more formal way to define the language used in creating automated tests.

3.1 Costs and benefits

Finances are somewhat of a lifeline for all for-profit companies and projects, and as such testing is something that will be in the books. Therefore, for a company to have any motivation to test their software, there has to be a monetary incentive. I.e. any decisions made should directly or indirectly increase the company's value. This incentive can come from comparing the cost of testing to the cost of having a bug end up in production. Slaughter et al. have defined *the cost of quality* [49].

The cost of quality can be further broken down into two main types: conformance and non-conformance [49]. They break conformance down to *prevention* and *appraisal*. Prevention contains the cost of preventing bugs before they happen, for example, by training the staff. Appraisal contains the cost of evaluating and auditing the software. This include, for example, testing. The price of non-conformance includes the costs when something goes wrong. These costs are made up of internal failures, i.e. within the developer company, and external failures, i.e. in the client company.

Slaughter et al conducted a study where they tracked the costs of quality in a big company currently improving its development processes and thus, raising its conformance costs.[49]. The results show that the overall quality costs went down while having the greatest effect on development, management, operations and quality assurance. They found that conformance costs didn't change much throughout the project, possibly due to the company having to keep certain processes in place [49]. They concluded that the project benefited the most from quality improvements at its early stages and that the rest of the project benefited from these improvements, that is, the cost of quality assurance at later stages was lower than what it would have been had they not made the improvements early on. They deduced that making larger investments into quality towards the end of the project would not be advisable. The results and conclusions they made seem to suggest that development models, which leave all testing to the end of the project, may not be as cost effective as one would hope. An estimate made by Masticola [37] supports investing in risk management which includes bug prevention.

3.2 Testing strategy

Testing strategy is a high level guide for testing. It should include any roles involved in testing, the environments, the objectives, the standards and technologies, and other resources needed for testing. Having studied it, testers should know what they are supposed to test, how they will do it, when they will do it and how much is enough.

The purpose of a testing strategy is to work as a framework so all testing is done in a predictable manner. This is especially helpful in a project involving multiple testers since everyone will know their roles and responsibilities which helps avoid overlapping work. Such a large project will also benefit from documenting which technologies and standards are to be used as it allows a tester to design and implement homogeneous tests which can easily be understood by the other testers as well.

While the purpose of similar high level documents is that they shouldn't change too often, a testing strategy should not be set in stone. It has such an important role in testing that if a problem is discovered, it should be possible to change the strategy. However, a change should be accompanied by a reasoning and a plan.

3.3 Continuous integration

Continuous integration(CI) is a concept, often attributed to Martin Fowler [18]. Various tools such as Jenkins [10] or Gitlab CI [26] exist for this very purpose. While the tools may vary in their capabilities the idea behind them is the same: at certain specified intervals — be it a change in the code, a certain duration or a manual trigger — all source code related to a module or a project is compiled, deployed and tested. This allows for automatic and manual execution of end-to-end tests.

The use of CI results in increased productivity without compromising quality according to a study conducted by Vasilescu et.al [55]. Their study included 246 GitHub [25] projects which at some point had adopted the Travis-CI functionality into their development process. The study concluded that the adoption of CI showed an increase in accepted pull requests and a decrease rejected pull request by non-core developers. They also noticed that the amount of bugs discovered by the core developers increased while user reported bugs did not, meaning CI likely helped developers discover more defects while having no negative effect on user experience. A core developer is a developer working in the project and has the ability to merge or reject pull requests.

3.4 White-box and black-box testing

White, clear or glass box testing means that the tests are being implemented and/or designed by a person with knowledge of the source code being tested — usually a developer. In essence this is the case for unit tests and integration tests in Profit.

One advantage in this approach is that the tester can test things that are only apparent to the developer. For example, if the tester has knowledge of the execution paths, they could potentially create a test which is unintuitive but has a very large coverage. A disadvantage is that in this approach the tests may not detect misinterpretation of the specification since the tester is likely the developer as well.

In black box testing the tester does not know how the underlying program works, i.e. the system is a black box. In this type of testing, the tests are created based on specifications. In Profit, this is the path usually taken in end-to-end tests.

3.5 Test automation

In terms of test execution, tests can be split into two types: manual and automated. Manual testing is where a tester, a person, is given a task to test a specific feature. It could mean modifying a program to force a certain execution path or, in the context of acceptance testing, using the interface to achieve a certain goal; for example, add a book into a library database using the provided software. An automated test aims to accomplish the same task but is executed by a computer rather than a person.

An automated test is like a program, executing one instruction after another, but aiming to be simple enough as to not require itself to be tested. It is immediately clear that creating an automated test is slower than doing one manual test. However, automated testing starts to show benefits over manual testing as time passes and automated tests are executed repeatedly [32].

To find out if automation has any tangible advantages over manual testing, Karhu et.al conducted an empirical study [32] by interviewing managers, developers and testers on the subject of test automation. According to the study, the biggest perceived advantage in test automation was increased test coverage as more testing could be done in a shorter time. While the main disadvantage was the costs involved in implementation, maintenance and employee training.

Not all products are equal when it comes to test automation; generic products seem to be favoured over customised ones [32]. This observation would indicate that products which are either entirely generic or share generic parts with their sibling products, would benefit from adapting test automation. On the other hand, a mostly custom product may never break even in the price/performance ratio.

Test automation efficiency is also affected by the need for human involvement[32]. This was especially apparent in customised systems requiring domain knowledge as testers would have to put themselves in the end-user's position. Even if automating such a test would be technically doable, it would still require training the automation engineer so she would have the necessary domain knowledge.

A system's technological infrastructure can also threaten to make tests obsolete, should it change often or quickly [32]. Such a behaviour would drive up the maintenance costs and may dissuade projects with a volatile technology stack from implementing automated testing.

The final observation made was that test system reusability facilitates automation[32]. As in many programming languages, this is effectively the same as building libraries that can be re-used in multiple parts of a project or possibly in multiple projects. Much like those libraries, setting up an automated system takes effort and is expensive, compared to a manual system.

With the observations gained from the study, one can conclude that test automation provides quality through better testing coverage and the ability to test more in less time but human involvement is necessary [32]. The main disadvantage is the cost involving implementation, maintenance and training [32]. There is also, a link between implementation and maintenance cost: bigger investment in implementation reduced maintenance costs [32].

3.6 Controlled natural languages

Domain Specific Languages are a set of languages constructed to help solve problems in a specific problem domain. Many of these languages do not necessarily have a name as they are created within organisations for a specific purpose and are never meant to be released for outside use. In fact, to support this kind of behaviour some programming languages, such as Scala [14], support DSLs. A subset of DSLs is known as *Controlled Natural Languages* (CNL); a set of languages that attempt to find the middle ground between formal languages, such as Prolog or C, and natural languages such as English or Finnish.

The concept of CNLs has been around since the 1930's [33]. Basically, a CNL is a language that is technical enough for a computer to process and yet expressive enough for non-technical people to understand it. That is, the language has strict rules — or grammar — which can be parsed easily by a computer yet allow enough freedom to mimic natural languages. A survey was done by Tobias Kuhn [33] where he listed 100 languages and classified them based on *Precision*, *Expressiveness*, *Naturalness* and *Simplicity* forming, what he called, the PENS classification system.

Since many languages exist and not all of them could be described as a CNL, a more precise definition is required. The definition provided by Kuhn has four requirements that must be met before a language can be considered a CNL. First, a language must be based on only one natural language. Second, it must be more restrictive than the language it is based on in terms of lexicon, syntax and/or semantics. Third, it must preserve most of the natural properties of its base language, i.e. native speakers should be able to understand its words and sentence like structures, even if they wouldn't understand the context. Fourth, it must be a constructed language, designed and not the product of a natural process, e.g. slang. While these definitions, especially second and third, are a bit vague, they are accurate enough to exclude languages such as Esperanto and common formal languages. However, ultimately, since a CNL is a language that should *feel* like a natural language, it is left for the reader to decide where the grey area is and which languages should be considered a CNL. For example, Kuhn has decided to exclude fictional languages such as George Orwell's 1984 *newspeak*, although it would technically fill the requirements.

Each property, of the PENS system, is ranked between 1 and 5. Precision means the language's ability to accurately convey messages. Natural languages are very imprecise since they require a lot of context information to deduce the real meaning and would rank at P1, whereas formal and fully specified languages would rank P5. Expressiveness means a languages ability to describe things. Natural languages can describe anything that can be communicated between two people and therefore rank E5, whereas E1 would include propositional logic which can be used to portray simple binary relations. Naturalness is the measure of how much a language looks like a natural language. Natural languages rank at N5 and languages such as ones that use a lot of symbols or unnatural keywords, such as the programming language Whitespace, rank at N1. Simplicity measures the ability to describe a language using rules and definitions. Natural languages cannot be described in this manner in a reasonable time and rank at S1. A language that can be described completely in 700 or less words would rank at S5. Kuhn's classification system also has nine additional properties describing the language's goals, method of communication and origin. However, these properties end up describing the environment more than that actual language itself.

In addition to the classification system, Kuhn evaluated whether CNLs actually provide benefit to their users. That is, do CNLs achieve the goal they were created for. The evaluation is done by answering the following three research questions which describe the goals of a CNL:

- (C) Does a CNL make communication among humans more precise and more effective?
- (T) Does a CNL reduce overall translation costs at a given level of quality?
- (F) Does a CNL make it easier for people to use and understand logic formalisms?

He concluded, through multiple conducted studies, that CNLs can be beneficial.

However, he notes that their usefulness depends on the problem domain and should therefore be evaluated before adapting one. Our interest lies mostly in *Behaviour Driver Development* (BDD) and Gherkin, and their suitability for describing tests. Therefore, we are interested in C and F. We will describe BDD and Gherkin more closely in chapter 4.3.2; for now it is enough to know that BDD is a development style derived from test driven development and uses Gherkin — a CNL — to describe use cases.

The creator of BDD, Dan North, defines one of goals of BDD to improve the communication between testers, developers and analysts [40]. With this in mind, we believe that BDD at least attempts to reach the goal *C*. However, we could not find any studies about BDD where communication would have been the main goal.

Hoisl et.al conducted a study [22] involving 20 software professionals where the participants were tasked with evaluating different types of notations based on a number of dimensions such as clarity and scalability. The study concluded that controlled natural languages are recommended for scenario based testing. The notation types in the study were: Gherkin, UML sequence diagrams and Epsilon. Gherkin was the favoured choice and a CNL. UML sequence diagrams describe how the control flow moves between the tester and the various components of the program being tested. Epsilon is a fully structured language, like Java. The results of the study seem to indicate that Gherkin was easier to understand than the other two notations. This leads us to believe that Gherkin does indeed make it easier for people to use and understand logic formalisms (F).

4 Testing levels

A common way to split testing is to separate it into levels. We have divided it into four levels: unit testing, integration testing, end-to-end testing and usability testing. We will take a look at each of them but only to the extent we deem necessary to explain PLP's current testing strategy and the improvements, later on.

We will go through the topics from the bottom up. Meaning that we start from the low level unit tests, go through integration testing and end-to-end testing. We will then introduce the testing pyramid which shows how the tests should be distributed. Finally, we cover usability testing. Most of our focus will be on unit testing and end-to-end testing as they are the most used form of testing in Profit.

4.1 Unit testing

Unit testing is the practice of testing the smallest testable piece of software. This small piece of software, a *unit*, could be a class, a function or a complicated algorithm, however, there is no clearly defined unit and it could envelop the entire program, e.g. a *hello world!* program. Unit testing can be done manually or automatically.

Manual unit testing is a practice that often happens during development. Usually it involves the developer forcing a certain execution path or manually setting variables and parameters to achieve a particular state. Because these changes are momentary and only made by the developer to herself, these tests are lost after they have served their purpose. These tests represent lost potential regression tests and could possibly provide a decent basis for regression testing [34].

In automated unit testing, the tests are not as integrated into the coding process as they are in manual unit testing; instead they are designed separately. This separation, while demotivating for a developer as it takes more time to implement features [34], does in general improve the software's quality [12, 29, 57]. Since it is important to keep the developers motivated, a lot of research goes into finding out how to make unit testing a more tolerable exercise. Eventually, some research endeavours achieve concrete results as popular unit testing frameworks adopt their proposed processes and methods.

Unit tests are the most common type of test. They are usually made by the developers as they write new code. Unit tests are usually very fast to execute and are run frequently during the program's development. Unit tests are meant to catch errors in a method's logic but not its interaction with other components. For example, a component may be developed using Java 8 but the rest of the components using Java 7. This could lead to a situation where the final software cannot be run on neither Java 7 nor 8.

4.1.1 Manual unit testing

Manual unit testing is ad-hoc testing where test cases appear and disappear very quickly while seamlessly integrating into the coding process. Very few studies have been conducted on manual unit testing, possibly due to its bohemian nature, its coupling with coding and simply because executing unit tests manually after each code change would take too much time. As a result, any guidelines or methodologies are hidden within coding guides and unit testing guides — which cover automated unit tests. However, manual unit testing has been compared to randomly generated unit tests [44] and the results seem to indicate that the randomly generated unit tests support manual unit testing by detecting defects not found by manual testing. Ramler et.al. predicted, based on the results, that the random tests may suffer from diminishing returns as tester learn from the random tests' results [44]. The study used a system resembling a legacy system, i.e. an outdated system whose original developers are quite often unavailable.

4.1.2 Automated unit testing

Since unit tests are small and abundant in a software project, automated unit testing is much more efficient than manual unit testing at providing regression coverage. Unit tests are almost always written by developers and are therefore white-box tests. They let the developers prove that a particular unit works as intended and, when done simultaneously with coding, support the development.

Unit tests are often built on top of existing frameworks such as JUnit [31] or CUnit [13]. These frameworks provide reporting, input data factories, test tagging and many other tools which allow developers to better understand test failures, avoid duplicate code and control test execution. Frameworks will do all this and more but come with a few caveats. Any framework comes with its limitations, be it the inability to modify reports or decide the testing order, and in some cases they can prevent the framework from being used. These limitations should be studied beforehand, especially since adopting a framework takes time. While testing frameworks may provide similar functionality, they do not do so in exactly the same way unless the aim is to be backwards compatible as is the case with TestNG [4] which provides most annotations of JUnit to allow for a smooth transition.

Due to the learning curve inherent in adopting a new testing framework, it is easier to do with a new project [45]. Many projects are built on top of existing, possibly completed projects which may already have their own obsolete unit testing frameworks. If the new project is to have a new unit testing framework to go along with it, the project management has three options: they keep the old framework and run two frameworks side by side, they take the time to transfer the old tests into the new framework or they simply discard the old tests altogether.

Having an old framework can have a few issues. An old, possibly deprecated, framework might not have support from its original developers. If the testing framework happens to be an open source project, the project management can hire developers and provide the support themselves, however, the costs involved can be high depending on the documentation, code clarity and the overall size of the framework. Without support the framework can be left dead in the water and should it become unusable in the future, the company using the framework will be forced to face the original problem. However, assuming no new tests will be created using the old framework, the management could simply let it be. The old framework and its tests would continue to provide regression testing effectively postponing the need for any decisions to a later date. This could be a valid option if the management simply cannot afford the resources to migrate the tests but still want to keep the testing coverage.

The amount of work required for transferring the tests to a new framework varies wildly. For example, moving from JUnit to TestNG might be as simple as modifying the configuration file and changing import sources. The group responsible for TestNG also provides a plugin for Eclipse [52] which allows for nearly automatic transition from JUnit. JUnit is in a peculiar position when it comes to Java testing frameworks as it is very prevalent and is considered to be something of a *lingua franca* of Java testing frameworks. For this reason many Java testing frameworks support JUnit in one way or another. However, if the new testing framework uses a DSL exclusively, porting may require rewriting the test cases completely.

The last option is to simply discard the old framework along with the tests it supported. This is quick and easy but might be costly should the old system fail at some point. Unit tests could help detect such a failure early. Discarding the old tests may be a good option if the system itself will be decommissioned very soon. That is, soon enough that no changes will be made to it. The system should remain frozen in time until it is ended.

4.1.3 Design

Unit tests have at least two factors which guide their design: *who* and *when*. The person responsible for unit test design is almost always the developer who wrote the unit being tested. He knows best how the unit is supposed to work and is therefore the most suitable person to verify its behaviour. However, there is an advantage in having someone else design the tests as it brings another interpretation of the unit's purpose to the table and could possibly detect a wrong interpretation of the specifications. However, detecting specification misinterpretations should also be the responsibility of a code review.

The design process for unit tests varies depending on if they were made before, during or after the development. When made before, the tests are made based on specifications and either on top of mock methods and functions or as a specification for a test. Creating unit tests before the actual program code is also know as *Test Driven Development* (TDD). TDD has been shown to work [5] but only if it is given the extra time it requires. Also, TDD is not a silver bullet and should be evaluated for each project separately [46].

Designing tests during the development is a more relaxed form of TDD. In pure TDD tests are always done first but a developer may wish to focus on the tests separately rather than as a part of the development process. Like in TDD, the developer still has a fairly good understanding of the unit's behaviour and can therefore design the tests efficiently. A downside is that the tests are more likely to be implementation driven rather than specification driven and will reinforce the developer's interpretation of the specification, irrespective of whether it is correct or not.

Sometimes, a developer might create unit tests once a significant amount of time has passed since they worked on the unit. When this happens, the developer may not remember exactly how the unit is supposed to work so verifying its behaviour with tests will take additional time as the developer has to analyse the unit first. Additionally, if someone modifies the unit, they have no way to verify their changes beyond their own intuition.

4.1.4 Contract Driven Development

According to a study conducted by Leitner et.al [34], unit testing is viewed by many developers as arduous, time consuming and boring. However, the same study found out that developers also believe unit testing to be useful. To combat these issues and make unit testing easier they introduce a concept called *Contract Driven Development* (CDD) and a software tool called Cdd. While the software tool was developed for Eiffel [50], the concept is applicable to any language or style which can provide contracts [43].

The basic idea behind CDD is that test cases are created automatically. Although such tools already exist, Leitner et.al recognise three main problems with them:

- Automated tests are not very good at adapting to changes and they will not spot errors they were not meant to catch, i.e. an automated test lacks the insight of a developer.
- Assuming there are no accurate specification tools, an automated test generator cannot distinguish between meaningful and meaningless input data.
- Measuring test set effectiveness is done using meters such as branch coverage or number of bugs found, but picking the right meter can be difficult for an automated tool.

CDD looks to overcome these shortcomings by tapping into a resource not used by frameworks such as Junit [31]: the implicit tests created by developers during development. As developers develop a new feature, they often create short tests that only exist for a little while. These tests are simple, require no maintenance and are often just small variations of each other. The fact that they require no maintenance and are easy to make, are probably the reason why developers use them [34]. There are two main methods how the tests are created: by a developer providing the right input or by changing parts of the program to force a specific execution path [34]. By monitoring these two changes, a CDD enabled system could extract tests from these implicit tests to create a reasonably effective regression test base [34].

4.1.5 Mutation testing

In short, mutation testing is a method for checking how well your existing test set can detect errors in the code, i.e. it is a method for testing tests. The idea is that you introduce mutations into the program's source code. These mutations include, but are not limited to, changing operators (+, -, *, /) to other operators, possibly changing parameters to different ones or removing a line of code altogether. However, in order for mutation testing to be successful, the resulting *mutant* program should still compile and should be syntactically correct. The idea is to produce errors similar to those a developer might introduce.

Andrews et.al have written a paper concerning the viability of mutation testing [1]. In their paper they analysed if hand created faults and automatically generated faults are representative of real faults. They concluded that not only are mutation faults viable but may actually perform better than hand picked faults, as hand picked faults tend to underestimate the test detection capabilities of a test suite.

4.2 Integration testing

Integration testing is where components are tested for interoperability. In terms of scale it is broader than unit testing but narrower than end-to-end testing. Its purpose is to take multiple components to form sensible groups and test those groups and their interactions. A sensible group means a group of components that work closely together. For example, a network component and an email component. When choosing component groups, it helps to view the interactions as dependencies. In the network-email example, the email component depends on the network component.

Integration testing has many philosophies by which the actual testing is done. Four relatively famous ones are: top-down, bottom-up, big bang and sandwich. Solheim et.al conducted a study [51] on which of these testing strategies produced the most reliable systems.

On top of the general testing methodology or philosophy, one must consider the testing order in which the various dependencies are tested, and the effort required to simulate not yet implemented components, also known as mocking. The testing order means picking the most fault prone dependencies and testing those since testing all dependencies would be infeasible [6].

4.2.1 Testing philosophies

Integration testing philosophies help keep track of how much testing has been done and how much is left to do. We will introduce top-down, bottom-up, big bang and modified sandwich philosophies briefly and evaluate them based on Solheim's study [51]. It is important to note that the study conducted by Solheim et.al used artificial systems which are close to but not exactly like real systems in development [51].

In Figure 2 we have a simple architecture for a calculator. We will use this architecture to visualise the different testing strategies. In the example, each box represents a collection of components which have been logically grouped together. The actual components are not important, merely the groups they belong to.

Top-down

The top-down approach is one where the components are tested starting from the highest level downwards — highest in terms of abstractions. This strategy involves stubbing or mocking the lower level components. In our calculator example the testing order would be: Calculator→(UI / Maths)→Functions.

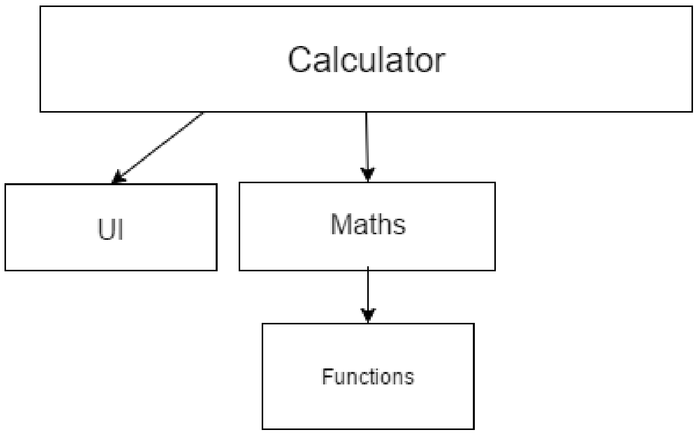

Figure 2: Calculator architecture

A study [51] conducted by Solheim et.al concluded that a top-down approach is the most of effective of the four in terms of defect correction. They also observed that a top-down approach uses the greatest number of components. Another conclusion was that top-down approach produces the most reliable systems.

Bottom-up

The opposite of the top-down approach is the bottom-up approach. Instead of starting from the highest level components, the testing starts from the low level components. In our calculator the testing order would be: Functions→(UI / Maths)→Calculator.

Solheim's study [51] determined that the bottom-up approach produces less reliable systems than the other strategies, but they also note that it might be due to their definition of reliability which emphasises the higher level components. They also suggest that system height, the amount of layers it has, could have an effect on the testing strategies' performance because bottom-up actually performed better than big bang on a system with more layers.

Modified sandwich

The modified sandwich testing strategy is a combination of bottom-up and top-down approaches. The idea is that each layer is tested separately starting from the bottom and top. From there, testing advances towards the middle until the two testing strategies meet, forming a sort of sandwich. Like the top-down approach, subbing or mocking is used to test the top level. In our example the testing order would be: Calculator→(UI / Maths)←Functions.

Solheim's et.al study [51] observed that a sandwich testing strategy yields moderately reliable systems.

big bang

In essence, the big bang approach is where integration testing is done once all the components are ready. All the components are brought together to form a complete system which is then tested. This saves time during the development as no integration testing needs to be done until at the very end.

According to Solheim's et.al study [51] the big bang approach produces reliable systems, much like the top-down approach.

4.2.2 Integration test order

Since testing every possible dependency between each component would require too many resources, it is important to decide on a testing order [6]. The testing order describes the order in which each dependency is tested. Two important things to consider in the test order is the importance of a given dependency and the amount of simulation of unfinished components the testing would take [6].

By only considering the simulation effort, one could come up with an optimal testing order. However, this approach may devote time to less important dependencies and leave important ones untested. Borner et.al examined some of these approaches [6] and added testing focus as a factor. They concluded that two heuristic algorithms, simulated annealing and a genetic algorithm, faired best when considering test focus and simulation effort [6].

4.3 End-to-end testing

A system can be described as a set specifications. Once all the functionality in a specification has been implemented, it can be tested as a whole. This is what end-to-end testing is: testing a part of a system so all the required components are available and complete, instead of being replaced by mock interfaces. However, sometimes mocks are unavoidable if, for example, the system requires a service that is not available during testing.

4.3.1 Database testing

Database data consistency

When conducting system testing, usually the user interface is tested. However, the underlying database should be tested as well. Database applications can damage the data inside the database, namely, actions which alter, delete or create records can cause data corruption or simply store valid but nonsensical data. Setiadi and Man have created a model [47] with which databases can be tested using special consistency rules. These rules are derived from specifications and can be executed as tests.

The main mechanic of their model is to look for inconsistencies. The tests to detect inconsistencies are created in three steps: rule extraction, rule translation and query execution. In *rule extraction* the designer goes through the specification and the business rules to find definitions for what is considered valid data and in what state should the database be after each operation. This step is done manually and is error prone [47].

The basic idea is to create consistency rules that are built from three parts: rule domain, domain rule and rule formula. The rule domain represents the dataset that the rule applies to, for example, a table or multiple tables joined together. The domain rule specifies where in the dataset this rule is appropriate, for example in an SQL query, it could be the *where* statement: *select * from students WHERE average_grade > 3*. Finally the rule formula specifies what is being tested, e.g. age > 21. As an example: a specification of an animal registry defines that there should be no ponies whom are older than 10 years old. The rule domain is the table *animals*, the domain rule is that the column *species* should be a pony and the rule formula is that the *age* column should be less than 11. These are the combined into After-State Database Testing (ASDT) rules which have a strict form. From here their ASDT tool can transform the rules into ASDT queries, which are just SQL queries in the form *SELECT * FROM domain WHERE (domain conditions) AND -(rule formula)* [47].

Setiadi and Man conducted an empirical study of this model [48]. The study was done on an example "*Employees Sample Database*" provided by Oracle's MySQL [11]. The database has 6 tables and 4 million records. However, there is no specification available and therefore they made their own assumptions about the possible business rules [48]. The study revealed two business rule violations and two issues that could be violations but since no specifications were provided with the database, the two issues could not be declared as violations. The empirical study shows that ASDT is, in fact, capable of detecting inconsistencies. Irrespective of, a more in-depth study and possibly comparisons to other methods are required to evaluate the model's viability.

4.3.2 Automated test case design

A good test case is such that it covers as much as possible, is easy to maintain and understand, and is quick to execute. Unfortunately, there is no single methodology or process which when followed would always produce perfect test cases. Therefore, most testers use guidelines, rules of thumb and intuition when trying to figure out how and where to focus their testing efforts.

Designer

When a test involves a system that requires a lot of business knowledge, the designer is usually a business analyst or equivalent. A business analyst can accurately translate use cases and documentation into meaningful test cases. However, as it is not always necessary for a business analyst to know programming, they may not be well versed in creating maintainable tests. This is relevant because tests are like small programs; they behave like programs: they can crash, have bugs and are made up of source code. This means that creating maintainable tests is as important as creating maintainable source code for any program.

Behaviour Driven Development (BDD) was first introduced by Dan North [39]. His motivation was that the traditional *Test Driven Development* (TDD) was somewhat confusing. From his experience he knew that developers weren't always sure what was to be tested and what was not. For this reason he introduced BDD which is a modification over the original TDD rather than a completely new methodology.

The main difference between TDD and BDD is that BDD aims to bring developers, business analysts and testers together and to create a common language using which they can agree on feature requirements. In practice this usually manifests as *Gherkin* which is a DSL and very close to a natural language with only a few reserved words.

The original BDD merely added a 'should' word to each test to better describe what behaviour the test is verifying. What this means that instead of writing tests like:

```
public class addCustomerTest {
  testAddCustomer() {...}
  testAddDuplicateCustomer() {...}
}
```

The tests are transformed into:

```
public class addCustomerBehaviour {
  shouldSucceedWhenAddingACustomer() {...}
  shouldFailWhenAddingDuplicateCustomer() {...}
}
```

This is not quite the modern Gherkin keywords used in Cucumber [36] or Robot Framework [19] but it still conveys the idea behind BDD: instead of testing for inputs, we are testing behaviour. This level of abstraction distances the tests from the actual source code making it more accessible for non-technical people.

Robot Framework

Robot Framework [19] (RF) is a testing framework which allows testers to write tests so they mimic a natural language such as English. Writing tests like this has the added benefit of being readable by non-technical people as well. RF is primarily used in acceptance and end-to-end testing and has built-in support for Gherkin. For this reason, BDD behaviour scenarios double as RF test case descriptions so a tester need not design the test case from the ground up but can focus on implementation instead.

Tests in RF are contained within test suites; each suite containing one or more tests. Each test is built from keywords which are essentially functions: they can have parameters, a return value and can invoke other keywords. A keyword generally does not have a state associated with it, however, it is possible.

Page objects

Page objects are a tool for test automation frameworks and programs which wish to access a view in a program. The view can be anything that can be displayed on a computer monitor. In our examples, all user interfaces will be made up of web pages and so each view will be a page. The idea is to build an interface on top of each web page or view which then provides access to the web page so the tests no longer directly access the web page but use the interface instead. In the case of RF, page objects can be written at least in Java, Python and as RF keywords. The effect of page objects is that any maintenance attributed to UI changes will be moved from test code to the page object code.

The role of page objects is shown in Figure 3. Our Java-like page object contains methods which can access the user interface. The methods may, for example, use Selenium to access a web page.

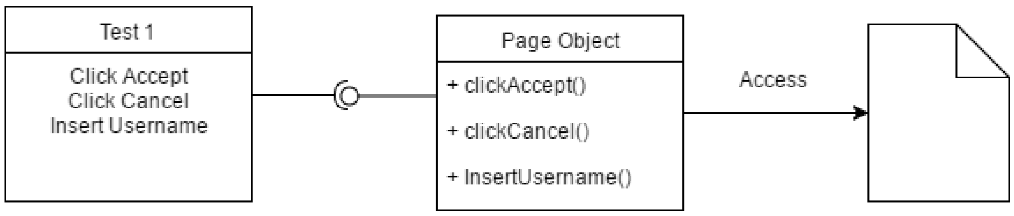

Figure 3: Page object's role

Leotta et.al conducted a case study [35] on the use of page objects and their effect on test maintenance. The study was conducted in a single company by having two versions of their test set: one built with page objects and one without. They then proceeded to make changes to the program which would cause some elements to be realigned or had their ID's changed so the automated tests would no longer work. Leotta et.al would then measure the time and effort it took to fix the tests. The results indicated that realigning the tests using the page objects pattern was 65.32% faster [35]. Peculiarly, the test showed lower productivity when using the page object pattern and using the measurement *Lines Of Code / Time*. However, this was attributed to testers copying and pasting solutions from one test to another rather than real productivity. Based on the study, the page object pattern seems to make automation testing more efficient as less time is spent fixing broken tests.

4.3.3 REST interface testing

Interfaces allow chopping up a big system into smaller components and has the advantage of allowing the use of external libraries. An interface may be anything from a programming language specific feature, such as Java's interface classes, to a simple text file where two programs read and write messages from and to. We will introduce here a fairly popular interface type that is used, also, in database-centric systems: REST.

REST

The *Representational State Transfer* (REST) is an architectural style originally introduced by Fielding et.al [16]. It was created to redefine the *Hypertext Transfer Protocol* (HTTP) and the *Universal Resource Identifiers* (URI) while still preserving what made WWW so popular. In effect, REST places constraints on architectures in an effort to minimise latency and network communication while allowing for independent components and scalability.

Structure

Originally REST was designed to provide an interface for a heterogeneous set of devices exchanging various types of data, ranging from pictures to binary files [16]. It was originally used in universities by scientists and researchers.

REST architectures are client-server architectures. The basic idea behind one is that it maps identifiers to resources and that it is stateless [16]. Being stateless, from the server's point of view, means that each interaction consists of a single request and a response, this provides several advantages [16]:

- No need to store state data.
- Since requests are made up of sequences, interactions can be parallel.
- Requests can be understood by an intermediary without causing excessive latency.
- All information is present in each request and response allowing for easy caching.

The server does not need to store state data. This frees up resources which might otherwise be taken by an inactive client. It also mitigates a Denial of Service (DoS) attack in which an attacker starts up sessions to hog all of the server's resources, however, should the server operate on TCP it is still vulnerable against opening TCP connections.

Since requests are self contained, interactions can be parallel. Quite often a client wishes to make multiple requests in a sequence to achieve a goal. For example, in order to get the names of every person in a group, a client might first have to ask a group for person identifiers — like keys in a database — and then request the name of each person one by one. But from the point of view of the server, each of these requests — for a group or a person — is not tied to any particular sequence and so the server can handle any number of these requests at the same time.

Requests can be understood by an intermediary without causing excessive latency. This is advantageous in an environment where messages are routed based on content. If the request would be in a sequence, the intermediary would have to remember any previous requests, possibly indefinitely.

All information is present in each request and response allowing for easy caching. The server can store responses to the most common requests and update them as necessary saving processing time.

Contracts

To create a meaningful test, a specification is required; a specification which adequately describes all the inputs, actions and outputs of a system. This specification is sometimes called a contract and it is created early on in the development process. While agile methodologies allow and even promote change during development, an interface contract is what keeps multiple modules, and thus the end product, in coherence. For example, it is not uncommon for two interacting modules to be developed by entirely different teams.

RESTful API Modeling Language or RAML [56] allows developers to describe REST interfaces in a manner which can easily be interpreted by programs. RAML is a subset of YAML, a data serialisation language. It allows developers to list resource paths, responses and payloads both ways, and a list of JSON or XML schemas depicting the contents of the payloads. RAML specifications are readable by humans and allow for meaningful testing of the REST APIs described. It is also possible to automatically generate the REST APIs directly from a RAML so the testers can focus on overall correctness rather than implementation details.

4.4 Testing pyramid

The testing pyramid is a model introduced by Cohn [9] shown in Figure 4. The purpose of the pyramid is to divide different types of tests into layers; each layer's size then corresponds to the portion of all tests a particular type should take. The idea behind the pyramid is that once the lower components have been properly tested, the higher level components can be built on a solid foundation.

Starting from the bottom, the lowest layer represents tests for a single module, also

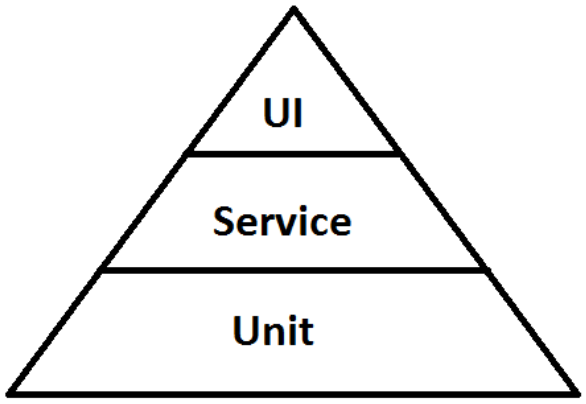

Figure 4: Test distribution pyramid

known as unit tests. These tests create the basis for a stable system [9]. As the individual units get tested, it is easier to build a reliable system on top of them.

The assumption behind the testing pyramid is that all systems can be thought of as a collection of services. The definition of a service is very broad; for example, a library class or even a collection of services could be considered a service. This means that the *service* layer of the pyramid contains all testing which is not *unit testing* or *UI testing*.

The top layer, UI, should require the least amount of tests. The benefit of UI testing is that the system is tested as a whole, as if a user were using it. However, a UI test is generally slower than a test which can invoke methods and functions, or even just RESTful interfaces, directly. Other problems include higher response times and the volatility of a representation, such as a UI, especially in the beginning of a project. The result of excessive UI tests are extended built times which slow down development.

4.5 Usability testing

The goal of usability testing is to find out how intuitive and satisfying the software is. It should be done on actual users or people who can act as if they were a user. A general rule of thumb is that the developer of a system cannot be a usability tester since their vision of what is good UI design may differ from the average user.

Some popular methods of usability testing are: remote usability testing, expert review and A/B testing. Remote usability testing is required when the user and the usability evaluators are far away from each other, for example in different countries. It involves setting up sufficient [53] monitoring systems such as a shared screen and audio recording. Expert review means bringing in a company which specialises in usability reviews. In A/B testing users are given two workflows to accomplish the same task. The feedback is then used to select the better option. Other methods exist, for example the hallway testing method in which seemingly random people are asked to participate in a usability test.

5 Metrics to support testing

A software metric is a tool used to measure different aspects of software. They can be used to show the effect a change has on the software being measured. Our interest is in metrics which can show how testing affects software and metrics which can show the testing itself is affected by changes in the testing strategy or the adoption of new technologies, methods and methodologies.

The metrics we are going to use will be chosen with the aid of the Goal Question Metric (GQM) [3] approach. The idea behind it is that: organisations have goals and metrics should be used to help achieve those goals.

A popular metric is code coverage and its various forms. We will briefly describe them and provide a use for them. We will also show why it is not a sufficient metric for measuring quality when used on its own.

To support code coverage, we will introduce the ripple effect in the context of Java. The ripple effect allows an automatic tool to analyse which classes are most prone to bugs by evaluating their connections to other classes. The basic idea behind it is that when a class is changed, each class using it may produce a failure. Therefore, a class with multiple connections is more likely to produce failures the more connections it has.

With the help of code coverage, we will do a brief explanation on testing adequacy. That is, how much testing is required until the test set is deemed adequate.

We will begin by briefly describing GQM. It is followed by coverage where we will go through a few common types of code coverage. Next, we will explain the ripple effect which can be used to target testing. We will close off this chapter with testing adequacy.

5.1 GQM

Originally defined in NASA, the GQM approach allows organisations and projects to find meaningful metrics to support their operation. Before it can be used, an organisation should have set goals for its projects. For each goal, a set of questions is derived. Questions, which when answered, should be enough to show if a goal has been reached. The answers come in the form of metrics, measuring some aspects of the project.

GQM is a hierarchical structure as displayed in Figure 5. At the very top are the goals. They are themselves derived from the organisation's and/or project's goals and should clearly define the purpose of the measurement, the target object, target issue and the viewpoint. The purpose of the measurement should describe what the organisation or project is hoping to achieve. The target object is one of: products, processes or resources. Target issue is the target object's property or a feature which we wish to address in our goal. The final part of the goal is the viewpoint which defines from which angle we look at the measurement. For example, the client or the food taster. Once the goal has been defined, it is split into relevant questions.

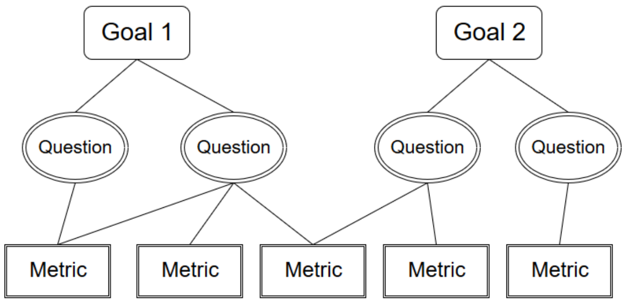

Figure 5: GQM [3]

In Figure 5, the middle layer contains questions which, when answered, should clearly indicate whether the goal has been reached or not. This means the questions should at least describe the current situation and whether the project is moving towards the goal. It is possible for multiple goals to have the same question, however, the viewpoint will likely change the answers — metrics.

The bottom layer in Figure 5 contains the metrics. The metrics are chosen so they provide answers to the middle layer questions. Each question may have multiple metrics attached to it and a single metric can serve a partial answer to multiple questions.

5.2 Coverage

Code coverage (or test coverage) is a metric for measuring how much of the source code is being invoked by the test set. The main types are [38]: function, statement, branch and condition. Function coverage tracks that each function or method is called at least once. Statement coverage is more meticulous and tracks that each statement has been executed at least once. Branch coverage means tracking that each path branching off of decisions is taken. For example, if-clauses redirect the program execution depending on if it evaluates as true or false. Condition coverage is a broader version of branch coverage: it tracks that every boolean sub-expression is evaluated true and false each at least once. Code coverage is almost always tracked automatically using tools such as Jacoco [23].

While measuring code coverage is a good way to find untested parts of the system, it is not a good measurement of quality, nor is high coverage an indication of an effective test suite [28]. This makes code coverage a somewhat deceptive metric as it easy to conclude, that if a part of the code gets executed without failures then it is bug free. However, it is easy to show that this may not be the case. For example, take the following library function which, according to its documentation, calculates the absolute distance between two numbers:

```
int difference(int a, int b) {
  return b - a;
}
```

The program works fine as long as $b \geq a$ but produces a negative number otherwise. A single test, such as verifying that *difference(1, 3) == 2* will result in a 100% code coverage but will not find the failure induced when $b < a$.

To truly prove that a function is bug free, one would need to employ tests which call the function with every possible input and compare the output to an expected value, that is, 100% *input coverage*. However, even our simple *difference* function, when assuming 32 bit signed integers, would require 2^{62} tests; clearly infeasible. Therefore, testers usually create tests where the input parameters form significant n-tuples. What this means is that there are input ranges for the parameters and all tests with parameters in a specific range will likely pass or fail together.

In our example function, a and b range from -2^{31} to $2^{31}-1$. We are also aware that a difference is calculated using subtraction; therefore, if $a, b \in [-\frac{(2^{31}-1)}{2}, \frac{2^{31}-1}{2}]$, then any calculation using a and b will not overflow. The absolute distance does not have any special inputs to consider, such as division by zero. With this we only need to consider the relative sizes of the parameters as long as they are within the specified range. Therefore, we only test $a < b$, $a = b$ and $a > b$ when $a, b \in [-\frac{(2^{31}-1)}{2}, \frac{2^{31}-1}{2}]$ to cover all the inputs in the range. Similar deductions can be used to cover rest of the ranges but in practice there is no time to construct such formalisms and testers need to rely on "rules of thumb": try the highest and lowest value, try one above the highest and one below the lowest, and so on.

In conclusion, a high code coverage is not a good indicator for software quality. It is possible that a correlation between *input coverage* and software quality exists, however, this has not been studied well enough as of now. While code coverage may not be a good indicator for quality, it can show which parts of the system go untested and where more tests are potentially required.

5.3 Ripple effect

When a change is made to a class, any classes depending on that class may need to adapt to the change which in turn may require changes in more classes; this is called the ripple effect: how a change propagates through a system. We will examine it from the perspective of a Java program.

From a testing perspective the interesting consequence of the ripple effect is that if a class is prone to the ripple effect — i.e. it is tied to many classes — then the class becomes a very likely source for bugs since changes to any of the classes it depends on may affect its behaviour. Arvanitou et.al have proposed a method for measuring the ripple effect: the Ripple Effect Measure (REM) [2].

REM works by capturing the amount of dependencies of each class and the propagation factor, unlike previous coupling metrics which do not take both into account [2]. The propagation factor is a number $x \in [0, 1]$, and describes a change's probability of propagating from the source class through a dependency into other classes. A dependency is one of: generalisation, containment or association. Generalisation represents a "is-a" relationship and is considered in three occasions: invocation of the *super* method, accessing protected fields and overriding abstract methods [2]. Containment represents a "has-a" relationship and means the use of a public interface of the source class [2]. Association means the use of the source class as a variable, parameter or a return type [2]. The formula for calculating REM (the propagation factor) for the dependent class A and source class B is as follows [2]:

$$REM_{A(B)} = \frac{MC + P + PrA}{M + At}$$

MC The number of method calls made from the dependent class to the source class, including the *super* method.

P If generalisation is being used, the number of protected methods in the source class.

PrA If generalisation is being used, the number of protected attributes in the source class.

M The number of all methods in the source class.

At The number of all attributes in the source class.

To calculate the propagation factor on a class level, one must take into account the propagation factor $P(A_i)$ for each pair $(A, B_i), i \in \mathbb{N}$ where A is the dependant class and B_i is a source class. The way to do this is with the *inclusion-exclusion* principle:

$$P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1 \le i_1 < \cdots < i_k \le n} P(A_{i_1} \cap \cdots \cap A_{i_k}) \right)$$

Once we have calculated the propagation factor for all classes, we can now focus testing the classes which scored a high value. However, while the theory is sound, there have been very few studies which would evaluate REM's effect on the quality of the software. Also, when used in context of other languages, such as C++, one needs to consider friend classes and methods, and similar functionalities [2].

5.4 Testing adequacy

Determining when enough testing has been done is no trivial task. However, a line should be drawn somewhere lest testing costs go up to the point where fixing the software afterwards becomes cheaper. Formally, the limit to testing is called test adequacy criteria. Once the criteria is satisfied, no additional testing is deemed necessary.

Test adequacy criteria can be can be defined in many ways, including but not limited to: detected faults, code coverage or use case coverage. We will introduce two types of testing criteria: for high level black box testing and for low level white-box testing. In black-box testing the tester does not know the underlying implementation and will test based on given documentation which can be, for example, a use case or a verbal description. In white-box testing, the tester is aware of the underlying implementation. An example would be a developer creating unit tests for a unit he has created.

5.4.1 White-box testing criteria

In 1975 in a book called *The Mythical Man-Month* [7] written by Brooks, he estimated that roughly 50% of development time in a software project is spent on testing. This estimation still lives on as a rule of thumb for estimating development time. In 1997 Hong Zhu et.al wrote a paper about testing adequacy [58]. In their paper they present criteria for what is an adequate test and a test set. The first thing they note is that there is no such method that could reveal all errors in a program and that testing should instead aim for a reasonable goal. They present the following criteria for testing coverage: statement coverage, branch coverage, path coverage and mutation adequacy.

While we showed with a simple example why testing coverage can be misleading in 5.2, it has been shown to be a decent indicator of test set quality. Specifically, statement coverage [21] and branch coverage [20] have been shown to be of use. However, Rahul Gopinath et.al [21] stated in their survey, that the other coverages they examined — branch, path and block — seemed to also do well and may work better depending on the project.

Another way to determine test set adequacy is by mutation adequacy. It is a well studied topic and the tools have reached a mature state [30]. We covered mutation testing in 4.1.5.

The conclusion we draw is that at least statement coverage, branch coverage and mutation adequacy are decent indicators for test set quality, with the caveat that forcing developers to up code coverage by any means necessary may lead to lazy tests, which execute a lot of code but may not be very effective. In addition to code coverage, one can employ mutation testing on the test set to measure the set's adequacy.

5.4.2 Black-box testing criteria

Since a black-box tester has no knowledge of the underlying implementation, there is very little motivation for coverage based test adequacy criteria. Instead, the tester needs to rely on documentation which can vary depending on the project. One way to use documentation to measure testing is to track which use cases have been tested.

6 The tools and methods currently used in PLP testing

In this chapter we will describe the current testing strategy of PLP. Our description will be based on our experience as an automation tester with almost two years of experience and on the knowledge gained through interviews and the occasional discussion. In the company, there are five groups of people who actively engage in testing: business analysts, manual testers, automation testers, developers and the management. We interviewed members of the other four groups to gain insight on how they participate in testing and how they fit into the testing strategy.

We had separately scheduled interviews with a business analyst, a developer and a manager. We did not schedule a separate interview with a manual tester as business analysts use the same tools and methods when testing. The interviewees had been on the company for at least six months and had experience from previous, similar occupations from other companies. We selected them so they could explain the tools they need to participate in the testing process and would have improvement suggestions or could point out weaknesses in the current testing strategy. The scheduled interviews had a set of prepared questions which are included in appendix 1, but we allowed the interview to flow when the interviewees brought up ideas and points of view we did not consider when making the questions. We recorded the interview with a microphone and by taking notes and then afterwards extracted the tools the interviewees used, the problems they pointed out and the improvement suggestions they had, and included them in our analysis and improvement suggestions.

Our description of the testing strategy begins by describing our target system, PLP. We will follow that with a brief description of the current testing strategy. Next, we will examine the goals set on testing by Profit. After that, we will describe how testing is done currently by describing unit testing, integration testing, manual end-to-end testing and automated end-to-end testing. Next, we will describe how communication plays into testing as a whole. Finally, we will describe the most common tools used by people involved in testing.

6.1 The Profit Life & Pension system

Profit Software is a software company providing insurers tools and solutions for managing insurance sales and management. Their flagship product, The Profit Life & Pension (PLP), is a DCS intended for managing investment insurances from when they are sold to the moment they eventually expire; i.e. when the money contained within the policy runs out due to being claimed by the beneficiaries or due to various fees reducing it to zero.

PLP stores more information than just the insurance policies it manages. The simplified form of the PLP's database is shown in Figure 6. The figure shows how a policy should have at least one client attached to it. However, it is common for an insurance policy to have a separate policyholder — the person taking the insurance — and an insured — the person the insurance is for. In addition, it is possible to have additional roles which may change throughout the policy's life. Other common features of investment insurances are the investment targets, of which there should be at least one, and any payments made.

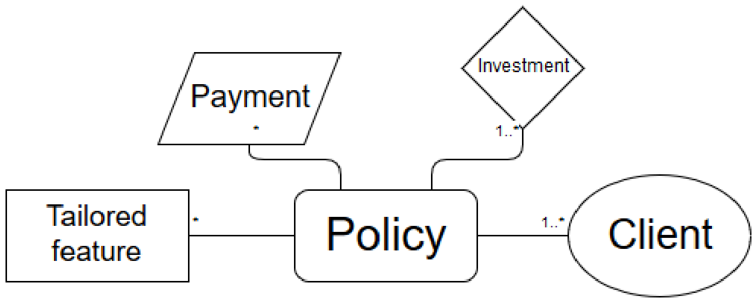

Figure 6: Data within PLP

It is these common features that allow PLP to have a core product on top of which Profit Software can add tailored features and modify the existing ones. Tailoring is an important part of PLP and it can affect essentially all features and data within it. In fact, PLP is built so that it can be configured to enable, disable or change most existing functionalities without having to rebuild the system from the source code; rebuilding is only required for new functionalities. The modularity is required because all client versions will differ from each other; at the very least, the insurances being offered through PLP are different for each client.

For testing this means that testing the core product alone will not guarantee that any of the tailored versions will work. Additionally, the core product's tests may not work in a tailored version which has lead to a situation where some tests have to be written more than once since the differences between versions are so great.

6.2 Current testing strategy

PLP's current testing strategy is mostly undocumented and most of the information is passed by word of mouth. Some of the agreed upon technologies and standards can be found from the documentation but the information is somewhat spread out. What makes matters worse, is that each tailored project has its own testing team and the teams only communicate sporadically. Fortunately, Profit Software has less than 50 testers which is likely why knowledge has spread reasonably well. I.e. the projects use the same technologies for the most part. Regardless, each project has its own version, a slight variation, of the testing strategy.

Recently, there has been initiative in bringing all projects to the same state in terms of technology and design. In practice, this means sharing the currently used technologies and tools between projects, and introducing BDD into projects which do not use it currently. This has definitely had a positive effect but it may not last for long once it is finished and each projects continues to evolve on their own.

6.3 Goals

At Profit, testing is used to improve the quality of the software by making it easier to maintain and by eliminating bugs as early as possible, preferably before a new feature is even completely implemented and merged into the code base. Maintainability, for each piece of code, is improved by creating regression tests which will detect some of the possible bugs if the code is changed. This is left mostly to unit tests as they take a relatively short amount of time compared to the end-to-end tests which can take several hours. The current goal is to move towards a BDD type of development where testers work alongside developers throughout the development of a feature. True to BDD, analysts would also have to get involved but isn't pushed as hard as getting testers to work with developers.

In addition to supporting development, testing is used to measure the integrity of a release — internal releases more than others. There are two kinds of releases: internal and customer releases. The former takes testing into account as a release can simply be deferred if a test does not pass. The latter, however, is almost entirely governed by other factors, mostly the client. That is not to say that testing results wouldn't affect a customer release, rather, the effect is dependent on the customer and any speculation or analysis on their decision process is beyond the scope of this thesis.

6.4 Unit testing

Unit testing should provide a steady foundation for regression testing but in PLP, they take a back seat to end-to-end tests. They are written by developers for new features but old features have not been tested all that much. Partially, this is due to the design which, in the past, has made testing difficult. There is no consensus on how unit testing is done; it is merely encouraged and each developer may choose to write tests before, during or after development. The developers do acknowledge the importance and usefulness of testing but sometimes testing is skipped in order to save time. In addition, when making changes to legacy components, testing is often omitted.

Unit tests are executed automatically once code is inserted into a repository. They are also included in the build chain so that if a unit test fails, the build chain stops. They may also be run locally at the developer's discretion.

The only metric in use for unit testing is code coverage — more accurately, statement coverage. It is used to create goals for unit testing, such as reaching 80% code coverage.

6.5 Integration testing

Integration testing is all testing not attributed as unit or end-to-end testing. This is where a partial system is deployed for testing. In Profit its main use is when a new module or a feature is being developed outside the main product. The new features usually mock PLP and therefore do not qualify as end-to-end tests. However, once the feature is merged into the main product, the integration tests start to run on top of the actual system and change into end-to-end tests. Although, calling these tests integration tests rather than end-to-end tests is merely an exercise in semantics.

Actual integration tests, which are reasonably far from unit or end-to-end tests, are used very little. In fact, we have only observed their use once; and they are not talked about very often when testing comes up. This leads us to believe that integration testing is not seen as necessary or worth the time and could therefore be an area of improvement.

6.6 Manual end-to-end testing

End-to-end testing comes in two forms: manual and automated. Manual testing is most commonly used for verifying bugs and new features before an automatic verification, or test, is implemented. If a bug is discovered, manual testing happens three times before automation: right after a bug has been discovered to verify it is actually a bug, once the bug has been fixed and, in some cases, before the release which ships with the bug fix. If a new feature is requested, manual testing is done once the feature is ready and before its release. Automation generally follows manual testing and is implemented as regression tests once a feature passes manual testing. Although, sometimes manual testing is skipped and an automated tests is implemented directly; this is especially true in a project where a BDD style of development is used.

Manual tests are made by business analysts or manual testers themselves. The process is started by a business analyst who creates a specification for a feature. A manual test flow is then created from this specification. In order to verify the feature efficaciously and efficiently, the test flow is designed in such a way that it attempts to test as much as possible with as few actions as possible; in other words, the test is optimised with respect to speed while still proving a meaningful test. Unlike automated tests, a manual tester does not necessarily have to stop if they encounter a bug during testing. Instead, the tester may continue and thus, create multiple bug reports over a single test flow execution.

An example of a manual test flow could be the following. Suppose there is a system X with views A, B and C. as shown in Figure 7. Suppose further that each view has an action to be executed which somehow changes the system and reflects that change in view A. A typical manual test flow would be the following: *Start from A, move to B, execute action in B, move to A, verify changes from the action in B,* *move to C, execute action in C, move to A, verify changes from the action in C.*

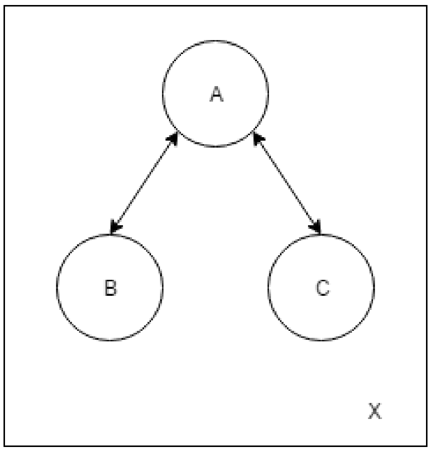

Figure 7: System X

6.7 Automated end-to-end testing

Automated end-to-end testing is done mainly through the UI with the exception of a few high level RESTful APIs. The test are based on three sources: specifications (or use cases), which describe a feature accurately enough for a developer to create an implementation based on it; manual test cases, which are a refined version of specifications and portray, step by step, how a user would use the system; and free descriptions which may vary in their accuracy. The specifications and use cases are always written by business analysts whereas the free descriptions may be written by anyone. Most automated tests are made for features but on a few occasions automated testing has been used to verify bugs as well.

The tests offer regression testing and play a significant role in the testing strategy. There aren't as many automated end-to-end tests as unit tests, but since the scope of a unit test is very small they end up covering a smaller portion of the system. However, the relative effectiveness of the unit tests when compared to end-to-end tests is unknown as unit tests, unlike end-to-end tests, are fast enough to be executed while a code change is still in the developer's own branch. For this reason, the bugs caught by the unit tests rarely show up after a change has been merged.

The effectiveness of end-to-end tests is measured by counting and inspecting regression bugs reported by automation testers. There have been attempts at integrating code coverage measurements into the end-to-end tests but thus far they have been unsuccessful.

The process of creating automated end-to-end tests for a feature is the following. A tester is given a specification, a manual test or a free description. A specification isn't necessarily following a standard notation such as UML but is loosely standardised within the company, that is, no formal guideline exists but the use cases follow common patterns. From here, automated tests follow three schools of design: a reflected manual test case, Gherkin oriented test case or a merger of the two.

A reflected manual test case, true to its name, attempts to mimic the manual test case which it is based on. This will produce a test case which has been reviewed by a business analyst and requires very little knowledge of the feature being tested but the test case will be lengthy will be designed for a manual tester rather than automated execution. One benefit that a manual test case has, is that a manual tester can continue working through the test case after they discover a bug — provided the bug does not prevent it — and could potentially discover multiple bugs during a single execution. However, an automated test should not continue once it discovers a failure, and thus, this benefit is lost. What we mean by this is that if we wanted an automated test to continue after a failed assertion, it would have to be programmed to switch to an alternative flow and since the set of possible failures is very large and mostly unknown, the test would quickly become convoluted.

The Gherkin notation has been adopted quite recently. It is used as a stepping stone while aiming for a BDD type development. The test cases written in this style are usually short and simple. When they are being created, the specification, manual test case or free description is broken down into multiple small tests, each testing a single action. In this approach, the other tests are not skipped even if one of them fails. However, extracting the relevant parts of a specification or a source requires a better understanding of the feature in question. The short form and near natural language test descriptions enable anyone to easily pick up what is being tested with the caveat that some business knowledge is usually required.

The third form is a merger of the two. These types of tests only appear when the testers are still learning the concept of BDD and are often plagued by the shortcomings of both styles — i.e. the tests tend to be long and the descriptions unclear.

In addition to UI tests, automation testers also write tests for some RESTful interfaces. These tests tend to use Java and sometimes resemble unit tests rather than the other end-to-end tests. Java is mostly chosen due to the company having a history with Java — and therefore a lot of expertise in it — but also because Java has a broad selection of third party libraries specifically created for REST interface testing, and is faster than the tools used for UI testing, even though those tools could also be used.

6.8 Communication

There are five groups of people who actively engage in testing: business analysts, manual testers, automation testers, developers and the management. The business analysts build specifications and create the manual test case flows. Manual testers then follow those flows to confirm that a feature works as intended. Automation testers usually follow manual testing and automate the test cases for regression testing. Developers implement features based on specifications and then create unit tests based on their implementations. They also work as consultants when a possible bug is found or if an existing behaviour is not present in the documentation. The management takes a bird's eye view of testing and allocate more testing to where it is needed.

As of now, most communication about testing happens through a static medium: documentation; and while it is reasonably comprehensive and somewhat formal, it can still be interpreted in different ways by different people. Of course, documentation in itself is a good thing but as long as the language and the style is created by only some of the groups, the rest may find it difficult to understand — or at the very least it may be a hindrance. Since the managers use the documentation only a little bit, only testers, developers and business analysts read the documentation on a regular basis. However, the business analysts govern how it is written.

6.9 Testing tools

Testing can be though of simply running the program, giving it predetermined inputs, and checking the product results against expected results. However, in this form, testing does not scale well as it requires the tester to spend a lot of time setting up the environment, running the program and other related tasks. This is where the tools come in. In Profit, there are tools for each group involved in testing: manual testers, automation testers, developers, business analysts and managers. Out of these groups, the automation testers are the most dependent on tools. Indeed, the very definition of an automation tester requires the presence of a framework or software which allows automated tests to be run. In this context, by tools we mean any program, methodology or a process which somehow enhances testing efficiency. As a direct result, JUnit is as much a tool as the page object model. We will go through the tools used by each group and explain where and how they are used.

6.9.1 Tools for automation testers

Automation testers operate on the UI level with the addition of select few RESTful interfaces. Therefore, most of their tools assist in end-to-end UI testing. We will go through the tools of an automation tester and then provide a plan which represents how a UI test is implemented including all the tools involved.

Gherkin

Gherkin is the name given to a CNL used originally by Cucumber [36]. It was created to go hand in hand with BDD. The grammar of the language is simple, reserving only a handful of words and only when they are the first word in a sentence or a paragraph. In profit, Gherkin is used mainly for automated tests and occasionally for documentation. However, its current use does not reflect the original purpose where Gherkin is only a part of BDD.

The original implementation of Gherkin was introduced with the testing framework Cucumber. It is a framework used for executing automated tests described as features containing scenarios. The scenarios then contain test steps which are implemented in Java, for example. This style of describing tests reflects Cucumber's strong support for BDD. However, Gherkin has been implemented in a number of tools and one of them is currently in use in Profit.

The current implementation in use at Profit is the one provided by RF. RF does not support BDD as strongly as Cucumber and instead of features and scenarios, RF has — more traditional — test suites containing tests. They do have a one to one correspondence so one can simply name test suites as features and tests as scenarios. Our examples will use the implementation in RF.

Like other CNLs the purpose of Gherkin is to provide the means to write natural language-like code. The language has very few reserved words and only three of them are at the core: *given*, *when*, and *then*. While not necessary, some implementations reserve *and* and *but*, to make the source code more fluent. Cucumber and RF also reserve words for describing the structure of the test file. Keywords such as *feature* or *scenario* are reserved in the original, Cucumber, implementation while RF reserves expressions like *Test cases* and *Variables* to separate different sections of a test file. Here's an example test case written in RF's language:

```
*** Test Cases ***
Scenario: delete an account
  Given an account exists
  When I log in as the administrator
  And I delete the account
  Then the account is no longer found within the system
```

The test serves two purposes: an automated test and a documentation of the *behaviour* being tested. Since documentation is usually written before a test, using this style for documentation provides test automation as a side effect, i.e. the automation tester need only to implement the test steps rather than trying to extract the relevant flows from a specification. Furthermore, conforming to the *given-when-then* pattern means that the tests are often simple since they can only test one particular action, the *when* part. This in turn translates into easier maintainability. As a side effect, using Gherkin in tests brings projects closer to using BDD which could bridge the gap between tests and specifications.

Page objects

Page objects are in use in Profit but have mostly been written in RF keywords. However, there have been fruitful instances where the page objects have been written in Java. The advantages of Moving to Java are the expertise already present in the company and the possibility to uncouple the current testing framework from the page objects, thus allowing the page objects to be used elsewhere, with JUnit, for example. However, rewriting the current RF keyword page object libraries in Java would take time and, therefore, money. In addition, the benefits of such an undertaking are questionable unless there is a need to switch from RF.

Support libraries

In addition to page objects, automation testers have libraries which provide useful keywords for the RF tests. These libraries include parsers, HTTP request libraries, SSH libraries, etc. They have been written using any of: RF keywords, Python or Java. Out of which Python keywords have almost all been converted to Java. In addition, RF keywords may execute Unix shell scripts.

Java support libraries tend to work well and have the advantage of a general purpose programming language. This generality allows the Java libraries to handle all tasks required by the tests and are therefore an adequate choice for a test support library language. RF keywords, however, are not.

RF keywords were not built to be a general purpose programming language. Their strength lies in their expressiveness but they, for example, lack the ability to do basic arithmetic or parse JSON strings.

Database manipulation tools

Almost all data in PLP is stored in a database. Therefore, the state of the database is effectively the state of the system. When creating tests for the more complex scenarios, the database sometimes has to be brought to a certain state before the actions being tested can be performed and checked. There are two ways to go about this: interaction with the UI and directly accessing the database. Both are in use in automated tests but the majority of tests use the former. The biggest disadvantage, when using the UI for the set up, is the speed at which tests execute: UI adds another layer which will invariably slow tests down. The problem is that setting up the system takes a much larger portion of the total execution than the action being tested and its assertions. For example, bringing the system to the desired state, so the action can be performed, can take up to ten minutes, while the action and the assertion may take less than ten seconds.

Modifying the database directly is done very meticulously and rarely as it requires knowledge of the underlying system. Currently two types of tools are in use when modifying the database state: importing and exporting tools. This fits automated tests fairly well because even if test data needs to be created through the UI, it only has to be made once, exported and then imported when the test runs. However, a tool for creating the test data, while bypassing the UI, would hasten test creation.

6.9.2 Tools for manual testers and business analysts

We will include manual testers and business analysts as a single group since their tools are practically identical. The only difference is that business analysts tend to focus more on the documentation unlike manual testers who focus on PLP. All the documentation is stored using an external service. The service contains all documentation except for the source code.

Manual testing is the most flexible type of testing as a tester can adapt to sudden changes unlike an automated test but manual testing is also much slower than an automated test — assuming we do not count the time it takes to create an automated test. The biggest time sink in manual testing is setting up the system to a desirable state. Therefore, the vast majority of the tools for manual testing should help prepare the target system. Currently, the tools consist mostly of: scripts, some of them integrated into web pages for anyone to use; virtual machines, running personal copies of the software; GUI tools, for accessing databases; and the same exporting and importing tools available for automation testers.

6.9.3 Tools for developers

Within profit, developers are responsible for unit testing and integration testing. The former is more widely used and is the main form of testing done by developers. The latter only happens occasionally. The current testing plan does not enforce unit testing but merely encourages it. This leads to situations where sometimes unit testing is skipped in order to save time.

Most unit tests are done using a testing framework such as JUnit or TestNG. The finished tests are then run at least during every build and every release, that is, they become a part of continuous integration. A developer can freely choose their environment and most development tools. There are a few mandatory tools including: a dependency management tool, build tool, the testing framework and version control. On one hand, having the freedom to choose their own tools has the benefit of not making developers use tools which they would dislike. On the other hand, this heterogeneity means that you cannot create a plugin for a specific IDE and expect it to work for everyone.

6.9.4 Tools for managers

The managers do not test directly. However, they are responsible for managing projects and are are, therefore, interested in testing. They approach testing mainly through reports and issues in the issue management software. The information required for any reports come from testers and so, require no tools besides a messaging application. The issue management software, however, is a tool which allows managers to monitor how much time needs to be allocated to testing.

7 Suggested improvements

This chapter is a collection of improvement suggestions we are making to the PLP's current testing strategy. The improvements in this chapter have been chosen on purpose so that they are small, reasonably easy to implement and do not require giving up the current tests. Our reasoning for this is that small changes are more likely to be adapted than big ones.

In the first part of this chapter we will describe our suggestions pertaining to automated end-to-end test design. We will then continue with end-to-end testing by describing the current issues with lacking debugging tools. Next, using GQM, we will produce a set of metrics which would help tracking the goals of testing and the effect our suggested improvements are having. After that, we will describe and provide motivation for a tool which could be used to directly create test data. Then, we make suggestions as to how testers could gain better domain knowledge and how it would improve their testing efforts. After that, we will offer suggestions for unit testing. Finally, we recommend adding the testing strategy into the documentation so it can be easily found by all those concerned.

7.1 Designing automated end-to-end tests

The two biggest problems in the current set of automated tests are execution time and complexity. The execution time is generally longer than 10 hours and therefore cannot be leveraged by each developer in the development process. Rather, the test set can only be run once a day. The second problem is test complexity. The test set has gone through multiple frameworks and design patterns, some of which have been automatically generated, which has resulted in a rather heterogeneous barrage of tests and styles to design them.

We believe the best way to address these problems is by standardising the design and providing a guideline to match so new tests, and refactored tests, will generally look the same. Simply put, the standard we propose is to follow a Gherkin style notation, whenever possible, while keeping the tests as small as possible. Small tests are easier to read and understand which will reach our first goal of reducing complexity.

The second goal of decreasing execution time has three parts to it: parallel execution, test code optimisation and test tagging. Parallel execution from the point of view of design means deciding on a smallest test unit, which cannot be split further into smaller parallelisable units. Code optimisation can then be used to further increase the execution speed of these units. Finally, test tagging can be used to execute relevant tests only so a developer could conceivably use the automated tests without having to run the entire 10 hour long test set.

7.1.1 Decreasing execution time by design

Parallel execution

The purpose of tests is to catch bugs as early as possible so future development isn't built on top of misbehaving software. For this reason, all tests should be run as often as possible. For end-to-end tests, there are at least two ample opportunities: once a developer is ready to merge their changes to the main branch and when all mainline components of the software are merged together. The former does require that any changed components can be built and combined with the current version of the system. Both opportunities allow for parallel execution but the developer's own test run could also benefit from test tagging.

Combining all the mainline components is done automatically and will therefore have only one user, the build system. So, it requires only one set of environments on which to perform a parallel test. However, if a developer wishes to test their changes against the end-to-end test set, they'd need their own set of parallel environments. This may not be feasible as it would bear additional costs and would not be wise as the environments may only be used sporadically. Therefore, a shared set of parallel environments would likely be a better alternative. This could offer developers faster feedback on their changes.

However, even with an optimal parallel test run with 10 environments, it would still take more than an hour to complete the test run. This is way too long for a developer to just sit around and wait. Therefore, we further suggest the use of tags in tests. The tags could be used to indicate accurately which features a particular test tests. Additionally, this is supported by the current testing framework.

Since all features are tied to a particular piece of documentation — a use case — the use cases could be given unique IDs which could then be used as tags in the tests. This would allow a developer to narrow the tests being run to a relevant subset and would help discover bugs directly related to the changed component much faster when the less critical tests are skipped. It is possible that a bug silently propagates through a system and is only visible on a completely different component, but we believe this is an acceptable risk, as the bug would then be caught later in the mainline merge — assuming the test set could catch it.

The mainline merge would benefit from a parallel test run because, at the moment, the test run takes a lot longer than the rest of the build chain. Right now, there is only time for one test run per day, but with the decreased testing time of the parallel test run, it is possible to do multiple test runs each day. The total time for a parallel test run has been roughly $\frac{x}{n}$, where x is the total time for a normal test run and n is the number of environments available for the parallel run. Although, this trend will not continue indefinitely as more environments are introduced and will finally come to a stop when the total execution time is the execution time of the slowest test unit. Parallelism is already being used in some projects in Profit and is in the process of being added to the other projects as well. Since all projects use the same testing framework, most of the groundwork related to parallelising the test run has already been done.

Optimisation

Another way to decrease execution time is to optimise the test code. Since optimisations will vary depending on each situation we will introduce two very common optimisations as examples. The optimisations are: combining test cases and returning to a particular view.

When combining test cases, one should take care not to potentially hide bugs. What this means is that if you have two consecutive assertions in a single test and the former of those fails, the latter never gets executed as the test fails right there. This is not a problem, however, if the two assertions depend on each other, i.e. the latter cannot pass if the former fails. Take the following two test cases:

```
Scenario: search bar exists
  Given a browser is open
  When I type the address of my favourite search engine to the address bar
  Then the page I’m looking for has a search bar

Scenario: search bar is editable
  Given a browser is open
  When I type the address of my favourite search engine to the address bar
  Then I can type text into the page’s search bar
```

The execution time for both test cases is quite short. Suppose, opening a browser takes roughly five seconds, loading page takes a second and the rest of the actions put together, when performed by a web driver, will take one second. In reality, the durations may vary but these estimations will be enough for a relative comparison. With these estimations the execution time for both tests together come up as roughly 14 seconds. However, since you clearly cannot type text into a search bar if the search bar isn't there, you can combine the tests:

```
Scenario: search bar exists and search bar is editable
  Given a browser is open
  When I type the address of my favourite search engine to the address bar
  Then the page I'm looking for has a search bar
  And I can type text into the page's search bar
```

By combining the two, the execution time is reduced by almost half as now only one browser is opened and the web page is opened only once. Additionally, the new test case conforms to the Gherkin model. The trade-off is that the new test is a little bit longer than the first two.

Sometimes it is not possible to combine tests without potentially hiding bugs. In that case, you can decide that each test assumes it starts with the search engine page already open. It would look like this:

```
Setup:
  Open a browser
  Type the address of my favourite search engine to the address bar

  Scenario: search bar exists
    The page I'm looking for has a search bar

  Scenario: search bar is editable
    I can type text into the page's search bar
```

The *Setup* is executed only once. The trade-off here is complexity as the test no longer starts from scratch. Additionally, if our tests were to change the current web page, they would need to return the browser to the initial state, adding more complexity into the mix.

7.1.2 Standardising tests

One of the reasons to adapt *Gherkin* was to reduce complexity in test cases, thus making them easier to maintain and understand. Especially old test cases have problems with complexity, mainly because they are easy to chain. To explain chaining we should consider this simple model for a test:

Initial state: The system is in a specific state

Actions: Actions are executed which affect the system somehow

Assertions: The results of the actions are checked

In and of itself this produces simple test cases which can be easily split into three phases but the problem comes when one realises that the end state, after the verification, is actually the initial state of another test, whose end state is the initial state of yet another test and so on. A *chained test case* such as this can become exceedingly long. Chained test cases have at least three problems which make them undesirable: added complexity, increased debugging time and the fact that a failure causes all following assertions to be skipped.

Our recommendation is to solve this issue by creating all tests using Gherkin, whenever feasible. This leads to tests which have an initial state (Given), actions (When) and assertions (Then), rather than a chained test case which would in result in, by abusing Gherkin, *Given-When-Then-When-Then-...*. We acknowledge that this requires testers to actually follow the idea behind *Given-When-Then* and do not mix actions with assertions which would qualify as *side effects*.

We have discovered that some of the keywords used in tests introduce side effects. That is, a keyword claiming to be an action keyword may actually perform assertions and the other way around. Some of these are legacy code from years past and should be removed as they no longer serve any known purpose. However, some of them have been added to make debugging the tests easier. In our experience the new assertions have not caused any problems but should be kept in check and have comments so they will not eventually become legacy code.

7.1.3 Gherkin and page object use case study inside the company

While working at Profit software we were put on charge at automating the testing of a new project. In addition to creating automated tests we had two additional goals:

evaluate the use of behaviour scenarios and the use of Java in test automation. To reach the first additional goal, we designed all tests using behaviour scenarios and reported how the other members of the project responded to it. The behaviour scenarios were written using Gherkin, i.e. the *Given-When-Then* model described in chapter 6.9.1. The second goal was reached by using Java to create page objects and libraries for the automated tests.

We used technologies already in use in other projects — most prominent ones were Robot Framework, Selenium and Java — but page objects, libraries and tests were created from scratch. The project included a test automation engineer, two developers, an architect and a project manager. Testing was done incrementally, as new features were implemented, with the help of developers whom would review the Gherkin test cases and informed us if there were scenarios that weren't being tested.

We found, that Gherkin allowed the developers to understand the test cases without the need to learn Selenium or Robot Framework and were easily able to verify our test scenarios even though they had no previous experience with Gherkin. We conclude from this that Gherkin can be used as a common language between developers and testers to describe functionality.

In previous projects, where most of the functionality was written using Robot Framework, we have noticed that writing complex algorithms can be a very arduous task. For example, even a simple calculation, such as $a+b$, where a and b are variables, must resort to calling a separate Python expression as the Robot Framework's language does not have such functionality. We found, isolating such algorithms and expressions into Java made the Robot Framework tests much shorter and easier to read.

While the tests were implemented using Robot framework and Selenium, we also used a page object model to separate the user interface from the robot keywords. This creates a three layer hierarchy in the test cases. The top layer consists of the robot tests. The tests comprise of the Gherkin keywords written in documentation, other robot keywords and the keywords from the page object library. The middle layer is the page objects library, providing access to the browser for the robot tests. This separation allowed us to easily fix test cases when they were broken by updates to the user interface. For example, if a field changes its type from *input* to *select*, only the page library needs to change as the keywords used by the tests remain the same. However, in this case in the name of readability, we changed the keyword's prefix from *insert* to *select*, e.g. insertPhoneNumber to selectPhoneNumber, which in turn did require a small change in the test cases. The bottom layer contains the user interface, the browser and the Selenium API. These are being accessed in the page objects library using the Selenium WebDriver.

7.1.4 Test failure rates

In optimal conditions, each failed automated test indicates a bug has been found. However, automated tests are not perfect and can fail just like any piece of code. Thus, when an automated test fails, a tester has to analyse the failure and determine its source. The source is usually one of: a bug in the system, a bug in the test due to a change in the specification, an environment problem or an unstable test. The first two require a bug report about the system or the test but only happen once. Similarly, the environment issues are generally ignored unless they happen constantly. However, an unstable test is harder to ignore. An unstable test is a test which tends to fail either constantly or at random. This can make analysing the test very difficult and will continuously create extra work for testers unless it is fixed. Furthermore, it may create work for others if the failure looks like a bug in the system.

Our proposition for fixing this issue is to separate — or quarantine — these known unstable tests from the other tests. This has already been done in some projects but should spread to the others as well. The reasoning behind this is that randomly failing tests cause a lot of work for testers but provide very little in terms of quality assurance. This is because tests like these quickly lose credibility and their failures will simply be ignored in subsequent test runs. We also propose a policy that an issue is added into the issue tracker for each quarantined test case so there will be a constant reminder to fix them.

7.2 Debugging end-to-end tests

When an unstable test fails, a tester may decide to fix it. The current method for fixing a test case is the following: run the test, change the code, run again and repeat until the bug is no longer present. The reason behind this method is because it is often not possible to continue from where the previous run failed. Since it is very common for the procedure to take multiple changes and test executions, chained test cases pose a problem as it can take a long time to get feedback on the code changes. Sometimes it is possible to run only a small part of a test which hastens the procedure. Although, this is an exception to the rule. In short, test debugging sessions usually take a fairly long time and more so for chained test cases.

We propose three solutions to shorten test code debugging session. First, we suggest test cases be split into smaller pieces wherever applicable or at the very least break chained test cases into its constituent parts. This would take potentially a lot of time upfront but would make debugging much faster. Second, a tool for easily setting the system to a desired state. We will look into this more in-depth in chapter 7.4. The third is a debugging tool.

Two important features of a debugging tool are *breakpoints* and *step-by-step execution*. A breakpoint is a feature commonly available in IDEs. It allows a developer to create a marker in the source code which will cause code execution to stop until the developer allows it to continue, usually step by step all the while allowing the developer to monitor data used by the program, such as variables and arguments. Currently, breakpoints can be simulated by stopping execution. However, step-by-step execution is only available for Java libraries and for all end-to-end tests when using a special IDE for Robot Framework called RIDE. A tool like this may not exist at the moment but even a rudimentary one would help the debugging process.

7.3 Additional metrics

Metrics can be used to monitor a software project at the expense of some overhead work. They are presented as a sequence of measurements done during the project's life. We will use GQM to select metrics which support the goals set for testing.

The current testing metrics are basically the same throughout all the projects in Profit Software. The metrics are: number of test cases implemented, number of failed test, number of passed tests, the total execution time of all tests and code coverage (for unit tests only). Infrequently, there have also been reports done on feature coverage but these have been rare as they require a fair bit of manual labour. These metrics can provide insight into testing but without clearly set goals are not as useful.

7.3.1 Goals for testing

Testing goals should support business goals and provide meaningful information for decision making [8]. Since PLP is an investment insurance management system, it must routinely handle calculations involving money. It is imperative that the calculations are done accurately so policies do not get billed or paid too much or too little. If such errors were to happen it would reflect negatively on the insurance company, and if PLP would be the cause of the error, the insurance company would be less inclined to continue using PLP. Therefore, one goal for testing is to reduce the number of defects reported by the client.

Within Profit, there have been issues with test maintenance taking too much time. The maintenance is usually brought on by tests which fail on their own, even in the absence of bugs. This is a good indication that such a tests should be properly fixed. Therefore, the second goal — and the goal for our improvements — is to improve the stability of the unstable tests from the testers' perspective.

7.3.2 Suggested metrics

With GQM we present Table 1 for tracking the overall goal of testing. To find out if testing is reaching its goal, we have devised two metrics. Any valid bugs the client reports on a delivery should be counted up until the delivery becomes obsolete. This means that the metric M1 will not be available immediately. Once both metrics M1 and M2 have been measured for a delivery, they should be stored. Once more than one pair of measurements have been stored, they can be compared to see how the number of bug reports from a client changes as the time spent on testing, relative to development, changes. We note that even if such a correlation exists, it does not necessarily indicate causation. However, if there is a causation between time spent on testing and bug reports, a correlation should exist as well.

In Table 2 we introduce metrics to track our improvements. The metrics are meant to track the amount of time spent on unstable tests to provide motivation for fixing them. We have also included a metric for the number of unstable tests so once they are being fixed, the progress can be monitored.

7.4 Inserting data directly into the database

Because PLP needs to handle records on the order of 10^8, it requires a structured database to store it all. In fact, the state of the database is effectively the state of the system. During manual and end-to-end testing this state is changed through the UI, with very few cases where the database is manipulated directly. Changing the state through the UI is slow as it introduces an extra layer to the data, yet it

Goal	Purpose:	to reduce
	Issue:	the number of
	Object:	defects
	Viewpoint:	reported by the client
Question	Q1	How many defects are reported after a delivery, relative to the amount of development time?
Metrics	M1	$\frac{B}{t_d}$
Question	Q2	How much time was spent on testing relative to the amount of development time?
Metrics	M2	$\frac{t_d}{t_t}$

Table 1: Tracking the overall goal of testing
B = Reported bugs in the last delivery
t_d = Time spent on development (only the time developers used)
t_t = Time spent on testing

Goal	Purpose:	to improve
	Issue:	the stability
	Object:	of unstable tests
	Viewpoint:	from the testers' perspective
Question	Q3	How much of the testers' time is spent on the unstable tests?
Metrics	M3	Total time spent on testing
Metrics	M4	The testers' estimate on how much time is spent on unstable tests
Question	Q4	How many tests are affected?
Metrics	M5	Testers' estimate on the number of unstable tests

Table 2: Tracking our suggestions

is necessary to test all the logic which resides between the UI and the database. However, testing almost always aims to test a particular behaviour or functionality which can only be invoked once the system has been brought to a specific state. Achieving this specific state often takes more time than the target functionality and any assertions associated with it. For example, we have observed automated tests which take five minutes each from which only 10 seconds are spent on the target functionality and assertions. This brings us to suggest a tool which could be used to bring a system to the desired state by directly inserting data into the database.

Currently, Profit Software has tools which accomplish a similar task but are somewhat limited. There is a tool which allows users to import and export basic data to and from the database. However, it is limited by the fact that it cannot create new data and does only basic validations on the imports, such as checking data types. Another, more powerful, tool called the conversion engine — which is used when importing data to PLP from other systems — has better validations but is also limited to importing existing data. If there was a tool, capable of creating input files for the existing tools, it would be possible to chain them to insert completely new data into the database.

Such a tool would especially benefit manual testers and business analysts if they could easily instruct the new tool chain to create a policy, for example, rather than clicking through the UI and running batches which can take more than an hour. Automation testers might also benefit from it but not as much since automated tests are generally fine with the import/export strategy. We acknowledge that creating a tool like this would definitely take time, experienced developers, business analysts and testers to build. But, the potential amount of time saved should at the very least warrant an investigation as to whether it is worth it or not.

7.5 Domain knowledge in automated testing

If a test engineer were to be able to implement a test case defined only using *Gherkin* and only using a few rows, e.g.

- Given An Ice Cream Product Exists
- When A User Orders An Ice Cream
- Then The Ice Cream Is Delivered To The User

The test engineer needs to know what it means for ice cream to exist in the system, how a user can order ice cream and how can one know when the ice cream is delivered. This is what is known as *domain knowledge*, i.e. knowledge of the underlying system and how it operates. This knowledge may not be inherently available to the test engineer but is, for example, known by a business analyst.

This creates a controversial situation where a test engineer is testing a feature she does not understand. It is still possible to test it because the business analysts have created a step by step instruction on how to manually test the feature. However, now the test engineer may not be able to break the manual test into logical pieces but instead automates a lengthy chained test case which we talked about in 7.1.2.

One possible solution is that the test engineer and the analyst discuss the matter or possibly create the test case together. Though, this approach would mean that both the tester and the test engineer should be available at the same time. Asynchronous communication, such as email, would also be possible but might delay the test case.

Another possibility, and the one we recommend, is that the tester participates in the meetings where the features are being designed. This way, the test engineer will not only understand how to feature works but can also provide input about how and what could be tested using automation.

7.6 Unit testing

The problems we uncovered in unit testing are the following: they are difficult to create for the old parts of PLP and there aren't enough of them. The former would require more developer insight than we currently have and will leave it for future work. For the latter, however, we can suggest an improvement.

Currently, the test distribution is not according to the testing pyramid. A lot of the testing effort is put into the very top layer, the end-to-end, or UI, tests. This poses a problem as they are too slow to be executed within a reasonable by a developer, whereas unit tests are fast enough. We suggest that testing effort should be geared towards unit testing. We know that some testers are able to understand and write Java and could therefore be tasked with writing additional unit tests. This would require some training as to how PLP works and how it is being developed, however. Assuming more unit testing will be done, in order to target the testing efforts, we further suggest using the ripple effect to find error prone classes.

Simply adding more tests and increasing coverage is not necessarily a good way to measure test set effectiveness. Therefore, we suggest the use of mutation testing on the current unit test base to find out how well it can find faults — i.e. kill mutants.

7.7 Documenting the testing strategy

While studying the current testing strategy we discovered that there is no single source of information and that a lot of the knowledge is passed by word of mouth rather than having been documented. We suggest that the testing strategy is added is added into the documentation so testing can follow common patterns and standards.

We acknowledge that many projects may have technological or other limitations and restrictions imposed by the clients. Therefore, a company wide testing strategy would have to be such that it would allow projects to accommodate these limitations and restrictions. The details of a company wide testing strategy would have to be agreed upon by a committee consisting of developers, managers and testers from each project. In addition, we suggest that each project have its own testing strategy document detailing the project specific details. We still recommend these project specific strategies to be as similar as possible so testers can be moved from project to project without having to learn everything from scratch.

8 Conclusions and future work

We set out to provide improvement suggestions for our target system, PLP. To help determine if we reached that goal, we divided it into these three research questions:

- Q1 Does the current testing strategy contain problem areas?
- Q2 How could we improve those areas?
- Q3 How could we add to the current testing strategy to make it more comprehensive and more effective?

The first questions we answered by examining the current testing strategy in chapter 6. We managed to spot problem areas in all levels of testing, especially in automated end-to-end testing as we worked actively on that across multiple projects.

The second question we answered in chapter 7. We took the problems we found in chapter 6 and made suggestions on how to improve them. The most important improvements we suggested were how end-to-end tests should be designed and implemented. We suggested that end-to-end tests should follow the Gherkin design so the tests would become easier to understand and maintain.

The third question we also answered in chapter 7. We pointed out possible processes that were missing in the testing strategy and described how they could improve testing. The most important missing feature we suggested is adding the testing strategy into the documentation so it can be viewed by all.

For future work we found that a tool which could be used to directly insert data into the database, bypassing the UI could be beneficial but would require further investigation as to its viability.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" In: *Proceedings of the 27th International Conference on Software Engineering.* ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 402–411. ISBN: 1-58113-963-2. DOI: `10.1145/1062455.1062530`. URL: `http://doi.acm.org/10.1145/1062455.1062530`.

[2] E. M. Arvanitou et al. "Introducing a Ripple Effect Measure: A Theoretical and Empirical Validation". In: *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on.* 2015, pp. 1–10. DOI: `10.1109/ESEM.2015.7321204`.

[3] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering.* Wiley, 1994.

[4] Cédric Beust. *TestNG.* 2016. URL: `http://testng.org` (visited on 11/07/2016).

[5] Thirumalesh Bhat and Nachiappan Nagappan. "Evaluating the Efficacy of Test-driven Development: Industrial Case Studies". In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering.* ISESE '06. Rio de Janeiro, Brazil: ACM, 2006, pp. 356–363. ISBN: 1-59593-218-6. DOI: `10.1145/1159733.1159787`. URL: `http://doi.acm.org/10.1145/1159733.1159787`.

[6] L. Borner and B. Paech. "Integration Test Order Strategies to Consider Test Focus and Simulation Effort". In: *Advances in System Testing and Validation Lifecycle, 2009. VALID '09. First International Conference on.* 2009, pp. 80–85. DOI: `10.1109/VALID.2009.30`.

[7] Frederick Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975. ISBN: 0-201-00650-2.

[8] Yanping Chen, Robert L. Probert, and Kyle Robeson. "Effective Test Metrics for Test Strategy Evolution". In: *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research.* CASCON '04. Markham, Ontario, Canada: IBM Press, 2004, pp. 111–123. URL: `http://dl.acm.org/citation.cfm?id=1034914.1034923`.

[9] Mike Cohn. *The Forgotten Layer of the Test Automation Pyramid.* 2009. URL: `https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid` (visited on 03/19/2016).

[10] Various contributors. *Jenkins*. 2016. URL: `https://jenkins.io/` (visited on 11/07/2016).

[11] Patrick Crews and Giuseppe Maxia. *Employees Sample Database*. 2015. URL: `https://dev.mysql.com/doc/employee/en/` (visited on 06/06/2015).

[12] L. Crispin. "Driving Software Quality: How Test-Driven Development Impacts Software Quality". In: *Software, IEEE* 23.6 (2006), pp. 70–71. ISSN: 0740-7459. DOI: `10.1109/MS.2006.157`.

[13] CUnit. *CUnit*. 2016. URL: `http://cunit.sourceforge.net/` (visited on 11/13/2016).

[14] EPFL. *Scala home page*. 2016. URL: `http://www.scala-lang.org/` (visited on 09/04/2016).

[15] Facebook. *Facebook*. 2016. URL: `www.facebook.com` (visited on 11/07/2016).

[16] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 407–416. ISBN: 1-58113-206-9. DOI: `10.1145/337180.337228`. URL: `http://doi.acm.org/10.1145/337180.337228`.

[17] Finanssivalvonta. *FIVA homepage*. 2016. URL: `http://www.finanssivalvonta.fi` (visited on 08/29/2016).

[18] Martin Fowler. *Continuous Integration*. 2006. URL: `http://www.martinfowler.com/articles/continuousIntegration.html` (visited on 12/20/2015).

[19] Robot Framework. *ROBOT FRAMEWORK*. 2016. URL: `http://robotframework.org/` (visited on 10/17/2016).

[20] Milos Gligoric et al. "Comparing Non-adequate Test Suites Using Coverage Criteria". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: ACM, 2013, pp. 302–313. ISBN: 978-1-4503-2159-4. DOI: `10.1145/2483760.2483769`. URL: `http://doi.acm.org/10.1145/2483760.2483769`.

[21] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code Coverage for Suite Evaluation by Developers". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 72–82. ISBN: 978-1-4503-2756-5. DOI: `10.1145/2568225.2568278`. URL: `http://doi.acm.org/10.1145/2568225.2568278`.

[22] B. Hoisl, S. Sobernig, and M. Strembeck. “Comparing Three Notations for Defining Scenario-Based Model Tests: A Controlled Experiment”. In: *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the.* 2014, pp. 95–104. DOI: 10.1109/QUATIC.2014.19.

[23] List of authors: https://github.com/jacoco/jacoco/graphs/contributors. *JaCoCo - Java Code Coverage Library.* 2016. URL: https://github.com/jacoco/jacoco (visited on 10/15/2016).

[24] IBM. *IBM DB2.* 2016. URL: https://www.ibm.com/analytics/us/en/technology/db2/ (visited on 11/07/2016).

[25] GitHub Inc. *Github.* 2016. URL: https://github.com/ (visited on 11/07/2016).

[26] GitLab Inc. *GitLab Continuous Integration.* 2016. URL: https://about.gitlab.com/gitlab-ci/ (visited on 11/07/2016).

[27] reddit inc. *Reddit.* 2016. URL: www.reddit.com (visited on 11/07/2016).

[28] Laura Inozemtseva and Reid Holmes. “Coverage is Not Strongly Correlated with Test Suite Effectiveness”. In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: ACM, 2014, pp. 435–445. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568271. URL: http://doi.acm.org/10.1145/2568225.2568271.

[29] D.S. Janzen and H. Saiedian. “Does Test-Driven Development Really Improve Software Design Quality?” In: *Software, IEEE* 25.2 (2008), pp. 77–84. ISSN: 0740-7459. DOI: 10.1109/MS.2008.34.

[30] Y. Jia and M. Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[31] JUnit. *JUnit.* 2016. URL: http://junit.org/ (visited on 11/07/2016).

[32] K. Karhu et al. “Empirical Observations on Software Testing Automation”. In: *Software Testing Verification and Validation, 2009. ICST ’09. International Conference on.* 2009, pp. 201–209. DOI: 10.1109/ICST.2009.16.

[33] Tobias Kuhn. “A Survey and Classification of Controlled Natural Languages”. In: *Comput. Linguist.* 40.1 (Mar. 2014), pp. 121–170. ISSN: 0891-2017. DOI: 10.1162/COLI_a_00168. URL: http://dx.doi.org/10.1162/COLI_a_00168.

[34] Andreas Leitner et al. "Contract Driven Development = Test Driven Development - Writing Test Cases". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ESEC-FSE '07. Dubrovnik, Croatia: ACM, 2007, pp. 425–434. ISBN: 978-1-59593-811-4. DOI: `10.1145/1287624.1287685`. URL: `http://doi.acm.org/10.1145/1287624.1287685`.

[35] M. Leotta et al. "Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study". In: *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on.* 2013, pp. 108–113. DOI: `10.1109/ICSTW.2013.19`.

[36] Cucumber Limited. *Cucumber home page.* 2016. URL: `https://cucumber.io/` (visited on 09/04/2016).

[37] S. P. Masticola. "A Simple Estimate of the Cost of Software Project Failures and the Breakeven Effectiveness of Project Risk Management". In: *Economics of Software and Computation, 2007. ESC '07. First International Workshop on the.* 2007, pp. 6–6. DOI: `10.1109/ESC.2007.1`.

[38] Glenford J. Myers. *The Art of Software Testing, 2nd edition.* Wiley, 2004.

[39] Dan North. *Introducing BDD.* 2006. URL: `http://dannorth.net/introducing-bdd/` (visited on 10/15/2016).

[40] Dan North. *What's in a story?* unknown year. URL: `https://dannorth.net/whats-in-a-story/` (visited on 05/22/2016).

[41] Oracle. *Core J2EE Patterns - Data Access Object.* 2016. URL: `http://www.oracle.com/technetwork/java/dataaccessobject-138824.html` (visited on 11/07/2016).

[42] Oracle. *Oracle Database.* 2016. URL: `https://www.oracle.com/database/index.html` (visited on 11/07/2016).

[43] JonathanS. Ostroff, David Makalsky, and RichardF. Paige. "Agile Specification-Driven Development". English. In: *Extreme Programming and Agile Processes in Software Engineering.* Ed. by Jutta Eckstein and Hubert Baumeister. Vol. 3092. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 104–112. ISBN: 978-3-540-22137-1. DOI: `10.1007/978-3-540-24853-8_12`. URL: `http://dx.doi.org/10.1007/978-3-540-24853-8_12`.

[44] R. Ramler, D. Winkler, and M. Schmidt. "Random Test Case Generation and Manual Unit Testing: Substitute or Complement in Retrofitting Tests for Legacy Code?" In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on.* 2012, pp. 286–293. DOI: `10.1109/SEAA.2012.42`.

[45] P. Runeson. "A survey of unit testing practices". In: *Software, IEEE* 23.4 (2006), pp. 22–29. ISSN: 0740-7459. DOI: `10.1109/MS.2006.91`.

[46] Giuseppe Scanniello et al. "Students' and Professionals' Perceptions of Test-driven Development: A Focus Group Study". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing.* SAC '16. Pisa, Italy: ACM, 2016, pp. 1422–1427. ISBN: 978-1-4503-3739-7. DOI: `10.1145/2851613.2851778`. URL: `http://doi.acm.org/10.1145/2851613.2851778`.

[47] R. Setiadi and Man Fai Lau. "A Structured Model of Consistency Rules in After-State Database Testing". In: *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International.* 2014, pp. 650–655. DOI: `10.1109/COMPSACW.2014.109`.

[48] R. Setiadi and Man Fai Lau. "Identifying Data Inconsistencies Using After-State Database Testing (ASDT) Framework". In: *Quality Software (QSIC), 2014 14th International Conference on.* 2014, pp. 105–110. DOI: `10.1109/QSIC.2014.39`.

[49] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. "Evaluating the Cost of Software Quality". In: *Commun. ACM* 41.8 (Aug. 1998), pp. 67–73. ISSN: 0001-0782. DOI: `10.1145/280324.280335`. URL: `http://doi.acm.org/10.1145/280324.280335`.

[50] Eiffel Software. *Eiffel Software.* 2016. URL: `https://www.eiffel.com/` (visited on 11/07/2016).

[51] J.A. Solheim and J.H. Rowland. "An empirical study of testing and integration strategies using artificial software systems". In: *Software Engineering, IEEE Transactions on* 19.10 (1993), pp. 941–949. ISSN: 0098-5589. DOI: `10.1109/32.245736`.

[52] TestNG. *TestNG Eclipse plugin.* 2016. URL: `http://testng.org/doc/eclipse.html` (visited on 11/13/2016).

[53] Katherine E. Thompson, Evelyn P. Rozanski, and Anne R. Haake. “Here, There, Anywhere: Remote Usability Testing That Works”. In: *Proceedings of the 5th Conference on Information Technology Education.* CITC5 ’04. Salt Lake City, UT, USA: ACM, 2004, pp. 132–137. ISBN: 1-58113-936-5. DOI: `10.1145/1029533.1029567`. URL: `http://doi.acm.org/10.1145/1029533.1029567`.

[54] Inc. Twitter. *Twitter.* 2016. URL: `twitter.com` (visited on 11/07/2016).

[55] Bogdan Vasilescu et al. “Quality and Productivity Outcomes Relating to Continuous Integration in GitHub”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 805–816. ISBN: 978-1-4503-3675-8. DOI: `10.1145/2786805.2786850`. URL: `http://doi.acm.org/10.1145/2786805.2786850`.

[56] RAML workgroup. *RAML webpage.* 2016. URL: `http://raml.org/` (visited on 07/01/2016).

[57] N. Yahya and N.S. Awang Abu Bakar. “Test driven development contribution in universities in producing quality software: A systematic review”. In: *Information and Communication Technology for The Muslim World (ICT4M), 2014 The 5th International Conference on.* 2014, pp. 1–6. DOI: `10.1109/ICT4M.2014.7020666`.

[58] H. Zhu, P. Hall, and J. May. “Software Unit Test Coverage and Adequacy”. In: *ACM Computing Surveys* 29.4 (1997).

Appendix 1. Questionnaires

For all

- What are the tools you use when testing?
- What is the goal of testing in your opinion?
- Is there anything you dislike about testing and is there something you would improve?

For manual testers

- What are the steps you take when testing a use case, bug, feature, etc. ?
- Are you ever instructed to do exploratory testing?

For developers

- Do you test your own code? (when unit testing)
- Do you create tests for all the code you write?
- What about legacy code?

For business analysts

- Besides verifying test cases, how are you involved in testing?

For managers

- Do you use testing to measure quality and how do you do it?
- What quality metrics do you use?
- Are there linchpins in testing?