M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Requirements as test

Automatic generation of test from structured requirements

Kim Rostgaard Christensen (s084283)

Kongens Lyngby 2015

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

The world of software has over the years been tainted with stories about failed projects. Almost everyone has his or her own story about a software system, that did not solve the task, which it was designed to. This indicates either a mismatch between requirements and solution, or simply an erroneous requirement specification.

While in the recent decades, a paradigm shift towards agile methods has occurred, focusing on smaller time-boxed iterations intricately containing every development phase: Design, implementation, documentation and validation. Requirement refinement and elicitation, however, have been left dead-in-the-water in this paradigm shift and remained a *de facto* waterfall phase with neither feed-back from implementation and validation, nor feed-forward to validation.

Adding enough structure, will automatic generation of artifacts such as documentation, diagrams and even tests from requirements. The latter will provide us with the feedback loops needed. The questions are then, how much structure is needed, how much can be generated, and is it feasible?

“*Misunderstandings and neglect occasion more mischief in the world than even malice and wickedness. At all events, the two latter are of less frequent occurrence.*”

– Johann Wolfgang von Goethe, *The Sorrows of Young Werthe*, 1774

Preface

This masters thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a masters degree in Computer Science and Engineering.

Kongens Lyngby, August 11, 2015

[illegible]

Kim Rostgaard Christensen (s084283)

Acknowledgements

The people that have endured the most, during the writing of this thesis, are probably my (soon-to-be) wife and kids, whom I owe a great deal of hugs and kisses for putting up with my non-presence.

I would also like to send thanks to my supervisor Ekkart Kindler who have shown a degree of patience and ability to provide solid feedback on my confused writings, and kept me focused throughout the writing of this thesis.

Contents

CHAPTER 1

Introduction

This thesis proposes, discusses and evaluates a technique and tool for mapping requirements to tests. The technique and tool is designed to alleviate some of the workload of continuous manual acceptance testing and requirement change integration. The idea is that code generation – specifically test code generation, meant to be run automatically – will keep test code in sync with the system under test and its requirements. An idealized model of the process of the technique is shown in figure 1.1. The overall concept is: Construct requirements as use cases, map them to the implementation, generate tests from these requirements and finally execute and validate these tests.

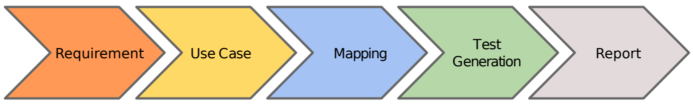


Figure 1.1: Ideal development flow.

The thesis also presents and briefly discusses the origin of the idea. It extracts generally usable concepts from that discussion. Additionally, it discusses a number of design concepts and how they would be able support the technique. After this, an analysis is performed to identify usable design models for use in an implementation of a tool, that supports the technique. Finally, a discussion and conclusion of the different concepts introduced in the project closes the thesis.

The remainder of this introduction presents the problem statement and outlines the proposed solution.

1.1 Problem statement

This section is a presentation of the problem, that the technique and tool of this thesis tries to solve.

Given the high rate of software project failures[VSC08][Cha05] and the general widespread requirement/implementation mismatch, we should treat requirements as incomplete and continuously evolving.

During the development life-cycle of a software system, additional domain knowledge is bound be acquired. This knowledge improves the general understanding of the problem domain towards a better solution. Often, it also affects the requirements by making them more elaborate, complete, correct – or even invalid. In any case, software development is an ongoing process and requirements must be, in a strict definition, expected to be an incomplete, inaccurate part of the software development domain.

Integrating requirements deeper into the development can be done by adding a reference system from requirement documentation to implementation artifacts. This way, automated system can send out reminders to developers on which documents should be reviewed, whenever a change has happened in a component. Documentation, however, is usually written in natural language, with all the befits and ambiguities that follows. The general concept is that requirements maps to implemented system to provide additional analysis and feedback.

In this thesis, one of the main goals is to add this feedback channel via generated acceptance tests. It is believed that if these tests are strongly linked to the requirements, then implementation changes that affect requirements, will change the tests, which then will verify that the system still works as intended. But to be able to actually generate requirements from tests, we need to integrate some measures into the development process.

The thesis use the behavioral aspect – namely use cases – as requirements and tool that will be a product of this thesis, should provide the mapping technique so that software developers are able to map the essential information from use cases. This must be done in respect to the model of use cases and not change it.

The next section outlines the project.

1.2 Project

Specifically, we want to convert requirements into tests. A concrete example of the abstract process in figure 1.1 is provided below.

Requirement: User must be able to log in to the system.

Use case: User logs in.

Mapping: Link use case to login subsystem.

Generation: Generate test that assert login functionality of the user against the implemented login subsystem.

Report: Run the generated test and report back if it succeeded or not.

The requirement involves the “User” actor that must be able to log in. To describe the behavior of this requirement, we expand it into a use case. The body of the use case is not provide here, in order to keep the example simple. The use case is the description of the intended behavior of the system under development, from an actor point of view. This description may then be mapped to artifacts of the system under development (as illustrated in figure 1.2). In this case, we link this use case to a login subsystem which needs to work in order for this use case to pass. Once the requirements have been provided with an implementation link (a mapping), we should be able to generate tests that tests the behavioral model from the use case, directly against the system under development. Once the test is done, its result should be reported back to the developers of the system under development.

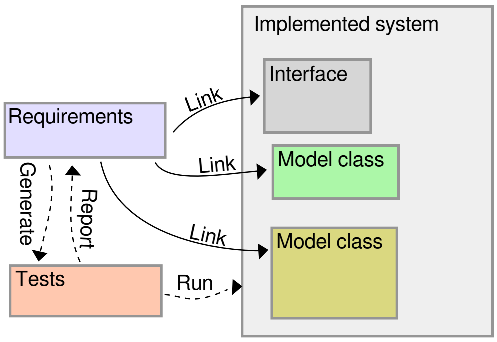


Figure 1.2: Concept of mapping from requirements to implementation.

The next section describes the concept of the implementation mapping technique.

1.2.1 Technique

The basic idea of the technique is that for every use case – or change to an existing use case – the system should generate a new set of tests, possibly replacing old tests. This process requires the participation of three different roles:

Writer: The person responsible for writing the use case. May be the customer of the system under development.

Mapper: The use case mapper will provide the mappings from the use cases to the system under development.

Testing system: The system designed for generating and running tests. It also reports back to the developers of the system.

The technique is presented in chapter 3 and refined during the course the the thesis. The next section outlines the main parts of the thesis.

1.3 Outline

The project will use the requirements from an existing system as a case study and build up a structure of these, so that test generation from them is possible. During this process, we will investigate how to structure requirements so that we can generate tests directly from them and map implementation to requirements. In doing this, we want to identify general patterns and constraints in the structure introduced to our requirements in the case study system to be able to apply them to other projects. In general, we will investigate to which extent this idea can be applied and implement a translation tool that is able to translate a (structured) representation of a use cases into a tests case.

The project is derived from the test-driven development methodology (see section 2.1.2). Lifting it from its typical application of integration testing onto the new level of acceptance testing. This technique/process is meant to be tool-assisted, so that tests may be generated automatically, but the mappings to the system needs to be done by hand.

The tool we want to build here aims in usage to be integrated in an existing development procedure, with a low learning curve that enables a broad range of programmers to gain better integration of use cases into the development process.

The development of the case study system (see section 2.2) coined the idea of test generation from structured use cases and spawned two implementations, than founded the basis of this project. The first was test generator, built as an auxiliary project, with no direct link to main project, other than the knowledge of which interfaces it exposed and the serialized data structures it transmitted. It, thus, had a no direct implementation link and every change in implementation had to propagated manually.

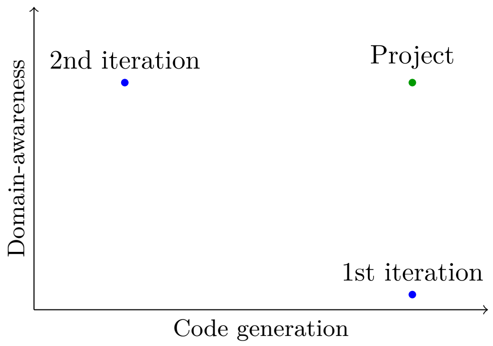


Figure 1.3: Project parameters and key points.

In the second iteration, the generated tests were later manually translated (programming language-wise) into a new set of tests that re-used the implemented code and added what we defined as "Domain-awareness". This was done by, again manually, writing up a set of support tools that tried to mimic the domain model of the case study system. So, any actor or general domain concept would have an appropriate source code class in this tool set. It imported most of the interfaces and model classes from main code base of the case study system, so any implementation change would be propagated to – or lead to compilation error in – the test support tools (section 3.2.3). Having the domain-awareness helped linking requirements to tests and – indirectly – implementation.

Having written *everything* manually in second iteration had two problems; the coverage was questionable – did we cover all branches of the use cases? And, how should we propagate requirement changes to these tests? The idea of combining the test generation with domain-awareness would allow for a mapping between requirements in implementation that enable changes in either to be propagated to the other. Figure 1.3 shows the parameters that we design after and how the different implementations a located in this space. The figure will be repeated throughout the thesis to indicate where in this space the current section operate.

The thesis will look closer into the concepts of first two iterations of the technique and summarize which them may generally needed, how the use case structuring should look like and if the technique and/or tool is feasible for broad development purposes.

1.4 Related work

This section contains a brief overview of the related works, founded during the background research phase for this thesis.

John Rushby[Rus08] proposes that tests can be generated from requirements, *if* these are already in an executable form, commonly found in model based software engineering. He identifies that there is a problem with loops in models as well. In the project from this thesis, the requirements are not executable, but merely text that is explicitly mapped to implementation.

Angelo Gargantini et al. propose model checking techniques to generate tests from requirements[GH99]. The scope of this, however, appears to be mostly on software cost reductions in the field of safety-critical software and is too formal in nature to mainstream software development.

A, not strictly related, but very interesting concept that may support this project, is the textual analysis with the purpose of building up an initial conceptual (domain) model for use in the early development stages[KFM10]. This would aid our project with a mapping suggestions, based on an analysis of what is written in the use cases.

In summary: There already exist methods and tools that support the formalization or structuring of requirements, but there seems to be a lack of motivation for applying them widely. Their scopes are either model-based software engineering, or safety-critical systems. The scope of this project is to provide an approach that is applicable in a wider array of development environments – without restrictions on software engineering paradigm, or target system market. It would, however, *not* be suitable for safety-critical systems, as it is very informal.

CHAPTER 2

Background

This chapter provides a brief introduction to software testing, test-driven development and how these relate to, and differ from this project. It also provides the historical background and motivation for the idea that eventually became the topic of this thesis. Additionally, it gives a brief overview of the design and implementation of the case study system used in this thesis, why we chose to perform the tests on the level that we did, and a brief overview on how it was done. This overview is continued in the next chapter. The purpose of going into detail with the case study system, is to build up an understanding of how it fits into this thesis and to explain the architecture of the system in order to evaluate its role in the technique of this thesis.

2.1 Software testing

In software engineering, continuous testing is one of the most effective weapons against software bugs. This section outlines some of the basic concepts handled in this thesis, and discusses the application and similarities to the technique of the thesis, of them.

2.1.1 Testing terminology

The terminology described here is provided, either because the term is used in the thesis, or for completeness – e.g. black-box/white-box testing.

Test coverage: The tests coverage is an indication of how many of the branches system are actually covered by the tests. So, a simple function with a single *if-else* branch, that had a test that covered only the *if* branch would have a coverage of 50%. If there was also a test for the *else* branch, the coverage would reach 100%. Test coverage applies to every level of testing, from acceptance tests to unit tests.

White-box test: A white-box test treats the system as a transparent box where, all the internals are known. From these known internals, tests can be written to take a specific code path within the component under test. Unit test is an example of a white-box test, as it tests the known internals of the unit. In the case study system, white-box tests are used to verify API's and system functionality – effectively testing multiple API's in a single test.

Black-box test: Treats the system as an unknown entity which responds to external stimuli. Nothing is known about the internals of the entity, and by by applying the stimuli, the entity will respond – hopefully – in the expected way. The Stimuli-organism-response (figure 2.1), which is used areas such as behavioral psychology, graphically depicts the concept very well.

Unit test: Is a detailed test of a single unit in the system. While the size of the unit is not strictly defined, a unit is typically a single file or class in software systems. A unit test is a white-box test that focus on broad test coverage.

Integration test: Is a test that treats a component, or set of components as a combined entity, and tests it as a black-box system. Integration testing is linked to functional requirements, but may also refer to non-functional requirements, such as behavior under load – which is known is stress-testing.

Test harness: An artificial environment that surrounds a test, supplying it with the resources and API's it needs. An example of a resource could be a database layer that supplies the test with the objects it needs. The harness does not need to provide real objects, but is free to return mock objects to the test, rather than having to deal with actually connecting to a database.

Acceptance test: An acceptance test is a test conducted to determine if the requirements of a specification or contract are met. If the contract specified that a ship that could float was to be built, the acceptance test would be to put it in the water and see if floats. If so; the acceptance test was a success.

System under test: The implemented system that is to be tested. Sometimes also referred to in the literature as "testee". In this thesis, we are using the case study system (section 2.2) as system under test, but may refer to system under test, in the context of general application of the approach presented in this thesis.

Stimuli
Organism
Response

Figure 2.1: Stimuli-organism-response model. Used – for instance – in psychology, but is analogous to black-box testing.

2.1.2 Test-driven development

A development methodology that emerged around the millennium, and have been looking very promising, is test-driven development. It focuses on – as the name would also indicate – tests before anything else. The basic work-flow is to first write tests, then refactor the code base of the system under development until the test passes. Then, to make sure the refactored code does not break existing functionality, all the previously written tests must be run. This work-flow is illustrated in the flow chart shown in 2.2. Some of the arguments for using test-driven development are that

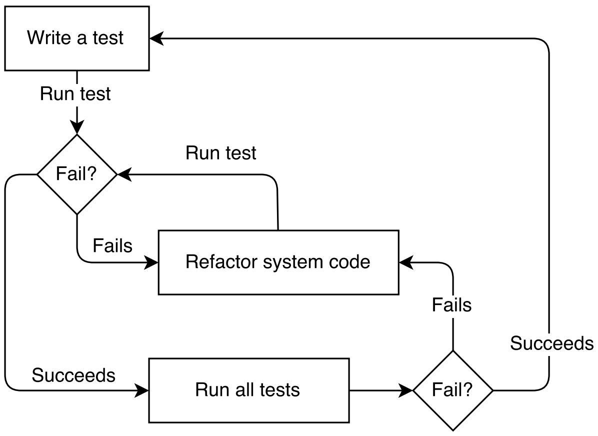


Figure 2.2: Basic work-flow of test-driven development.

it enables continuous regression testing. It focuses strongly on building the object and components that are needed rather than the ones that are thought to be needed. It also lowers the gap between component design, and developer feedback – whether the component works or not[GW03].
Test-driven development, in its nature, encourages code to be written testable. If a test is written prior to the code, then – obviously – the code written will be testable, or it will not pass the tests that originated it. This often means better code reuse and smaller, more loosely coupled components[GW03]. This is probably due to the fact that when you write tests, it is often the case that you repeat some actions over and over again.

In this thesis, the technique of generating tests is similar to the one found in test-driven development, in the way that you *can* build tests from your use cases before the implementation. But, unlike test-driven development, it does dictate a specific point in development process where you should add the tests.

Instead it focuses on a tool-assisted generation of tests scopes to achieve a very broad coverage of requirements. So basically, it gives you the outline of what to test, and which domain concepts are needed for the test.

2.1.3 Continuous integration

Continuous integration is a methodology, that supplements test-driven development quite well. In essence, it is an automated service that continuously runs tests on code, or produces new artifacts (for example; executable binaries) from it. This is done either periodically, (nightly, weekly, monthly) or triggered whenever a developer commits a change to the code base.

2.2 Case study system

This section presents the case study system that is the *raison d'être* of the technique and tool presented in this thesis. The section provided a brief overview of the problem domain and (software) architectural details that are important basis for the discussions later in the thesis. A problem domain glossary is provided in this section to introduce the terminology of the discussion.

The case study system is briefly discussed in this section to provide an overview of the functionality and implementation of it. This serves as an introduction to the "suitability" of the system design as a system under test.

The case study system started its life back in the fall of 2011. It was created to serve as a drop-in replacement for the customer's[1] current call-center system, or more specifically "reception hosting". About one year after, the first conceptual prototype emerged, and by the end of 2013, an alpha version of the product was ready to be tested and stabilized for production use. However, due to the high level of asynchronism, and low level of explicit coordination – which was by design – requirement regressions became an increasing problem. When the requirements also started to change, a dire need for automated, and usage-oriented testing became evident.

2.2.1 Problem domain glossary

In this section brief glossary from the problem domain is provided to cover the basic terminology used in the use cases.

Customer: The person in the role of purchasing the software. Assumed to have little or no knowledge about formalism, modeling or programming.

Contact: A person or a group known to the system – i.e. previously created with contact details such as phone numbers and email addresses.

Receptionist: A user in the system able to handle incoming calls by forwarding them or taking a message.

Caller: Anyone who dials a phone number handled by the system. They are not known by the system *a priori*, but the system *may* store previously entered data that serves as a cache.

[1]The Danish business Responsum (`http://responsum.dk/`)

PBX: Private Branch Exchange. A local phone switchboard with built-in logic that determines the flow and destination of a phone call based on dial-plans. Common PBX's capabilities include call queues, Interactive Voice Response (IVR) menus and transfers to either local – or external – extensions. A PBX can be either a special-purpose hardware device, or a software implementation running on regular general-purpose PC hardware. These are referred to as hard- and soft-PBX's, respectively.

Dial-plan: A decision system that decides what to with a call from a set of rules, such as "if the time of day is after 17 o' clock, send to voice mail", or "if the callee extension is +45 1234 5678", put the call straight trough to manager's extension". The concrete syntax is, of course closer to a programming language, and largely dependent on which PBX is used.

This glossary is incomplete, with regards to the domain model, but should be sufficient background for understanding the problem domain and overall role of the components in the component diagram (figure 2.5).

2.2.2 Targeted requirements

The (simplified) requirements to the case study system, that involves the "receptionist" actor point of view are:

Manage calls: Being able to technically handle calls by performing receive, park, transfer and hangup action.

Process calls: Being able to process calls in the context of a dialed reception. This involves having access to data about the reception and its contacts. Being able to dial them, or send them a message.

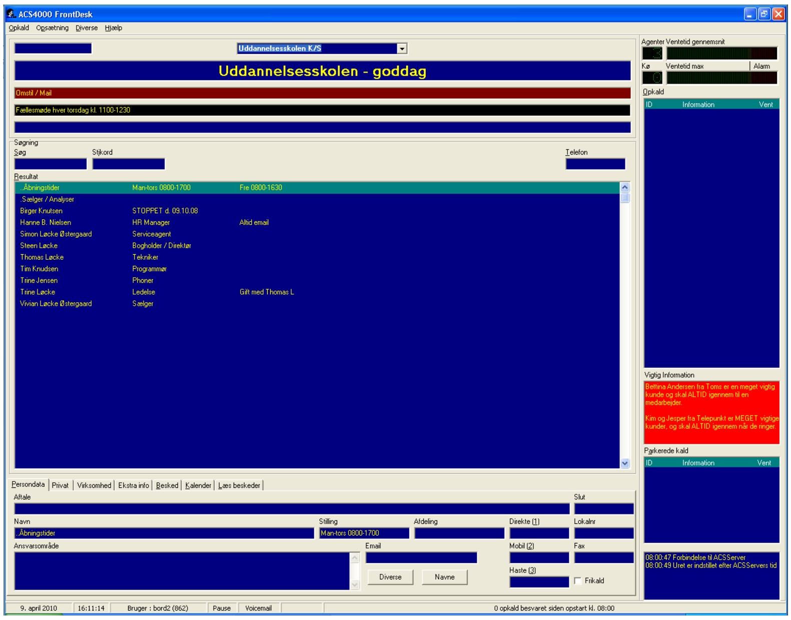
Manage message: Being able to send out messages to contacts, view and resend existing messages.

2.2.3 Business model and existing system

This section explains the business model of the case study system, who the customer is, and their motivation for having a new system developed.

The customer sells what is called a hosted reception service. Their primary customer segment are other businesses – both large and small. They employ a number of receptionists that answer phone calls on behalf of these businesses. They perform receptionist tasks as though they were physically located on the premises of those businesses, and have access to a lot of employee data, such as employee information, phone numbers and calendars. This information is continuously maintained by a number of service agents, usually by transferring data between systems manually. In other words, the customer was a specialized call-center, with needs unique enough to not be able to find a newer commercial off the shelf (COTS) system.

Throughout the last ten years, they have used a COTS system for this task, that is now showing its age. The system handles phone call routing, call handling (playing greetings and Interactive Voice Response (IVR) menus), presents reception information, manages call routing plans and reception information. The client used by the receptionists are shown in figure 2.3. It provides the receptionists with a visual call queue and the company information of the call they are currently handling.



Uddannelsesskolen K/S

Uddannelsesskolen - goddag

Omstil / Mail

Fællesmøde hver torsdag kl. 1100-1230

..Åbningstider	Man-tors 0800-1700	Fre 0800-1630
.Sælger / Analyser		
Birger Knutsen	STOPPET d. 09.10.08	
Hanne B. Nielsen	HR Manager	Altid email
Simon Løcke Østergaard	Serviceagent	
Steen Løcke	Bogholder / Direktør	
Thomas Løcke	Tekniker	
Tim Knudsen	Programmør	
Trine Jensen	Phoner	
Trine Løcke	Ledelse	Gift med Thomas L
Vivian Løcke Østergaard	Sælger	

Agenter 3
Kø 0

Bettina Andersen fra Toms er en meget vigtig kunde og skal ALTID igennem til en medarbejder.

Kim og Jesper fra Telepunkt er MEGET vigtige kunder, og skal ALTID igennem når de ringer.

Navn: ..Åbningstider
Stilling: Man-tors 0800-1700

```
08:00:47 Forbindelse til ACSServer
08:00:49 Uret er indstillet efter ACSServers tid
```

9. april 2010 | 16:11:14 | Bruger : bord2 (862) | Pause | Voicemail | 0 opkald besvaret siden opstart kl. 08:00

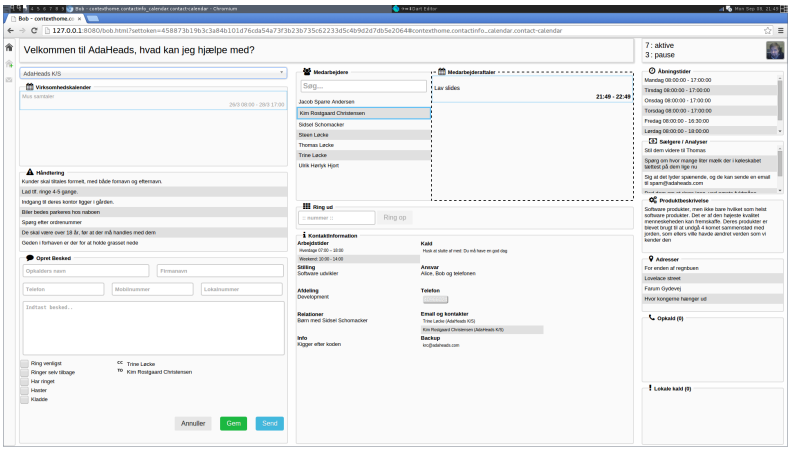
Figure 2.3: Screenshot of the receptionist frontend of the existing system.

The system is proprietary and abandoned by the developers and is still affected by open bugs and issues. Furthermore, the customers of Responsum are expressing an increasing interest in integrating their own system with the Responsum's system. An example of an integration could be calendar synchronization to lower the manual workload of their service agents.

2.2.4 Replacement system

This section gives a short introduction to the replacement system (the case study system of this thesis), its architecture and design and the development process that ultimately motivated the test approach, that is the topic of this thesis.

"OpenReception" is the brand name of the replacement system. It is a web-oriented software/telephony system. It is a system designed to enable receptionists to handle incoming calls, and provide them with the appropriate information so that they may



Bob - contexthome.contactinfo_calendar.contact-calendar - Chromium

127.0.0.1:8080/bob.html?settoken=458873b19b3c3a84b101d76cda54a73f3b23b735c62233d5c4b9d2d7db5e2064#contexthome.contactinfo_calendar.contact-calendar

Velkommen til AdaHeads, hvad kan jeg hjælpe med?

AdaHeads K/S

Virksomhedskalender

Mus samtaler 26/3 08:00 - 28/3 17:00

Håndtering

- Kunder skal tiltales formelt, med både fornavn og efternavn.
- Lad tlf. ringe 4-5 gange.
- Indgang til deres kontor ligger i gården.
- Biler bedes parkeres hos naboen
- Spørg efter ordrenummer
- De skal være over 18 år, før at der må handles med dem
- Geden i forhaven er der for at holde grasset nede

Opret Besked

Opkalders navn | Firmanavn | Telefon | Mobilnummer | Lokalnummer | Indtast besked..

- Ring venligst
- Ringer selv tilbage
- Har ringet
- Haster
- Kladde

CC Trine Løcke
TO Kim Rostgaard Christensen

Medarbejdere

Søg...

- Jacob Sparre Andersen
- Kim Rostgaard Christensen
- Sidsel Schomacker
- Steen Løcke
- Thomas Løcke
- Trine Løcke
- Ulrik Hørlyk Hjort

Medarbejderaftaler

Lav slides 21:49 - 22:49

Ring ud

:: nummer :: Ring op

KontaktInformation

Arbejdstider
Hverdage 07:00 – 18:00
Weekend: 10:00 - 14:00

Kald
Husk at slutte af med: Du må have en god dag

Stilling
Software udvikler

Ansvar
Alice, Bob og telefonen

Afdeling
Development

Telefon
40966024

Relationer
Børn med Sidsel Schomacker

Email og kontakter
Trine Løcke (AdaHeads K/S)
Kim Rostgaard Christensen (AdaHeads K/S)

Info
Kigger efter koden

Backup
krc@adaheads.com

7 : aktive
3 : pause

Åbningstider

- Mandag 08:00:00 - 17:00:00
- Tirsdag 08:00:00 - 17:00:00
- Onsdag 08:00:00 - 17:00:00
- Torsdag 08:00:00 - 17:00:00
- Fredag 08:00:00 - 16:30:00
- Lørdag 08:00:00 - 18:00:00

Sælgere / Analyser

- Stil dem videre til Thomas
- Spørg om hvor mange liter mælk der i køleskabet tættest på dem lige nu
- Sig at det lyder spænende, og de kan sende en email til spam@adaheads.com

Produktbeskrivelse

Software produkter, men ikke bare hvilket som helst software produkter. Det er af den højeste kvalitet menneskeheden kan fremskaffe. Deres produkter er blevet brugt til at undgå 4 komet sammenstød med jorden, som ellers ville havde ændret verden som vi kender den

Adresser

- For enden af regnbuen
- Lovelace street
- Farum Gydevej
- Hvor kongerne hænger ud

Opkald (0)

Lokale kald (0)

Figure 2.4: Screenshot of the receptionist frontend of the replacement system.

divert or directly handle the calls. The system is designed with high availability in mind with many – largely independent – components that are loosely coupled. This limits the Domino-effect, where one faulty component can take down another for no other reason than the fact that they are partitioned together.

This component-oriented design has also helped the testing process, as it enabled individual components to be tested and verified independently of the others. A screenshot of the receptionist client of the replacement system can be seen in figure 2.4 and provides the same information as the original one.

2.2.5 Chosen architecture

This section discusses the chosen architecture of the case study system. A component diagram is provided, but only the essential components will be covered in this section

As the existing system was considered critical infrastructure, the replacement system was designed with simplicity and high fault resilience in mind. This means that we tried to provide fall-back mechanisms for most of the system rather than over-eagerly handle every potential fault. The component diagram in figure 2.5 shows the architecture with the critical path highlighted by a red line. The critical path is the one that originates from the “ReceptionistClient” and ends in the “SIP trunk” component. These components are considered soft real-time components and are essential for an operation. Fallout of any of these components means direct financial loss for the customer. Fallout of any of the other components (except for SIP Phone, which is actually part of the critical path mentioned above) are tolerated in the design, and the stateless nature of REST (see section 2.2.6) enables us to maintain caches that can supply clients with the data they need for handling calls.

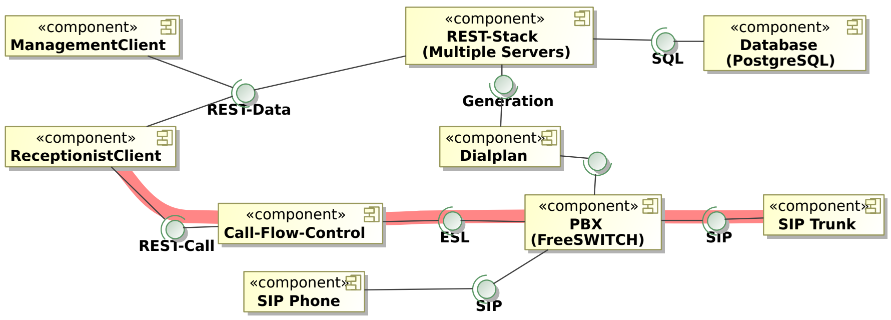

«component» ManagementClient
«component» REST-Stack (Multiple Servers)
SQL
«component» Database (PostgreSQL)
REST-Data
Generation
«component» ReceptionistClient
«component» Dialplan
«component» Call-Flow-Control
ESL
«component» PBX (FreeSWITCH)
SIP
«component» SIP Trunk
REST-Call
«component» SIP Phone
SIP

Figure 2.5: Component diagram.

Being able to test the entire critical path as a whole is one of the motivations for implementing the test technique discussed in the section 2.2.8. The important interfaces, in the context of this thesis is the "REST-Data" and "REST-Call", which are stateless (covered in the next section) REST interfaces that provide the "ReceptionistClient" with domain data and call-management actions – respectively.

2.2.6 Stateless architecture

The case study system is, like many others contemporary, distributed applications, using REST[2]. It is a reasonably new, and non-standardized (by any committee) technology for building Web-connected API's. It is a client/server protocol that bases itself upon some very simple principles and enables high scalability by its stateless design. The stateless design enables API providers to partition and cache their resources better, as they do not need to synchronize state across data partitions.

System testing is also simplified quite a bit by this technology, when you don't need to take into account a protocol state.

2.2.7 Loose coupling of components

In order to minimize the damage of a component failing, we have divided the REST API into smaller services, each responsible for only handling one single task. This is what is, informally, defined as the "bulkhead pattern". Originating from naval vessel floating compartments, which were composed of several individual bulkheads. These bulkheads could be closed off, in case of a leakage in one of them. A parallel to software engineering is to divide your application into separate operating system processes that can be terminated if they start to leak memory, consume excessive amounts of CPU time, or merely lock up. Sometimes, killing worker processes is used as a preemptive method of assuring that processes are kept under control. The

[2](**RE**presentational **S**tate **T**ransfer)

Apache HTTPD web server uses this strategy by maximizing the number of requests[3] a worker process may serve before being terminated and replaced.

In our architecture, we have divided the database operations, dialplan generation, CDR[4] into dedicated servers, that may be replicated and replaced – even in a live production environment.

2.2.8 Testing strategy

As development progressed, it became increasingly difficult and time consuming to verify the correctness of the implementation. This was caused by the workload og manually performing the acceptance tests. These basically involved: Performing an incoming call via a phone. Picking up the call via the system. Accept the call on another phone and *then* perform the actual use case scenario. This could, for example, be to forward the call and signal an idle state to system. Other issues with this manual approach was that it was easy to perform in an incorrect order (or make other errors), thus leading to false negatives in test runs. Clearly the project could benefit from investing time in scripting the setup and tearing down of the state which was need to perform the manual tests. During the progression of the project, it became more and more clear to us that we were constrained by two parameters, in regards to testing:

- We needed to verify the the functionality of system as a whole, and argue that it was free of instabilities. This functionality was defined in the use cases.
- We had very limited man-hours available and, thus, had to prioritize very aggressively on what level to test on.

As system consisted of a number of loosely coupled components, and some of them were not under our control, we decided to focus on writing up black-box tests focusing on verifying the behavior of the individual components, or multiple connected components. Furthermore, we wanted to automate this using a continuous integration service that could run our tests for us, and provide us with reports on regressions, or identified bugs. But first, a proper level of testing had to be chosen.

2.2.9 Level of testing

The tests were built from the perspective of how the system was meant to work. We basically treated the system as a block-box system and abstracted away as many implementation details as possible. For this purpose, we and built a "robot-receptionist" and a "robot-caller". These "robots" acted as clients and connected to the **REST-Data**, **REST-Call**, **REST-SIP** interfaces from figure 2.5.

[3]http://httpd.apache.org/docs/2.2/mod/worker.html
[4]Call Detail Records: records of call duration and other information used for invoicing

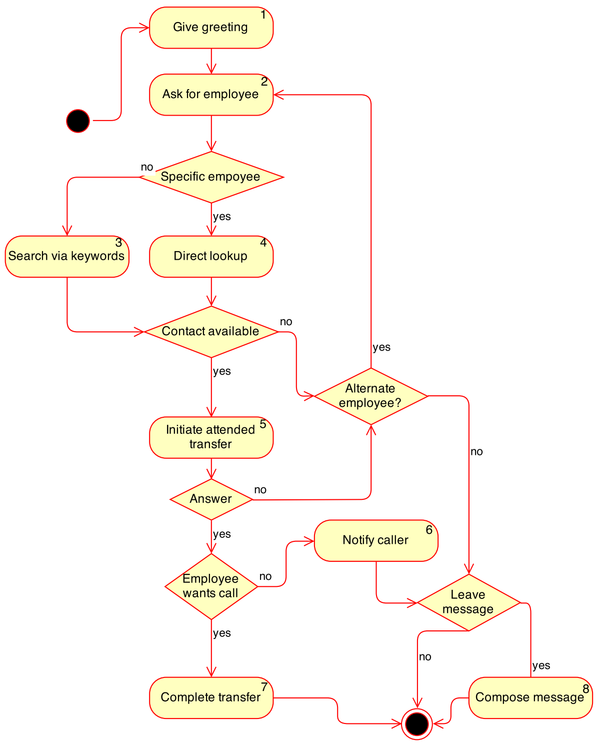


Figure 2.6: Labeled activity diagram of the basic workflow of a receptionist.

2.2.10 Coverage of testing

In order to test the system, using the black-box method, we needed to formulate the tests in a way that provided the broadest coverage possible. To achieve this, we crafted a set of activity diagrams from the use cases of the system (see figure 2.6). From this, we could assert which paths of the activity diagram were covered.

2.2.11 Generation of tests

To make sure that all paths were covered, every activity node got labeled. Any unique path through the activity diagram should then correspond to a use case, but more importantly; be realized by a test. To achieve this, we constructed a set of tools that, among other things contained the “robots” discussed briefly in section 2.2.9.

By letting a continuous integration service automatically re-deploy the software stack and, run our acceptance tests (along with a number of integration tests) every time there was any change to the code base, we both strengthened the confidence of the correctness (according to specification) of the software stack, and verify that no regressions arise. The continuous integration server can be found at `http://ci.bitstack.dk/`

2.3 Summary

The high level of modularization, and low coupling of components made the system easily testable. Most components could be be tested individually, and some of them emulated. The modularization effectively also led to many well-defined interfaces, which is the basic premise for black-box testing. The project, in itself, is quite suitable for a case study system, from a testing perspective.

The use cases of the system are, however, not very complex and may not be able to expose scalability issues in the tool. The already developed "robot" programs are, on the other hand, very suitable for building controllable "robot-actors" that may be used for supporting the functionality of the generated tests.

Prior to the conceival of the idea for this thesis, the test concept described above was implemented and part of the case study system development. During the writing of this thesis, a second iteration was written, in order to aid the tool for this thesis. These iterations are described in the next chapter, which covers the first two iterations of the test generation tools.

CHAPTER 3

First iterations

This chapter provides anecdotal background that supports use case tool concept discussed in chapter 4. It describes the initial and second iteration of the test generation tool in order to define which parameters the third iteration (the project of this thesis) should – and shouldn't – be optimized by. It covers the basic concept of implementation mapping and presents two different views on the level of integration of these. The chapter provide an implementation-driven discussion on the test mappings, that is reused later in the thesis.

The first iteration was developed using a simple form of code generation. It was a simple experiment that proved itself useful enough to refine. The second iteration was linked directly to the shared model and interface classes of the system that were exposed in a common framework (section 3.2.1). The second iteration created a common ground for the requirements translation project, as many of the actors and concepts that we need to map to are provided to us by it (section 3.2.3).

This chapter also covers the motivation and scoping for the tests and test tools. Later on, it explains how it was implemented and refined. The first iteration of the project focused primarily on enabling the generation of test.

The second iteration was done as part of this thesis, to create the supporting test tools needed for test generation.

3.1 Original project

This section is provided as reference, and documents our original motivation and scope for the first version of our use-case tests. The first iteration focused on generating use case tests from requirements – without mapping it to the domain model. The first iteration is highlighted in figure 3.1 that emphasize this relation.

3.1.1 Motivation and scope

During the stabilizing of the case study system system, the high level of asynchronism of the system, and low level of control with the PBX that made feature regressions a daily pain. Unit tests were of little, or no use, as we were basically writing and testing components, without knowing if they belonged in the system, or not. For this problem; the solution of applying a test-driven development methods seemed very

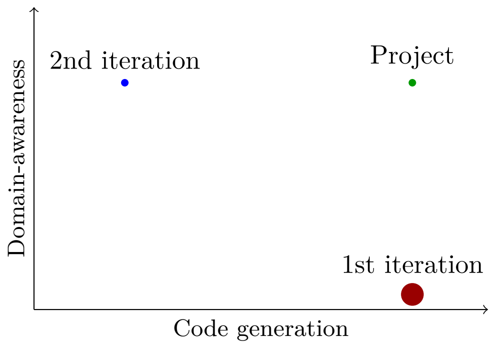


Figure 3.1: First iteration focused mainly on generation. The dark red dot highlights the 1st iteration.

suitable, but we wanted to raise the abstraction level a bit further, and start directly from requirements and test the system as a whole.

Testing a complete system can basically be done in two ways; either by providing a complete test environment (sometimes called “test harness”) that emulates the behavior of the interfaces that the component under test requires, or by testing a deployed instance of a system. In our case, the option of emulating the interface of an entire PBX wasn’t really a solution for our problem. Basically, if we knew the exact behavior of the PBX and IP-telephony domain, then we would just build the logic to handle it. The best solution we could think of, was to include the PBX as a required – and included – component for the component tests.

We wanted to add requirement-level tests, and decided on writing up use cases, instead of the requirements document that existed prior. This document contained a very verbose description on how the case study system was supposed to behave, along with screen-shots of the old system.

3.1.2 Test specification

From the requirement document mentioned earlier, activity diagrams – such as the on in figure 2.6 were created. These were very useful communication tool for deriving use cases. When manually walking through the paths with a representative from the customer of the system, the use cases could be derived from their description of what would happen at a given step – and what shouldn’t. The motivation for using the activity diagram, was that is was very simple for the customer to relate to – without being distracted by the more specific level of use cases. I.e. not much “what happens if”, that could then be covered in-depth in a dedicated discussion in each activity.

To be able to generate test from this, we decided that every statement, or line, in the use case could be mapped to a block of manually written source code block. So, basically, if we had an action that was “Receptionist dials extension of contact”, we would have a corresponding re-usable function that performed this action. We named this “the use-case oriented scripting environment”, and used it to write up tests of the system that originated directly from the requirements.

3.1.3 Mappings

Every use case statement was analyzed, and concepts and actors were extracted from them, effectively giving us the minimal requirements for which (programming) objects needed to participate in the test.

In the scripting environment, we then outlined the domain actors (such as caller and receptionists). When external resources were needed, they were provided by the main system (under test) via service interfaces. The final thing to outline and add to classes were the non-actor domain concepts, such as calls and messages.

The next step was then to assign every action, precondition and postcondition line a unique identification and map the identification to individual code block. We also wanted to generate documentation along with the tests, so the mapping from on use case step was done to three separate chunks, each serving its own purpose.

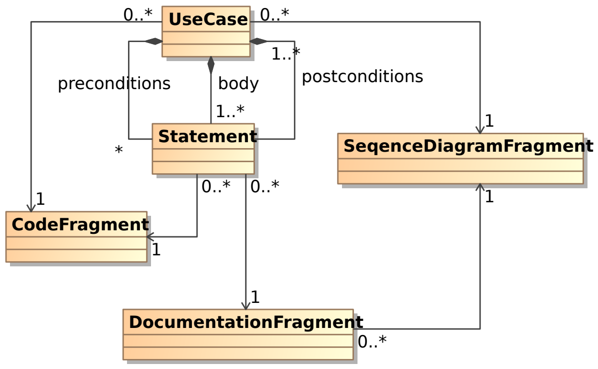


Figure 3.2: Class diagram outlining the composition of the initial test-generation model.

Visualization: A graphical presentation of the use case in the form of a sequence diagram. In our specific case, we used a tool called seqdiag, and an example of the input is shown is shown in listing 3.1. The main concept of this was to provide a convenient overview of how, and when, the interaction between the actors happened, and the full realization within the system. The fragment of the visualization is represented by the `SequenceDiagramFragment` in figure 3.2.

Testing The final, and most significant part of the system, was the ability to generate use case tests from the actual use cases. This meant that we needed to describe every action in the use case as block of code (`CodeFragment` in figure 3.2) and then patch it together using the list of actions, pre- and postconditions provided in the use case descriptions. An example of an assembled test is shown in listing 3.2.

Documentation: In addition to tests and diagrams, documentation was also generated from the use cases. These were, once generated, uploaded to a Wiki. This meant that we could provide a generated web link to the documentation documents, in the generated tests. The fragment of the documentation is represented by the `DocumentationFragment` in figure 3.2.

A class diagram showing the relations between the different components in our initial use case translator is shown in figure 3.2. It depicts the association between the different types of code fragments, and how they are related to a use case model.

```

Receptionist ->> System [label = "Dials contact"];
System       ->> Receptionist [label = "Call rejected"];
```

Listing 3.1: Example seqdiag input fragment.

```
# \${WIKI_URL}
from forward_call import Test_Case
class Sequence_Diagram (Test_Case):
  def test_Run (self):
    self.Preconditions ()
    self.Receptionist_Dial_Contact()
    self.Call_Is_Denied()
    self.Postconditions()
```

Listing 3.2: Example generated Python code output.

The observant reader will probably already have noticed that this code cannot stand by itself. For instance, the `self.Receptionist_Dial_Contact()` statement references a method found within its own class[1]. In order to keep the code block complexity low, we decided to abstract a lot of the complexity into macro functions that were provided to the use case through a **test support framework**. Each use case was given its own programming class that then provided these macro functions via class members. Each class would then need to be self-contained and also provide setup/teardown functions, and setup pre- and postconditions. The setup and teardown functions were defined as technical prerequisites, whereas the pre- and postcondition functions were for setting up the use case prerequisites and would, thus, map to the use case pre- and postconditions. Programing-wise, this was done by adding the

needed macro methods required by the use cases to an abstract superclass that represented the overall use case. Each use case path – or use case variant – would then be

[1]For those unfamiliar with Python, `self` is a reference to the current object instance, similar to the `this` keyword in Java.

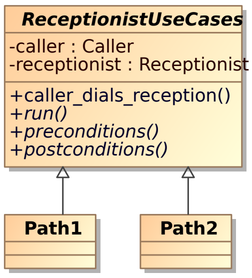


Figure 3.3: Class diagram outlining the overall hierarchy of executable use cases.

an extension of this superclass. A simplified class diagram illustrating this is shown in figure 3.3, where you can see that every variant (path of an extension) of a use case may reuse methods from its superclass. Each use case variant must also supply the abstract methods `run`, `Precondition` and `Postcondition` to fulfill the interface.

Missing methods, or problems in generated code were identified by static analysis using the pylint tool[2].

3.1.4 Discussion

The generation of tests from use case descriptions proved itself to be a useful concept, but was very time-consuming in its current form. The main disadvantage was that it became a very decoupled "project within the project". Every use case now lived in a separate form with three different files that needed to be kept in sync every time a change occurred. The sequence diagrams provided no value for the customer, and limited value for the developers working on the system.

The idea of giving each use case step an identity, for generating their test function signatures, lived on in the next iterations.

3.2 Second iteration

We soon realized that doing individual mapping for every line was tedious, time consuming and error-prone. Especially due to the fact that a lot of implicit knowledge was needed to perform the manual mapping. We thought that by linking the requirements to implemented classed into the test framework we could inject additional domain knowledge into the tests. Hopefully it would reduce the work-load associated with mapping the tests manually, as in the first iteration. Specifically, by moving our macro methods to the appropriate class – in an object-oriented fashion – would enable reuse. For example when a sentence states that a receptionist actor hangs

[2]http://www.pylint.org/

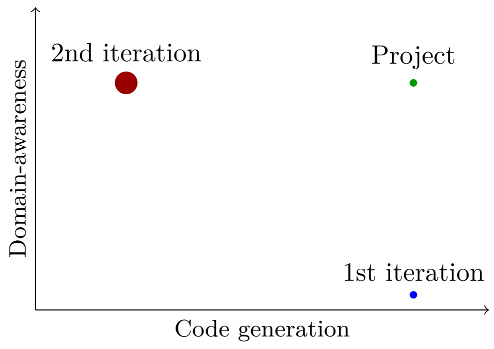


Figure 3.4: Second iteration involved manually writing up the use case tests while linking it to the domain model, but didn't include any generation.

up a call, we add a "hangup" method to the "Receptionist" class that takes a "Call" object as an argument. The second iteration is highlighted in figure 3.4. It is built during the course of this thesis, and is thus considered an auxiliary part of it.

3.2.1 Sharing domain model knowledge

During the evolution of the software, platform, language and architecture changes eventually landed us in a space where we had the opportunity to use the same programming language both on the sever and the client. initially, we wanted to exploit this by sharing the model classes between the server and client, but it soon evolved into a larger framework, also covering interfaces and REST resource definitions. We could now, by re-using the domain classes and interfaces from the framework, build tests with a higher domain awareness than in the previous iteration.

This section explains the processes of using the models and interface of the framework for a third application: Namely, testing. By exposing these interfaces and model classes to the tests, we had our link directly to the implementation.

3.2.2 Domain framework content

The domain framework is a set of library-level classes that is divided, into the following categories (and code packages). It takes care of tedious low-level tasks such as serialization/deserialization object building from databases and URI encoding. It is divided into the following different parts:

Models: Every model of the framework is either a part of the domain model, or closely related to a concept in it. Each class that is meant to be distributed is provided with serialization and de-serialization methods for use in the service layer.

Storage: Abstract interface descriptions. An example is an interface that dictates which primitives a storage layer for messages must have. In our case, this is CRUD operations (see appendix A), and and "enqueue" that puts a message into a delivery queue.

Client: These classes provide client classes (services from the client point of view), that exposes methods for remote procedure calls. The clients use the framework model de-serialization methods to automatically build typed objects on the client side. This eliminates the hassle of having to deal with low-level encoding and decoding. It enables the clients to work directly on the domain objects via the supplied service interfaces.'

Database: Database access layer. Contains all SQL query functions. The functions cast from database queries to domain objects that may be serialized for transport to remote clients via – for instance – HTTP.

Resource: Encoding library that turns objects into a REST-resource identifiers. Used to abstract away the URI encodings on both the client and the server.

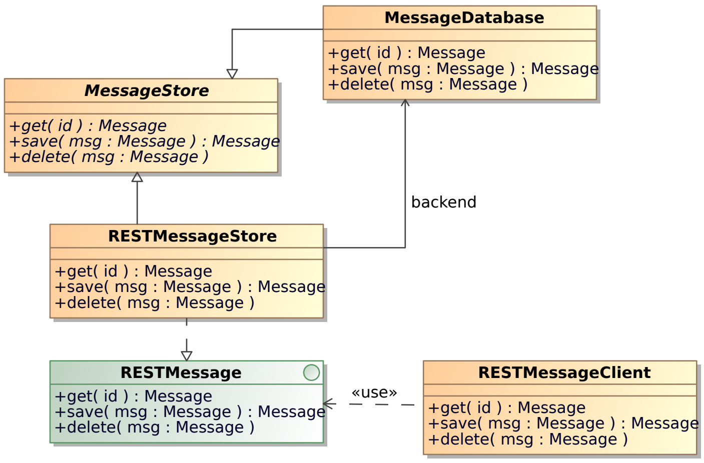


Figure 3.5: Example of the framework interface class structure.

The framework served an important purpose in writing the code that modeled the actor behavior needed in the acceptance tests that we wrote. An example of a class configuration the framework is shown in figure 3.5. The framework exposes the "RESTMessage" interface that is used by the "RESTMessageClient" class. The latter class is used by both the client user interfaces, the tests that we write, and the servers. The server-side database layer realize the "MessageStore" interface to enable easy delegation of client requests to the database, and enables integration tests to be re-used – but that is a different story entirely, and out of scope of this thesis.

3.2.3 Test support tools

In order to perform fully automated end-to-end testing of the system, the framework was not quite enough. A box test of a system – in general – requires external stimuli in order to produce a result.

So, to perform tests from a use case perspective, we needed to add more domain-awareness to our test environment. We did this writing up classes for use case actors that corresponded to a caller, callee and receptionist actor and aggregated the needed client interfaces and concepts. Most of this was provided by the framework (see section 3.2.1), such as the user and reception information. But, we also wanted to perform actual phone calls whenever a use case stated that this was what happened. We developed a small software phone, that could be integrated into, and controlled by, our test tools. On top of that, an abstraction library was created – PhonIO[3]. This library provided just the basic functionality (dial, hangup, ...) of a phone, exposing a general interface. This allowed us to scale up tests a lot easier, as we could just add additional soft-phones. It also allowed us to use the tests tools to both assert the functionality of a physical deployment (with hardware phones), and the functionality of an automated test-deployment (using soft-phones). The test system also supplies causality-checkers, such as "wait-for" methods that expects a specific event to occur within a bounded time-span.

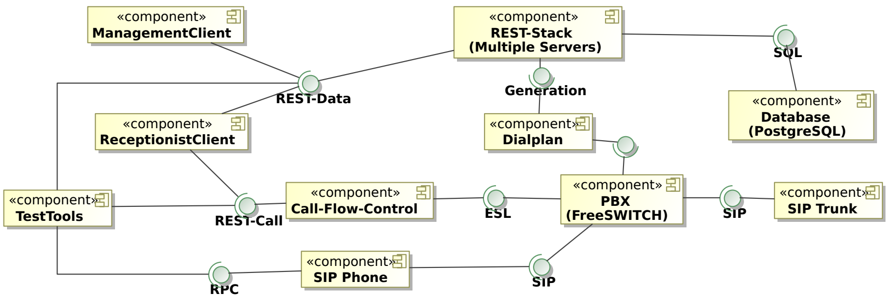

«component» ManagementClient
«component» REST-Stack (Multiple Servers)
SQL
REST-Data
Generation
«component» Database (PostgreSQL)
«component» ReceptionistClient
«component» Dialplan
«component» TestTools
REST-Call
«component» Call-Flow-Control
ESL
«component» PBX (FreeSWITCH)
SIP
«component» SIP Trunk
RPC
«component» SIP Phone
SIP

Figure 3.6: Component diagram – extended with tests.

The extended component diagram in figure 3.6 depicts how the supporting test tools fit into the overall architecture of the system. It re-uses the REST interfaces (REST-Data and REST-Call) (exposed in the shared framework) and controls phones via the library mentioned before. By reusing the model and interface classes from framework, test code base size was reduced significantly.

[3]https://github.com/Bitstackers/Phonio

3.2.4 Resource pools

A crucial component that was added to the tests tools is the resource pool. This component has also been found to be important in generation, but this is discussed later on.

Tests require resources as part of their test harness. Resources that are available to them prior to the test body, and they need to get these from somewhere. For this purpose we defined the abstract component “resource pool”. For those familiar with manual memory management from – for example – the C programming language, will recognize the pattern of explicitly allocating and deallocating resources from the `malloc/free` system calls. This system is analogous, but does not manage memory blocks – it manages resources.

Resources are quite a broad term, but specifically in our test tools, resources are pre-configured actors; domain objects – such as user and reception data. Before a test can run, it must specify which resources it requires to be able to complete. For instance; a test involving a caller, dialing a reception – handled by a receptionist, needs a caller and reception actor, and reception data (stored within a reception object). These will be allocated from three separate resource pools that must be available before starting the tests.

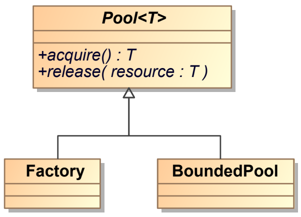


Figure 3.7: Abstract resource pools with specializations.

There are three different specializations of resource pools in our current implementation of the test tools. These cover a broad variety of scenarios, and should be sufficient for most implementations.

Bounded pool: A Bounded pool hold resources that are limited in the system. For example, our system holds a limited number of configured receptionist actors, and when that last one is allocated, there is no way of acquiring more. The only way to get around it is to manually lower the required number of actors required for a test or add more receptionist actors.

Factory: Any resource that can be acquired an unrestricted amount of times that may – for instance – be realized by a mock object generator. It is either a resource that may be shared (allocated multiple times), or a resource that easy to replicate.

These concepts are reused in the next iteration of the tests, which is the refined tests-generation tool.

3.2.5 Actor classes

Another thing that was added to the test support tools in the second iteration, was the actor classes. These classes contained the high-level functions that an actor was able to perform. These actions were either a direct link – or very close related to – the actions performed by the actor in the use cases.

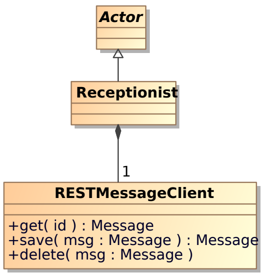


Figure 3.8: Example composition of a receptionist actor class.

Actor object would also know about which service interfaces they would require. In figure 3.8 we see a Receptionist actor that needs to store messages. This leads to a dependency on the RESTMessageStore via a RESTMessageClient instance. The receptionist is now able to provide method calls that include "Message" objects belonging to a RESTMessageStore by delegating the calls to that.

3.3 Summary

In this section we have identified a number of concepts that have support the implementation mapping.

Actor classes: Wrapper classes that represent domain actors. Provides the basic macro-functions that realize actions the actor performs in a use case step.

Concept classes: The classes that represent non-actor domain concepts. For the most part, these classes were already located in the framework supporting the case study system.

Resource pools: A convenient abstraction of filling the need for various actors or concepts in tests.

Domain framework: The supporting models and interfaces. Interfaces are exposed for a server to realize, and a client – which includes tests – to use.

Test support tools: The supporting library that contains the actor and concept wrapper classes, resource pools, and utility functions that the tests may require.

The first iteration showed us that the direct relationship to the code base matters. The separate code base that included the use cases quickly grew out of control, even though the code was generated. The sheer amount of work involved in creating the client interfaces and model classes from the case study system *twice* was too much.

In the second iteration, the tools that used the domain framework provided excellent support for the tests, and being that it shared the code base with case study system, most of the models and interfaces were now provided for us, and automatically tracked the changes to the code base.

The addition of the test support tools, enabled an abstraction away from the concrete implementation, as it provided the interface for testing the system from the point of view of an actor.

Both of the frameworks from this chapter have been identified as essential concepts for test generation, from a use case perspective.

CHAPTER 4

Conceptual design

This chapter contains a discussion of possible designs for use case structuring methods – with user interface mock-ups. The discussion is started by defining an appropriate view on use case and then presenting three conceptual solutions along with a discussion on their applicability in the problem domain. It is assumed that a domain framework (such as the one discussed in 3.2.1) is present, and the test support tools are written alongside – or prior.

The goal is to identify a suitable concept for a tool that enables us to inject better domain-awareness in our tests. It should also be able to support tests generation from them. The goal is to identify a concept that supports loose structuring of use cases. It should still provide enough information to be able to map to the system under test for test generation.

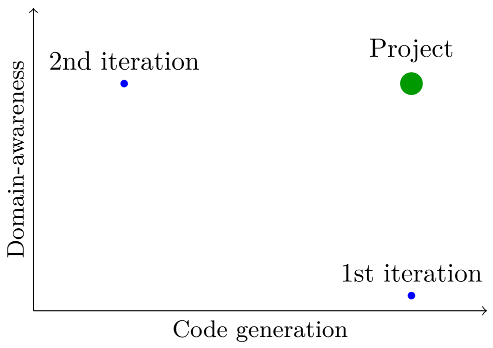


Figure 4.1: Finding a good combination of domain-awareness and generation.

By discussing and iterating through a number of different conceptual designs, levering their benefits and disadvantages, a suitable working model will be refined. Figure 4.1 shows the ideal placement of the tool, as something that combines both concepts from the previous iterations.

4.1 Interpreting use cases

This section discusses and chooses a use case representation model that is suitable for structuring.

In order to be able to structure use cases, we need to define to it what a use case is. But, since use cases do not have a widely adopted standardized format, we need to pick a suitable representation.

While we did find an existing formalized use case model[KS10], it focused on model checking and semantics. While this is indeed an important field of study, our scope is a bit different. As the tool we want to build here is meant as a support for existing development procedure – rather than replacing it, and introducing formalism – we constrain ourselves from constructing it from a formal model.

We focus on the structure provided by the template, and build our way up from there. But, before we try to build up a model for use case that support test translation, we need to establish a common ground of what a use case is, what it should include, and what it shouldn't include. From [Coc00], we get:

> "... The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfy the stakeholders' interests."

which basically means that we should write our use cases, solely focusing on behavior and intent of the involved stakeholders. Any non-essential information should be left out. Another point that is stated[Lar05] is that use cases should not be used to describe user interface actions, so for example an action "user presses submit button" is not suitable writing level for a use case.

So, in essence, use cases express *expected system behavior from a stakeholder's point of view.* For this thesis we use the "fully dressed" use case template[Lar05], as it already provides very good structure to build upon. It specifies the need to include a stakeholder list, a primary actor, main scenario, pre- and postconditions and a list of extension that are linked to main scenario. An overview of the terms used for the remainder of the thesis is shown below.

Use case: Contains – as a minimum – a primary actor, which signifies to aspect of the use case, and a scenario. The use case represents a high-level goal that the primary actor wants to achieve.

Scenario: The scenario is the list of steps that the primary actor needs to go through in order achieve the use case goal.

Steps: The steps of a use case. A step may be decomposed into smaller components or just be a text, based on which level of detail we want.

Extensions: Extensions are alternate scenarios that branch out of the original scenario. They are not meant as use case extensions, but alternate paths that a use case scenario may take.

Preconditions: A list of predicates that needs to be true for the first step in the use case to be executed.

Postconditions: A list of predicates that needs to be true in order to make use case goal succeed, after the steps in the scenario (possibly including the extensions), have been performed.

Actors: May be considered primary or supporting. Is typically a human performing a role in the system, but may also be another system that provides provides a service or information. An actor has a set of goals, which he/she wants to realize through the system. A good goal has a verb/noun combination, as a rule of thumb. In the context of a use case, an actor may act in a given role.

Concepts: Concepts are general terms that are usually part of the domain model for the system under development. Basically, most terms that can be interacted with in the system under development count as concept. In our example system, a “call” is considered a concept.

This use case representation is used in the next section, where three use case editor concepts are presented.

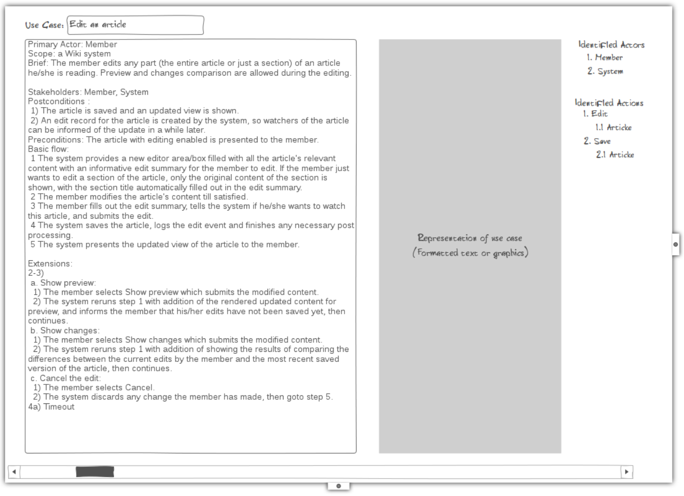
4.2 Concept 1 - Markdown editor

The first concept for a tool that aided use case structuring, was an editor that used a markup language that enabled the use case writer to tag specific words as keywords using a special syntax – such as surrounding text with parentheses, or other significant characters. The concept was dropped in an early state, but is documented for completeness, and to reference some of the ideas introduced by it later on. The

benefit that was hoped to be gained gain from this procedure was that both the use case, and the extra data needed to generate tests, was stored in the same textual representation.

This concept was only implemented as an experimental prototype model without user interface, and proved too loose in structure to be able to implement within the time frame of this Thesis. The main problem was that it effectively required a domain-specific language to be able to support the structure needed for test generation. The creation of a domain-specific language was considered out of scope, and a too large task that would dominate the project, taking focus from the actual task.

Figure 4.2 roughly shows how the user interface was planned to look like. Left part of the UI shows a structured textual representation of a use case. The middle part was reserved for graphical representations of the use case, which could be sequence diagrams or activity diagrams, and the right panel is for “use case analysis”, which is associations that will become evident during a use case analysis. The tool was meant to be backed by a domain model, which were to link the tagged keywords to domain concepts, or domain actors and the actions they performed.



Edit an article

Primary Actor: Member
Scope: a Wiki system
Brief: The member edits any part (the entire article or just a section) of an article he/she is reading. Preview and changes comparison are allowed during the editing.

Stakeholders: Member, System
Postconditions :
1) The article is saved and an updated view is shown.
2) An edit record for the article is created by the system, so watchers of the article can be informed of the update in a while later.
Preconditions: The article with editing enabled is presented to the member.
Basic flow:
1 The system provides a new editor area/box filled with all the article's relevant content with an informative edit summary for the member to edit. If the member just wants to edit a section of the article, only the original content of the section is shown, with the section title automatically filled out in the edit summary.
2 The member modifies the article's content till satisfied.
3 The member fills out the edit summary, tells the system if he/she wants to watch this article, and submits the edit.
4 The system saves the article, logs the edit event and finishes any necessary post processing.
5 The system presents the updated view of the article to the member.

Extensions:
2-3)
a. Show preview:
1) The member selects Show preview which submits the modified content.
2) The system reruns step 1 with addition of the rendered updated content for preview, and informs the member that his/her edits have not been saved yet, then continues.
b. Show changes:
1) The member selects Show changes which submits the modified content.
2) The system reruns step 1 with addition of showing the results of comparing the differences between the current edits by the member and the most recent saved version of the article, then continues.
c. Cancel the edit:
1) The member selects Cancel.
2) The system discards any change the member has made, then goto step 5.
4a) Timeout

Figure 4.2: Crude mock-up of a user interface using a markup language for writing use cases.

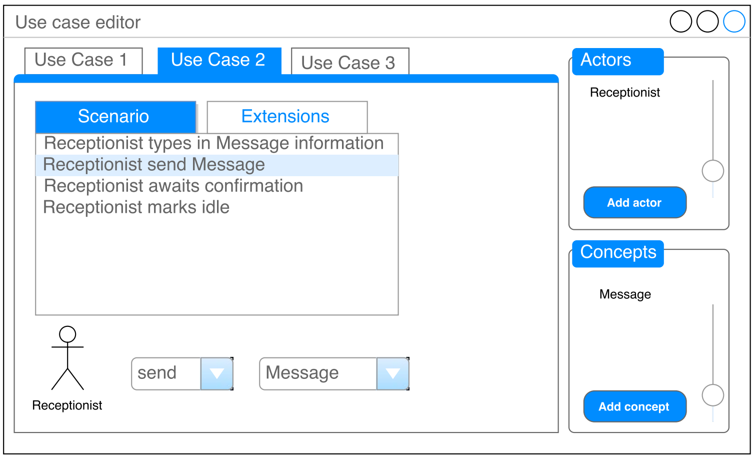
For example, "user sends email" implicitly states an association between the "user" actor and "email' concept, and the verb "sends" is the active action that the actor performs.

The tool was meant as an online Wiki-like tool that enabled collaboration. The concept of markup language was dropped, as it quickly obfuscated the textual representation of the use cases with markup syntax. To try a different path, a more strict component/structure editor concept was studied, and the idea of clear-text analysis was not revisited again before the third concept.

4.3 Concept 2 - Component/structure editor

This section presents the second requirement-to-test translation concept, proposes a rudimentary meta model and evaluates the approach in terms which parts should be refined in a later concept, and which shouldn't. The second concept originated from the idea that structure could be added by a component-oriented tool. The user interface only provides graphical components that represent domain actors and concepts than can be connected and re-arranged to create the use cases. The user interface should then – similar to the first concept – provide immediate visual feedback in the form of textual use case representation (or a diagram), like in the first concept. The concept was implemented as an initial prototype and the evaluation of it is partly based on this.

The procedure of the concept is to define actors and concepts beforehand, and then create use cases from combining these. So, if starting from scratch, a user would be expected to initially define at least one actor and the actions that the actor perform and the objects that this action would be performed on. This should be done prior to actually writing the use case.



Receptionist types in Message information
Receptionist send Message
Receptionist awaits confirmation
Receptionist marks idle

Figure 4.3: Use case editor UI mockup.

A user interface mock-up is shown in figure 4.3. In the top of the screen is a tab selector that selected the use case currently being edited. In the selected use case panel, we can see the main scenario (the selected tab), where the list use case steps are shown. The selected step is “Receptionist send Message”, which is also shown in the bottom of the user interface, where it can be edited. The available actors and concepts are shown on the right side of the user interface. The pre- and postconditions are not included in the user interface mock-up, but are included in the meta model discussion.

The component/structure concept had the big advantage, that it was a good fit for – and aided the development of – the meta model. This concept was not chosen, due to the significantly increased workload that it involved, and the added meta model complexity. This is elaborated in the summary. The concept and its corresponding meta model was ultimately simplified for the next concept.

4.3.1 Meta model

This section contains a brief discussion of a meta model that could be used for translating use cases into test cases in this concept. The discussion is supported by the high-level graphical model depicted in figure 4.4, which shows the concepts that are being discussed.

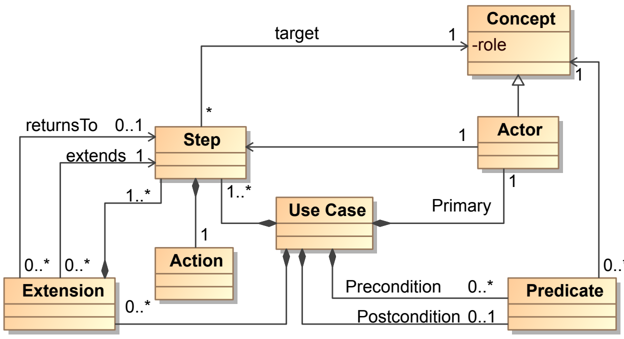


Figure 4.4: Partial meta model for creating use cases models in concept 2.

The central class of the meta model is the use case. It is composed of stakeholders and a primary actor, pre- and post-conditions, use case steps (the `Step` class) and a number of extensions. The primary actor is mostly for information purposes, as the Actor class (which models the domain actor) is also contained within the steps. The `Step` class models the use case's steps, and are not very flexible as they expect a use case step to consist of an actor, a target and an action. The target may be either a domain actor or a domain concept.

The extensions are treated as lists of use case steps, as well as the main scenario. In addition, they have extension points, as well as optional return points to use cases. These map to the formulations where an actor returns to a specific step in the main scenario – for example "user returns to 2." – in extensions. The pre- and postconditions are modeled as predicate classes, that ensures that some property holds for an associated concept.

4.3.2 Mapping to application domain

In order to map the models produced with the tool, we need to translate them to models usable by the "test mapping domain". Figure 4.5 shows the conceptual model of how a meta model supporting this domain could look like. The discussion in this section will not go into too much depth with the meta model, as it has not been chosen as the implementation model. There may also be inaccuracies in it, for that exact reason. It specifies, like the meta model in figure 4.4, that the use case consists of an ordered list of steps, where actions consist of; one or more actors, one verb describing the action and one target for the action (object for verb). It also contains the corresponding associations to predicates (pre- and postconditions) and actors. But, unlike the simpler model for building the use case models, it is extended with additional classes, needed to construct models suitable for test generation. The mapping process is expected to be done by a developer, and could be realized by a textual test mapping language, discussed later in this section.

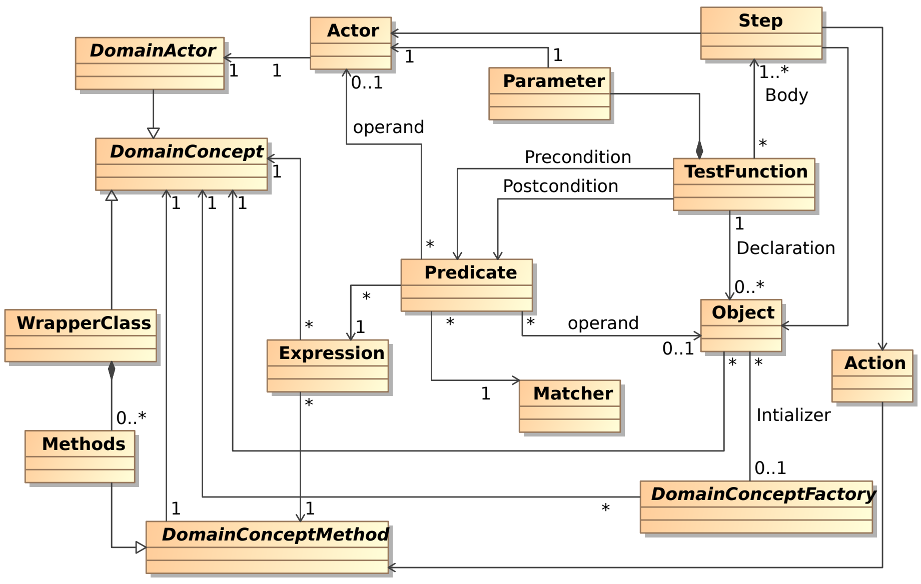


Figure 4.5: Concept 2 use case mapping.

In the application domain, we have a set of domain actor, and domain concepts. The actors from the use case domain may be mapped to domain actors, acting in a specific role. In our system – for example – a “contact” actor may act as the “callee” in some use cases, which effectively is a role. A domain actor needs to be specialized by application domain actor classes, that need to be mostly coded by hand, but could be stubbed out by the tool. The wrapper class has the purpose of covering the domain concepts, functionality-wise. It will supply a set of methods, available for the developer to map to use case actions – hence the association to the action class.

Predicates are treated as functional expressions, that link to a matcher which must be realized (in code) by a function that returns a boolean value, based on input. For instance, quantifications such as “greater than” and “equals” are examples of matchers.

One thing that proved useful, was the parameters of the test class. Basically, to be able to specify which domain concepts should be part of the signature of the test function. An example would be that a test function involving a receptionist actor, would need to have the signature `exampleTest(Receptionist r)`, or similar. The basic is just that the involved concepts, need to be supplied to the test function from the test support framework.

4.3.3 Mapping language

In order to define the mappings from the use cases to the concepts in the meta model, the concept of a mapping language emerged. While the language never left the conceptual state, it is described in this section for completeness.

The mapping language is written in the aspect of an actor. It is conceptually the same as a recipe for a holder-class (class as i object-oriented programming) that knows about every interface that it needs to access, and every resource that needs to be provided to it.

The example language shown in figure 4.1 illustrates how a language like that could look like. The example uses indentions to indicate ownerships. The label **Receptionist:** indicates that the mappings and properties below are owned by that actor. The requirements – which could translate to class fields – are listed (new-line separated) below the **requires:** keyword. The **maps:** keyword translates to provided methods (as those from the figure 4.5). The methods have a signature, and a function body that should translate into a class method with the supplied body. These mappings are meant to be done, manually, by a developer (a mapper actor) trained in the method beforehand.

```
Receptionist:
  requires:
    MessageContentGenerator messageContentGenerator
    ReceptionistState currentState = ReceptionistState.Unknown

  provides:
    changeState (ReceptionistState rs) -> currentState = rs

  maps:
    sends_message (Message msg) -> msg.enqueue()

    types_in (Message msg) ->
      msg.content = messageContentGenerator.next().content

    returns_to (ReceptionistState newState) -> this.changeState (newState)

  properties:
    is_ready -> receptionistState == ready
```

Listing 4.1: Example language for mapping concepts.

4.3.4 Summary

The concept outlined in this section was implemented on a prototype-basis – without the mapping language or a user interface. The concept quickly proved infeasible for several reasons.

- In order to create a use case, you first had to define every component (actor, action, concept) of the use case. This work-flow is quite cumbersome, and increases the work-load of creating use cases significantly. Having to decompose the use case before actually writing it is not very user friendly and shifts the focus from use case writing, to *ad hoc* modeling.

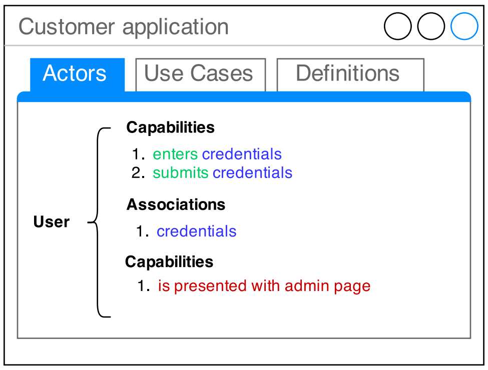
- A potential problem with this approach is that the rigid structure of having the actor-action-object constraint could quickly lead to artificial use cases that fitted the tool, rather than the problem domain. As it would be very close to the implementation, it was also speculated that terms from it would find its way back into the requirements. This would lead to a too technical jargon in use cases. This is generally a bad idea as it alienates the customer [CK92].
- The predicates provided little, or no value in the experimental prototype that was implemented, and revealed that they could be written in the mapping code very easily without the need to include them in the meta model.
- The problem with the mapping language is, that it came very close to a programming class, and it felt very much like re-inventing object-oriented programming classes. The very big benefit of having a mapping language, was that the meta model of a system under test could be linked to that of our use cases, and provide a much better analysis of the use case translation. It would, thus, enable tool-level analysis awareness of missing mappings.

There were some good concepts that proved useful in the next concept. Mainly, the idea of inferring knowledge about *who* (domain actors) is performing an action and *what* (domain concepts) is the target for the action. This is very useful information when we need to extract information about what the resource requirements for the tests should be. The idea of pre-defining some of the concepts and actors for the use cases to share is also carried on to the next concept.

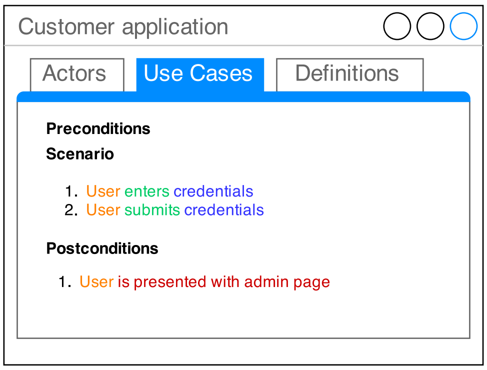
4.4 Concept 3 - Hinting tool

The third concept is a hybrid between the first second concept introduced earlier in this chapter. It is placed – functionality-wise – somewhere in between. It kept the ability to predefine actors and concepts, but didn't enforce the composition structure as the previous concept. The use cases were kept as text-only steps that could be analyzed using the defined actors and concepts at any point – after they are written. The basics of the concept is "type hinting" or "visually annotating", as it will derive actor types an concepts from the use cases, using text-matching. From these type hints, the tool would then be able know which resources should be provided to tests in the test generation step. The concept is implemented as the current prototype.

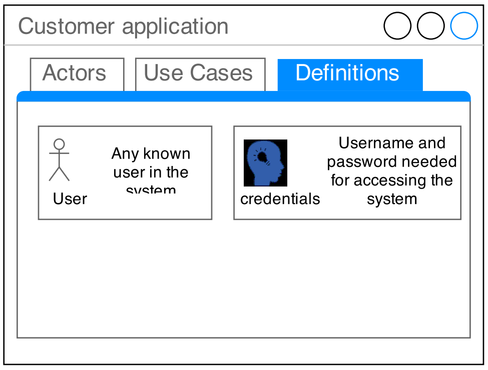
Figure 4.6 outlines the screens expected in a user interface of realizing this concept. The first tab of mock-up – depicted in figure 4.6(a) – shows an overview of the actors defined. Currently, only the "User" actor is defined, and this actor has some associations and capabilities, that are extracted from the definitions created, and use



(a) Actors tab


Customer application
Actors
Use Cases
Definitions
Preconditions
Scenario
1. User enters credentials
2. User submits credentials
Postconditions
1. User is presented with admin page

(b) Uses cases tab


Customer application
Actors
Use Cases
Definitions
User
Any known user in the system
credentials
Username and password needed for accessing the system

(c) Definitions tab

Figure 4.6: Mock-up screens of the customer user interface..

cases that the user partakes in. The next tab, which is depicted in figure 4.6(b), shows the selected use case with the different parts highlighted. One color for actors, one for actions, one for objects. Figure 4.6(c) shows a tab that contains the current definitions. A **definition** is the representation of a meta-model concept (or actor) in a particular role. This definition needs a unique name, which should correspond to a concept already found in the domain model. The domain model, if defined beforehand, could also be thought to be imported to the declarations.

4.4.1 Hinting process

The purpose of this section is to explain the hinting process through an example. The section starts from a use case description, and tries to go through the steps needed to convert it into an acceptance test. The use case used in this text is simplified and contains only the basics needed for interpreting and translating the use case. It is shown in verbatim text seen in listing 4.2, and does not illustrate any user interface specific details. It contains three lists; the main scenario, the postconditions and the preconditions. Each of these lists are simple text strings.

```
Scenario:
  Receptionist types in message
  Receptionist sends message
  Receptionist marks state as ready
Preconditions:
  The receptionist is logged in
Postconditions:
  The message is stored
  The receptionist is ready to handle the next call
```

Listing 4.2: Use case example.

The first thing we need to do to translate this example, is that we identify the domain concept contained within the list steps in the text. In this case, we here observe that it includes the concept "message" and a "receptionist" actor. Additionally, some interaction between the message and the actor, which we define as actions.

In listing 4.3, we have highlighted the different parts of the use case, using an orange color for actors, green for actions, blue for domain concepts, and a dark red for attributes. Using this highlight, it illustrates the abstract interpretation and the interaction between the different parts very well. In this case, the actor role names match their domain actor names, but it is important to note that the participating actors (and concepts), participate in a role – rather than as an actor. The role of definitions are also illustrated in the domain model later on.

```
Scenario:
  Receptionist types in message
  Receptionist sends message
  Receptionist marks state as ready
Preconditions:
  The receptionist is logged in
Postconditions:
  The message is stored
  The receptionist is ready to handle the next call
```

Listing 4.3: Use case example with its different parts highlighted.

Based upon the information in listing 4.3, we extract the components from the use case and translate it into test code that could look like listing 4.4. In the tests, each of the concepts in the example is mapped to a class, that is given an object identifier. In this case, we have the "Receptionist" and "Message" classes that have the identifiers "receptionist" and "message" – respectively.

Given the amount of information in (and definitions of) the use cases, we can easily generate code such as the one shown in listing 4.4. But these functions are still very high-level, and don't assert anything about the system being tested, it's just an alternate representation of the use cases. To be able to actually test the implementation, we need to fill in the methods used on our receptionist and message object. The methods will act as the link between the tests and the implementation, which is the mappings. The mappings are – in this conceptual model – meant to be written manually by a developer. Most of the code can probably be generated as stubs. The next sections goes through the test function, discusses the individual parts in more detail, and suggests conceptual implementation mappings by classifying them.

```
boolean usecase_test (Receptionist receptionist, Message message) {
  // Preconditions
  expect (receptionist.is_logged_in(), true);

  // Scenario
  receptionist.types_in (message);
  receptionist.sends (message);
  receptionist.marks_state (idle);

  // Postcondition
  expect (message.is_stored(), true);
  expect (receptionist.is_ready(), true);
}
```

Listing 4.4: Suggested structure of generated test case.

Prior to running the test function, the supporting test tools (covered in 3.2.3) need to supply the objects "message" and "receptionist" pre-initialized. This makes them a very explicit part of the resource dependencies to the test, and should be added to the signature of the test function. The dependencies are extracted from the resources requirements of the test function block.

The second thing that needs to be done is to add the preconditions. Having tagged the types and the property of the use case that must hold, we can translate the precondition into an "expect" expression.

Next in the test, the statement `receptionist.types_in (message)` is a method that belongs to the receptionist actor, and requires knowledge of the "message" domain concept. The action of typing in a message is conceptually the creation of message object. So, to be able to make meaning of the test, we need some way of simulating the message creation. This infers the need for a content generator mechanism. A concrete implementation could be a simply fixed "dummy object" or an object that is generated with content from a randomized pool. In the pseudo-code in listing 4.5 a MessageContentGenerator is associated with the Receptionist class as a basic class field.

```
class Receptionist {
   MessageContentGenerator messageContentGenerator= ...;
   ReceptionistState receptionistState = ...;
```

```

  void types_in (message) {
    message.updateContent(messageContentGenerator.content);
  }

  void sends (Message message) {
    message.enqueue();
  }

  // Alias function.
  void returns_to (State newState) {
    changeState (newState);
  }

  void changeState (State newState){
    receptionistState = newState;
  }

  boolean is_ready() {
    return receptionistState == ready;
  }
}
```

Listing 4.5: Pseudo code representing Receptionist domain actor.

The next statement in listing 4.4; `receptionist.sends (message)` takes the argument “message” and makes it available for other actors to access later on, by storing it persistently. This is typically done using a database or file store. By sending a message to a message store, we can delegate the action to an “enqueue” method of the message object, which needs to have a notion of where it is, or should be stored. This is realized by having a “messageStore” interface which is a service interface object that, can actually be originating directly from the code base of the system under test.

The final statement in the scenario is `receptionist.returns_to (ready_state)`. This is actually a mutation function that alters the state of the receptionist actor. This state change could be global and should then be updated multiple places, which then leads to additional “state-store” dependencies. Here, we also note that there is a concept of a ready_state. This is an explicit state change that could, possibly be linked to a state machine (see 5.3.3 contained within the receptionist actor object).

The final thing done in the tests, is the postconditions, which are translated to expressions conceptually equivalent to preconditions.

An open question that was raised during the development of this concept is; how much extra information can we add. Can we use some extra classifiers to increase the test generators understanding of the annotated use cases? For example, if we were to add a “persistent concept” annotation to message, would it improve test generation?

The pseudo code for the “Message” class is shown in listing 4.6 for completeness, and will not be discussed in detail.

```
class Message {
  RESTMessageClient messageStore = ...;

  void enqueue() {
    messageStore.enqueue(message);
  }

  boolean message.is_stored() {
    messageStore.contains(this);
  }
}
```

Listing 4.6: Pseudo code representing Message domain concept.

4.4.2 Summary

There is a lot of manual work associated with this process, as the only help you will receive from the conversion tools, are method stubs. This, however provides programmers with good guidelines on what the method should do. The lowered value compared to the second concept is, that we no longer have a complete view – from the use case tool – of the test model. On the other hand, in this concept, developers can write the tests using a general purpose language that they should be more comfortable with, rather than a mapping which must be learned first.

This concept is – as previously mentioned – implemented in the current prototype of the tool. The form is, however not identical to the concept introduced here, but simplified further, which we shall also see in the next chapter.

A common denominator for all three concepts, is that manual mapping seems to be a necessity. In the second concept, a mapping language was introduced, but the fact that it almost mirrored the concept of a programming class, made it infeasible. In particular, because learning developers a new language is costly and the gain from learning to use it, did not match the questionable value of getting a mapping model.

CHAPTER 5

Analysis

This chapter documents the analysis performed. This is done in order to support the major design decisions in the scope of building a tool that implements the hinting tool concept from 4. It focuses on building models that enable translations from the use case representations to tests – focusing on broad test coverage. This chapter also presents an execution model that is suitable for translating use case steps to executable source code functions.

5.1 Meta model

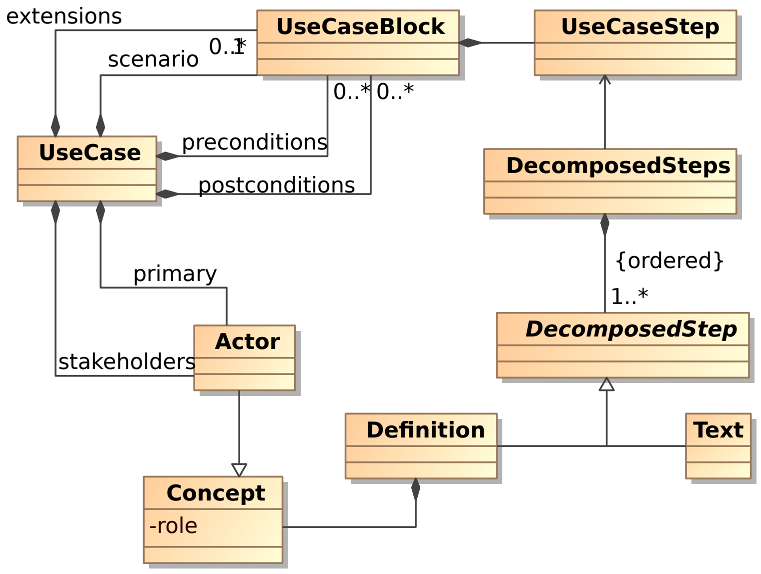


Figure 5.1: Meta model of the third concept.

This section discusses an evolved and simplified version of the meta model introduced in chapter 4, and covers the changes that happened when the model transitioned from concept to design.

A central concept that was removed, is the actor actions. While they provided additional information about what actors were capable of, they added little or no value to tests. The predicates that signified the pre- and postcondition needs to be mapped as code in the test templates (see section 5.4). This effectively means that the mapper actor needs write up a function that explicitly break the execution within the conditional steps, in the case of an unmet condition. An example function that realizes this is shown in listing 5.1 where an `AssertionError` is thrown if the pre-/postcondition fail.

```
void the_user_is_logged_in(User user) {
  if (!user.loggedIn) {
    throw new AssertionError("User is not logged in.");
  }
}
```

Listing 5.1: Example pre-/postcondition code mapping (written manually).

When performing a translation, every use case step will be broken down into a set of decomposed steps. A decomposed step is an *ordered* list of either a definition or just a text. The decomposition is a reversible process, that will be performed by the use of a definition set. A definition set is the information which added by the use case writer and links domain concepts to use cases. Definitions are used to infer which concepts or actors need to passed to the generated test case functions, when performing the use case translation.

5.2 Translating a use case

Once we have built the use case models, translation of them is a matter determining which paths they contain and translating them to the test function calls. The signatures of the functions are automatically inferred by the concept types linked in definitions.

During the design process, three major constraints for use cases – in regards to test generation – were defined.

Steps: Every step in a use case scenario is a synchronous, self-contained action – every step must finish before the next starts. Single steps may mutate the system state.

Termination: A use case scenario (the main scenario) should always terminate. It should have distinct start and finish. Otherwise, it would be impossible to generate tests from it.

No extensions nesting: A design decision that simplified the path determination.

We treat the use case scenario with extensions as a directed graph – potentially containing cycles. Every node of the graph represents a use case step and every edge the invocation of the next step. An example of a use case graph is shown in figure 5.2. It consists of a main scenario, which are all the s_n states (state s_1 to s_5) and an exit point. The exit point is provided to signify explicit termination for the main scenario. Use case extensions that never return to the main scenario, but merely terminate the use case returns directly to this node. The extension node $e_{3.1}$ represent an extension that illustrates this exact behavior.

The next subsection goes into more detail on how to extract the different paths from the use case graph.

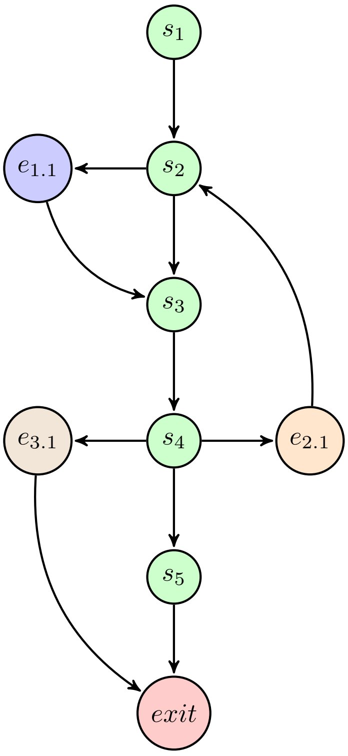


Figure 5.2: Graph depicting different paths through a use case.

5.2.1 Determining paths

A path of a use case is any graph path that traverses the graph from the initial state s_1 to the (*exit*) node. In the simple use case graph in figure 5.2, we enumerated 11 paths - not counting loops.

The basic path of an extension (the one that diverts from the main scenario, and goes straight back to it), once finished can be represented as the union of the path leading in, the path of extension scenario, and the path from the exit point of the extension, to the exit point of the use case represented as a *ordered* set of nodes (5.1).

$$p_{exten} = \{s_1 \ldots E_{entry}\} \bigcup E_{scenario} \bigcup \{E_{entry} \ldots exit\} \tag{5.1}$$

Harvesting each additional path (p_{exten}) will be a matter of combining the scenario of the use ($E_{scenario}$) with the all the different possible entry and exit paths, as seen in the itemized list above.

Once a list of paths is retrieved, it is possible to convert every path, which is essentially a list of active actions either performed by, or affecting, the primary actor. This list will be translated into a test by converting it into code snibblet, and joining them together in a code block that becomes the body of a test function.

Until now, the fact that back edges (edges that go backwards in the directed graph) can occur, has been ignored. These edges raise the bar for identifying unique paths, as loops are introduced. Loops are potentially infinite and need to be detected and mapped appropriately.

The loop detection is quite simple, if we disallow nested extensions. In that case, the loop detection can be performed solely by detecting that the return point of the extension is not before the entry point in the main scenario. When a loop is detected in an extension, the extension may be flagged as “looping”.

Loops make it impossible to achieve 100% test coverage, but we could approximate the behavior by defining a fixed number times that the loop must execute and a fixed path that it should always take during the loops. While this is far from optimal, the test generation tool could provide some convenience functions. These should aid the use case writer in choosing which paths should be taken in which loop execution and how many time it should loop. The general use case coverage, however, should be high, as we have the opportunity to generate tests for every possible path.

5.3 Mapping the test to the domain

This section discusses the individual components of a use case with the perspective of converting it into an executable test. In order to write up functions that implement the generated function block stubs in actual system behavior, a programming model is presented in this section.

The programming model builds upon the constraint of having the use case represented as an ordered list of steps that are unrolled into paths, that represent the different ways a use case can play out. The steps of these paths need to be executed in order.

5.3.1 Step execution

Executing a test of a given use case path may be considered a long function call-chain. Each new function call passes its computed state onto the next function, which is how variables get passed also. This method of passing along the state is a common pattern in interpreters and compilers, where this state is referred to as "the environment", and the state S_n initially fed into the tests with a list of variable and function definitions – similar to those our meta model. In our test-case compiler we adopt this concept. One of the benefits of this is to have the ability to have an exit procedure that performs state clean upon exit of every use case.

An execution of a use case path that consists of n steps could look like the statement in 5.2, where every S_n, $Precondition$ and $Postcondition$ are all functions taking the mutated environment (the current state) as an argument, returning the environment with the changes they made. Th chain is evaluated right-to-left, and the first function to be called is, thus, the $Precondition$ function.

$$Postcondition \rightarrow S_n \rightarrow S_{n-1} \rightarrow \cdots \rightarrow S_2 \rightarrow S_1 \rightarrow Precondition \rightarrow env \tag{5.2}$$

The function is applied to the statement, then the result is applied to the matcher which then returns a success or failure value depending on the outcome of the evaluation. For anyone familiar with the concept of monads in functional programming, this concept is quite similar. Here every function carries the state needed by the next function, and forms an execution chain. The model may also be, represented in procedural languages by a list of statements that only operate on input parameters and return their result.

An application of this, in a concrete syntax, is shown in listing 5.2. It shows the nesting of functions in a use case that merely logs in a user, to log it out again. The change is made to the environment and then passed up the call chain to finally be processed by the **postcondition** function. This approach is very suitable for functional programming languages, as it is functionally pure (assuming each function has no side effects).

```
postcondition (
  user_log_out (
    user_log_in (
      precondition (environment)));
```

Listing 5.2: Example evaluation of a functional call chain.

Alternatively, this can be done in an imperative fashion, using a procedural programming language as shown in listing 5.3. The concept of chaining is the same as above, but now the mutated environment is no longer returned, but instead passed implicitly from one statement to the next.

```
precondition (environment);
user_log_in (environment);
user_log_out (environment);
postcondition (environment);
```

Listing 5.3: Example evaluation of a procedural call chain.

The latter example is the most readable of the two, and the one that maps (visually) the use case steps. It, very explicitly, states which steps are taken for the use case to succeed.

5.3.2 Test case environment

In order to run the tests, we need to supply them with some input values, which are the actors and domain concept that they need to operate on. These values are inferred by the definitions added to the use case model by the writer of the use case. So, a use case that involves a "Bookkeeper" actor needs a pre-built object of that type handed to it by the test environment.

Typical test runners use a test model that consist of three basic steps; setup, run and teardown. Setup and teardown are different from pre- and postcondition in that they are unrelated to the test itself. They merely make sure that objects are initialized with right values and, are in the state that the test expects. This fits the need of having to input values to test quite nicely.

Assuming that the actor and concept classes that map to the domain model are already written as code, we may use these test runners (or harness) as scaffolding for our test, providing them with the necessary inputs. An example: For the use case "Send message to contact" (see appendix D). We need, beforehand, at least a class representing the actor "Receptionist" and a class representing a message. Initialization and de-initialization of these objects can then be done by the setup and teardown function of the test runner, respectively.

The next obvious question is then; how do we define which objects are available for the test harness? In the test case system, we have made a configuration file that outlines which actors are available, and simply added the data to initial state of the system under test. Another approach could be to turn the creation of resources needed by tests into a use case itself and feed the output of that generated test into the next test. This approach is covered in more detail in the next chapter.

5.3.3 Target system state

This sections outlines, from a test runner point of view, a design that – ideally – provides a clean view on development system state and how the use cases should mutate it in. It, roughly, re-applies the design from section 5.3.1 on a higher level and encapsulates the (test) system state into an even larger environment.

Assuming that the use cases are self-contained[Lar05] in the way that every action and alternative, for a given actor, may be put into a single (large) use case, then it should also be possible to treat use cases as functions that operate on a global system state.

Use cases may also specify some expectations to the system state. This is what is defined as preconditions. If we, however, maintain a system state analogy, we may model use case executions as a set of mutation functions that affect the global system state.

A simple example: An actor *accountant* has a use case *accountant creates invoice*. This use case requires that the *accountant* actor has previously been created. The creation could be provided by the *admin* actor's *admin creates accountant* use case.

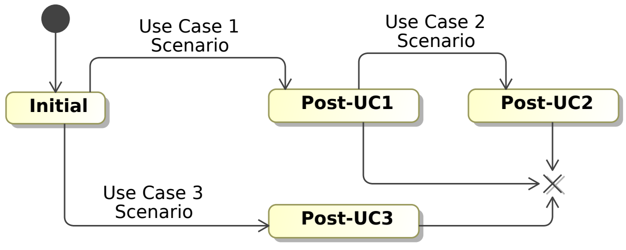


Figure 5.3: State machine of the perceived system state.

So, the *accountant creates invoice* effectively has a precondition that is provided by *admin creates accountant* to make the global state match what is expected to execute the *accountant creates invoice*. The concept is illustrated in figure 5.3, where *Use Case 1* is a prerequisite to *Use Case 2*, but *Use Case 3* has no prerequisites. In order to reach completion (termination) and coverage of *Use Case 2*, *Use Case 1* has to be executed.

Each use case state (an example is **Post-UC1** in figure 5.3) is a super-state that contains the state space of every alternative scenario. Basically, every node, of every path extracted from the graphs described in section 5.2.1 is also a state, and every edge, the mutation function.

This model is useful to consider when programming the test support tools and may even be supported by a concept, such as the event stack validation described in appendix E.1, where the states and transitions are logged and replayed through a set of state machines. But, for the majority of implementations, it should be considered a programming model. Using this model, developers may also chose to build their precondition mappings as other use cases.

5.3.4 Simulating error conditions

Use cases may branch out and go to alternatives actions. These branches may signify that an error has occurred, and may be difficult to emulate. Injecting errors in systems to see how they respond to it, is a field of study on its own. For this purpose, some of the failure scenarios could be described as use cases. That is, a sequence of steps that lead up to the error and how the actor experiencing the error must handle it. Simulating error conditions outside of the control of the system under test may be impossible by a mapper. One possible solution is to add an assumption here and merely treat the event or decision as having occurred. An **assumption** is a no-operation statement that, in some way, must notify the test runner about its occurrence in a test. This may be realized by just injecting a message into the log stream of the use case test.

5.4 Technique overview

The technique in this thesis may, at this point, be summarized as the following steps:

1. Write up use the cases in the tool. This process can be supported by using activity diagrams as a base. This is also covered in section 3.1.2.
2. When the use cases are written, they need to be annotated using the hinting tool. Each actor and concept role need to be marked as their respective types.
3. When this is done, a template needs to be created. The template should support the individual functions and function stubs with the correct signatures could be generated from the use cases – once annotated.
4. The templates serve a mappings from the use cases to the implemented system and the generation of executable tests from the use cases may now be performed.

Once the tests are generated, they can be checked into a source code repository for a continuous integration service to use or be run locally – depending on the chosen test runner.

5.5 Summary

The path coverage, despite the fact that it is a sound concept, does not really scale. For every new extension in the use cases, every test need to be regenerated taking into account this extension. If the system had n paths before, adding a new use case extension, adds an exponential number number of new paths – especially in the presence of loops. This is very impractical in a development environment with continuously evolving requirements, and can make the running of tests – because of the large amount of them – so time consuming that result is useless, once it arrives.

The same basic problem applies to the target system state, and its associated use case mutation functions. If we were to build up complete model for this, it would lead to a state-space explosion very fast.

The simplified design of the meta-model does, however, support a more intuitive workflow, where the use case writer may write up use cases, almost as they would write up use cases normally. The process of this will also described in the next chapter.

The implementation discussed in the next chapter will use the meta model introduced in this chapter as its model for use cases. The graph model and path extraction will be implemented, but the test case environment and target system state are left as design patterns for a test mapper to use when writing mappings and test support tools.

CHAPTER 6 Implementation

This chapter describe the tool that was implemented as part of this thesis, to enable the generation of tests from use cases. The chapter provides a high-level architectural view of the test system, and an example on how a use case goes from description to test.

As already mentioned in chapter 5, the actions of users were eventually dropped, as they provided little or no value in test. This will become more evident, when an example of a generated test is shown.

The protocol specification for the implementation is shown in appendix F, test results for it can be found in appendix G, and the database schema is located in appendix H. The handbook, written from the point of view of the user interface, can be found in appendix B.

6.1 Architecture

This section provides a description of the architecture used in the implementation of the use case mapping and generation tool.

The implementation is done in a client/server architecture. It consists of three basic components; a client, a service and a shared library. These map to the folder structure. It is web based to allow easy collaboration. The distributed model is built around a

layered model-view-control (MVC), that uses a dedicated test backend. Our backend built upon the Dart unittest library[1] configured for junit xml output, the Jenkins continuous integration server, and Git revision control system. Git is used, first and foremost, to control revisions, but also as a test code distribution system. The Jenkins server will see a change in the Git repository, update to this version and trigger a new build. Being that the tests output junit-format xml documents, Jenkins can parse this and provide detailed reports on success/failure output, run-times and historical data.

The layered MVC introduced above, is a model commonly used in web programming. It is illustrated in figure 6.1 and is a method for maintaining the MVC paradigm – in a distributed system. The service interface provide an API that delegates to an internal controller. This is allowed to modify the model. This model is only stored

[1]`https://pub.dartlang.org/packages/test`

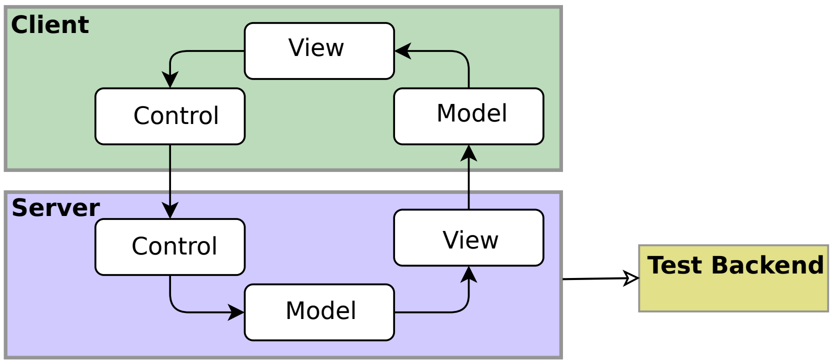


Figure 6.1: Layered model-view-control architecture with test backend.

on by the service, but may be retrieved (or even pushed) to a client's model via a representation, which then becomes the service's view. The client's model is pushed to a local view, which is a user interface, that may send change requests to its local controller that delegates it to the controller of the service.

The client and the server share their code-base around a library package that contain (meta model) models, serialization/de-serialization tools, and the (use case) translation tools. The translation can, thus be done either by the client, or the server.

6.2 Use case translation and generation

This section contains a description of the implemented components and utilities needed to support the use case to test translation. A description on how the use case is unfolded into paths is also provided.

A use case that is unfolded to a set of paths, each having its own test function, will produce a log transcript on every run. This output log includes a list of assumptions (see section 5.3.4) and potentially an error. Errors arise when an expectation fail, or a system error occurs. These error are fatal for the test run – i.e. the run will stop at the point where the error occurred. Assumptions are for use case steps that are mapped to assumption statements and are non-fatal.

The actual translation of use cases into tests are handled by the server, but requested by the client. When the server receives such a request, it generates the test files and runs a static code analysis on them, and gives back the client an analysis report, along with the generated test code.

The tests are generated from test templates, that are explained in the next section.

6.2.1 Test templates

Test templates are, in this thesis, hand-written source code files that are missing the actual test code function bodies. They provide the tests with the implemented functions of what they need. So, if a test makes a function call to a function called "receptionist_answers_call(receptionist, call)" with the passed parameters "receptionist" and "call", then the template is expected to provide this function.

The templates are implemented in a simple form. They are source code files that have a single placeholder entry that is replaced with the generated code by the tool. The placeholder is located within a comment, so template files can still be parsed and compiled before the generation step, enabling them to be verified prior to use.

The templates are the mapping link between the use case and the system under development, and are – for the example use in this chapter – linked to our case study system (section 2.2.

They assume the presence of a set of test support tools, in the location where they are generated. Actor and concept classes (explained in the next section) are part of these test support tools.

6.2.2 Actor and concept classes

Actor classes represent a domain actor and should be able to support the behavior that is expected from an actor of that type. The classes will be hand-written by a developer, but the functionality that is needed may be inferred by the use case steps involving the actor/concept and the actions it performs.

6.2.3 Definition

Definitions are mappings from use case roles – which are simple string identifiers – to actor or concept types. Definitions are stored in a global definition set that may be reused by different use cases. The definitions are used to determine which actor and concept types that should be part of the signature of; both the individual use case steps, but also the source code function signature of the use case path functions.

6.2.4 Normalization of use case steps

To be able to make a use case step into a function call, we need to normalize it to a string that follows the general conventions of function call in programming languages (and specific for Dart, as this is our target language). We also need to be able to recognize identical steps in different use cases. It gives us the following constraints for a normalization function.

- Programming conventions must be followed. This means no spaces in function identifier, no redeclaration of function identifiers with different parameter set, and no identifiers must start with a number.

- Function should be deterministic. So, given specific input, they always produces the same output.
- Not specifically a constraint; but a normalization function should – if at all possible – produce human-readable output, to make it easier to trace generated code.

The normalization described above could be realized by a good hashing algorithm – prefixed by a constant string to avoid prefixed numbers in identifiers. This, however would collide with our desire to have human-readable functions. The implementation ended up doing the following:

- Replace all non-allowed characters with underscores.
- Prefix the functions with an underscore
- Transform every character in the string of a step to lower case

A function identifier for "Receptionist answers call" then becomes:

```
_receptionist_answers_call
```

One thing that is needed for a function *call*, are the function parameters. These are supplied by the definitions associated with the use case. So, given that the set of definitions contains the "receptionist" and "call', the signature of the function would look like this:

```
_receptionist_answers_call (receptionist, call)
```

6.3 Example translation

This section goes through the steps associated with creating tests from use cases in this implementation. The example is done disregarding the concrete user interface details, which can instead be found in the handbook, in appendix B.

The example reuses the use case from the 3rd concept section (4.4), that is also repeated in figure 6.1 for convenience.

```
Scenario:
  Receptionist types in message
  Receptionist sends message
  Receptionist marks state as ready
Preconditions:
  The receptionist is created
  The receptionist is logged in
Postconditions:
  The message is stored
  The receptionist is ready to handle the next call
```

Listing 6.1: Use case example revisited.

As mentioned earlier, the modeling of an "action" was dropped because it provided no real value in test generation. With this ignored, the highlighting of actors (orange) and concepts (blue) – done by the use case writer – will look like listing 6.2, and lead to two declarations; one declaration of the role of "receptionist" actor of type "receptionist" and a domain concept role of a "message" of type "message". The concrete user interface has visual containers that signify how to build up pre- and postconditions and main scenario. This effectively gives us the composition and structure of the use case.

```
Scenario:
  Receptionist types in message
  Receptionist sends message
  Receptionist marks state as ready
Preconditions:
  The receptionist is created
  The receptionist is logged in
Postconditions:
  The message is stored
  The receptionist is ready to handle the next call
```

Listing 6.2: Use case example with its different parts highlighted.

When the declarations are in place, we can pass the structured use case, along with them to the generator. It will perform normalization on the steps (and actor roles), in order to build a test that does not break the syntax of the programming language. Listing 6.3 shows the generated code with normalizations applied. A thing to note is that the function signature for the generated functions includes every concept and actor that is used within their bodies. This is done to support chaining model presented back in section 5.3.1. The code generated here is clearly not runnable, as the called functions are not declared. We need to write up a template that we can place the functions in.

```

void preconditions(Receptionist receptionist) {
  the_receptionist_is_created(receptionist);
  the_receptionist_is_logged_in(receptionist);
}

void postconditions(Receptionist receptionist, Message message) {
  the_message_is_stored(message);
  the_receptionist_is_ready_to_handle_the_next_call(receptionist);
}

void scenario(Receptionist receptionist, Message message) {
  preconditions(receptionist, message);
  receptionist_types_in_message(receptionist, message);
  receptionist_sends_message(receptionist, message);
  receptionist_marks_state_as_ready (receptionist)
  postconditions(receptionist, message);
}
```

Listing 6.3: Example of generated code without a template applied concept.

Listing 6.4 shows the template that supports the generated use case from listing 6.3. Initially, we require some services, so we instantiate a client for them. The specifics of passing the configuration to them are omitted in this example for simplicity.

The first mapped function elaborates the behavior of the statement "receptionist is created". This means, in our implementation, that the receptionist actor's user is known by the user service. If not, the test should fail. The unmet condition is, in this example, handled by writing a code block that uses the `fail` function provided by the test framework. It simply makes the entire test fail, with message passed as parameter.

In the test support tools of the case study system, the receptionist actor is automatically logged in, prior to running the tests. This means that we can safely assume, that this actor is logged in. Nevertheless, we may explicitly state it for the log using "assume". (listing 6.4 l. 16).

In order to simulate the creation of a "Message" object, we have instantiated a MockMessageFactory, which is a factory resource pool (see section 3.2.4). The factory will return a constructed "Message" object, ready for use in the sending step.

The step "the_message_is_stored" uses the MessageService to re-retrieve the message saved in the main scenario. In order to do so, it uses an `expect` function, that is part of the test framework. A thing to note, is that equality must be defined for the `equals` matcher to work.

The use case placeholder is located at the very bottom of the template and will contain the generated code from listing 6.3, once this template is applied to the use case.

```
 1 UserService userService = new UserService(...);
 2 MessageService messageService = new MessageService(...);
 3 MessageFactory messageFactory = new MockMessageFactory(...);
 4
 5 void the_receptionist_is_created(Receptionist receptionist) {
 6   try (userService.get(receptionist.user.id) {}
 7   catch (NotFoundException e) {
 8     fail("Receptionist " + receptionist + " not created");
 9   }
10 }
11
12 /* Later steps will fail if the receptionist is not logged in. */
13 void the_receptionist_is_logged_in(receptionist) {
14    assume("The receptionist is logged in by the test tools");
15 }
16
```

```
void the_message_is_stored(Message message) {
  Message fetchedMessage = messagService.get(message.id);
  expect (fetchedMessage, equals (message));
}

void the_receptionist_is_ready_to_handle_the_next_call(Receptionist r) {
  expect (r.state, equals(ReceptionistState.Idle);
}

void receptionist_types_in_message(Receptionist r, Message message) {
  // Generate() takes a Receptionist parameter - signifies author.
  message = messageFactory.generate(r);
}

void receptionist_sends_message(receptionist, message) {
  messageService.send(message);
}

void receptionist_marks_state_as_ready(receptionist) {
  userService.changeState(receptionist.user);
}

/*[USE-CASE-PLACEHOLDER]*/
```

Listing 6.4: Example template methods (written manually).

As we see from the listings, both the generated and written code is quite readable, and encourage decomposition into small and intuitive steps. It may be desirable to have a common set of utility functions that are shared among use cases, and to move some of the function to the belonging actor to build a more object-oriented set of test support tools.

The initial state of the test support tools are provided by a global setup and teardown function that encapsulates the entire pool of tests.

Generated tests are source code files meant to be compiled and run by a test runner. This means that they may be added to the same source code management (SCM) system, as the test tools for easy distribution. If the SCM system also supports revision management, then requirement changes that lead to regeneration of tests, will be evident in – and traceable from – the revision log.

6.4 Technical details

The client, server and library are built using the web-oriented programming language Dart[2]. It is optionally typed and class-oriented. It was originally meant as a replacement for the popular JavaScript as a natively supported browser scripting language via a virtual machine, built into the browser, like with JavaScript. This effort has recently been abandoned in favor of a cross-compilation to JavaScript instead.

[2]http://dartlang.org/

One of the big advantages of Dart is that it supports both web-client and server-side development and hereby also code sharing between server and client. It has a built-in package manager with many component with many uses, ranging from markdown visualization library to database bindings.

This service of the tool makes use of a HTTP-server and router package called "shelf", which handles all of the low-level work associated with server programming. Focus can then be placed on writing API response handlers instead.

Persistent storage is done in a PostgreSQL database, via the response handlers. The schema is quite simple and most of the tables focus on storing encoded objects, rather than enforcing database constraints. A latter implementation would most likely have these constrains. But, during development it has been convenient to be able to focus on the model, rather than database design, when the – inevitable – changes to the design occurred.

The client is web-based and uses some HTML5 features for presentation – the notification API[3], just to name one. It is built around the model-view-control (figure 6.1) and has no external dependencies, other than the service. Once compiled to JavaScript, it should be able to run on every modern browser.

6.5 Summary

While the mappings here were done to a specific domain, the concepts of mapping to actors, concepts, mock objects and interfaces should be applicable for most system. Most test frameworks (junit, testNG and others) should also have built-in matcher functionalities, such as `expect` and `fail`. If not, simple if-else statement and thrown named exceptions will yield roughly the same result.

The mapping procedure is simple, and can be done in whichever IDE or editor a developer prefers. The task of writing a single step function that performs one specific (high-level) task is easily distributed, as there shouldn't be shared state between the steps – other than the one they forward.

The definition sets, from which we can detect actors and concepts from textually analyzing the text of every use case step, is an *ad-hoc* solution, and should be improved in a later iteration. The simple analysis of looking for occurrences of the definition string by comparing strings, may result in ambiguity when two strings share a sub-pattern.

In the test support tools written for the test case system, we make heavy use of resource pools and object factories to supply the tests with the, quite large, resource blocks that they need.

[3] `http://www.w3.org/TR/notifications/`

The implementation contains open ends, a rough user interface, and although tested (appendix G) should be considered a conceptual prototype not suitable for deployment.

6.5.1 Known bugs and limitations

This section documents the identified bugs and errors in the application.

Textual analysis too primitive. The current textual analysis is performing raw substring matching on whichever definition is matched first. This means that if you have a definition of “something” and add another definition of “thing”, then the textual analysis will only match one of them. Which one, depends on which one occurs first in the definition set.

CHAPTER 7

Discussion

This chapter discusses and evaluates the implementation from the thesis, establishing some quantifiable goals, while discussing the general usefulness and feasibility of the approach. It also raises some issues that should be solved to increase feasibility of the technique of the thesis. Additionally, it identifies a number of the side-benefits that followed the approach and evaluates the overhead associated with it, based on collected data. The general applicability is discussed in the next section. This is followed by a section that describes the impact of changes in either requirements or system under test, how this technique handles it and how much extra work is associated with it.

7.1 General applicability

This section discusses the general applicability of the technique by identifying the general traits that made it suited for the case study system.

The technique described in this thesis was a good fit for the case study system. The architecture described in chapter 2 had clean well-defined (and exposed) interface clients and models readily available in the common framework already developed[1]. While other developed software systems may expose similar models and interfaces, they may have implementation-specific constraints that disallows re-using these components. This can either be because their dependencies disallow it (they depend on a complicated configuration of external libraries), or simply because they do not provide interface clients. While the latter can be resolved by writing new clients, the workload of the technique increases as well and the problematic decoupling from implementation occurs (see section 7.5).
A thing that may have made the case study system a good platform for this technique, is the fact that the use cases of the system are quite simple. This can also be seen in appendix D. The scope of the case study project has not been to provide a complex usage model, but rather the opposite. The target businesses for the product focus on low average call-handling time and, thus, will try to keep the use cases simple.

Other development projects, that have more complex use cases or interfaces, may experience larger complexity in writing mapping functions that realize their use cases.

[1] `https://github.com/Bitstackers/OpenReceptionFramework`

Another observation, that may have an effect on the specific fit, is the fact that the case study system uses a stateless architecture. This may make the use case steps easier to write, as the state is carried by the tests, rather than the services.

There is a correlation between the number of external dependencies and workload of mapping functions. By external is meant out of the developers of the system user tests' control. An example of an external dependency in this context is a remote banking interface, or a hardware device that behaves in a certain way. In this case, a harness (see section 2.1.1) needs too be written specifically for this purpose that either emulates or directly interfaces with this external interface.

The general applicability of the technique of this thesis is strongly linked to the number of external dependencies, complexity of use cases and architecture of the system. The higher the number of external dependencies, the higher the workload, and complexity of the test support tools themselves.

7.2 Impact of changes

This section tries to uncover the relative additional workload of this approach, from a requirement or implementation change perspective. Changes are, thus, discussed from these two points of view.

7.2.1 Requirement changes

One of the most relevant question of this project is: How do requirement changes impact the generated tests? The answer to this is defined by the characteristics of the change, which is either the modification of an existing requirement or the addition of one.

The action of modifying an existing requirement needs to follow a very specific procedure. First, the use case needs to be modified to accommodate the change, which could involve the rephrasing a single use case step, the addition or removal of an extension, or the (re)definition of a concept or actor. When this is done, the template will need to be changed to match the new/changed steps by either modifying existing mapping functions or writing up new ones. The regeneration of tests is possible to do automatically by creating an association between use case and template, which is not present in the current implementation. This step is thus manual. In essence, the workload of changing requirements is a relative 1:1 relationship between changed/added steps and mapping functions. So, one step changed means one mapping function needs to be changed.

An addition of a new requirement is defined by the same 1:1 workload, but has the additional task of creating the template needed to map to the implementation. If the new requirement defines a new actor – or domain concept – additional work must be calculated to build up the classes for these. A test tool overhead is presented in table 7.5 that suggests that at least 15% extra time is needed for writing up these classes.

The removal of a requirement or used step mappings may be detected by a good static analysis of the generated source code. It will inform the developer of unused functions, that are suitable for removal.

7.2.2 Implementation change

Most implementation changes should be transparent to the tests, and only interfaces changes would really have an impact. As the test support tools from the case study system share the interfaces classes with the implementation, inconsistencies will be detected in a re-compilation of the test case sources. It is therefore important that the test support tools track implementation. In the case study system, this is done by putting an incremented version number on every release of the shared framework (see section 3.2.1). The test runner will upgrade to the latest framework version, prior to building and running, hereby ensuring that the tests work with the latest interfaces.

If there is no sharing of interface code between tests and the system under test, it will very challenging and time consuming to keep the tests synchronized with the system. This is discussed further in section 7.5.

Any change to actors or concepts seem unlikely, as they are created solely for the purpose of tests, and should only be changed if the requirements change.

7.3 Benefits

This section presents some of the benefits identified in using the technique introduced in this thesis.

7.3.1 Requirement/test relationship

The big gain from generating tests from use cases are that there is now a strong link between tests and requirements. Whenever a change occurs in a requirement, we may simply re-generate the tests and run them again. Some additional work is, however, required as we also saw in section 7.2. But nevertheless, the requirement/test relationship is valuable benefit.

7.3.2 Simple use cases model

The structuring of the use cases have been kept simple in order to make it flexible. Instead of just supporting one view of how, for instance, a use step would look like, mere textual representations are used for them. The mappings to the domain models of the system under test are provided by keyword identifications that refer to a domain actor or concept, via a definition. This makes it flexible in two ways: The first is that you do not need to define every actor and/or concept beforehand, but may simply write your use cases – as use cases – and add the definitions later. The second benefit is that the simple structure and textual representation makes it easy to export to other platforms, for example a Wiki platform for documentation.

7.3.3 Mappings as source code

The use cases mappings to system under test are defined as being source code that is to be written by developers (acting in the role of mappers). This gives the benefit of not having to train developers in a mapping language, but instead enables them to use a tool that they should already know quite well – general purpose programming. If the developers are the same as ones that develop the system under test, then there will be additional benefit of them already knowing how to interface with the system, in the context of writing up the test support tools.

Mappings as source code also give access to a wide range of static analysis source code tools, that provide useful information about the generated tests and written mappings, and detect errors in both.

7.3.4 Side benefits

An observation that was made during the writeup of the tests using the second iteration of the test support tools was, that writing general integration tests became a lot easier. The code written becomes very verbose in the way that you state the test from the point of view of an actor performing an action. Informally, what is actually possible, is writing a requirement as a test. Not as in the topic of this thesis, but as a hand-coded test that asserts that a feature is present. Having the tools (the test framework) makes it very intuitive and easy to write. The concrete code for testing the example outlined in section 7.4.3 is shown in listing 7.1. This code is not generated, but written by hand, using the test support tools.

```
static void pickupAllocatedCall(Receptionist receptionist,
                                Receptionist receptionist2,
                                Customer callee) {
  String receptionNumber = '12340002';
  Model.Call allocatedCall;

  log.info ('Customer ${callee.name} dials ${receptionNumber}');
  callee.dial (receptionNumber);
  log.info ('Receptionist ${receptionist.user.name} hunts call.');
  allocatedCall = receptionist.huntNextCall();

  expect (receptionist2.pickup(call), throwsA(Forbidden));
  log.info('Test done');
}
```

Listing 7.1: Test code for single call allocation.

Some may argue that this piece of code maps very nicely to a requirement. While it may not be entirely possible to completely generate it from one, a requirement can most definitely be derived from the code. The example also captures an error condition, which may be challenging to describe from a use case in sufficient detail, so that test generation is possible.

Another side benefit from having use cases written as tests – whether they are generated or not – gives new developers a good reference point on how the system is intended to work. By reading the class files representing actors and concepts and observing how these interface with the system provides a good high-level overview of the general architecture and how it should be used.

7.4 Challenges

This section contains some of the challenges faced during the analysis and implementation of the tool.

7.4.1 Scalability

As mentioned in chapter 5, the open issue on path coverage – and especially scalability of it, remains. The number of paths generated quickly makes the size of the generated tests very large.

In order to overcome this, additional heuristics, such as stricter design guidelines for use case writing, must be enforced. If the use cases were limited to having a maximum number of extensions, or to be decomposed into several smaller use cases, it would be possible to keep the number of generated tests manageable.

Another angle on this would be to let the use case writer define which paths of the use case should be selected for test generation. As an alternative to automatic generation, that had to generate a test for every path, it would be focused on the scenarios that are likely to happen when the system under test is in production.

As the approach presented in this thesis has not been applied to a system with more complicated use cases, it is difficult to say with certainty that it would not be possible to apply it on that system. It is however, very unlikely – given the problem stated above – that it would be possible to use it without the heuristics also discussed.

7.4.2 Side effects of use case steps

While the view of having the system viewed as an environment – or global state – that is passed around and mutated, it does not solve the issue that sometimes use case steps will have side effects. Sometimes, however, they are necessary for the next steps in the use case. Consider the following use case steps:

1. Receptionist pickup inbound call

2. Receptionist parks inbound call
3. Receptionist locates an employee for dialing
4. Receptionist dials employee
5. Receptionist transfers the inbound call to the employee

In this example, we are dealing with two calls which we can map to a “Call” class in our support test tools. The first call can be provided to test via a precondition. The other, however introduces two problems. The first is that the call has no explicit reference to a call being created. The keyword “dials” may be considered an active action, but as actions are no longer part of the meta model, this cannot be used. “Employee” is a role that is mapped to actor, so, this leaves us with no information about that a “Call” object is being generated here. The other problem is derived from this. As we have no indication of the call being created, we will have means of tracking it either. So the last action – “Receptionist transfers the inbound call to the employee” – will not be able to be mapped without some way of sharing the created “Call” object between the two last steps.A solution to this, which is also the way it should be solved in the current implementation, is to share these created objects via global variables – which is very error-prone and impure (software design-wise).

7.4.3 Parallelism

A case that is not able to be covered by this approach, is when a use case contains multiple actors that perform the same action simultaneously.

We have a use case, where two Receptionist actor battles for the same call. In order to test this, we may wish to describe the scenario like so – assuming a call is has arrived and is ready for pickup:

1. Receptionist 1 tries to pickup call
2. Receptionist 2 tries to pickup call
3. Call is assigned to either Receptionist

A postcondition may read “The call is assigned to *only* one receptionist”, in order to emphasize on the actual intended behavior of the system, but the description above should be sufficient.

Studying the use case a bit closer, what is actually implied is that 1. happens simultaneously with 2. In practice, it would mean that a test would have to emulate the simultaneous behavior by spawning multiple threads and collecting their return values once they have terminated. But there is another problem with the scenario above, which is that the test tools do not know when to parallelize. As of now, every step in a use case scenario is modeled as a synchronous action and will wait until the step has completed its execution before starting the next one.

A method for solving this, is to add the asynchronism in the mapped test code, but this is a very bad idea. This would lead to very unexpected behavior, if requirements change in the specific block. This would lead to threads being spawned, expecting to perform a specific action that no longer existed in the requirements, perhaps deadlocking while waiting for an event to happen – or changing the state of the system that would lead an error later in the test.

A better way of solving it, is to add a keyword. For example, the word **simultaneously**. So, the use case would then read:

1. Receptionist 1 tries to pickup call
2. Receptionist 2 tries to pickup call **simultaneously**
3. Call is assigned to either Receptionist

Making the keyword **simultaneously** refer in step 2 refer to the previous step, 1, will signal to the test generator that a parallel step is to be performed. This feature is neither implemented, nor conceptualized further, but included in the discussion as it is an actual problem that was encountered during the development of the second iteration of the tests. There has been developed an *ad hoc* test that spawns threads and collects returned values, so there exists a technical solution for the problem.

This next sections presents the empiric evidence gathered with the purpose of determining the usefulness, additional workload factor of the approach in this thesis. We also try to determine the significance of the presence of a domain framework, integrated with the main code base. This is done by comparing failure levels for the first two iterations of the requirement translation tool (chapter 3).

7.5 Code sharing significance

This section compares the average failure (and success) levels of the two iterations of the test support tools. The data set that is collected are the average success/failure rates from our continuous integration service (Jenkins CI). The most significant difference between the two implementations is the fact that one of them shares code

Project	Avg. success rate	Avg. failure rate
First iteration	53.4%	46.6%
Second iteration	97.1%	2.9%

Table 7.1: Success/failure rates for the two implementations.

with the code base of the system under test, while the other does not. This will, thus, serve as the evaluation point. The two datasets have different number of total tests, which is why the evaluation will be done on the basis of the *relative* failure rate, instead of the absolute. Table 7.1 document the average success and failure rate of tests. The numbers clearly state that the failure rate of the second implementation is significantly lower than the first.

In the first implementation of the test tools (see section 3.1.2), there was no direct link to implementation code base. The tests were written in a different programming language than the main code base and the implementation of clients that used the API interfaces was manual and duplicated.

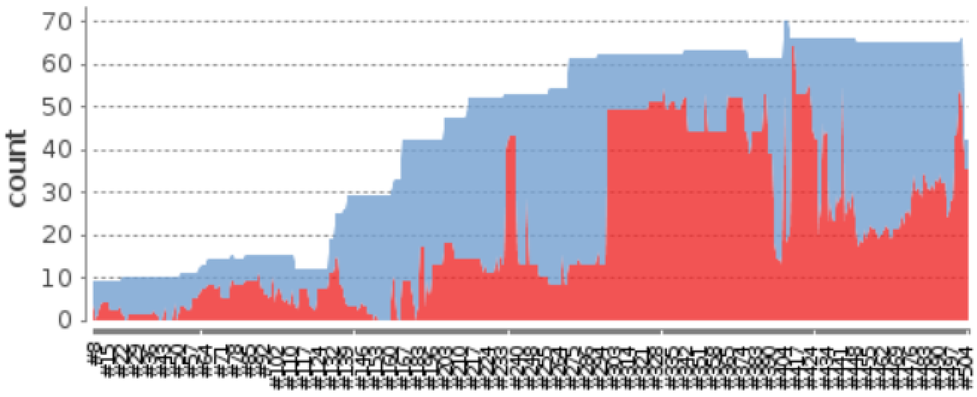


Figure 7.2: Iteration 1 Jenkins build trend. Blue area is total tests and red area is failures.

Iteration 1 build trend in figure 7.2[2] show an increasing relative number of errors compared to the second iteration shown in figure 7.3.

The second iteration treated tests as an intricate part of the main project, rather than an auxiliary part and made use of the domain framework that was built for the application. This, ultimately, led to a much lower failure rate, and the correction of errors much faster. First iteration has many longer-running errors, whereas in the second iteration, an error is typically fixed before new errors arise. Most of the errors are false negatives that simply occur from the fact that the tests are out of sync with the main code base.

By this, we conclude that the level of integration with the main code base matters. By sharing language and domain framework, the tests became an intricate part of the

[2]Image quality is low, but as the Jenkins server that ran the test has been *physically* garbage collected, it was the best that could be procured.

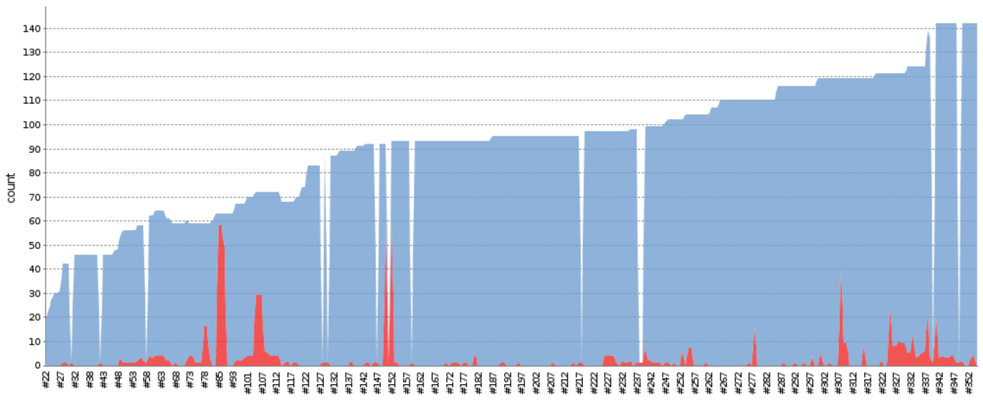


Figure 7.3: Iteration 2 Jenkins build trend. Blue area is total tests and red area is failures.

development, rather than an auxiliary part and the presence of a domain framework is concluded to be a critical part of the technique of this thesis.

7.6 Workload overhead

This section estimates and evaluates the workload associated with the technique presented in this thesis.

7.6.1 Test support tools

To be able to establish a quantitative measure for how much extra work is involved in writing the test support tools, the number of code lines was enumerated and support-only files were counted as overhead. Table 7.4 shows these metrics.

Project	Total (kLoc)	Test support (kLoc)	Overhead
Test framework	8.9	1.1	10%
PhonIO library	2	2	100%
Total	10.9	3.1	28%

Table 7.4: Test framework metrics.

The rest of the test framework code is the (manually written) tests, but the numbers gives an indication of how much extra work is put into building these tests. In table 7.4, we see that the overhead of test support tools in the test framework is 10%, not counting the PhonIO library, which is built for the specific purpose of supporting our use case tests, but is actually not designed to be used only in our system. If we, however, do include it, our overhead increases to 28%, which is a significant increase in the code base size.

In order to get an additional measure for the workload, the relative number of commits to the revision control system (RCS) was counted. The motivation for including an additional measure is, that lines of code may be a bad measure[Fra+13].

Project	Total (commits)	Test support (commits)	Overhead
Test framework	355	53	14.9%
PhonIO	68	68	100%
Total	10.9	3.1	28.60%

Table 7.5: Number of commits (as of 2nd of July, 2015).

Table 7.5 shows the number of code commits performed over the last 5 months. It also shows how many of these commits that involve the files that are support-only. Calculating the percentage overhead, we arrive at 14.9%. However, taking into account the PhonIO library, we again arrive around 28%. This is a lot higher than the overhead from a larger study of test-driven development methods[GW03]. In this study, they approximate that roughly 16% more time is spent on development when using test-driven development methods.

The metrics gathered in this section have given good indication of how much extra effort is involved in writing test support tools. 28% extra work is a substantial amount and too much for a wide adoption, as this will also mean an increase in man-hours spent,and a direct cost increase of the project.

The study of test-driven development methods mentioned above also showed that test-driven development approaches do produce higher quality software, so the extra time spent *may* be worthwhile for some applications that are not cost-driven, but require a higher quality. There are, however, no metrics that support that this technique produces higher quality software. The only thing that substantiates this claim, is the fact that feature (from a use case perspective) regressions may be automatically detected, and may be corrected quickly after.

7.6.2 Test mapping

The mapping functions are in a 1:1 relationship with the number of unique steps in the use cases. So, one step in a use case, means one mapping function.

If the interfaces and most of the domain models are provided to the tests via a domain framework, then the test mappings should be rather simple. The mapping functions in the use case tests of the case study system are, on average, 4-6 lines of code – including log statements.

We do, however, also have more complicated tests, which are not part of the use cases. As the message-sending architecture is defined to be a work-queue where the dispatcher is decoupled from the message sending, we need additional support code to check if a message is received. If the postcondition for one of our use cases had been; "Message is received by contact", then the function that provided this functionality would become quite complex. This is because our test support tools would need some way accessing an email mailbox. The single step of asserting that a message is actually received, would then result in hundreds of new lines of code.

The example above is an actual issue that we, in the development of the case study project, had to deal with. As we wanted to have our tests to be as close to a deployed scenario as possible, we defined a test (not use case) that sent a message to number of recipients and checked if the message was there. Other projects, applying the technique of this thesis, may face similar challenges with, for instance, physical hardware devices that need to be in certain state before or after the tests. Interfacing may not be provided and must then be implemented, in a harness, for the sole purpose of testing.

In general, the use tests we have in the case study system are limited to functionalities provided via the domain framework and not any external systems. This makes the case study system feasible for this technique, with regards to the test mapping function overhead. The 4-6 lines pr. use case step is not substantial – given that the use case are short and simple. On the other hand, if another system had many external systems that needed to be interfaced with, then the test mappings become increasingly large and time demanding.

7.7 Sources of error

This section contains a discussion of identified sources of error related to this chapter.

The case study system and test system has been developed by the same group of people, which have also played a large role in the use cases creation and refinement process. This is problematic when the general applicability of the approach presented in this thesis needs to be evaluated. Having the same people work on requirements, implementation and tests lead to a unified understanding of the system as a whole, and a very short path from requirement change, to implementation change. While this is generally a good thing, it does not scale to larger projects and important domain-specific details may unknowingly go undocumented, as everyone has a common understanding of how things work overall. The problem with this, is also that requirements may have an "artificial feel", in the way that they reflect the actual workings of the system under development, rather than the intended workings.

For this evaluation, it means that our requirements may be better suited for acceptance-test-writing than others, trying the same approach. It can, however, be argued, that as any (good) requirement should be testable[HJD10], then it shouldn't matter significantly how the project groups are structured.

CHAPTER 8

Conclusion

This chapter presents the findings of this thesis and evaluates the general feasibility based on them. The feasibility evaluation serves as an overall conclusion. In the next section a brief overview of the project work of this thesis.

During the development of the tool for this thesis, a number of conceptual designs were proposed in order to optimize the technique along with the tool. An analysis of how to model the different components was performed and a tool was implemented. The tool is in a proof-of-concept state as not all features are implemented. The discussion and the metrics served as good tools for evaluating feasibility.

8.1 Findings

This sections identifies and describes the overall findings of this thesis.

Code base sharing: As seen in section 7.5, a common programming language for tests and main code base is important. The implicit tracking of the main code base, by the tests, result in fewer false negatives. The same section also concludes that a domain framework, that the test support tools can use, to perform their test is also critical for the application of this technique.

Test support tools: The support tools that provide the code that exposes the system functionality required by use case steps, need to be present in order to be able to execute the generated tests.

Coverage issues The general find-all-paths method for generating tests from use case paths is not scalable. This makes the generation non-feasible without constraining the use cases.

Additional workload: The overhead is estimated to be around 28% (section 7.6) of the code size. We use this as estimated workload increase, which – in practice – may be higher because additional activities (an example is use case mapping) are involved in applying this technique.

Based on these findings, a conclusive feasibility is performed.

8.2 Feasibility

The feasibility of the technique and tool for broad development purposes cannot be substantiated by the work of this thesis. The mapping simply unfolds into a too large state space with a structure so complex that it is impossible for an end-user to specify it – and very hard for a trained professional to elaborate it. When applied to projects with simple use cases (such as the case study system), the test coverage generation can be similarly kept simple.

In conclusion: The goal of having our tool users write use cases – without changing the form of use cases, is not feasible in regards to automatic test generation without enforcing restrictions on use case writing. As this thesis focused on enforcing as little restrictions on use case writing as possible – the solution proposed here is faulty by design, as it contradicts itself.

While the tool implemented in the context of thesis may not be feasible as a general technique, the artifacts that it produced (the domain framework, and test support tools) have helped the case study system in two major ways: The use case driven development approach aided in getting the system stabilized and aided to focus on functionality, rather than features. The other aid was the test support tools that provided an excellent component-based structure for writing additional tests. The actor and concept classes, along with the service-oriented architecture made the test very readable and could even implicitly be read as requirements.

As a final note: The technique presented in this thesis is not really feasible as a requirements-to-tests approach, but may be refined to disregard test generation and work as a tests-as-requirements approach instead.

APPENDIX A

Glossary

This appendix contains a list of terms introduced or used in this thesis.

Domain framework: A hand-coded set of source code files that exposes from, and shares interfaces, and model classes, with the system under test and test support library.

Test support framework: A hand-coded set of source code files that provides interfacing between the sources of application under development, and the tests files that are generated. In this thesis, it builds upon an existing test library designed for unit testing. It provides the needed "setup", "teardown", grouping and tests runners that catch and report unhandled exceptions.

CRUD: Abbreviation of **C**reate, **R**ead **U**pdate and **D**elete is an acronym denoting the four primitives operation of persistent data storage. An interface for persistent storage may expose all or a subset of these primitives.

Requirement analysis: Requirement analysis, for the purpose of this project, meant as a concept that uses some the information stored in the requirements to check it for validity, ambiguity and provide alternative representations. An example of an alternative representation is that a set of use cases becomes a use a use case diagram, just on a different abstraction level.

System under development: The software system that is being developed, and serves as the system under test in the context of this thesis.

System under test: The software system that is being tested against. Equivalent to "system under development" in the context of this thesis.

APPENDIX B
Handbook

This appendix contains a handbook of the use-case writing and test generation tool implemented in the context of this thesis. The handbook presents the software through a number of screenshots and an explanation on how to write a use case – and how to generate a test for it. This handbook covers the client interfaces, but makes reference to the server. For more details, see chapter 6.

B.1 General usage

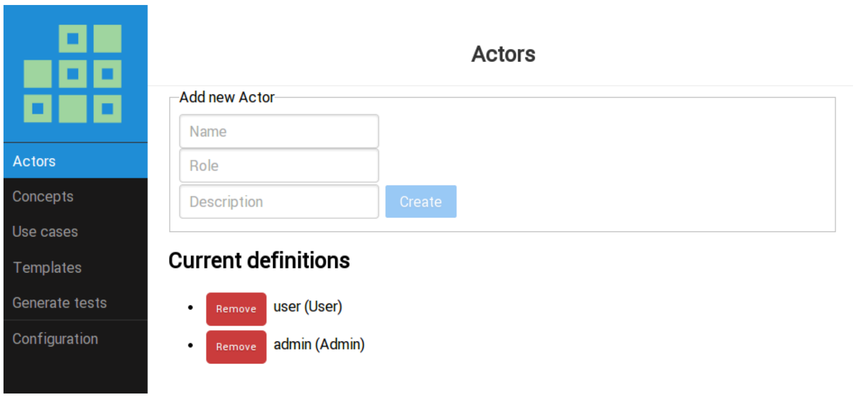
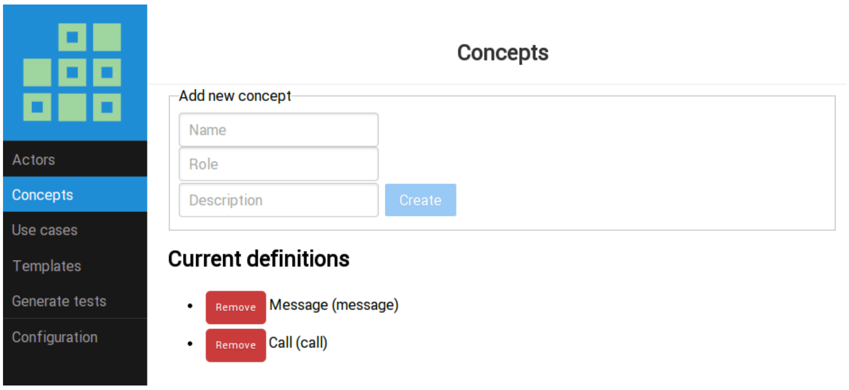


Figure B.1: The actors panel of the application.

Figure B.1 show the user interface, currently navigated to the "Actors" panel. The left side of the interface show the navigation options. The "Actors" panel has the possibility to a new actor definition. In order to do this, the actor must be named and given a role. The description is optional and the actor definition will be created, once the "Create" button is pressed. An existing definition may be removed from the list below by pressing the "Remove" button to the left of it.

The next navigation option is the "Concepts" panel. This panel is equivalent to the "Actors" panel in functionality, and we refer to this section for usage information. A screenshot of the "Concepts" panel is shown it figure B.2.

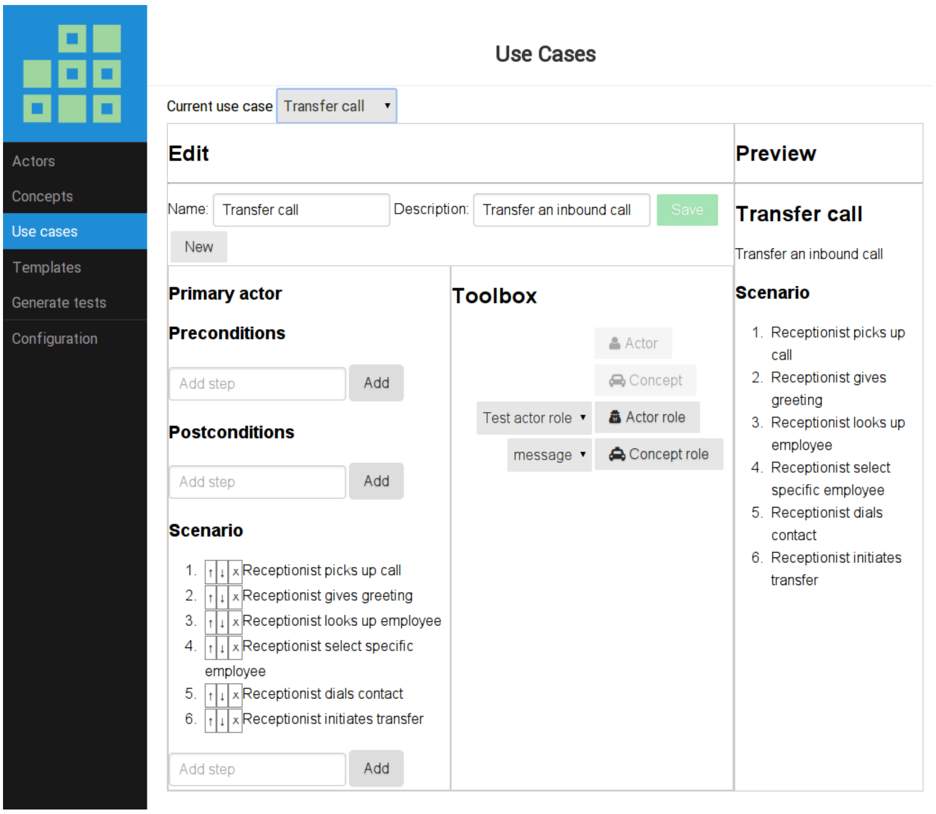
The "Use cases" panel (figure B.3) contains a number of different components. The topmost one is the "Current use case" selector, which is a drop-down that will change



Current definitions

- Remove Message (message)
- Remove Call (call)

Figure B.2: The concepts panel of the application.



1. Receptionist picks up call
2. Receptionist gives greeting
3. Receptionist looks up employee
4. Receptionist select specific employee
5. Receptionist dials contact
6. Receptionist initiates transfer

Transfer call

Transfer an inbound call

Scenario

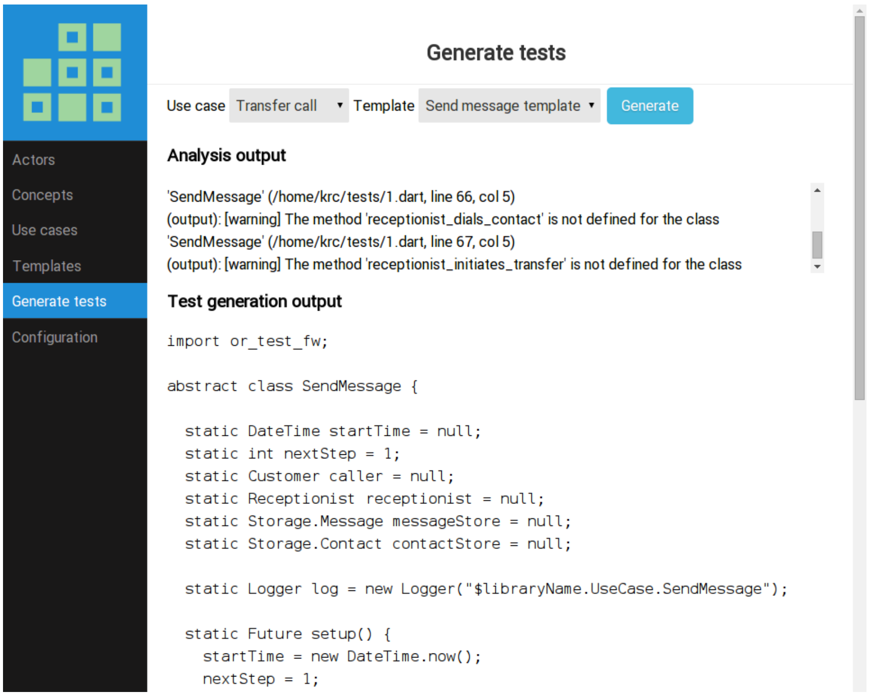
1. Receptionist picks up call
2. Receptionist gives greeting
3. Receptionist looks up employee
4. Receptionist select specific employee
5. Receptionist dials contact
6. Receptionist initiates transfer

Figure B.3: The use cases panel of the application.

the current use case. The remainder of the panel is split into an “Edit” region and a “Preview” region. The “Edit” region allows editing of a use case, such as name or description change or modification of the scenario, pre- and postconditions. Each step may be modified by the navigation buttons to the left of the list item. The up arrow moves the step up in the list, the down arrow moves it down. The “x” removes the step from the use case. New steps can be added via the associated input field and the add button.

The middle panel is the toolbox, its sole purpose is to be able to quickly mark selected text as either concepts or actors, or one of them in a specific role.

The rightmost panel is a review of the structure of the current use case, it updates on changes, so there is a live feedback on how the use case looks like – without the formatting tools.

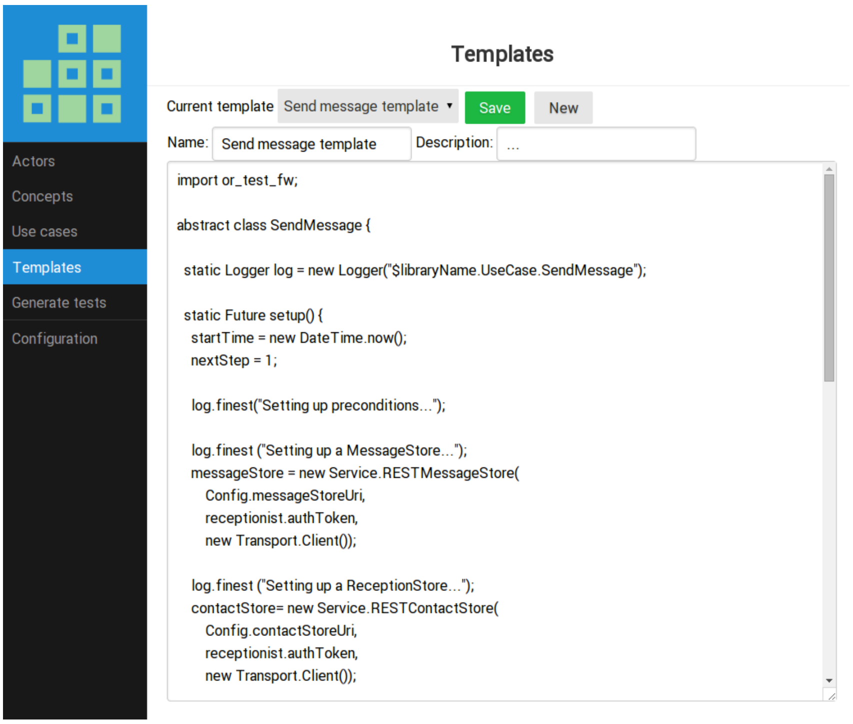


```
'SendMessage' (/home/krc/tests/1.dart, line 66, col 5)
(output): [warning] The method 'receptionist_dials_contact' is not defined for the class
'SendMessage' (/home/krc/tests/1.dart, line 67, col 5)
(output): [warning] The method 'receptionist_initiates_transfer' is not defined for the class
```

```
import or_test_fw;

abstract class SendMessage {

  static DateTime startTime = null;
  static int nextStep = 1;
  static Customer caller = null;
  static Receptionist receptionist = null;
  static Storage.Message messageStore = null;
  static Storage.Contact contactStore = null;

  static Logger log = new Logger("$libraryName.UseCase.SendMessage");

  static Future setup() {
    startTime = new DateTime.now();
    nextStep = 1;
```

Figure B.4: The test generation panel of the application.

The “Generate tests” panels (figure B.4) provides a tool for converting a specific use case to a test, using a template and the provided definitions (actors and concepts). When the panel is initially selected, only the use case, template selector and “Generate” button is visible. When a use case and template is selected and the “Generate” button is pressed, a request to generate the tests will be sent to server, which then will generate and analyze the generated test and return it along with the analysis output. This output will be displayed below the selectors and button and the generated test case right below the analysis output.



Current template: Send message template

Name: Send message template

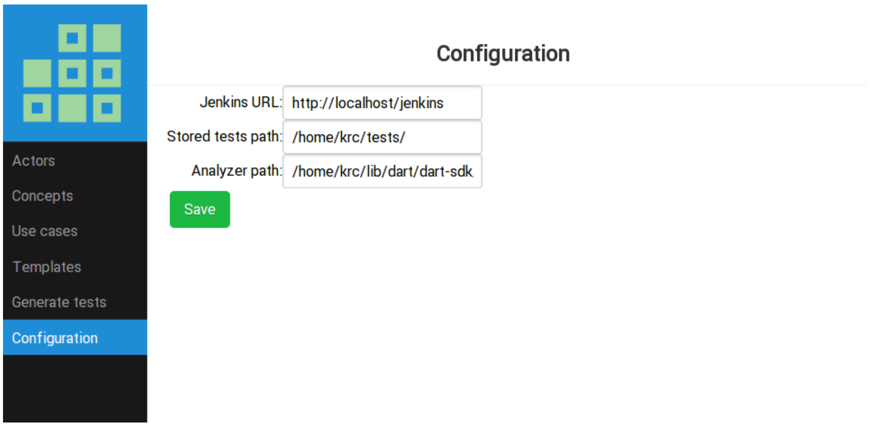
Description: ...

```
import or_test_fw;

abstract class SendMessage {

 static Logger log = new Logger("$libraryName.UseCase.SendMessage");

 static Future setup() {
  startTime = new DateTime.now();
  nextStep = 1;

  log.finest("Setting up preconditions...");

  log.finest ("Setting up a MessageStore...");
  messageStore = new Service.RESTMessageStore(
    Config.messageStoreUri,
    receptionist.authToken,
    new Transport.Client());

  log.finest ("Setting up a ReceptionStore...");
  contactStore= new Service.RESTContactStore(
    Config.contactStoreUri,
    receptionist.authToken,
    new Transport.Client());
```

Figure B.5: The test templates panel of the application.

Test templates can be modified in the “Templates” panel. The top of the panel contains a selector and buttons for saving the current template and creating a new – respectively. Name, description and the template itself may be edited using the input fields. The “Templates” panel is shown in figure B.5.

The last panel in the tool is the “Configuration” panel. It provides access to the server-side configuration, such as location of the binary used for analyzing the generated test code. It allows for changing the path where the server should store the generated test files. This location should be one where test support tools are also located. The final option is the URI location of the continuous integration service, which is currently unused. A screenshot of the “Configuration” panel is shown in figure B.6.



Configuration

Jenkins URL: http://localhost/jenkins

Stored tests path: /home/krc/tests/

Analyzer path: /home/krc/lib/dart/dart-sdk

Figure B.6: The configuration panel of the application.

B.2 Installation

Please refer to the README.md files in the source code directories the most up-to-date version of the installation procedure.

APPENDIX C

Additional figures

This appendix contains out-of-place figures that are no longer part of main document, but are still kept for completeness and reference.

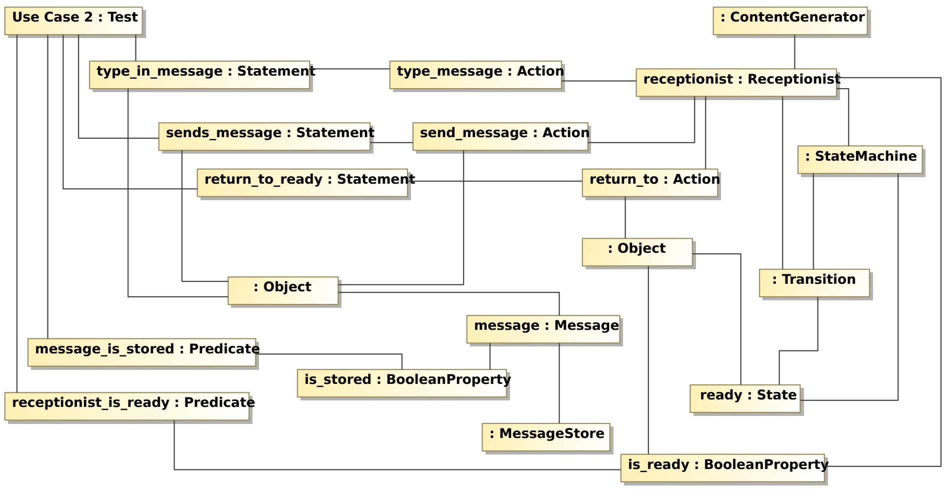

Use Case 2 : Test
: ContentGenerator
type_in_message : Statement
type_message : Action
receptionist : Receptionist
sends_message : Statement
send_message : Action
: StateMachine
return_to_ready : Statement
return_to : Action
: Object
: Transition
: Object
message : Message
message_is_stored : Predicate
is_stored : BooleanProperty
ready : State
receptionist_is_ready : Predicate
: MessageStore
is_ready : BooleanProperty

Figure C.1: Object diagram illustrating the use case as mapped test, using the meta model from concept 2 (see chapter 4).

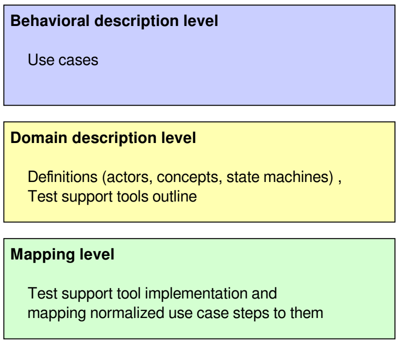

Behavioral description level
Use cases
Domain description level
Definitions (actors, concepts, state machines) ,
Test support tools outline
Mapping level
Test support tool implementation and
mapping normalized use case steps to them

Figure C.2: The different levels of the technique of the thesis.

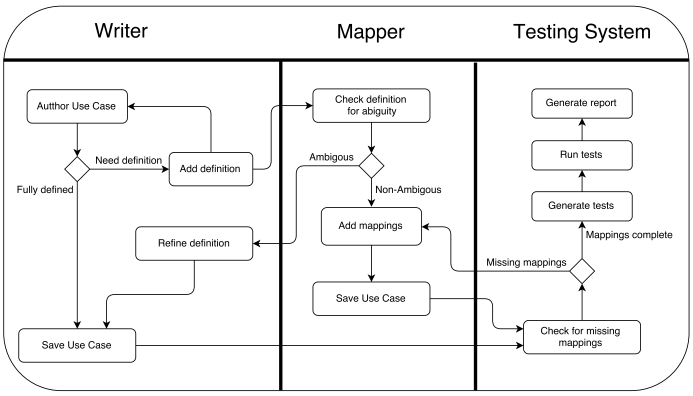


Figure C.3: Activities associated with the technique of the thesis.

APPENDIX D

Use cases

This section contains selected use cases from the OpenReception project, used within this thesis and as examples for the tool.

Use Case 1	**Transfer to contact**
Scope:	System-wide
Description:	The receptionist actor must be able to transfer an active call to a chosen – dialed – contact associated with the currently active reception
Level:	User-goal
Primary Actor:	Receptionist actor
Stakeholders and Interests:	• Receptionist: Wants to process the call with regards to the caller's wishes • Caller: Wants to reach a specific contact
Preconditions:	• Receptionist has picked up incoming call from caller • Receptionist has parked incoming call
Postconditions:	• Caller and contact's phones are connected • Receptionist is no longer in call and ready for next call
Main Success Scenario:	

1. Receptionist dials a number of the selected contact
2. The contact accepts the call (picks up)
3. Receptionist has a dialogue with contact
4. Receptionist transfers contact to caller
5. Receptionist marks his/her state as idle.

Extensions:

2.a Contact cannot be reached

1. Receptionist tries alternate contact
2. Receptionist returns to step 1

3.a Contact declines transfer

1. Receptionist hangs up contact
2. Receptionist picks up caller
3. Receptionist offers caller to leave a message

4a. Caller wishes to leave a message

   1. Receptionist types in message and caller information
   2. Receptionist saves message

4b. Caller does not wish to leave a message

5 Receptionist ends call with caller

6 Receptionist returns to step 6

Use Case 2	**Send Message to contact**
Scope:	System-wide
Description:	A receptionist must be able to send a message – via a distribution list – to a contact, typically containing information received verbally via a call. An example use case, from the receptionist actor point of view is outlined below.
Level:	User-goal

Primary Actor:	Receptionist actor
Preconditions:	• Receptionist have selected a contact who will serve as message recipient
Postconditions:	• Message is stored and ready for dispatching • Receptionist is idle
Main Success Scenario: 1. Receptionist types in message 2. Receptionist sends the message via the system 3. Receptionist marks his/her state as idle.	

APPENDIX E

Additional concepts

This section contains concepts that was worked on, during the preparation of this thesis, but did not find its way into the main application or design. The concepts are very rough, but left in for reference.

E.1 Event stack validation concept

Event stack validation is a concept that was coined in parallel with state concept in chapter 5. It is not exclusively related to use cases – but more validation in general. It is a stack-replay concept. It is neither implemented, nor designed further than this concept.

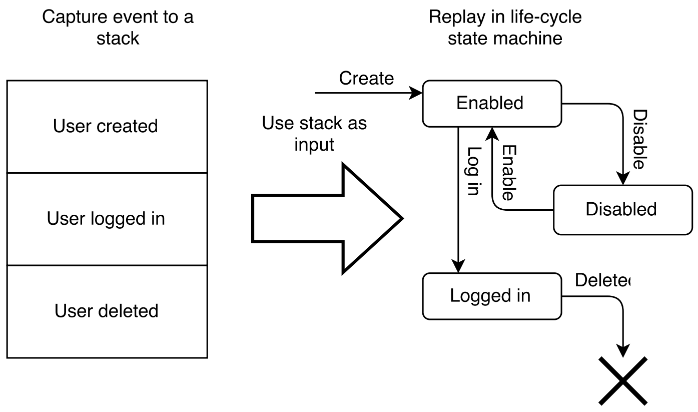


Figure E.1: Concept; validate event stack using life-cycle state machines..

Example; in the use case it is stated that a call is hung up and a callee awaits this event. The lifeline of the call is however not tracked and to be able to properly assert the true state of this, the code macro needs to into account this lifeline and reflect on which assertions hold for every stakeholder that has knowledge of the call.

If we are to define for the tests system what valid transitions are, by creating state machines, and then store object state transitions, we can assert that no objects within the system will break causality – at least in the situations elaborated in the use cases.

E.2 Further analysis of use cases

When we have our use case represented as a graph, we can perform these additional analysis':

Skipping actions may be prohibited: Should it be possible to jump ahead in the use case?

Primary actor must participate: The primary actor is important, as this is the stakeholder that defines the perspective and scope of the test. The primary is the person that starts the use case via an active action, or receives a start signal from another actor – the system for instance. The primary actor must also be part of the main scenario, and an analysis error should occur if this is not the case.

Additionally, if we assumed that we had a complete semantic model of a use case, as in a full linguistically analyzed sentence with subject, verb object *and* mappings to the implemented system, we could do additional checks for errors in the use case, via the tool. We would also be able to extract capabilities of an actor easily by just getting a list of verbs where the actor acted as subject.

APPENDIX F

Protocol specification

On the client interface side, we decided that the functions providing these interfaces should return a normal response on 2xx series HTTP codes. 4xx and 5xx would raise an exception. 1xx and 3xx series are unused in our stak, and thus, unmapped.

On the server side: Any interface will return 500 error codes on server errors, and 400 bad request upon bad client requests. The protocol is outlined below. Text within curly brackets – {} denote named request parameters.

`GET /actor`: Retrieve the list of actors currently defined.

`POST /actor`: Create a new actor. The actor object is passed in the request body.

`GET /actor/{id}`: Retrieve a single actor, identified by `id`.

`PUT /actor/{id}`: Update a single actor, identified by `id`. The actor object is passed in the request body.

`DELETE /actor/{id}`: Remove a single actor (un-define), identified by `id`.

`GET /concept`: Retrieve the list of concepts currently defined.

`POST /concept`: Create a new concept. The concept object is passed in the request body.

`GET /concept/{id}`: Retrieve a single concept, identified by `id`.

`PUT /concept/{id}`: Update a single concept, identified by `id`. The actor object is passed in the request body.

`DELETE /concept/{id}`: Remove a single concept (un-define), identified by `id`.

`GET /template`: Retrieve the list of templates currently stored.

`POST /template`: Create a new template. The template object is passed in the request body.

`GET /template/{id}`: Retrieve a single template, identified by `id`.

`PUT /template/{id}`: Update a single template, identified by `id`. The template object is passed in the request body.

DELETE /template/{id}: Remove a single template, identified by `id`.

GET /usecase: Retrieve the list of use cases currently stored.

POST /usecase: Create a new use case. The use case object is passed in the request body.

GET /usecase/{id}: Retrieve a single use case, identified by `id`.

PUT /template/{id}: Update a single use case, identified by `id`. The use case object is passed in the request body.

DELETE /usecase/{id}: Remove a single use case, identified by `id`.

POST /usecase/{id}/testsfromtemplate/tplid: Generate a ste of tests from the usecase identified by `id`, using the template identified by `tplid`.

GET /configuration: Retrieve the current run-time configuration of the system.

PUT /configuration: Update the current run-time configuration of the system. The new configuration object is passed in the request body.

APPENDIX G

Test results

Due to the young age of the tool, the tests only cover the database layer for now. The next iteration of tests would cover, in implementation order; the data models, the service layer (REST), user interface components, and unit tests of non-trivial components. Current test results are shown in table G.1.

Test	Expected value	Actual value
Actor create	OK	OK
Actor get	OK	OK
Actor remove	OK	OK
Actor list	OK	OK
Actor update	OK	OK
Concept create	OK	OK
Concept get	OK	OK
Concept remove	OK	OK
Concept list	OK	OK
Concept update	OK	OK
Config save	OK	OK
Config load	OK	OK
Template create	OK	OK
Template get	OK	OK
Template remove	OK	OK
Template list	OK	OK
Template update	OK	OK
UseCase create	OK	OK
UseCase get	OK	OK
UseCase removal	OK	OK
UseCase update	OK	OK

Table G.1: Test results for database tests.

APPENDIX H

Database schema

```
--------------------------------------------------------------------------------
--  System users: For later login system.

CREATE TABLE users (
   id                 INTEGER NOT NULL PRIMARY KEY,
   name               TEXT    NOT NULL
);

CREATE TABLE groups (
   id   INTEGER NOT NULL PRIMARY KEY,
   name TEXT    NOT NULL
);

CREATE TABLE user_groups (
   user_id  INTEGER NOT NULL REFERENCES users (id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
   group_id INTEGER NOT NULL REFERENCES groups (id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,

  PRIMARY KEY (user_id, group_id)
);

CREATE TABLE auth_identities (
   identity  TEXT    NOT NULL PRIMARY KEY,
   user_id   INTEGER NOT NULL REFERENCES users (id)
                     ON UPDATE CASCADE
                     ON DELETE CASCADE
);

-------------------------------------------------------------------------------
--  Domain concepts (also Actors).

CREATE TABLE concept_types (
   name TEXT NOT NULL PRIMARY KEY
);

CREATE TABLE concepts  (
   id          INTEGER NOT NULL PRIMARY KEY,
   name        TEXT    NOT NULL,
   role        TEXT    NOT NULL,
   type        TEXT    NOT NULL REFERENCES concept_types(name),
   description TEXT    NOT NULL DEFAULT '',
```

```sql
   UNIQUE (name, role)
);

-------------------------------------------------------------------------------
--  Use cases

CREATE TABLE use_cases (
   id                INTEGER NOT NULL PRIMARY KEY,
   name              TEXT    NOT NULL UNIQUE,
   primary_role_id   INTEGER REFERENCES concepts (id),
   scenario          JSON    NOT NULL DEFAULT '[]',
   extensions        JSON    NOT NULL DEFAULT '[]',
   preconditions     JSON    NOT NULL DEFAULT '[]',
   postconditions    JSON    NOT NULL DEFAULT '[]',
   description       TEXT    NOT NULL DEFAULT ''
);

CREATE TABLE configuration (
   client JSON NOT NULL
);

CREATE TABLE templates (
   id          INTEGER NOT NULL PRIMARY KEY,
   name        TEXT    NOT NULL,
   body        TEXT    NOT NULL,
   description TEXT    NOT NULL
);

-------------------------------------------------------------------------------
--  Create key sequences.

CREATE SEQUENCE users_id_sequence
  START WITH 1
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 1;
ALTER SEQUENCE users_id_sequence OWNED BY users.id;
ALTER TABLE ONLY users ALTER COLUMN id
  SET DEFAULT nextval ('users_id_sequence'::regclass);

CREATE SEQUENCE groups_id_sequence
  START WITH 1
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 1;
ALTER SEQUENCE groups_id_sequence OWNED BY groups.id;
ALTER TABLE ONLY groups ALTER COLUMN id
  SET DEFAULT nextval ('groups_id_sequence'::regclass);

CREATE SEQUENCE concepts_id_sequence
  START WITH 1
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 1;
ALTER SEQUENCE concepts_id_sequence OWNED BY concepts.id;
ALTER TABLE ONLY concepts ALTER COLUMN id
  SET DEFAULT nextval ('concepts_id_sequence'::regclass);

CREATE SEQUENCE templates_id_sequence
  START WITH 1
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 1;
ALTER SEQUENCE templates_id_sequence OWNED BY templates.id;
ALTER TABLE ONLY templates ALTER COLUMN id
  SET DEFAULT nextval ('templates_id_sequence'::regclass);

CREATE SEQUENCE use_cases_id_sequence
  START WITH 1
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 1;
ALTER SEQUENCE use_cases_id_sequence OWNED BY use_cases.id;
ALTER TABLE ONLY use_cases ALTER COLUMN id
  SET DEFAULT nextval ('use_cases_id_sequence'::regclass);

-----------------------------------------------------------------------------
--  Set ownership:

ALTER TABLE users OWNER TO tcc;
ALTER TABLE groups OWNER TO tcc;
ALTER TABLE concepts OWNER TO tcc;
ALTER TABLE templates OWNER TO tcc;
ALTER TABLE use_cases OWNER TO tcc;

ALTER SEQUENCE users_id_sequence OWNER TO tcc;
ALTER SEQUENCE groups_id_sequence OWNER TO tcc;
ALTER SEQUENCE concepts_id_sequence OWNER TO tcc;
ALTER SEQUENCE templates_id_sequence OWNER TO tcc;
ALTER SEQUENCE use_cases_id_sequence OWNER TO tcc;

-----------------------------------------------------------------------------
```

Listing H.1: Database schema for the test generation tool.

Bibliography

[Cha05] Robert N Charette. “Why software fails [software failure]”. In: *Spectrum, IEEE* 42.9 (2005), pages 42–49.

[CK92] Michael G Christel and Kyo C Kang. *Issues in requirements elicitation.* Technical report. DTIC Document, 1992.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases.* 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201702258.

[Fra+13] Gordon Fraser et al. “Does automated white-box test generation really help software testers?” In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis.* ACM. 2013, pages 291–301.

[GH99] Angelo Gargantini and Constance Heitmeyer. “Using model checking to generate tests from requirements specifications”. In: *Software Engineering—ESEC/FSE’99.* Springer. 1999, pages 146–162.

[GW03] Boby George and Laurie Williams. “An initial investigation of test driven development in industry”. In: *Proceedings of the 2003 ACM symposium on Applied computing.* ACM. 2003, pages 1135–1139.

[HJD10] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements engineering.* Springer Science & Business Media, 2010.

[KFM10] Christian Kop, Günther Fliedl, and Heinrich C Mayr. “From Natural Language Requirements to a Conceptual Model”. In: *International Workshop on Design, Evaluation and Refinement of Intelligent Systems (DERIS2010).* 2010, page 67.

[KS10] Radosław Klimek and Piotr Szwed. “Formal analysis of use case diagrams”. In: *Computer Science* 11 (2010), pages 115–131.

[Lar05] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3/e.* Pearson Education India, 2005.

[Rus08] John Rushby. “Automated test generation and verified software”. In: *Verified Software: Theories, Tools, Experiments.* Springer, 2008, pages 161–172.

[VSC08] June Verner, Jennifer Sampson, and Narciso Cerpa. “What factors lead to software project failure?” In: *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on.* IEEE. 2008, pages 71–80.