The impact of using a contract-driven, test-interceptor based software development approach

by
Arend Justus Posthuma

Thesis presented in fulfilment of the requirements for the degree of
Master of Philosophy in the Department of Information Science,
Centre for AI Research,
School for Data Science and Computational Thinking,
Stellenbosch University

Supervisor: Dr. Fritz Solms
Co-supervisor: Prof. Bruce Watson

December 2023

This research was conducted with and supported by the Centre for AI Research (CAIR) — Computational Thinking for AI

Contents

List of Figures

List of Tables

Abstract

Contract Driven Development, also known as Design by Contract (DBC), Contract Programming and Programming by Contract, is a well-known methodology for designing software. The main aim of the methodology is to reduce quality assurance costs, and to improve reusability and software quality through the use of formalized component contracts. Companies are spending large amounts of money and resources on quality assurance and testing in the pursuit of correct and bug-free software, yet contract driven development is not currently used extensively in industry. This is because the specification of formalized component requirements within component contracts is perceived to be complex, tedious and expensive.

In this study, we introduce the concept of test-interceptors, which are automatically generated from component contracts. The function of the test-interceptors is to validate whether, in the context of rendering component services, the component contracts are satisfied. These test interceptors can be used for unit testing, integration testing, operational testing and external service provider oversight. It is expected that such an approach improves verifiability, enforces separation of test logic and test data and assists with recuperating part of the requirements formalization costs through lower test development costs and lower costs associated with bug fixes.

This study aims to assess the impact of introducing contract-driven development to both, the quality attributes of the software development *process*, and the quality of the *software* produced by the process.

Glossary

Agile methodologies Methodologies based on the agile principles as laid out in the Agile Manifesto. Examples include incremental development, frequent testing and deployment, welcoming changes to the requirements at any stage, ... 32

boundary values values that lie on the edges or boundaries of input or output equivalence classes. These values are of particular interest because they are more likely to cause errors or trigger unexpected behavior in the software.. 63

compliance Conformance to standards applicable for the artifact.. 12

correctness Meets stakeholder requirements.. 13, 16

equivalence partitioning a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once.. 63, 76

quality software Software which meets the stakeholders' quality requirements for software.. 12

traceability The ability to trace activities or artifacts to other related artifacts. For example the ability to trace from requirements to design and tests, from design to code). Traceability may be uni- or bi-directional. In the case of bi-directional traceability one can trace both, forward and backward (e.g. from code back to design decisions back to requirements).. 33

validation An activity whose purpose it is to establish that all stakeholders have been identified, that the requirements are correct and that the component meets those requirements.. 11, 16

Acronyms

ATDD Acceptance Test Driven Development. 10

AWS Amazon Web Services. 78

BDD Behaviour Driven Development. 10

CDD Contract Driven Development. 10, 48, 56, 95, 98, 99

CORBA Common Object Request Broker Architecture. 2

COTS Commercial off-the-shelf components.. 33

EDD Example Driven Development. 10

IDL Interface Definition Language. 2

MDE Model-Driven Engineering. 32, 33

OCL Object Constraint Language. 15

OMA Object Management Architecture. 3

OMG Object Management Group. 2

ORB Object Request Broker. 3

QA Quality Assurance. 36, 51, 82

REST Representational State Transfer. 79

RMI Remote Invocation Interface. 3

STDD Story Test Driven Development. 10

TAO The ACE ORB. 3

TDD Test Driven Development. 10

URDAD Use-Case Responsibility Driven Analysis and Design. 31, 33, 34

Chapter 1

Introduction

1.1 The earliest example of CDD

In the early 1990s, the IT industry was faced with a major problem: integrating different systems using different technologies was very difficult. Factors like different operating systems, different programming languages, and different hardware made communication between software components difficult. Software developers ended up having to create custom protocols using sockets, which were not standardized and ended up being different for every project. The Object Management Group (OMG) attempted to solve this problem by creating a standardized protocol called CORBA (Common Object Request Broker Architecture) to allow software components on different hardware, written in different programming languages and used in different operating systems, to communicate with each other. This allowed developers to create heterogeneous distributed systems much easier, and CORBA became the bleeding edge of technology and popular. Due to the fact that CORBA components were very complex to create, this opened the door for vendors that specialized in creating these components, which could be re-used by developers and software companies so that they did not have to create their own. Many mission-critical systems were created using this technology, and vendors had to ensure that their components adhered to certain quality attributes. One of the key features of CORBA is its use of IDL (Interface Definition Language) to specify the interfaces of objects in a distributed system. IDL is a formal, machine-readable language that can be used to specify the inputs, outputs, and other properties of an object, such as its methods and data types.

CORBA's IDL is one of the earliest examples of contract-driven devel-

opment in that it allows developers to specify the behavior of objects in a formal, machine-verifiable way. By using IDL, developers can create more robust, reliable, and maintainable distributed systems by ensuring that objects behave as expected and by providing clear and precise documentation of their behavior.

Java leveraged off CORBA by building its Remote Invocation Interface (RMI) in version 1.1 on top of CORBA in February 1997. Early Java Enterprise Edition (Java-EE) application servers were also built on top of the CORBA technology but as direct cross language integration was increasingly replaced with XML and JSON based web services, the need for CORBA in the enterprise systems domains started to diminish. Furthermore, the CORBA Object Management Architecture (OMA) resembling a reference architecture for cross language application servers was never adopted by industry. The use of CORBA was increasingly confined to legacy systems and integrating real-time and embedded systems[16] where it is still actively continuing, for example The ACE ORB (TAO) which is a real-time Object Request Broker (ORB), designed to meet the demanding requirements of real-time and embedded systems. [17]. An ORB is a middleware that enables distributed applications to communicate with each other, allowing objects to interact across a network as if they were local objects.

1.2 Software quality management

Software is playing an increasing role in the electronic components market. From performing simple tasks like controlling lights in a home, to complex tasks like controlling the different components in a smart car like climate control, cruise control and collision detection to name but a few, embedded software is an integral part of electronic systems. And since the development and manufacturing of electronic components are subject to stringent quality assurance checks, embedded software has to be developed and provided with similar quality attributes as the hardware and its interface and specifications need to be included within the component catalog for the component. In order to produce quality software components, *good* software is needed. But a quality *process / methodology* is just as important in the production of quality components. This process is used to determine the actual quality requirements, designs, code, tests and component specifications and documentation.

Contract-driven development is a programming paradigm that requires

formalization of component requirements into testable contracts which specify the behavior of the component. By using contracts, developers can create more robust, reliable, and maintainable software components by ensuring that those components behave as expected and by providing clear and precise documentation of their behavior. For example, in embedded software for an electric car, contracts can be used to specify the behavior of the charging system, battery management system, and other components. By using contracts, developers can ensure that the charging system behaves as expected and that the battery management system is safe and reliable. Furthermore, contracts can be used to specify performance characteristics of the embedded software, such as its maximum response time or maximum number of requests per second. This can help to ensure that the embedded software can operate efficiently and effectively in the specific environment it is intended for.

1.3 Software testing and creating good software

In a world-wide survey focusing on the period between 2012 and 2019, 1660 senior executives in corporate IT management roles reported the proportion of their budget that was allocated to quality assurance and software testing. The average percentage steadily increased from 18% in 2012 to an estimated 39% in 2018 and 40% in 2019 [1]. These figures confirm that IT companies have realized the importance of, and are invested in, efforts to produce good software components.

Some commonly used quality requirements that help to produce "good" software are:

- Documented and published with a defined specification: This allows interested parties to quickly and easily determine what a component does and how it does it. It should also contain the context of where the component fits into the larger system and list other systems and components that it interacts with. It is very important that the specification of the component is clearly defined in order for someone who wants to reuse the component to understand what the component requires in terms of inputs to function correctly, and what outputs it produces. Documentation must also be updated when the code is updated.
- Single responsibility: The Single Responsibility Principle (SRP) in software states that every class, module or function in a program should

have only one responsibility or purpose. The main reasons for this principle are to enhance simplicity, maintainability and reusability.

- Meets requirements: The component satisfies all the requirements that were gathered during the requirement elicitation phase of the software development methodology. In order to determine if the requirements are satisfied, the component needs to be thoroughly tested. However, software testing does not show the absence of bugs, just their presence. The reason for this is that it is practically impossible to test all possible combinations of inputs and scenarios, so testing typically covers a subset of possible conditions. Another important factor is that human error like incomplete test coverage or mistakes in test case design can occur. While testing cannot guarantee the absence of all defects, thorough testing does help to identify a significant number of defects and by combining various techniques like unit testing, integration testing, system testing and user acceptance testing, the overall code quality is improved.

- Functional: The component is stable and performs its task without errors. This includes operational considerations like speed (the time it takes for the component to complete its task should be within an acceptable time-frame) and system resource usage (memory and processor usage should be within acceptable parameters and resources used should be released when the task is completed).

- Maintainability: The software should be designed and implemented in a way that makes it easy to maintain and modify over time. It should be well-structured, well-documented, and follow industry best practices

- Reusability: The component is written such that it is able to be reused in other parts of the system, or when building new systems.

- Is robust with regard to invalid usage: Should the component be used with invalid input values or not as intended, it should not fail catastrophically. Rather, it should fail gracefully with relevant error messages.

- Security: The software should be secure and protect against unauthorized access, data breaches, and other security threats. It should use appropriate encryption, authentication, and authorization mechanisms to protect sensitive data and user information.

- Scalability: The software should be able to handle increasing amounts of data, traffic, and users without significant performance degradation or downtime. It should be designed with scalability in mind and use scalable architectures and technologies.

- Compatibility: The software should be compatible with other systems, devices, and software products, without causing conflicts or compatibility issues. It should be designed with interoperability in mind and use open standards and protocols.

- Testability: The software should be designed and implemented in a way that makes it easy to test and verify its correctness and reliability. It should have well-defined interfaces and use automated testing tools and frameworks.

These requirements are based on industry best practices and standards, including ISO/IEC 9126-1, IEEE Std. 1061, and the ISO/IEC 25010 standard for software quality. [20]

In order to establish the quality of produced software components, one performs both, **verification** and **validation**.

Verification aims to establish that:

- the component meets the specified requirements, and

- the code is "good code" in the sense that it meets the required coding standards (including being free from typical coding errors).

Validation aims to establish that:

- all stakeholders who have requirements, have been identified;

- the requirements themselves are correct (complete, consistent, minimal);

- the component meets its functional and non-functional requirements.

Verification is commonly performed through code and artifact analysis, whilst software validation is commonly performed through functional and non-functional testing. Both these activities can be very resource intensive and hence expensive. In order to reduce cost and make the processes repeatable, both verification and validation are automated as much as possible, and their execution is commonly bound into the build cycle.

For software verification, one commonly uses manual and automated analysis. Manual analyses normally consist of peer and formal artifact and code reviews, and automated analyses are commonly performed through model/specification and code analyzers. These activities verify that the artifacts comply to certain standards and are free of specific undesirable elements. Automated testing is used to automate aspects of software validation. This may include both, functional and non-functional testing.

Non-functional testing involves testing non-functional aspects of an application. These aspects include:

- Performance: How well a software component meets its requirements in terms of response time or throughput of single requests.
- Scalability: The ability of a software component to perform under load (number of requests processed per unit of time).
- Elasticity: The ability of the software component to scale up and down system resources depending on the load, like processing power, memory, storage and communication resources.
- Robustness: The ability of the software component to function and recover from errors (like invalid inputs) or stressful conditions.
- Security: The vulnerability of the software against malicious attacks.
- Usability: The quality of the user experience when interacting with the software.
- Modifiability: The cost associated with requirements or customization changes.

However, not all aspects are commonly automated. For example, usability testing generally includes user testing above automated aspects like user workflow complexity evaluation or screen complexity evaluation.

Functional testing uses either black box testing in which the tests are purely derived from the requirements, or white box testing in which someone with knowledge of the implementation is able to create more comprehensive and detailed tests. Types of functional tests include:

- Unit testing: Testing individual parts of a system in isolation, ensuring that inputs result in the desired outputs.
- Integration testing: Ensuring that individual modules of code work together properly as a group. This is often done in concert with unit testing.

- Regression testing: When developers update code, regression testing is used to ensure that the component still functions as expected and that the updates did not introduce any errors.
- System testing (also referred to as end-to-end testing): The whole system is tested in its entirety. Testers can also provide feedback on the performance and usability of the system.

Functional testing is commonly automated and executed within the component build process, whilst test logic and test data are often specified manually. Only the test execution is automated. Furthermore, tests need to be verified to cover the requirements and the different execution paths.

Manual testing has a number of disadvantages [46, 40, 21, 31, 13]:

- It requires more time and resources. Manual test cases have to be defined and these need to be executed every time the code changes. This can be a barrier to agile development and can slow down the release cycle.
- Manual testing is performed by human testers, which can introduce errors and inconsistencies in the testing process. This can lead to false positives or false negatives in test results.
- Performance testing cannot be done manually. It is not practical to organize hundreds of machines and testers to determine if a software component is able to handle the load.
- Manual testing is difficult to scale, as it requires additional resources and time to perform additional tests. This can be a challenge for organizations that need to rapidly scale their testing efforts in response to changing business needs.
- It can be expensive, as it requires dedicated resources and a significant amount of time to perform. This can make it difficult for organizations to justify the cost of manual testing.

Automated testing, on the other hand, addresses all the disadvantages of manual testing:

- Much less time and resources are required. The tests are written once, and can then be executed as many times as is needed. Test cases themselves can also be automatically generated from contracts, which is discussed in detail in Chapter 2 and 5

- Tests are much more accurate because once the logic is correct, the testing tool will execute the exact same logic every time the test is run.
- It is possible to do performance testing since multiple instances of the testing tool can be executed (for example by using threads or virtual machines).
- Software change requests can be done much faster since running the automated tests will quickly show if an error was introduced by the update without having to wait for a tester or quality assurance person.
- The cost is lower. Although the initial setup of an automated testing framework can be expensive, once the tests are created, they can be run multiple times without additional cost. This makes automated testing a more cost-effective option in the long run, particularly for large or complex software projects.

Additionally, automated testing has the following advantages:

- Automated testing is normally part of the build process of the application - every time changes are made, and the application is built, the tests are executed, and the developer will immediately see if his updates to the code have introduced an error.
- Test scripts can run unattended, freeing up resources.
- Automated testing can cover a wider range of scenarios and edge cases, which can help increase test coverage and identify defects that might be missed with manual testing. Test automation tools can execute tests quickly and consistently, and can be used to cover large volumes of tests, which can improve the overall quality of the software product.

Irrespective of the type of testing performed, testing can only be meaningful if we have testable requirements. In order to make requirements testable, they need to be formalized and quantified. The formalization of requirements involves expressing the requirements using a formal requirements specification language whose vocabulary, syntax and semantics are based on mathematics. This allows statements to be evaluated by software tools to prove that the program conforms to its specifications. The quantification of requirements involves categorizing the requirements and defining common attributes for similar requirements. These attributes need to be measurable and unambiguous.

In both functional and non-functional testing, requirements and tests need to be refined across levels of granularity. Clean layers of granularity

is an important principle of good software architecture design: each layer must adhere to the *dependency inversion principle*, meaning components in a lower level of granularity should not have any dependency on higher-level components. This allows one to understand a higher level workflow without having to understand the finer details of the implementation [37]. Grouping requirements and tests by level of granularity has the following benefits:

- It improves understandability since one is able to focus on specific functionality within the system. It is easy to get overwhelmed with all the requirements and tests of large, complex systems or components.
- Reusability is improved when a component at some level of granularity, along with its formalized requirements and tests, can be re-used within another system.
- Maintainability of the component is improved when changes and updates are necessary, and the updates to the requirements and tests can be localized to the component at the specific level of granularity.

1.4 Modern software development methodologies

There are a number of modern software development methodologies that are used by companies in their efforts to produce quality software [8, 3]. The two most popular methodologies today are:

- Test Driven Development (TDD) which requires software requirements to be converted into test cases before development starts, and then testing the developed software repeatedly against these test cases as it is being developed. TDD is primarily used to test units of code (like methods and classes) and aims to ensure a set of operations is performed correctly.
- Another popular methodology is Behavior Driven Development (BDD) which is similar to TDD in that it requires tests to be written first, but in the case of BDD, the tests describe software behavior by using a domain specific language (DSL). The DSL uses natural-language constructs that describe the required behavior and expected outcomes, and is converted into executable tests by specialized software. BDD is also sometimes referred to as Story Test Driven Development (STDD), Acceptance Test Driven Development (ATDD) or Example Driven Development (EDD). [2]

The main disadvantage of TDD and BDD is that it requires a great amount of extra effort from developers to write tests. In the case of BDD, developers additionally have to learn and use the DSL and specialized software, resulting in extra time and costs on a project. Another major disadvantage is that the tests need to be updated and maintained when the code is updated. But... is there a better methodology to create quality software that does not have these disadvantages?

1.5 Using contracts in software development

Contract Driven Development (CDD) is a development methodology that aims to produce correct software through formalized requirements in the form of component contracts which include the specification of pre-conditions, post-conditions and invariants [29]. The idea of using contracts to specify the behavior of software components can be traced back to the early days of computer science and programming.

In the 1960s and 1970s, researchers in the field of formal methods began to explore the use of formal, mathematical languages to specify the behavior of software systems. This work laid the foundation for the development of programming languages and tools that could be used to write contracts for software components.

The 1980s and 1990s saw the fields of programming languages and software engineering developing new languages and tools that could be used to write contracts for software components. This work included the development of new languages such as Eiffel and JML, as well as new tools and frameworks for specifying and verifying contracts.

In the 2000s and 2010s, contract-driven development gained popularity as more and more developers started to realize the benefits of using contracts to specify the behavior of software components. Their objective was to improve reliability, increase maintainability, improve documentation, and make testing easier.

The benefits of Contract Driven Development have been demonstrated, most notably through *The Lessons of Ariane:* After investigating the unfortunate and very costly crash of the European Ariane 5 rocket 40 seconds after launch, an international inquiry board discovered that this crash was the result of a software error that occurred when a 64-bit floating point value was converted into a 16-bit signed integer. This caused an uncaught exception that crashed the entire on-board computer system.

Jezequel and Meyer argues that it is not a management problem that caused such an error to remain undetected, since systematic documentation,

validation and management procedures were all in place. Neither is the programming language at fault, since the exception mechanism of Ada could have been used to catch the exception. The design is not at fault since the same code that was used in the Ariane 4 was reused, however the Ariane 5 had a different trajectory which caused the overflow to occur. They further argued that the implementation is not at fault because the engineers had a target to keep the workload of the computer at 80 percent and checking for all possible and impossible events would have exceeded that target. And finally they argue that testing cannot be blamed since, although you could test more, you can never test all and in fact the launch *was* a test launch.

Although a 10-year-old component was reused from the Ariane 4, Jezequel and Meyer concludes that the core of the problem is the absence of a precise specification associated with the reuse of the component: it should have stated explicitly that the value passed to this component must fit in 16 bits.
They emphasize the importance of Design by Contract in the construction of reliable software and stress that modules of a system should be governed by precise specifications, similar to contracts and that this approach is imperative for effective component reuse [23].

However, the CDD methodology has never really gained significant traction because of the following reasons [45, 42, 43, 22, 44, 19]

- Developer skills requirements and development process changes: CDD can add complexity to the development process, as it requires skills to create and maintain contracts in addition to the code itself. CDD may be less common in certain industries, making it difficult to find experienced developers who are familiar with it. Some developers may not fully understand the benefits of CDD and may therefore be resistant to using it.

- Additional effort and cost: CDD adds overhead to the development process, as it requires additional time and resources to create and maintain contracts. It requires more upfront work, as the contracts need to be written before the implementation. Furthermore, CDD may increase the cost of the project because of the extra effort required for creating and maintaining the contracts. Additionally, CDD may make it harder to debug the software because the contracts need to be checked and validated. Finally, CDD may limit flexibility because contracts need to be followed strictly. This means that should any changes occur, code and contracts and tests need to be updated synchronously for the benefit of having testability and reusability.

- Lack of tool support: CDD may be difficult to implement in certain languages or frameworks due to a lack of tool support.

1.6 The research problem

Due to these reasons, there is a perception that contracts do not deliver "enough bang for your buck" [36, 41]. But would Contract Driven Development contribute significantly to produce quality software if the complexity and tediousness of implementing contracts can be mitigated? Would the quality of the software produced by using a contract-driven methodology justify the effort and cost of utilizing contracts?

In this study, we attempt to answer these questions. We investigate whether the complexity and tediousness of implementing contracts in software can be mitigated by generating test interceptors from component contracts. The automatically generated test interceptors are component wrappers which, in the context of service provision, monitor contract compliance of the wrapped component. Empirical data is obtained from actual software development projects by using a Contract Driven Development methodology [38, 37] to specify component contracts. We evaluate a software product that was developed for a client, and we compare the results against a previous product of similar scope and size.

In order for a study to be performed on real software development projects, it is necessary to convince not only management, but also team members. Real software development projects have real clients, time-lines and budgets. Anything that might influence client relationships, deadlines and especially costs, make most managers uncomfortable. Many team members are also uncomfortable to change their work-flow, especially if it is a stressful environment with tight deadlines and complex projects. These factors make managers and team members negative and unwilling to participate in a study. The strategy that we used to convince managers and team members to participate in the study has been very successful and is discussed in detail in Chapter 7.

The study empirically assesses the impact on both, quality attributes of the software development process and quality attributes of the software produced by the process. Process qualities include *development productivity* (the rate/cost at which software is produced), *process certifiability* (the impact of contract-driven development on the ability to certify a software development process), *development reliability* (the ability of the process to reliably produce software at a given rate and quality), *process usability* (the ease with

which contract driven development can be introduced) and *process scalability* (the ability of introducing contract-driven development to large development teams).

This study also assesses the impact of contract-driven development on selected quality attributes of the produced software. In particular, the impact on correctness (the software meets the requirements), re-usability (whether contract-driven development leads to components which are more reusable), and simplicity are considered. For each of the process and software qualities, measures are identified, and the measurement process is specified.

Based on the empirical results of the study, it will be clear what impact a contract driven, test-interceptor based development approach has on the complexity of adding contracts to software components, if the extra work is worth the effort, if the approach delivers enough "bang for your buck" and ultimately if the approach produces correct, high-quality software.

Chapter 2 discusses related work and why it is not sufficient, as well as highlights the differences between what has been done and what we have done in this study. Chapter 3 discusses the problem in detail and also the specification for a solution and how the solution will be evaluated. In Chapter 4 we discuss the details of the methodology we use for the empirical study, based on the requirements from Chapter 3. Chapter 6 covers how our methodology was used in a real software development project, and we discuss the results that were obtained. Chapter 7 summarizes the solution and discusses future work.

Chapter 2

Related work

Contract Driven Development is not a new concept. In 1992, Bertrand Meyer wrote an influential article about design by contract and discussed the notion of contracts and pre-conditions, post-conditions and invariants in code [29]. These ideas were subsequently embraced by developers and academics and many development frameworks, tools and methodologies were created which promote the use of contracts in code.

Unfortunately, these frameworks, tools and methodologies never became popular because of their complexity and extra efforts required to implement contracts. But, we believe that modern programming languages make it possible to overcome these obstacles. The focus of this chapter is to look at current and previous tools, methodologies and frameworks for Contract Driven Development and discuss their advantages and disadvantages compared to our approach.

2.1 Requirements by Contracts Allow Automated System Testing

Nebut et al. describe a requirement-based testing technique that leverage use-cases in order to generate functional and robustness test objectives [30]. Requirements are specified as UML use cases, which are annotated with contract specifications attached as notes. The contracts are specified using a custom language to construct predicates for pre- and post-conditions from state assertions (e.g. `open(door-1)`) and standard logical operators. In addition, they support two use case relationships, *includes* and *generalization* which they effectively use for requirements aggregation and refinement. In order to facilitate automated test generation, Nebut et al. narrow down the

differentiation between use cases and services by supporting parametrized use cases and assuming that there is a mapping of use cases onto system services.

Nebut et al. then use pre- and post-conditions of use cases to construct a graph of all possible transitions between use cases. Any system process can then be related to a path through the graph. Test processes (called test objectives) are constructed as specific finite paths through the Use Case Transition System (UCTS) graph to satisfy certain test criteria. Nebut et al. found that selecting a set of test paths (test objectives) which ensure that each use case is instantiated by at least one path and each scenario for which all pre-conditions for a use case are satisfied is included on one of the test paths.

Finally, test scenarios are constructed by setting the system into specific states and providing particular user inputs, i.e. each test scenario is a combination of system state and system input.

Strengths of the approach is the proposed framework include that both, test processes (test functions) and test scenarios (test data) are generated and that the tests are generated from a semiformal requirements specification without taking design considerations into account. But even though the proposed framework provides automated requirements-based functional and robustness testing from contract-annotated requirements models, it has not been widely adopted for automated requirements testing. Possible reasons for this failure include

- **Scalability:** The framework was demonstrated in a very simple academic project. The approach does not test across levels of granularity. Instead, the entire requirements specification for the system is flattened into a single UCTS graph across which finite test processes are found using a breadth-first search algorithm. For industrial systems, this approach is unlikely to be feasible as the search space will explode with increasing system size and complexity. The approach investigated in this dissertation utilizing CDD and interceptors, aims to manage scalability through automated contract-based test function generation for services across levels of granularity.

- **Completeness:** Nebut et al. propose the use of a custom language for the specification of pre- and post-conditions. These are attached via notes to UML Model elements. The expressiveness of the language is very limited and does not support the specification of non-trivial constraints. For specifying constraints against an object graph, UML has included the *Object Constraint Language* (OCL) as part of the standardization of the UML. The OCL enables one to specify UML model

constraints using first-order predicate logic with quantifiers on finite domains. Furthermore, although the ability to generate test scenarios is one of the strengths of this approach, Nebut et al. have shown the generated test scenarios to be incomplete. If testing completeness is required (e.g. for certifiable projects) a manual process needs to identify scenarios which need to be added to achieve completeness.

- **Usability:** The use of a custom language for constraint specification requires new tools for validating constraint syntax and correctness. These tools are not available. Furthermore, the proposed approach is only usable in the context of model-driven engineering. Only a very small fraction of industrial projects is based on model-driven engineering. The CDD approach that utilizes interceptors analyzed in this dissertation uses code annotations of contract specifications in the programming language (e.g. against Java interfaces) using the programming language itself to specify the pre-condition, post-condition and invariant constraint predicates. Users thus need not learn a new language, and the compiler tools can themselves be used to verify the constraint specification. Furthermore, the approach can be used for non-model-driven engineering projects.

2.2 Consumer-Driven Contracts

Evolving service providers and consumers present a number of challenges, particularly issues that arise when service providers change their contracts or *document schemas*. Robinson puts forth two strategies for mitigating these issues: adding schema extension points and performing "just enough" validation of received messages [34]. He notes that both of these strategies help to protect consumers when the provider changes the contract, but they are of little help to the provider to know how the contract is being used and what obligations it must maintain as it evolves. Robinson continues to discuss the "Consumer-Driven Contract" pattern which addresses this shortcoming by drawing on the assertion-based language of the "just enough" validation strategy, which imbues the provider with insight into its obligations towards consumers.

To illustrate these points, consider as an example, a Product Search service provider which allows consumers to browse a product catalog. A typical search response is an XML message consisting of product entries. Each entry has the following fields: CatalogueID, Name, Price, Manufacturer and

InStock. Currently, the service is used by two consumers, both of which uses XSD validation to validate messages before processing them. One consumer uses the CatalogueID, Name, Price and Manufacturer fields. The other consumer uses the CatalogueID, Name and Price fields. Neither of them uses the InStock field.
When another consumer appears and asks for a Description field to be added, there are significant and costly implications for the provider and existing consumers due to the way in which they have been built. However, depending on the way in which the change is implemented, the cost can be distributed between the affected components. Firstly, the original schema could be modified and each consumer required to update its copy of the schema in order to correctly validate the search response. In this case, consumers who have no interest in the new functionality have to be updated. Alternatively, a second operation, as well as a new updated schema could be implemented in the service provider to service the new consumer, and the original operation and schema are still maintained to service the existing consumers. This way, the changes and related costs are limited to the provider, but at the expense of making the provider more complex and costly to maintain.

Robinson begins his investigation into the issues with changing contracts by looking into schema versioning. The W3C Technical Architecture Group (TAG) describes a number of versioning strategies, which range from very liberal *none* (services do not distinguish between different versions of a schema and must tolerate all changes) to very conservative *big bang* (services must terminate if they receive an unexpected version of a schema). The problem with "no versioning" strategies is that the resulting systems are unpredictable in their interactions, fragile and costly to maintain. "Big Bang" strategies, on the other hand, result in systems that are tightly coupled, and schema changes cause a domino-effect through providers and consumers resulting in downtime, retarding evolution and impacting revenue generating opportunities. A flexible versioning strategy is needed - what the TAG calls a *compatible* strategy.
A compatible versioning strategy provides for backwards- and forwards-compatible schemas. Backwards-compatible schemas enable consumers of newer schemas to still accept instances of older schemas, while forwards-compatible schemas enable consumers of older schemas to process instances of newer schemas.
Making schemas backwards- and forwards-compatible can be achieved by using the *Must Ignore* pattern, which recommends that schemas incorporate extensibility points to allow new elements and attributes to be added, and consumers to ignore elements that they do not recognize. This flexibility comes at the expense of increased complexity, since extensible schemas allow

for unforeseen changes that might never occur. This results in obscuring the expressive power of a simple design, and introducing meta-informational container elements into the domain language. And although extensibility does allow for backwards- and forwards compatible schemas, it does not help with "breaking" changes to a contract.

Robinson discusses the removal of the InStock field as a breaking change to the contract: Although none of the existing consumers actually use it, they will break if the field is removed. This is due to the use of XSD validation as well as static language bindings which are derived from the document schema - consumers implicitly accept the whole provider contract even though they do not use all the component parts, thereby implementing a form of "hidden" coupling. Robinson augments the Internet Protocol's Robustness Principle in the context of service evolution by stating that message receivers should only perform targeted validation of the data they receive which is directly related to the business functions they implement, as opposed to an "all-or-nothing" approach.

By observing contracts between providers and consumers, Robinson expresses the following insights:
A provider contract expresses a service provider's business function capabilities in terms of the set of exportable elements necessary to support that functionality. Provider contracts have the following characteristics:

- **Closed and complete** This type of provider contract expresses the business function capabilities of a service as a complete set of exportable elements available to consumers. It is closed and complete from the perspective of the available functionality of the system.
- **Singular and authoritative** This provider contract is singular and authoritative in its expression of the business functionality available to the system.
- **Bounded Stability and immutability** This provider contract is stable and immutable for a bounded period and locale. It typically uses a form of versioning to differentiate differently bounded instances of a contract.

Consumer contracts, on the other hand, are entered into when a provider accepts and adopts the reasonable expectations expressed by a consumer. Such a contract has the following characteristics:

- **Open and incomplete** A consumer contract is open and incomplete with respect to the business functionality available to the system. It expresses a subset of the system's business function capabilities in terms of the consumer's expectations of the provider contract.

- **Multiple and non-authoritative** A consumer contract is multiple in proportion to the number of consumers of a service, and each is non-authoritative with regard to the total set of contractual obligations placed on the provider.

- **Bounded stability and immutability** Similar to a provider contract, a consumer contract is valid for a particular period of time and/or location.

Robinson concludes that consumer contracts, by expressing and asserting expectations of a provider contract, allow us to know exactly which parts of the provider contract currently support business value by the system, and which parts do not. This leads to the suggestion that services might benefit from being specified in terms of consumer contracts from the start. In this way, provider contracts emerge to meet consumer expectations and demands - *consumer-driven contracts* or *derived contracts*.
Consumer-driven contracts have the following characteristics:

- **Closed and complete** A consumer-driven contract is closed and complete with respect to the entire set of functionality demanded of it by its existing consumers. The contract represents the mandatory set of exportable elements required to support consumer expectations during the period in which those expectations remain valid for their parent applications.

- **Singular and non-authoritative** Provider contracts are singular in their expression of the business functionality available to the system, but non-authoritative because derived from the union of existing consumer expectations.

- **Bounded stability and immutability** A consumer-driven contract is stable and immutable regarding a particular set of consumer contracts. This implies that we can determine the validity of a consumer-driven contract according to a specified set of consumer contracts, effectively bounding the forwards- and backwards-compatible nature of the contract in time and space. The compatibility of a contract remains stable and immutable for a particular set of consumer contracts and expectations, but as expectations change, the contract will also change.

Robinson identifies two significant benefits of consumer-driven contracts in terms of evolving services. Firstly, the focus is on the specification and delivery of functionality around key business value drivers - a service is of value only to the extent it is consumed. And secondly, consumer-driven contracts provide the fine-grained insight and rapid feedback needed to plan changes and assess their impact on applications currently in production. However, there are certain liabilities: the consumer-driven contract pattern is applicable in the context of a single enterprise or closed community - an environment in which providers can exert some influence over how consumers establish contracts with them. Providers and consumers must know about, accept and adopt an agreed upon set of channels and conventions. This adds a layer of complexity and protocol dependence on an already complex service infrastructure.
Although the pattern allows for better management of breaking changes to contracts, it is not a cure. At best, it provides better insight into what constitutes a breaking change, and as such may serve as the foundation of a versioning strategy.
Consumer-driven contracts do not necessarily reduce the coupling between services, but it does identify "hidden" couplings, which allow providers and consumers to better manage them. Finally, there is a risk that when consumer contracts drive the specification of a service provider, the conceptual integrity of that provider could be undermined - services are discrete, identifiable, reusable business functions whose integrity should not be compromised by demands that fall outside their mandate.

Pact [35] and Spring Cloud Contract [39] are popular testing frameworks that utilize Consumer Driven Contracts.

- **Pact** designers specifically state that, in the right situation, Pact will provide many benefits. The right situation is where:
  - You have control over the development of both the consumer and provider.
  - Both the consumer and provider are under active development.
  - The provider team can easily control the data returned to the responses of the provider.
  - The provider team can manage a relationship with each consumer team.

  The main advantages in this situation is that you are able to evolve the codebase and Pact will guarantee that the contracts are met. This will

allow you to know before deployment whether applications will work together. Pact is particularly well suited for developing and testing intra-organisation microservices.
However, the designers also state that in the wrong situation, using Pact will not provide any benefits. The wrong situation is where:

  - The team maintaining the other side of an integration is not also using Pact.
  - Consumers of an API cannot be individually identified (for example, public APIs).
  - You cannot load data into a provider without using the API that you are currently testing (for example, public APIs).
  - You cannot control the data used to generate the responses of the provider.
  - The functionality of a new or existing provider is not being driven by the needs of particular consumers (for example a public API or an OAuth provider where the API is completely stable)
  - Consumer and provider teams do not have good communication channels.
  - Performance and load testing is the main objective.
  - Functional testing of the provider is the main objective - Pact is about checking the contents and format of requests and responses.

- **Spring Cloud Contract** is an umbrella project that aims to help users successfully implement the Consumer Driven Contract approach. Currently, the project consists of the Spring Cloud Contract Verifier project.
The Spring Cloud Contract Verifier enables Consumer Driven Contract development of JVM-based applications and ships with a Contract Definition Language (DSL) written in Groovy or YAML. Contract definitions are used to produce:

  - JSON stubs for Wiremock, for testing consumer code.
  - Acceptance tests, in Spock or JUnit, for testing against the provider.
  - Messaging routes if used.

  The biggest disadvantage of Spring Cloud Contract is that contracts need to be written manually outside the code base. Keeping them synchronized is a potential problem.

Although it is outside the scope of this particular study, our approach can easily be adapted to take into account the expectations of consumers. The analysis phase of the URDAD methodology is perfectly suited to incorporate consumer contracts and absorb the advantages of the consumer driven pattern. Furthermore, our approach is provider and consumer agnostic in that it can be effectively applied in the development and evolution of both providers and consumers. Our approach also has the added benefit of specifying contracts on a much finer level of granularity (e.g. method and class level) in contrast to Pact and Spring Cloud Contract which only caters for high-level contracts between providers and consumers that ensure each component understands the message contents sent between each other. Finally, our approach specifies contracts using annotations in code - this makes it easier to keep the contracts in sync with code changes, and it does not introduce dependencies on external components such as Groovy or YAML. This results in arguably the most important benefit of our approach, which is that it is less tedious to implement.

2.3 Automated Generation of Test Cases from Contract-Oriented Specifications

Belhaouari et al. created an experimental platform known as Tamago-Test for software analysis and automated testing [5]. They focus on the automation of test-case generation from specifications written as Design by Contract and rely on First-order logic assertions to express contracts between components in the generation of test-cases. The preconditions are used to infer the relevant values to provide as method parameters, and the post-conditions are then used as natural oracles (an oracle determines whether the test results are correct [10]). The generation of values for method parameters that are correct regarding the specification corresponds to a constraint-satisfaction problem (CSP) [26]. Initially, the variable domain is defined by the type, and the CSP reduces the range by the various constraints represented by each atomic term extracted from the contract. This term is obtained in the disjunctive normal form [12] of assertions and is included in the preconditions. When the CSP must instantiate a variable, it draws a random value from its reduced domain. Finally, the CSP architecture Belhaouari et al. propose does not restrict the constraint language to finite-domain and predefined types - they came up with a "type builder" that enables the framework to be extended in a flexible way.

The most important characteristics of the Tamago platform created by the Belhaouari et al., are the provision of a specification language, a runtime, and a set of tools for analysis and support. Their approach also emphasizes the Separation of Concerns principle: The client provides the specifications for components, and the providers implement those specifications. The runtime and tools are responsible to realize the contract between these two parties. The specification language is similar to the grammar of assertions in Java Modelling Language (JML) and Belhaouari et al. use only a subset of the features currently available. The resulting language is abstract, and based on a model of co-algebraic observable properties with first-order logic assertions (preconditions, post-conditions and invariants). It also includes descriptions of service behaviors based on finite-state automata with conditional transitions.

An example of the abstract syntax used to describe a bank account service, is as follows:

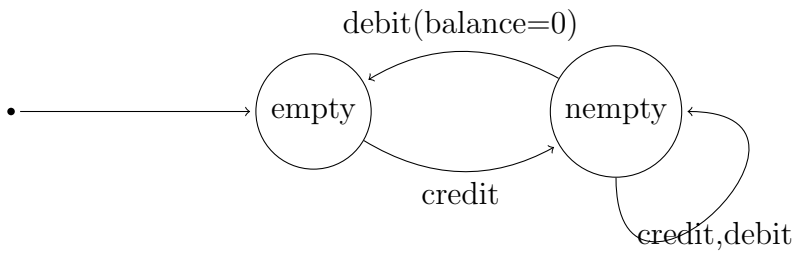
```
interface BankAccount {
  property readonly int balance init();

  property constant string iban;
  invariant balance ≥ 0 ∧ iban.size() == 34
    ∧ (iban.substring(0,2)=="FR"
      ∨ iban.substring(0,2)=="DE"
      ∨ iban.substring(0,2)=="GB" ... )

  void credit (int amount, string id) {
    pre amount > 0 ∧ id.size() == 34
    ∧ (id.substring(0,2) == "FR"
      ∨ id.substring(0,2) == "DE" ∨ ... );
    post balance = balance@pre + amount;
  }

  void withdraw(int amount, string id) {
    pre 0 ≤ amount ≤ balance ∧ id.size() == 34
    ∧ (id.substring(0,2) == "FR"
      ∨ id.substring(0,2) == "DE" ∨ ... );
    post balance = balance@pre − amount;
  }
}
```

The conditional activation of functionalities is described using a behavior automaton. Such an automaton is a good complement to logical assertions, and it plays an important role in the static analysis that can be performed on the specifications.



An account has two properties: the balance and the international bank account number (IBAN). It is possible to credit or to withdraw from an account, under the restriction that there is enough money in the account for the withdrawal operation.

Belhaouari et al. believe the contract language that they have designed is a good compromise between expressiveness and tractability. For tractability, they have developed a set of tools to analyze the contract specification at design time. The first tool performs structural analysis by doing type-checking and using finite-state automata techniques to detect unreachable states or unused functionalities in the contract. The second tool attempts to uncover inconsistencies in the dynamics of the contract by utilizing a symbolic interpreter to generate a set of scenarios from the service behavior. The constraints to be enforced are partially evaluated using the effective preconditions, invariants and the guard of the predecessor transition in the behavior automaton. With this conjunction of assertions, a CSP-based minimization algorithm is used to narrow the domains of observable properties. Complementary to the conjuncts of the effective post-conditions, the invariants and the guards of the next transition (if any) is able to expand/reduce the domain of properties. If a domain becomes empty, there is no more solution and another branch is inspected. This analysis uses various fixed point detection heuristics to ensure the termination of the analysis.

This approach is very similar to the Contract Driven Design with Test Interceptors approach, but differs in that Belhaouari et al. use an abstract language with logic assertions to specify the contracts instead of annotations. This could potentially be a more powerful mechanism to specify contracts, but a disadvantage might be that it becomes much more complex. Our CDD approach that utilizes interceptors has the added benefit that the same language as the programming language is used to specify the contracts, making our approach less tedious and complex to implement.

Additionally, with the approach of Belhaouari et al., testing is done against the code through static and dynamic code analysis. No functional test logic in the target language is generated. We consider the alternative approach of generating test interceptors containing the test logic which are used in both, unit testing and operational testing. Both these testing approaches have wide acceptance within industry.
Finally, the approach of Belhaouari et al. checks that post-conditions are satisfied, subject to the pre-conditions being met. The correct behavior under pre-condition violation (i.e. that the exception associated with that pre-condition is raised) is not verified in this approach.

2.3.1 Contract driven development equals test driven development minus writing test cases

Leitner et al. recognizes that unit testing is a resource-intensive and time-consuming activity [27]. They introduce Contract Driven Development as a method to solve this problem by extracting the contracts that are present in code, and use those contracts to generate test cases. This is achieved by taking advantage of the activities that developers normally perform during the development process: when a new feature is being developed, the developer will run the application in such a way so that the new feature is used. By placing assertions along the way, the developer verifies that the new feature works as expected. This is done by triggering the new feature with the correct input, and also with incorrect input (by changing parts of the application) to ensure that error-handling is done correctly.
These implicit human-generated tests are easy to create and run, and they don't require any maintenance since they are not permanent. Other advantages over automated tools are that the automated testing strategies cannot make up for the insights that a human tester has into the semantics of the software and the relationships between different components. Automated tools also cannot distinguish between meaningful and meaningless input data, and although the quality of the automated tests can be estimated using certain measures or combinations of measures (such as code coverage, number of bugs found, mutation testing, proportion of fault-finding tests out of total tests, etc.), the characteristics of the project under test make it difficult if not impossible for a tool to determine automatically which measures to use.
The drawback of the implicit human-generated tests is that they only exist for one or very few runs and are not kept for later execution. This is because the developer manually provided specific inputs, and if the application was changed to force a certain path to be tested, that change was undone after

the tests.

Leitner et al. proposes a method to capture these implicit tests and to make them explicit and persistent. They created a tool called Cdd (which expands to Contract Driven Development) which targets Eiffel code since Eiffel natively supports contracts, and is installed into the Eiffel Studio IDE. Cdd observes program executions and, when a failure occurs, Cdd detects the last uninfected state and takes a snapshot of this state. This snapshot is then recreated and serves as the starting point of the extracted test case. Cdd chooses the time at which it takes this snapshot so that it is early enough for the state not to be infected, but also late enough to reduce execution time. Also, in order to make the test case more robust regarding system change, the snapshot does not include that part of the state that is irrelevant for reproducing the failure.

Consider the following bank account example written in Eiffel:

```
class BANK_ACCOUNT
inherit
  ANY
    redefine
      default_create
    end
feature
  default_create
    do
      balance := 300
    end
  balance: INTEGER
  deposit (an_amount: INTEGER)
    do
    ensure
      balance_increased : balance > old
      deposited : balance = old balance
    end
  withdraw (an_amount: INTEGER)
    do
      balance := balance - an_amount
    ensure
      balance_decreased: balance < old
      withdrawn: balance = old balance
    end
  ...
invariant
  balance_not_negative : balance >= 0
end
```

This example represents an application still under development that con-

tains incorrect and unfinished code. When a developer runs the code as is in Eiffel Studio, he starts out with an empty unit test suite. In the debugger of the IDE, he invokes the deposit method of the *BANK ACCOUNT* class and indicates that he wants to deposit R30. This throws a post-condition error and stops the application. The reason why this failure occurs is that there is no implementation yet for the *deposit* method - only the contract is written before the implementation. Cdd becomes active when the failure is observed and extracts the test case, which is added to the empty test suite. The extracted test case looks as follows:

```
class TEST_CASE_1
feature
  test
    local
      ba: BANK_ACCOUNT
    do
      ba := new_object ("BANK_ACCOUNT")
      set_field (ba, "balance", 300)
      check_invariant (ba)
      ba. deposit (30)
    end
end
```

The developer does not have to fix the problem immediately - he can go on and test the *withdraw*method. This action will also cause a post-condition error, which will trigger Cdd to extract a test case and add it to the test suite. The extracted test case looks as follows:

```
class TEST_CASE_2
feature
  test
    local
      ba: BANK_ACCOUNT
    do
      ba := new_object ("BANK_ACCOUNT")
      set_field (ba, "balance", 300)
      check_invariant (ba)
      ba.withdraw (20)
    end
end
```

Since Cdd employs continuous testing, these test cases will be evaluated on every run. The IDE will show the tests as failed, until the developer has implemented the relevant methods, and they work correctly, in which case the IDE will show that the tests have passed.

Our approach is similar in that it can also be used to facilitate test-driven

development (writing the contracts first, and then writing the methods that implement the contracts), but our approach does not generate a test suite. Our project also is not integrated into an IDE, but this could be considered for future expansions. Another similarity with our approach is that as requirements are refined incrementally, the code is incrementally enhanced to satisfy the refined requirements.

One of the biggest differences to our approach is that we are focusing on more modern languages that do not have native contracts built in. Another important difference is that the approach of Leitner et al. cannot be used for operational testing (in a production environment) - once the code is compiled and deployed and out of the IDE, no further testing happens, unlike our approach which allows for operational testing since interceptors are compiled and deployed with the production code and contract validation can easily be enabled or disabled.

2.4 iContract - the Java Design by Contract tool

Kramer created iContract, a freely available source-code pre-processor that instruments Java source-code with checks for class invariants, as well as pre- and post-conditions that may be associated with methods in classes and interfaces [25]. His inspiration came from the Eiffel language, which has native support for design by contract. In iContract, special comment tags (@pre, @post, @invariant, added to the Javadoc of the classes and methods) are interpreted and converted into assertion check code that is inserted into the source code. It also caters for the four hierarchical type relations in Java: class extension, interface implementation, interface extension and inner-classes (collectively referred to as "type extension").

To use iContract, Java source code is annotated with three specific comment paragraph tags:

- @invariant, to specify class- and interface-invariants
- @pre, to specify preconditions on methods of classes and interfaces
- @post, to specify post-conditions on methods of classes and interfaces

The tool is run as a preprocessor over a set of annotated source code files. iContract instruments the methods in annotated files with assertion check

code that enforces the specified pre-, post-conditions and invariants. In the background it builds up a repository of contract related information about the classes, interfaces and methods. This information is used to support subcontracting which propagates pre-, post-conditions and invariants along class-extension, interface-implementation, multiple interface-extension and inner-class relations. The instrumented files are compiled instead of the originals, resulting in the same classes, interfaces and methods except that the pre-, post-conditions and invariants are being enforced by means of automatically generated specification checks. Despite being annotated with pre-, post-conditions and invariants, the original files remain fully compliant to standard Java at all times due to the annotations being a part of the (optional) comment paragraphs.

An example of a Java interface with annotations looks as follows:

```
interface Person {

  /*
  @post return > 0
   */
  int getAge();

  /*
  @pre age > 0
   */
  void setAge(int age);
}
```

To clients, the interface method *setAge(int)* specifies the obligation that the age must be greater than zero. In return, the benefit offered to clients of the interface *Person* is that they are promised to receive an age greater than zero, when calling *getAge()*.

Normally, a class that implements the *Person* interface, would look as follows:

```
class Employee implements Person {

  protected int age_;

  public int getAge() {
    return age_;
  }

  void setAge(int age) {
    age_ = age;
```

```
  }
}
```

To instrument the class Employee such that the pre- and post-conditions will be checked and enforced at runtime, iContract instruments the source code with extra checks:

```
class Employee implements Person {

  protected int age_;

  public int getAge() {
    //#*#-------------------------------------------------------
    int __return_value_holder_;
    /*  return age_;  */
    __return_value_holder_ = age_;
    if (!(__return_value_holder_ > 0))
  throw new RuntimeException ("Employee.java:5: error: post-
   condition "+
                "violated (Person.getAge()): "+
                "(/*return*/ age_) > 0");
    return __return_value_holder_;
    //-------------------------------------------------------#*#
  }

  void setAge(int age) {
    //#*#-------------------------------------------------------
    boolean __pre_passed = false; // true if pre-cond passed.
    // checking Person.setAge(int age)
    if (! __pre_passed ) {
    if (age > 0) __pre_passed = true; // Person.setAge(int age)
    }
    if (!__pre_passed) {
      throw new RuntimeException ("Employee.java:9: error:
   precondition "+
                  "violated (Employee.setAge(int age)): "+
                 "(/*Person.setAge(int age)*/ (age > 0)) "
    ); }
    //-------------------------------------------------------#*#
    age_ = age;
  }
}
```

It is not clear when work on iContract stopped, but an attempt was made to resurrect it in the form of Java Contract Suite (JContractS) which is available on SourceForge (https://sourceforge.net/projects/jcontracts/). However, at the time of writing, the last update to this project was made in April 2013.

Our approach is very similar to iContract/JContractS. In fact, had work continued on these projects to take advantage of the newer language features of Java, we might have incorporated iContract/JContractS in this study. The biggest differences that our approach has, are as follows:

- We are using native Java annotations to specify the pre- and post-conditions and invariants instead of using the comment paragraphs, and we are using reflection to generate interceptors.
- We generate separate interceptor classes instead of inserting extra checks and assertions into the original classes. This is a much neater implementation since the original code is left untouched and interceptors can be enabled/disabled with a simple switch.
- There has to be separate code generation runs for test and production since the test logic assertions and checks are inserted into the actual components. In our approach, the test logic is encompassed in the interceptors, separate from the actual components.
- The interceptors we generate enable self-testing components which can be used for unit and operational testing.
- The generation of the interceptors is a pre-compiler step and happens automatically.
- With iContract, pre-conditions are tested, and an exception is immediately thrown if the pre-condition is violated. This means that there is no verification that the component itself checks for pre-condition violations and behaves correctly in such scenarios. Our CDD approach that utilizes interceptors specifically verifies that the component behaves correctly when pre-conditions are violated.

2.5 URDAD as a semiformal approach to analysis and design

Contract-driven development falls into the class of semiformal methods as it includes the formalization of component requirements into component contracts. Use-Case Responsibility Driven Analysis and Design (URDAD) is a semiformal Model-Driven Engineering (MDE) approach with a defined analysis and design process including the following steps:

1. The scope of the system is determined by the use cases which are assigned to the responsibility domain assigned to the system.

2. Each use case is mapped onto a system service. The detailed requirements and design for each of the services (use-cases) are typically developed incrementally within an agile software development process.

3. For each service the following process is followed:

   (a) The service requirements are formalized by specifying:

      i. the data structures for the inputs and outputs of the service.

      ii. the pre-conditions under which the service is provided and the exception raised when a pre-condition violation is detected – pre-condition assessments are either specified fully using a constrained expression or assigned to a lower level service which is either available for reuse or whose detailed requirements and design are refined at the next lower level of granularity.

      iii. the post-conditions which must hold after service provision – post-conditions are either specified fully using a constrained expression or assigned to a lower level service which is either available for reuse or whose detailed requirements and design are refined at the next lower level of granularity.

   (b) The process used to realize the functional requirements is designed. This may include process orchestration across lower level services.

   (c) The transition to the next lower level of refinement is done by taking one of the lower level components as new subject and repeating the above process for its services. The process ends when all required services are either designed or available from the environment (e.g. external service providers, frameworks used, operating system).

Agile methodologies and principles are currently widely adopted in industry. Typically, they are used within DevOps approaches which aim to improve the agility of businesses themselves in order to continuously address evolving client needs and business opportunities. It is thus imperative that a model driven development method supports many of the agile principles, including incremental development and deployment of a software solution. URDAD based MDE naturally facilitates an Agile development method by

introducing a number of mechanisms which promote the core principles of *Agile* (see [4]):

1. *Support for incremental development* around use-cases (functionality or features). In particular, detailed requirements specification and design, together with automated test functionality and code generation, allows for early and continuous testing and deployment of functionality into a production system.

2. The ability to *effectively operate within an environment of rapidly changing requirements* through the ability to assess the impact of requirement changes on the design via bidirectional traceability and rapid addressing of changed requirements through automated test updates from modified component requirements and automated code generation from modified designs.

3. *To enable business and developers to collaborate* more effectively around a requirements model and documentation generated from it, which is more accessible to business than code.

4. *Continuous attention to technical excellence* through test functionality, which is automatically generated from semiformal requirements. Furthermore, decoupling layers of granularity via component contracts facilitates

   - *increased reusability and flexibility through pluggability.*
   - *unit testing across levels of granularity* via relatively simple unit tests which test only the component logic and not any logic of its dependencies[1]. Note that this in itself can improve agility, as a lack of rigorous automated tests across levels of granularity leads to a "do not touch if it's not broken" approach.
   - the *generation of tests for COTS components* to be reused within a system. Note that using a COTS component will usually require the addition of an adapter.

   Note also that studies have correlated quality with agility [18].

5. Having component tests which are generated from semiformal component contracts facilitates *continuous refactoring in order to continuously remove unwarranted complexity*, i.e. simplicity.

[1] This is achieved by using mock objects for component dependencies.

URDAD is typically embedded within a model-driven engineering process, within which agile principles and practices can be absorbed. URDAD aims to increase agility through a number of intrinsic process elements like decoupling across all levels of granularity, explicit search for service reuse resulting in lower cost and complexity reduction, automated documentation, complexity reduction through enforced responsibility localization and rapid informal contract and design specifications by domain experts followed by incremental design formalization by technology experts [38]. Klopper et al. used a comprehensive framework put forward by Avison and Fitzgerald to assess three software development methodologies for a pilot study at Strate Ltd., the central securities depository of South Africa [24]. The methodologies they assessed were The Architecture of Integrated Information Systems (Aris), The Rational Unified Process (RUP) and The Use-case Responsibility Driven Analysis and Design (URDAD) methodologies. The aim of the assessment was to determine which framework best suited the organization's requirements. The criteria of high importance for them were Planning and Control, Improved Quality, Teachability, Design for Change, Effective Communication, Simplicity and Product and Cost. Based on the scores obtained by utilizing the assessment framework, they found that the URDAD methodology scored the highest, confirming our belief that our work regarding CDD is very much relevant and will contribute to the quest of creating quality software.

2.6 The Agile Methodology

The Agile Methodology is well known and used by many software development companies. It is based on four key values and twelve key principles [4]: The four key values are:

- Individuals and interactions over process and tools - this value underlines the importance of communication with clients and that team members should prioritize client questions and feedback.
- Working software over comprehensive documentation - the most important duty of the team is to complete the final deliverables that were promised to the customer. Documentation is secondary.
- Customer collaboration over contract negotiation - this value states that the client must be involved in all project phases, instead of only before and after the project.

- Responding to change over following a plan - when circumstances change and clients demand extra features, teams must adapt and still deliver a quality product that satisfies the client.

The twelve principles that underpin Agile development practices were developed almost two decades ago and will likely remain relevant for years to come due to their *anthropocentric* nature, meaning they are based on human values with the goal of improving workspace quality. These principles are:

- Early and continuous delivery of valuable software - traditionally, clients would only see their product once it has been completed. Agile requires clients to be continuously involved via small, incremental updates that allow the client to see progress and introduce changes early on.

- Embrace change - by welcoming change and actively seeking to make improvements via client feedback, Agile provides a competitive edge to the business and ensures that the product satisfies the needs of the consumers.

- Frequent delivery - smaller, frequent releases reduces the chance of error since it provides the client with more opportunities to provide feedback.

- Cooperation - Agile requires all stakeholders to work in unison in order to achieve the project goals. This advances cooperation and communication and helps management understand the challenges faced by developers.

- Autonomy and motivation - project managers are required to foster an environment where team members are motivated and not afraid to voice opinions and make suggestions for the betterment of the product.

- Better communication - face-to-face communication is preferred over emails and memos.

- Working software - Agile requires stakeholders to minimize paperwork and maximize productivity. The only measure of success is a working product that satisfies the client.

- Stable work environments - the project team must aim to achieve a repeatable pattern of sustainable development that prevents team members from being overwhelmed while the project progresses.

- Quality assurance - when speed is prioritized over quality, it becomes difficult to adapt the product to changing requirements, making the process less agile. It is important to embrace continuous improvement.

- Simplicity - all process that do not contribute to the quality or progress of the project should be removed. Repetitive and time-consuming tasks should be automated if possible, and re-use of components from other projects is encouraged.
- Self-organizing teams - a team should have decision-making powers and be comprised of individuals with a variety of skills who can organize themselves as necessary to achieve the project goals. This allows the responsibility of satisfying the client to be shared, rather than be on a single project manager.
- Reflection and adjustment - learn from past mistakes, reflect on performance and discuss ways to improve.

The cornerstone of the Agile methodology is that it is *iterative* and *incremental*. The goal is to deliver value is fast as possible in increments instead of all at once. The development process is split into multiple iterations, with each iteration delivering additional functionality and improvements. After each iteration, the product is usable, and each iteration builds on the previous iteration, adding functionality until the product is finished. These iterations are also known as sprints. The typical steps of a sprint are shown in Figure 2.1

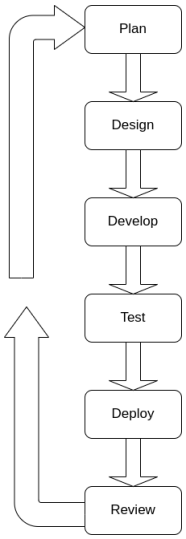


Figure 2.1: Typical Agile Sprint

The Planning step involves a discussion of the requirements and objectives of the sprint, as well as estimations of the time that it will take to

complete. During the Design step, the requirements are analyzed and a design is done to decide the best approach to implement the requirements. The Develop step is where the actual work is done, and the developers implement the requirements. During the Test step, the work that has been completed is tested by a member of the team responsible for quality control (QA) to ensure that the requirements are met, and no obvious bugs are found - this usually happens on a testing environment. When testing has been completed, the latest updates are merged into the final product and deployed to a QA environment - normally the client has access to this environment and is able to use the product and conduct their own testing and provide feedback in the work. Finally, during the Review step, the team looks back over the previous sprint and discusses any problems and make adjustments that should be incorporated into the next sprint.

In this chapter we looked at current and previous tools, methodologies and frameworks for Contract Driven Development. We identified a number of advantages that our CDD approach that utilizes interceptors have compared to these tools and frameworks. Some of the most notable advantages of our approach are that contracts are specified using code annotations, which means developers do not need to learn another language, and it is easier to keep the contracts in sync when code changes. With our approach, contracts can be specified on a fine level of granularity and the implementation of contracts in our approach is arguably less complex compared to various of these existing tools and frameworks. It is also worth noting that a number of these existing frameworks are no longer being maintained.
In the next chapter we discuss the experimental design to assess the impact of using a test-interceptor based contract driven development approach.

Chapter 3

Experimental Design

In this chapter we refine the problem statement posed in Chapter 1:

1. Can Contract Driven Development be used to improve the reliability of the software development process leading to fewer bugs whilst neither increasing the development cost/time nor the skills requirements of developers significantly?

2. Would the quality of the software produced by using a contract-driven methodology justify the effort of utilizing contracts?

In order to make the first question assessable, we add a third research question: For the purpose of the current study the first part of the research question is refined further into

3. Can one offset the overheads of a semiformal contract-driven development approach by

   (a) providing a mechanism which enables developers to efficiently formalize requirements using technologies they are familiar with and

   (b) providing tooling which generates test interceptors which can be used to assess contract compliance in unit testing, integration testing and operational testing?

In order to empirically assess these questions, we have to set up an experiment where one project is executed using the standard development approach of the organization (informal requirements and hand-coded tests), whilst a second development project of similar size, scope and complexity is executed within a contract-driven development approach. The latter team must use the tool set developed within this research project to support contract-driven development.

In this chapter, these problem statements are refined into a set of requirements including requirements for making contract driven development more efficient, and quantitatively assessing the quality attributes of the two development processes as well as the artifacts produced by the development processes. These requirements are further refined into requirements for the tooling required for both, improving the efficiency of a CDD process and for assessing the process qualities of both, the traditional software development project and the CDD based project.

3.1 Assessment requirements

In this section, we are refining the problem statement into requirements for the assessment of the empirical study. In order to assess the impact of our approach, we have to (a) specify the quality attributes which are being assessed across the study and reference study, (b) define and specify a measure which will be used to quantify each quality attribute, (c) define requirements for tools and/or processes used to extract the process data required for the calculation of measures for process qualities, and (d) define requirements for tools and/or processes used to extract the software data required for the calculation of measures for software qualities.

We have identified the following process qualities to assess and compare:

- **Development Productivity** - In the context of this project, we focus on the **number of man-hours to develop a product.** This includes requirements specification, requirements refinement and design, test and bug fixing. This measure does not take the skill and cost differentials of human resources into account because there is not an objective way to determine the skill of an employee (without extensive testing which could be a field of study on its own) and even in the same company the cost (salary) of employees varies too much and that type of information is normally confidential.

- **Correctness of the produced software** - the rate at which the produced software meets the client requirements. This entails producing correct results and handling exceptions properly, and can be measured by counting defects over a period of time with bug-tracking software. All errors should be logged via this bug tracking system from the start of user acceptance testing, until the software system is released to the client. It is also important to distinguish cosmetic and functional bugs.

- **Certifiability** - will the utilization of the process yield results that adhere to established industry standards, for example IEC 61508 (industrial controls), IEC 62034 (medical), ISO 26262 (automotive), IEC 60880 (nuclear energy) or EN 50128 (rail transportation). For software to promote certifiability, it needs to meet the following requirements:
  - Precision - a software development methodology must allow for precise control and monitoring of test inputs and code execution;
  - Formalized requirements - the methodology must allow for the formalization of requirements;
  - Traceability - contract violations must be traceable back to the original requirements.
- **Test Quality** - the quality of the testing itself, which include the number of tests and the level of detail of tests that can be automatically generated. By quantitatively assessing the number of errors which were not captured by the tests and removing the percentage which is due to incorrect requirements, we are left with only those failures which were in the code but not exposed by tests.

  Firstly, we need to calculate the average complexity of our project and a reference project. Complexity measures like cyclomatic complexity and Halstead measures can be used. This will give us a scaling factor to use when comparing the datasets of the two projects:

$$C_a = \frac{C_1 + C_2}{2}$$

  - C_a = Average complexity.
  - C_1 = Complexity of Project 1.
  - C_2 = Complexity of Project 2.

  Using this scaling factor, we can normalize the datasets for comparison:

$$E_u = \frac{C_1}{C_a} * (E_t - E_c - E_i)$$

  - E_u = Relative undetected errors (errors in code).
  - E_t = Total number of errors.
  - E_c = Errors captured by tests.
  - E_i = Errors due to incorrect requirements.

- **Usability** - This will be determined by the feedback from the developers regarding their experience and impressions of utilizing contracts, as well as working with the tool-set that we created.

Regarding the qualities of the software produced by our approach versus the software produced by following the standard approach, we have identified the following metrics to assess and compare:

- **Reusability** - there are two basic types of methods to measure reusability: **empirical** and **qualitative**. Empirical methods depend on objective data, which can be calculated by a tool or analyst automatically and cheaply. Qualitative methods rely on a subjective value as to how well the software adheres to certain guidelines or principles and requires manual effort. [32]. Metrics such as program size, program structure (low coupling), documentation (indicated subjectively on a scale of 1 to 10), programming language and reuse experience can be used to quantify reusability.

- **Simplicity** - also known as the inverse of the complexity measure. Although many methods exist to measure software complexity [6, 11], most use the following metrics:

  Cyclomatic Complexity
  Developed by McCabe [28], this metric is used to indicate the complexity of a program, and measures the number of linearly independent paths through the source-code of a program. Take for example the following section of code:

```
A = 10
IF (B > C) THEN
     A = B
ELSE
     A = C
ENDIF

Print A
Print B
Print C
```

  The control flow of the program is graphically depicted in Figure 3.1:

  The graph depicted in Figure 3.1 shows 7 shapes (nodes), 7 lines (edges) and 1 exit point. Mathematically, Cyclomatic Complexity is defined as:

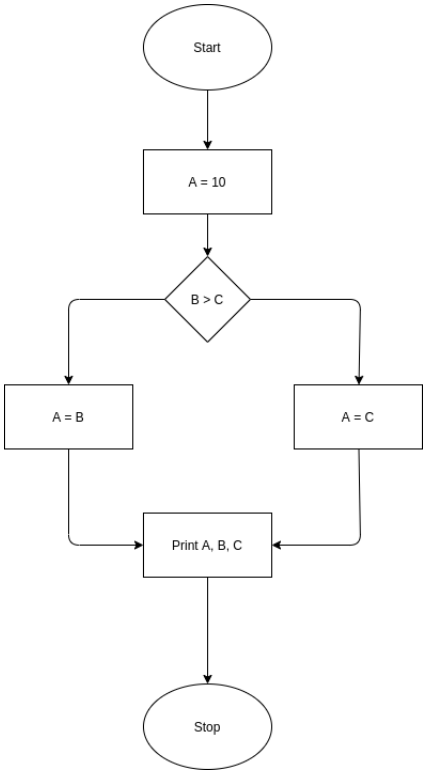


Figure 3.1: Control Flow Graph

$M = E - N + 2P$

- E = number of edges in the control flow graph
- N = number of nodes in the control flow graph
- P = number of nodes that have exit points

Halstead Metrics

Halstead [15] introduced software metrics as part of his treatise on establishing an empirical science of software development. He observed that language agnostic metrics should reflect the implementation of an algorithm in different languages, but be independent of their execution on a specific platform. His goal was to identify measurable properties of software and the relations between them.
The following base measures can be collected:

- η_1 = number of distinct operators
- η_2 = number of distinct operands

- N_1 = total number of operators
- N_2 = total number of operands
- LOC = total lines of code (without comments)

Length
$N = N_1 + N_2$
This metric calculates the total number of operator occurrences and the total number of operand occurrences.

Vocabulary $\eta = \eta_1 + \eta_2$
The total number of unique operator and unique operand occurrences.

Volume $V = N \times log_2\eta$
This metric calculates the relative program size.

Maintainability Index
Maintainability Index is a software metric which measures how maintainable (easy to support and change) the source code is. We use the improved four-metric model proposed by Welker[47] to calculate the Maintainability Index of the source-code. This maintainability index is calculated as a factored formula consisting of Lines Of Code, Cyclomatic Complexity and Halstead volume:
$MI = 171 - 5.2 \times ln(V) - 0.23 \times M - 16.2 \times ln(LOC)$

◦ **Bug Density** - usually defined as defects per thousand lines of code (KLOC) and measured by dividing the size of the module by the number of confirmed defects.

In order to definitively conclude whether CDD with test interceptors produces quality software, and if the complexity and tediousness of implementing contracts have been mitigated, we need to show that:

- The introduction of contracts does not significantly impact the duration of the project;
- The average number of bugs per day is less when using our approach;
- The bug density is less when using our approach;
- Component reusability is better when using our approach;

- The code complexity should not increase when using our approach;
- Our approach should not hinder the certifiability of the project;

3.2 Process requirements

When considering the methodology, we have to take into account the core qualities of good design and ensure that the methodology promotes them.

3.2.1 Core qualities of good design

The following are accepted design principles [37]:

- Responsibility localization. Also known as a design with high levels of cohesion, this design promotes the single responsibility principle, which means that each component is responsible for a single piece of functionality.
- Clean layers of granularity. This important principle entails that in the levels of granularity of a component, lower levels should not have any dependency on the higher levels. The higher level workflow should be understandable without knowing the details of the lower levels.
- Decoupling. This improves maintainability and flexibility, and is most commonly seen in client-server relationships. Clients should not be dependent on a single service provider - instead, the client should define its requirements using a contract, and this contract is then used by a service provider to determine which services to provide.
- Simplicity. The more complex a system becomes, the higher the risk, development and maintenance costs become. A simple solution which is easy to understand and explain is always preferable to a solution that is difficult to understand and explain.
- Architecture and technology neutral. Technology changes frequently, so ideally a design should be technology neutral and then mapped onto the choice of technology.

The following are the benefits of adhering to these principles:

- Understandability. By adhering to the principles of simplicity, good responsibility localization and the ability of understanding workflows at a higher level of granularity, understandability is greatly improved.

- Reusability. Adhering to responsibility localization results in reusability because classes whose function addresses a single responsibility and whose behavior is well-defined, are likely to be reused.
- Testability. By specifying a contract for each component, testability is improved since contract adherence is easily testable.
- Maintainability. The principles of simplicity, decoupling, testability and reusability all contribute to make a system more maintainable.
- Longevity. When a design is technology neutral, it will survive changes in technology.

A Contract Driven Design process focuses on identifying and assigning responsibilities in the early stages of the design. For each of these responsibilities, there should be a *contract* which all service providers (components) that realizes the responsibility, must adhere to. The functional aspects of a contract are defined by an interface with pre- and post-conditions on the services and, if required, invariance constraints on the service provider.

Pre-conditions
If any pre-condition of the contract is not met, the service provider may refuse the service. However, if all pre-conditions are met, the service provider has to provide the service. Service providers make use of exceptions to notify a client in cases where pre-condition violations have occurred.

Post-conditions
Post-conditions are the deliverables of a service provider and include return values and, in some cases, also the state information of the service provider. Post-condition violations are also communicated via exceptions.

Invariance constraints
Invariance constraints are the rules governing the states of the service provider. If at any stage the state of the service provider does not adhere to these constraints, the system is in an invalid state. [37]

Realistically, a software development company with real-world projects and real-world clients, with established processes and employees employed to run those processes, will not be agreeable to change these processes for an experiment when actual projects are affected, which might have consequences on revenue or reputation. So in order to get buy-in from the company to perform our experiment, our methodology needs to fit within the existing Agile

software development methodology so as not to have any negative impact on running projects.

For the purposes of this study, we need to utilize a reference approach. This entails that we need to use an approach that can be applied by a team of developers during a project that enables them to specify contracts for components. Ideally, this approach must fit into and complement current architecture and process requirements stipulated by the organization. In this case, the organization uses an Agile software development methodology, so our approach should leverage the methodology already in use. With that in mind, Agile elements like regular stand-ups and retro-actives are mechanisms for discussing contracts, problems and getting feedback regarding the process.

In order for a development team to utilize our approach and tool set, we need to provide documentation and training, especially the set-up and use of the tool-set. A formal training session to explain the concepts and goals of the study should be considered, and daily Agile stand-up sessions should be sufficient to address questions. A wiki-page or similar can be considered for technical details and how-to examples.

Most bug-tracking and project-management software allows exports of data to be done, but it would not be of value if the data that is exported is not accurate, so team-leads and team members must be diligent when logging bugs and features and updating the status of these items as they are worked on.

In order for us to analyze the code that is produced, a central code repository is crucial. Fortunately, the organization has a strict process in place which requires developers to commit their code on a daily basis to GitHub. This allows us to download the project(s) and have access to the raw code, which allows us to calculate the relevant metrics.

3.3 Development tooling requirements

We need to create the software that will be utilized to process contracts as annotations, and generate the test interceptor classes that validate the contracts. The requirements for contract-driven development tools include:

- The specification of the language developers will use to formalize the contract. This should be as close as possible to the native language (i.e. Java) so as not to introduce additional complexity and a learning curve that will impact on the project deadline;

- The specification of the tools used to generate the test interceptors, which are:
  - The tools need to be simple, fast and seamless to use;
  - If a pre-condition is not met, the associated exception must be raised;
  - If a post-condition is not met, the associated exception must be raised;
  - If an invariant fails, the associated exception must be raised;
  - The tools should be packaged into a re-usable library that can be incorporated into any project;
  - The tools should be developed such that it does not have an intricate and complex application programming interface (API) - it should be easy to implement in new projects as well as existing projects without too much refactoring.

In this chapter, we discussed the requirements needed to empirically assess our CDD approach. We specifically looked at the process qualities to assess, namely development productivity, correctness of the produced software, certifiability and usability. We also discussed the assessment of the qualities of the produced software, including reusability, simplicity and bug density. Furthermore, we discussed process requirements and which qualities a methodology should promote. Finally, we discussed the development tooling requirements that the tools we create to generate test interceptors should adhere to.
In the next chapter, we discuss the details of the CDD methodology that the team must follow.

Chapter 4

CDD Methodology Design

In this chapter, we discuss the details of the empirical study in which we aim to assess the impact of using a light-weight, efficient, tool-supported CDD approach on the quality attributes of both, the software development process and its outputs. Specifically, we look at tooling which will make CDD efficient and less intrusive for developers. Additionally, we discuss the design of an experiment which includes a project where the team followed the traditional approach, and a project where the team used CDD, the selection of projects to be executed and the execution of the development processes within an environment which captures the test and artifact qualities. Finally, we discuss the reference process as well as the measures and tools used to compare the relative process qualities of these two software development processes.

4.1 The development methodologies used

In this section, we discuss the details of the software development methodologies that were followed by the team during two projects. We also discuss the software architecture that is used, and how our approach integrates with it.

4.1.1 Traditional approach versus a light-weight CDD approach

The traditional software development approach that the company is using, is an Agile methodology. Since Agile is, at its core, about lightweight processes that provide flexibility and are unique to each team [7], it is well suited for our purposes to introduce CDD without major changes to the company stan-

dard way of working. By keeping process changes to a minimum, we are able to isolate the impact of CDD on the process. Furthermore, people are normally resistant to change, and our methodology would stand a better chance of being accepted if changes to the normal flow are kept to a minimum. By applying the experiment to an actual project of an actual client, it not only ensures that the experiment represents a realistic software development scenario, but also that the company is less resistant to allow us to perform the experiment.

The team that participated in the experiment created a software product using our light-weight CDD approach. We then identified another product that was developed by the same team, with a similar scope (time-lines, amount of code and complexity) which the team had done previously using the traditional approach. Figure 4.1 depicts the software development processes that were used by the team during the two different projects:

- **Planning:** Both projects started with a planning step, during which time the project manager presented the project to the team. This normally include the high-level requirements of the client, technical details and preliminary time-lines.
- **High level requirements and design:** The next step is high-level design sessions, which involve the whole team. During this step, the requirements were analyzed, and the technical design was done. The work was divided into sprint items and added to the backlog in the project management software that is used. This step was performed during the traditional approach, as well as the CDD approach.
- **Specify component contracts:** When the team followed the CDD approach, the developers performed this extra step, during which time the component contracts were informally discussed in plain text.
- **Develop code:** During both the traditional approach and the CDD approach, the developers were assigned items from the backlog to implement in code. However, when they followed the CDD approach, the contracts that were specified in plain text were now expressed in terms of the Contract Description Language (discussed in Chapter 5) and formalized in code.
- **Specify test data for test-interceptor-based unit tests:** While following the CDD approach, test interceptors based on the component

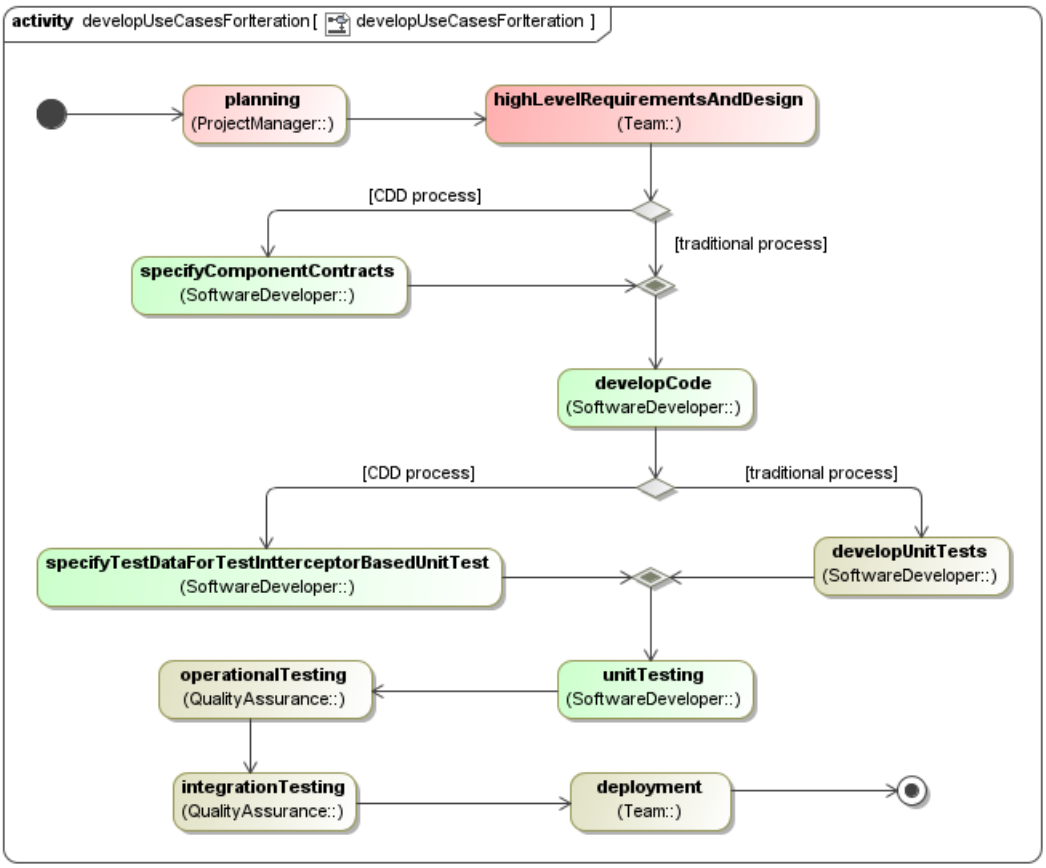


Figure 4.1: Methodologies followed by the teams

contracts were automatically generated. For the unit tests, the developers concentrated on specifying test data using equivalence partitioning (discussed in Chapter 5)

- **Develop unit tests:** While following the traditional approach, unit testing was left to the discretion of each developer and tests were created for which the outcome was known.

- **Unit testing:** For both approaches, the unit tests were performed in order to determine if the function worked as expected. All dependencies were mocked by mock objects that mimic the behavior of the actual components in an integrated environment. This step was part of the build process and was performed every time a component was built to ensure that no errors were introduced by updates to the code.

- **Operational testing:** For both approaches, the components were de-

ployed on the production environment, and only the QA people were given access to use them. This step tested the components with live, operational data. Any issues that were found, were logged in the bug tracking system.

- **Integration testing:** For both approaches, the components were deployed on the development and QA environments to test their behavior in their integrated environment. The mocked components in the unit tests were replaced by the real components in the system. The QA people tested the components end to end and verified that the requirements of the client were met. If any errors were found, it was logged in the bug tracking system.

- **Deployment:** All relevant users were given access to the component on the production environment, and the team monitored log files and support tickets.

4.1.2 Software architecture

The architecture design that the company uses is a layered architecture, which requires that an application be created using four layers, as depicted in Figure 4.2:

- User Interface layer - This layer contains the user interface components (screens) and the flows that determine which components to display based on user actions.

- Application Adapter layer - This layer is responsible for translating and mapping the information from the User Interface Layer into a Data Transfer Object (DTO), and passing those objects to the Infrastructure Layer. The DTO objects are found in the Domain Layer.

- Infrastructure layer - This layer is responsible for the processing of the data received from the Application Adapter Layer and performing business logic on the data. It is also responsible for interactions with databases and other mechanisms of data persistence.

- Domain layer - This layer contains objects that relate to the user domain, which encapsulates the data passed between the User Interface layer, Application Adapter layer and Infrastructure layer.

Using the layered architecture has the following benefits: Responsibility localization is promoted by designing and grouping the components of a system according to the four layers mentioned in Figure 4.2. Components have a

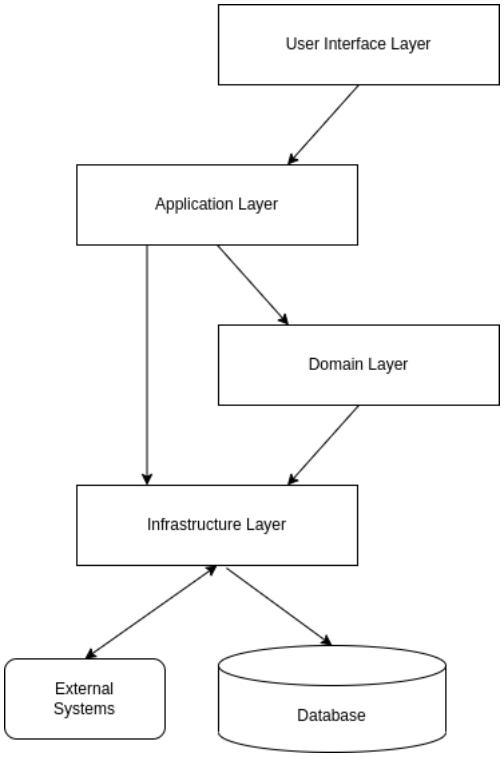


Figure 4.2: Layered architecture

single piece of responsibility according to the layer it occupies, which adheres to the single responsibility principle. It promotes clean layers of granularity in that the layers are not dependencies of each other - changes to the user interface (in the User Interface layer) does not affect the Application layer, Domain layer or Infrastructure layer. Similarly, any changes to the persistence mechanism (for example, changing from a MySQL database to an Oracle database) only affects the Infrastructure layer. The decoupling of components into the four layers also improves flexibility and maintainability, since updates to components in one layer normally does not affect components in the other layers. By designing and grouping components into the four layers, an otherwise complex system is simplified, which reduces risk and maintenance costs.

The resulting benefits we obtain from using an architecture that supports these qualities, are understandability since the layered architecture promotes simplicity and responsibility localization. The clean layers of granularity also contributes to enhance the understandability of a system. Responsibility localization results in increased reusability of components because when the

function of a component is well-defined, and it addresses a single responsibility, the component is likely to be reused. By adding contracts, testability is improved because contract adherence is easily testable. When a system is simple, and the components are decoupled and reusable and testing is easy, it results in the system being easily maintainable.

4.2 The Methodology for Assessing Process Qualities Between a Tool Supported CDD and Traditional Software Development Process

In Chapter 3, we identified the following process qualities to assess in order to determine the impact of our approach in the empirical study:

- Development productivity
- Correctness of the produced software
- Certifiability
- Test quality

We also identified the following qualities of the software produced by the process that we have to assess in order to determine the impact of our approach:

- Reusability
- Simplicity
- Bug density

To determine the impact of our approach, we need to assess these qualities in a project where a team utilizes our approach, and compare the results to a reference project of similar scope, size and complexity where the team utilizes the "normal" company development approach.

4.2.1 Development Productivity

In the context of this project, we focus on the number of man-hours to develop a product. In order to assess this quality, we need to determine the time duration of activities (like requirements specification, design, development

milestones, testing and bug fixing) and compare the results to the reference project. Bug-tracking software or project management software is necessary to keep track of all project related activities during both projects, and the software should be able to export the information we need (like activity name, start-time, end-time).

4.2.2 Correctness of the produced software

To measure the correctness of the produced software, we need to determine the number of defects (bugs) logged over a period of time. Diligent use of bug-tracking software is required for the duration of the project utilizing our approach, as well as the reference project.

4.2.3 Certifiability

We need to determine if our approach addresses the requirements for certifiability as stipulated in a formal standard, like DO-178. This standard lists the following requirements:

- Precision - a software development methodology must allow for precise control and monitoring of test inputs and code execution;
- Formalized requirements - the methodology must allow for the formalization of requirements;
- Traceability - contract violations must be traceable back to the original requirements.

4.2.4 Test quality

Testing is always a tedious activity, especially for developers who have to write unit tests. This generally result in poor test coverage - either not enough tests are created and important functionality is not tested, or too many tests are created that focus on the same functionality. We need to assess the tests that are used while utilizing our approach, and compare them to the tests that are used for the reference project.

In Chapter 3, we introduced a formula to use in order to compare the test quality of the two projects. In order to use this formula, we need to determine the complexity of the projects, as well as classify and aggregate the errors that were discovered. The complexity can be determined by an automated tool (see the discussion about Simplicity below). Bug tracking

Name	URL
Verifysoft Technology	https://www.verifysoft.com
Halstead Metrics Tool	https://sourceforge.net/projects/halsteadmetricstool
JHawk	http://www.virtualmachinery.com
SourceMeter	https://www.sourcemeter.com
PMD	https://pmd.github.io
Sonarqube	https://docs.sonarqube.org

Table 4.1: Automated code metric tools

software is needed to log the errors that are discovered during testing, and these errors need to be scrutinized and classified.

4.2.5 Reusability

An important consideration in software development is the reusability of components. The use of existing components or libraries greatly reduces the time needed by a developer to implement certain functionality in an application. But the developer needs to know that the component that is being reused, performs correctly. We need to make a qualitative assessment of the components created while utilizing our approach and compare it to the components created in the reference project.

4.2.6 Simplicity

In Chapter 3, we identified the following metrics to measure the code produced: Cyclomatic Complexity, Halstead Volume and Maintainability Index. To assess these metrics, we endeavor to use automated tools which are highly rated by the community, as well as academics, where possible. There are a number of tools available:

- Verifysoft supports metrics like Cyclomatic Complexity, Halstead metrics and Maintainability Index. However, the software is not free.
- Halstead Metrics Tool supports metrics like program vocabulary, length, volume, effort, difficulty and time. The software is free, but it was last updated in 2016, and it can only export results in HTML and PDF format.

- JHawk is popular in academic environments and supports metrics like Maintainability Index, Halstead metrics, Lines of Code and Cyclomatic Complexity. The software requires expensive licensing fees.
- SourceMeter offers a free version that covers the metrics we are interested in, is easy to use, and is able to export results into a comma-delimited file.
- PMD analyses software according to a multitude of editable rules. These rules can become very complex to create and maintain, and only a small amount of metrics are provided by default. It is free and open-source.
- Sonarqube is a popular software tool that is used to analyse code in order to detect bugs and known vulnerabilities, and is often used as part of a Continuous Integration (CI) pipeline in companies. It supports some metrics like Cyclomatic Complexity and Maintainability, but none of the Halstead metrics.

For the purposes of this project, we decided to use SourceMeter to calculate the code metrics because of the facts that the software is free, and that it is able to export the results in a format that is easy to work with. This entails that we need access to the code repository in order to download the source code of both the project utilizing our approach, and the code of the reference project and compare the results of the metrics calculated by the automated tool.

4.2.7 Bug density

This quality is measured by dividing the size of the module by the number of confirmed defects (bugs). By utilizing bug-tracking software and examining the code from the repository, we will be able to calculate the overall bug density of each component. Cosmetic bugs should be assessed separately, since contracts do not affect the design of the user interface.

4.3 Experimental setup

In order to determine the impact of our approach, we need to set up an experiment. This experiment consists of two projects that are similar in scope, size and complexity, but one project must utilize our CDD approach, while

the other project utilizes the standard company software development approach.

Ideally, two teams are required: one team completing a project utilizing the normal software development approach, and the other team completing a different project while utilizing our CDD approach. But depending on resource availability, it would also be acceptable when the same team completes both projects when one project starts after the other project has finished.

We have to ensure that the relevant mechanisms to gather the data discussed in Chapter 3 are in place at the start of both projects, for example bug-tracking software, project management software and software repositories. Fortunately, these items are standard in most IT companies.

It is important that the team working on the project that utilizes our light-weight CDD approach, understand the methodology. To this end, the author must be a member of this team in order to ensure that the methodology is followed correctly and to answer any questions that team members might have.

The team utilizing our approach must be shown how to use the tool set that we have developed, and it must be incorporated into the development environment. Any problems or enhancements to the tool set must be addressed as quickly as possible.

After both projects have been completed, the data from the bug-tracking software, project management software and repositories must be analyzed and assessed according to the requirements discussed in Chapter 3. The process qualities to assess are:

- Development Productivity - assess the man-hours taken to complete the projects.
- Correctness of the produced software - assessing the defects over a period of time.
- Certifiability - assess which aspects of the process makes certifiability easier.
- Test quality - assess the number of failures that were not captured by the tests.

- Usability - assess the feedback from the developers regarding their impressions of using our approach and tool-set.

The qualities of the software to assess are:

- Reusability - assess what the effect of our approach has on creating reusable components.
- Simplicity - measure and compare the cyclomatic complexity, Halstead volume and maintainability index of the software produced in both projects.
- Bug Density - measure and compare the bug density of the software produced in both projects.

The results of these assessments and measurements must be discussed and presented in a graphical format to illustrate the differences. In order to do so, it will be necessary to export the data from the bug-tracking software and project management software into a format that can be processed and analyzed. Custom programs and scripts will be necessary to transform the exported data into a format to facilitate analysis, for example the data will need to be converted into a comma-delimited text file in order to be imported into Microsoft Excel or other spreadsheet software.

Extracting the data from bug-tracking and project management software (data sources) requires some custom-built components. These software suites normally have the functionality to export data into a file in a format like JavaScript Object Notation (JSON), Extensible Markup Language (XML) or some other form of text. We will create a custom parser to process each of these files according to the format of its contents, and insert the data into a relational database. This allows us to easily do queries and apply filters in order to extract relevant data-sets. These data-sets are exported into text files as comma separated values (CSV), which is the standard format that can be imported by spreadsheet software, which enables us to do statistical analysis and generate graphs (see Figure 4.3).

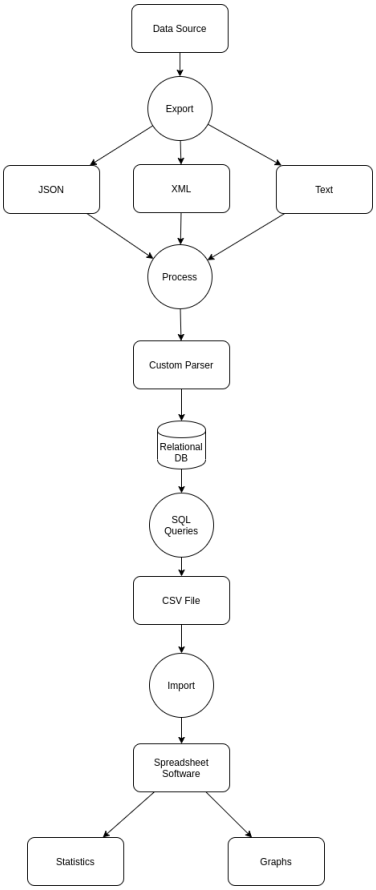


Figure 4.3: Assessment Process

In this chapter we discussed the details of the methodologies of our CDD approach, as well as the traditional approach used for comparison. Furthermore, we discussed the software architecture used by the company. We also discussed the details about how to assess and compare the process qualities of our CDD approach versus the traditional software development approach, and the tools used in order to generate the relevant metrics. Finally, we discussed the details of the experimental setup.
In the next chapter we discuss the details of the tools we developed to facilitate our CDD, interceptor-based approach.

Chapter 5

The Development of Tooling Support for CDD

In this chapter, we discuss the tools that we have created to facilitate CDD. This includes the contract description language that is used to define contracts, the inner workings of interceptors and how they are automatically generated, extension of contracts (contract inheritance), the testing triangle and equivalence partitioning.

5.1 Annotations in Java and how they are used with interceptors

Java classes and interfaces are annotated with pre- and post-conditions (and invariants) which represent component contracts. An *annotation* in Java is a mechanism to add syntactic meta-data to the source-code, without affecting the execution of the code. In this context, meta-data refers to "information about data" and can be applied to language structures in Java such as packages, classes, interfaces, variables and parameters.
In its shortest form, an annotation looks as follows:

@entity

The @ character denotes that the text following it, is an annotation. Annotations can also have attributes and values, which can be set. The following example shows an annotation with attributes and values:

@entity(tableName = "person", primaryKey = "id")

In this example, the attributes are *tableName* and *primaryKey*, and their values have been set to *person* and *id* respectively.

Annotations have been available in the language since the release of version 1.5 of the Java Development Kit (JDK) in September 2004 and are typically used for:

- Compiler instructions - these annotations are processed by the Java compiler and will trigger certain actions. One example is if a method is annotated with *@Deprecated* to indicate that it should no longer be used, the compiler will show a warning message if it encounters code that still uses this method.
- Build-time instructions - these annotations are scanned by automatic build tools like Apache Maven, and are used to generate source code or other files based on these annotations.
- Runtime instructions - these annotations are accessed when the program is running and are used to affect the flow of the program.

The Java language has a number of built-in annotations, and it is also possible to create custom annotations. In order to specify contracts in code, we have created a library in Java with the following custom annotations:
@Precondition(constraint = "boolean expression", raises = Exception.class)

@Postcondition(constraint = "boolean expression", raises = Exception.class)

@Invariant(constraint = "boolean expression", raises = Exception.class)

These annotations are used to specify contracts, and are used during build-time to generate the source code for test-interceptors via reflective programming (reflection), which is a language feature that allows a program to examine or introspect upon itself and manipulate its own internal properties, which we use to generate test-interceptor classes from the contracts specified in the annotations.

5.2 Contract Description Language

An Object Constraint Language (OCL) is a textual language used in software development to specify constraints or rules that apply to objects in a system. OCL is typically used in conjunction with Unified Modeling Language (UML) to provide a precise and formal specification of the behavior of a system.

OCL allows developers to express constraints on the attributes and operations of objects in a system, as well as on the relationships between objects. These constraints can be used to ensure that the system behaves correctly and consistently, and to check that the implementation conforms to the design.

OCL expressions can be used to specify a wide range of constraints, including data validation rules, business logic rules, and constraints on the behavior of the system. OCL expressions can also be used to specify pre-conditions and post-conditions for operations, allowing developers to define the expected behavior of a method in a precise and unambiguous way.

OCL is widely used in software development, particularly in the development of large, complex systems, where it can help to ensure that the system behaves correctly and consistently, and to reduce the risk of errors and defects.

We use a simple OCL called a Contract Description Language to specify the contracts that are used in pre-conditions, post-conditions and invariants. The *constraint* parameter in the aforementioned annotations is used as the predicate for specifying the constraint in the contract description language. This contract description language consists of simple Java-based expressions for which the following hold:

1. The result of the expression must evaluate to a boolean, for example: *constraint = "isSupportedPartner(rewardsPartner) == true"*

2. In a post-condition, you may use a pre-evaluation - this is an evaluation of some aspect of the state of the component when the service is requested. A pre-evaluation in the contract description language is a Java-clause enclosed in double forward-slashes, for example: *constraint = "getCounter() == //getInstanceCounter() + 1//"*

5.3 Interceptors

The test-interceptor classes encapsulate test-logic and are interface-compatible with their underlying component counterparts (see Figure 5.1).

This enables test-interceptors to intercept service requests to the underlying counterpart components in order to verify contract adherence by:

1. Assessing the truth value of each pre-condition on service request.

2. Delegating the request to the underlying component for processing.

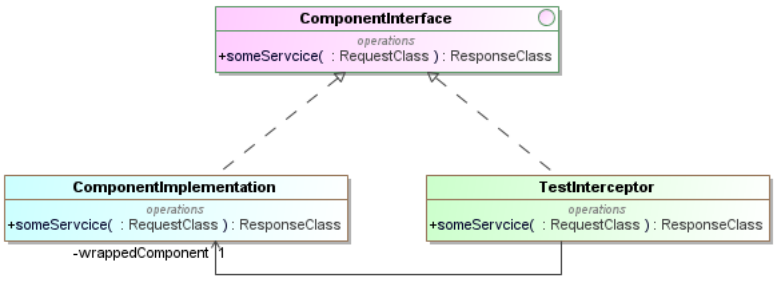


Figure 5.1: Interceptor class diagram

3. Upon return the test interceptor verifies:

   (a) if an exception was generated by the underlying component, verify that the associated pre-condition did not hold via a contract violation exception which is an error.

   (b) if the service was provided (no exceptions) verify that all post-conditions hold true.

The sequence of these events are depicted in Figure 5.2.
The test-interceptors can now be used in the following scenarios:

1. In a live system, components can be wrapped by test-interceptors which assess on the live system whether the wrapped component functions in accordance with the contract.

2. For unit testing, one would inject mock objects for any dependencies of the component under test and specify test data based on an analysis of equivalence partitioning and boundary values.

3. For integration testing, one would inject actual objects used for their dependencies and potentially use the same test data as the unit tests to the component.

The test logic tests whether the behavior is according to contract, but does not include any scenario data sets against which the test is executed. For example, the generated test interceptors contain only test logic. They will:

1. Assess whether all invariance constraints are met - if not, the system starts off in invalid state and the test fails.

2. Assess whether the provided test scenario satisfies the preconditions and store that information.

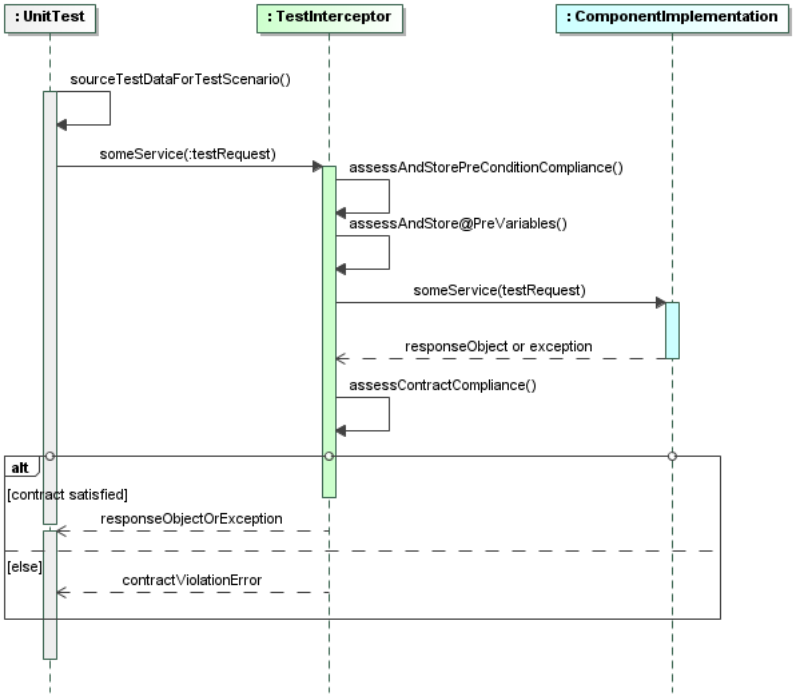


Figure 5.2: Interceptor sequence diagram

3. Perform any @pre evaluations on the supplied scenario data in order to have that information for the assessment of post-conditions which include @pre elements.

4. Call the underlying (wrapped) service with the test data from the provided test scenario.

5. Catch any exceptions and check whether a raised exception is due to the violation of a pre-condition (using assessments 2) - the test fails if not.

6. If no exceptions are raised check:

   (a) that all type-2 pre-conditions were met (using assessments 2) - the test fails if there was a pre-condition not met.

   (b) Assess all post-conditions using potentially assessments in 3 - test is failed if at this stage any post-condition is violated.

7. Assess whether all invariance constraints are still met - if not, the service has left the system in an invalid state and the test fails.

8. If this point is reached, pass the test.

The following code illustrates a Java interface with a method that is annotated with preconditions and post-conditions:

```
public interface Discount_interface
{
  @Precondition(constraint = "isIDValid(ID) == true", raises =
   InvalidArgumentException.class)
  @Precondition(constraint = "isSupportedPartner(RewardsPartner)
    == true", raises = InvalidArgumentException.class)
  @Precondition(constraint = "itemPrice > 0", raises =
   InvalidStateException.class)

  @Postcondition(constraint = "returnValue > 0"}
  @Postcondition(constraint = "getCounter() == //
   getInstanceCounter() + 1//")
  public double getDiscountPercentage(String ID, String
   RewardsPartner, double itemPrice) throws Exception;

}
```

The interceptor class that is generated from the annotations, looks as follows:

```
public class Discount_TestInterceptor implements
    Discount_interface
{
  private Discount_interface counterpart = null;

  public Discount_TestInterceptor(Discount_interface counterpart
   )
  {
    this.counterpart = counterpart;
  }

  @Override
  public double getDiscountPercentage(String ID, String
   RewardsPartner, double itemPrice) throws Exception
  {
    /*
    Evaluate the preconditions
    */
    boolean _pre_1 = isIDValid(ID) == true;
    boolean _pre_2 = isSupportedPartner(RewardsPartner) == true;
    boolean _pre_3 = itemPrice > 0;

    /*
    Evaluate the pre-assessments
```

```
    */
23  int _preAs_1 = getInstanceCounter() + 1;

25  double returnValue = 0;
    try
27  {
      /*
29    Call the wrapped method of the counterpart
      component to be tested
31    */
      returnValue = counterpart.getDiscountPercentage(ID,
    RewardsPartner, itemPrice);
33
      /*
35    If method was implemented correctly, it would have
      thrown exceptions if any of the preconditions were
37    false.
      */
39
      if (!_pre_1)
41    {
        /*
43      The method above did not throw an exception but
        precondition 1 is false.  This means there is a
45      problem with the implementation of the method -
        it does not properly enforce precondition 1.
47      */
        throw new PreconditionNotEnforcedException("isIDValid(ID
    ) == true", "getDiscountPercentage()");
49    }
      if (!_pre_2)
51    {
        /*
53      The method above did not throw an exception but
        precondition 2 is false.  This means there is a
55      problem with the implementation of the method -
        it does not properly enforce precondition 2.
57      */
        throw new PreconditionNotEnforcedException("
    isSupportedPartner(RewardsPartner) == true", "
    getDiscountPercentage()");
59    }
      if (!_pre_3)
61    {
        /*
63      The method above did not throw an exception but
        precondition 3 is false.  This means there is a
65      problem with the implementation of the method -
        it does not properly enforce precondition 3.
```

```
67          */
            throw new PreconditionNotEnforcedException("
        isSupportedPartner(RewardsPartner) == true", "
        getDiscountPercentage()");
69        }

71        /*
          Evaluate post conditions and their pre-assessments
73        */

75        if (!(returnValue > 0))
          {
77          throw new PostconditionNotMetException("returnValue > 0"
        , "getDiscountPercentage()");
          }
79
          if (!(getCounter() == _preAs_1))
81        {
            throw new PostconditionNotMetException("getCounter() ==
        //getInstanceCounter() + 1//", "getDiscountPercentage()");
83        }
        }
85      /*
        The wrapped method threw an exception
87      */
        catch (InvalidArgumentException iae)
89      {
          if (!_pre_1)
91        {
            /*
93          Precondition 1 is false, so the method was
            supposed to throw this exception.  This
95          means it implemented the check for this
            precondition correctly.  Rethrow the valid
97          exception.
            */
99          throw iae;
          }
101       if (!_pre_2)
          {
103         /*
            Precondition 2 is false, so the method was
105         supposed to throw this exception.  This
            means it implemented the check for this
107         precondition correctly.  Rethrow the valid
            exception.
109         */
            throw iae;
111       }
```

```
113         /*
            If this point is reached, it means that
115         the preconditions are valid, but the
            method still threw an exception.  So
117         there is a problem with the implementation
            of the method.
119         */
            throw new PreconditionsHoldButServiceRefusedException("
        InvalidArgumentException", "getDiscountPercentage()");
121       }
          catch (InvalidStateException ise)
123       {
            if (!_pre_3)
125         {
              /*
127           Precondition 3 is false, so the method was
              supposed to throw this exception.  This
129           means it implemented the check for this
              precondition correctly.  Rethrow the valid
131           exception.
              */
133           throw ise;
            }
135         /*
            If this point is reached, it means that the
137         preconditions are valid, but the method still
            threw an exception.  So there is a problem with
139         the implementation of the method.
            */
141         throw new PreconditionsHoldButServiceRefusedException("
        InvalidStateException", "getDiscountPercentage()");
          }
143       /*
          All preconditions and postconditions are true.
145       */
          return returnValue;
147     }
    }
```

The test data on the other hand is the different test scenarios. It encompasses request and state data for both, the component and its environment, i.e. for a specific test scenario one puts the system (including the component being tested) in a specific state as well as the data for the request and response objects.

A test consists of 3 parts:

1. Test interceptor containing the test logic.
2. Test scenario (test data) sets - for complex tests, this may include some logic around putting the system into the required state.
3. A unit test which reads the test data sets and requests the service being tested through the test interceptor.

5.4 Generating interceptors automatically

When using the Java programming language to create software components, the normal process of converting source code into an executable program works as follows (Figure 5.3):

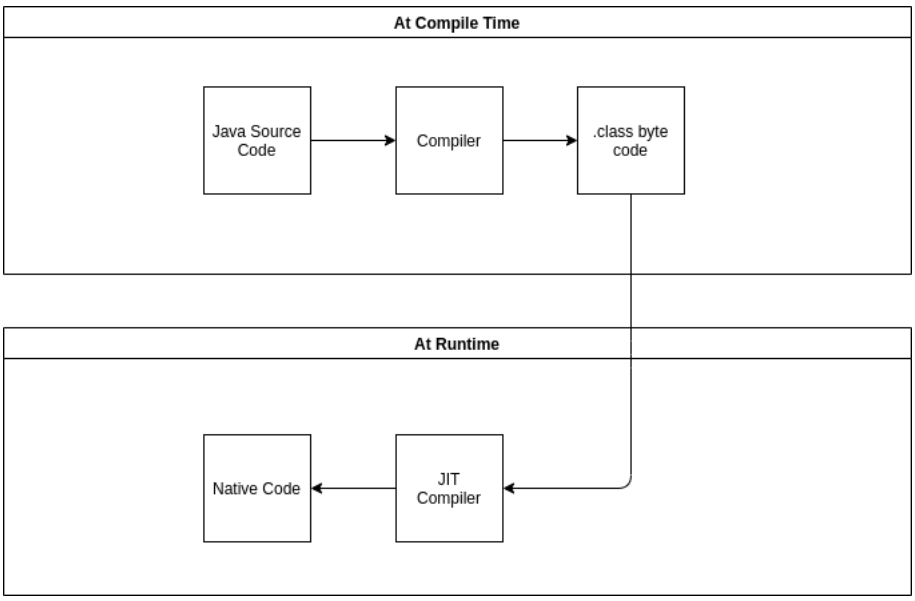


Figure 5.3: Normal Java Compilation Process

1. Source code is written and saved in text files with a .java extension.
2. The Java compiler is invoked. It processes all files with a .java extension and produces byte-code, which is a machine-independent instruction set that is processed by the Java Virtual Machine (JVM). The byte-code produced for each .java file is saved in a corresponding .class file.
3. At runtime, Java uses a Just In Time (JIT) compiler to convert the byte-code into native instructions for the machine it is running on (for example Windows, Unix, Mac).

Interceptor generation happens as a pre-compilation step, as shown in Figure 5.4

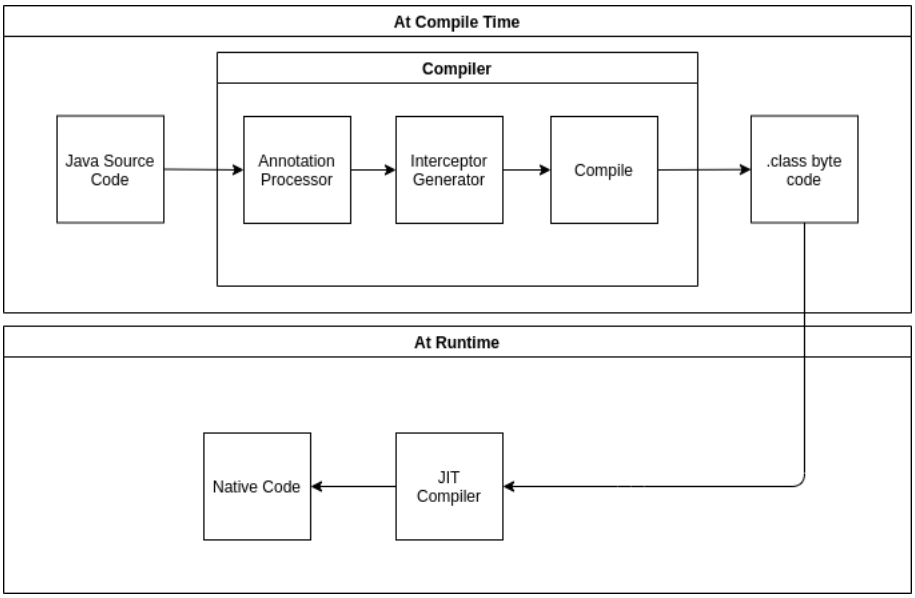


Figure 5.4: Java Compilation Process with Interceptors

The tool that we have created to process contracts as annotations and generate interceptors, consists of two libraries: Annotation Processor and Interceptor Generator.

1. When the source code is compiled, our Annotation Processor is invoked. During this step, the annotated contracts are processed and data structures that contain all the pre-conditions, post-conditions and invariants are created in memory.

2. These data structures are then passed to the Interceptor Generator, which generates .java source code files containing the interceptor logic.

3. The interceptor source code files are then converted into byte-code like every other source file.

The reason for having the interceptor generator logic separate, is because there are multiple ways to do it. This approach allows us to easily create and swap out different interceptor generator libraries that may offer different advantages, like memory utilization, speed and ease of use. As a proof of concept, we created an interceptor generator library that generates interceptors purely by string manipulation. We quickly discovered that this

was very tedious and complex, difficult to debug and slow, but at least it served to prove that our tool worked in principle. To optimize interceptor generation, we decided to use a template engine. This is a tool that uses a pre-defined template, and replaces the variables defined in the template with actual values. They are popular tools used to create websites and PDF files (among others) and have been optimized in terms of speed and resources. The biggest advantage of using a template engine is that, because the template is separate, it is quick and easy to update if anything in the interceptor needs to change. We decided to use the Apache Velocity template engine (https://velocity.apache.org/) because it is open-source, stable, fast and easy to use. Figure 5.5 depicts the details of this process.

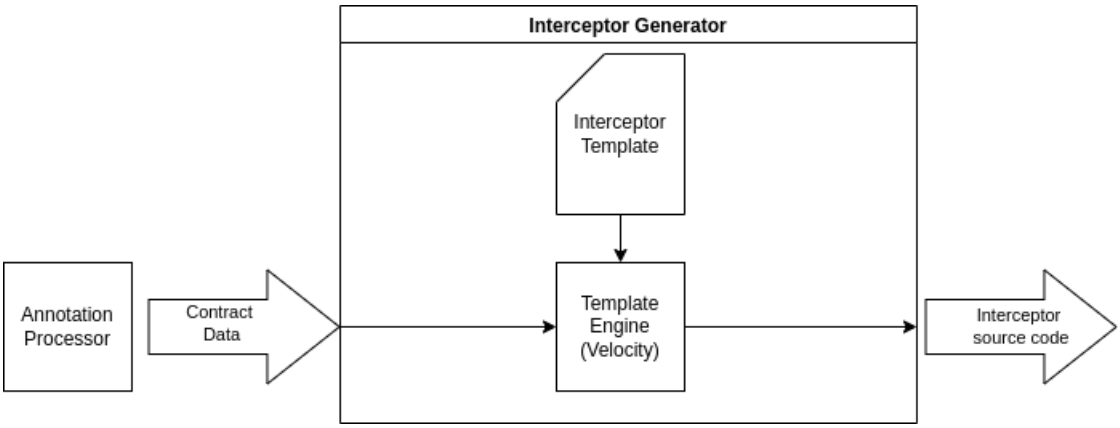


Figure 5.5: Interceptor Generator

5.5 Extension of contracts

In Object-Oriented Programming, inheritance is the mechanism which allows one class to be based on another class. The main class is known as the base class or super class, and the other class that are based on it are called subclasses or derived classes. The derived classes retain all the properties and behaviors of the base class (see Figure 5.6).

Using this mechanism, we have implemented the extension of contracts: when an interface (the base interface) with methods that are annotated with contracts is extended by another interface (the sub-interface), the contracts of the base interface are ”inherited” by the sub-interface. In practice, this means that when a class implements a derived interface that extends a base interface, the interceptor that gets generated will validate the contracts of both the derived and the base interface. This is a very powerful and useful

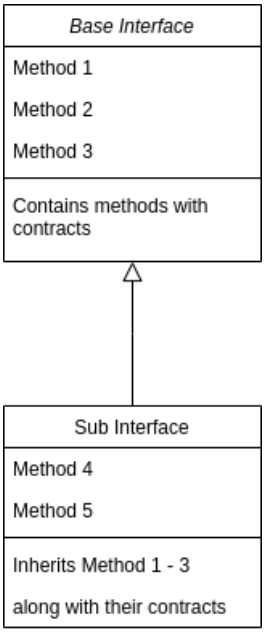


Figure 5.6: Inheriting Contracts

feature which promotes reusability (since contracts in base interfaces can be reused multiple times), as well as reliability (contracts in base interfaces will have been tested and debugged).

The test logic (e.g. the test interceptor) can be used for unit testing, integrations testing and operational testing.

5.6 Testing

The Testing Triangle (depicted in Figure 5.7) shows the different types of software testing and how they build on each other:

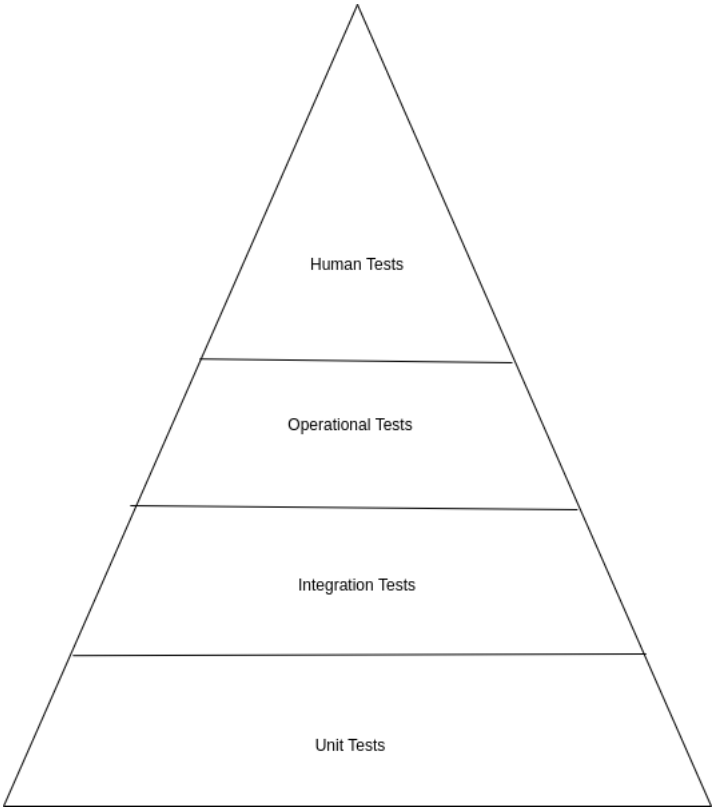


Figure 5.7: Testing Triangle

Unit testing

At the lowest level, this type of testing focuses on testing units of code (like methods or functions) and is arguably the most important step since it forms the foundation of testing - if the basic building blocks (methods and functions) are not correct, all the subsequent steps in the triangle will fail.

For pure unit testing, a component needs to be tested in isolation. That requires all dependencies to be mocked by mock objects which act like contract compliant components from the perspective of the object being tested (i.e. they need not mimic the full behavior of the component as then they would just be the component, they only mimic for the test scenarios a behavior which is perceived observable by the component being tested as correct behavior).

A pure unit test typically

1. creates an instance of a component with its dependencies initialized with mock objects,

2. reads each test scenario,

3. requests the service under test with the test data through the test interceptor from the wrapped component being tested, and

4. reports any TestFailedExceptions to some reporting channel (e.g. a test report).

Unit testing is often done in a non-pure approach, where one uses tested lower level components (assuming they do behave correctly) instead of mock objects.

Integration testing
Integration testing is about testing whether the component behaves correctly in its integrated environment. For this, the mock components satisfying the component dependencies are replaced by the real components, i.e. all component dependencies are part of the actual system. Note that with this approach you have unit and integration testing (and even operational testing) across levels of granularity. At the highest level, the component is the fully integrated system. If you don't use true unit testing with mocking, the distinction between unit and integration testing at any level of granularity starts falling away.

Operational testing In an automated operational test, all or a subset of the system components are replaced with test-interceptor wrapped system components which test contract adherence in the context of normal system operation (the test data/scenarios is now replaced with live operational data). In this case, any TestFailedexceptions resemble scenarios where under normal operation a component does not adhere to its contract.

Human testing
This type of testing involves humans doing manual testing to determine the user experience of the system. This includes non-functional qualities like responsiveness, error-handling and robustness. Although it is an important step in the testing phase of any software product, it falls outside the scope of this study.

Our annotation processor is invoked as a pre-compilation step by the Java compiler. Apart from the initial set-up and configuration of the project to use the annotation processor (in the project build-scripts or in the Integrated Development Environment (IDE) that is used), there are no extra steps re-

quired. This results in interceptor classes being automatically generated each time the project is built.

For unit and integration testing, the test-interceptor wrapped component is called with test data sets, which are systematically created from an analysis of equivalence partitions for the problem. Future work will look at augmenting the CDD tool suite with tools that automatically generate test data sets[14]. It is the clean separation of test logic and test data which facilitates the reuse of test logic, which also applies for operational testing. Generating the interceptor classes is trivial in terms of speed and resources, and even in extreme cases where large numbers of annotations are processed, the build process is not noticeably affected. An interceptor can process any contract as long as it is expressed as a valid boolean expression, although an important function that the interceptors cannot do (yet) is to test for errors or exceptions - a good example of this is date or time formats: it is common practice in Java to use a date parser (for example SimpleDateFormat) which throws an exception when a date or time is not in the expected format. It would be a very powerful feature if date and time formats can be specified in contracts.

5.7 Equivalence Partitioning

Known as equivalence partitioning, this testing technique can be applied to every level of the testing triangle. It involves dividing input data into equivalent partitions that can be used to derive tests, reducing the number of test cases considerably. For example, if an input value must be a number between 1 and 10, the following conditions are applicable:

- Numbers between and including 1 and 10 are valid, which implies that any number greater than 10 is invalid, and any number less than 1 (0 and below) is invalid;
- Any 3-digit number is invalid.

Instead of testing every possible value (which results in numerous tests) the possible values are divided into groups (partitions) as shown in Figure 5.8

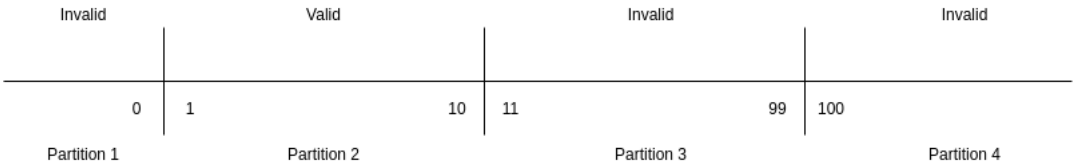


Figure 5.8: Equivalence Partitions

Only one value from each partition is used for testing - if one value passes the test, all others will also pass the test. Likewise, if one value in a partition fails a test, all other values in the partition will also fail the test.

In this chapter we discussed the technical details about how test interceptors are designed and used in testing. In particular, we discussed annotations and how they are used to specifying contracts. We also discussed the contract description language and how it is used to specify pre-conditions, post-conditions and invariants. A large part of this chapter discussed the technical implementation of interceptors and their inner workings and automated generation. Additionally, we looked at the extension of contracts and how inheritance is used to achieve that. Furthermore, we discussed the different types of testing and how interceptors are used with each type. Finally, we discussed a testing technique called equivalence partitioning.
In the next chapter we discuss the results obtained in the experiment.

Chapter 6

Results

In this chapter, we discuss the details of the two projects that we examined and compared in order to assess the impact of a CDD approach. We also show the methods and results of the measurements we obtained.

6.1 Experiment: Android Application

In January 2021 work started on a Covid-19 vaccination project at one of our clients. The goal of the project was the creation of software applications that enable the client to:

- gauge the interest of users to get the vaccine via a short survey;
- enable users to make appointments to receive the vaccine;
- sign digital waivers;
- push-notifications with reminders for vaccine appointments and other important information;
- record vaccine injections;
- report symptoms and side effects; and to
- provide digital certificates as proof of vaccination.

The system also provides additional admin functionality for officials to create appointment slots, record a medical history for each person, and generate statistics.

The system was developed using an Amazon Web Services (AWS) cloud infrastructure for the back-end. This infrastructure hosted a MySQL relational database and provided AWS Lambdas, an event-driven computing

service that runs code in response to events from clients that connect to it via a Representational State Transfer (REST) protocol. The types of clients that connect to the back-end include Android mobile applications, iOS mobile applications, and web-applications (see Figure 6.1).

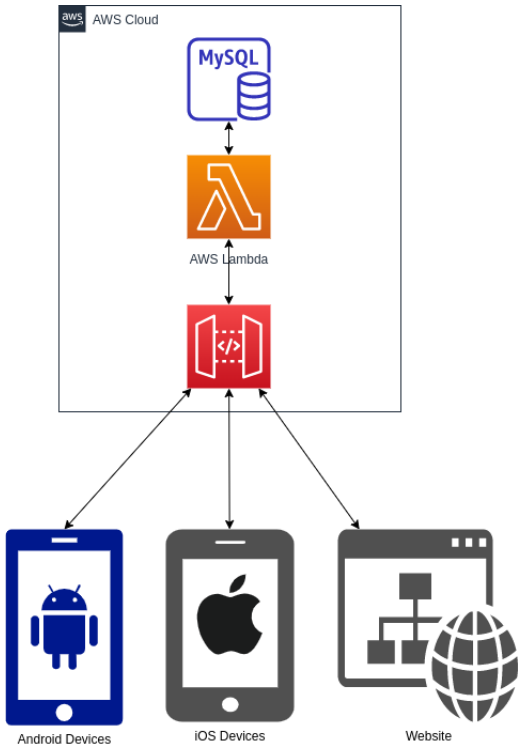


Figure 6.1: System Architecture

Different teams were responsible for each of the different components: One team focused on the back-end, another on the web front-end, and two other teams focused on the iOS and Android applications respectively. A light-weight contract driven design process with the following elements was followed:

- Continuous software delivery through the use of sprints. At the end of each sprint, workable components were delivered for testing,
- Changing requirements were accommodated.
- Frequent meetings with stakeholders were organized.
- Online face-to-face interactions were common.
- The teams were self-organizing.

- After each milestone was delivered, reflections (retro-actives) were held to discuss what worked, what didn't work and how things could be done better.
- Contract elicitation happened early in the sprint.
- Contracts were refined continuously.
- Contracts were expressed in terms of the Contract Description Language (discussed in Chapter 5).

The back-end team was responsible for setting up and maintaining the AWS environment, creating and maintaining the database and creating and maintaining the REST endpoints via AWS Lambdas. The web front-end team created web-based applications that run inside a browser, and the iOS team and Android team implemented the same functionality for the respective mobile platforms. Contracts were informally discussed between the back-end team and the other teams to determine what data the REST endpoints expected and what they provided.

The Android application was created using the standard company layered architecture (see Figure 6.2):

1. A **User Interface layer**, which contains both, the user views (forms) and the user work-flows around assembling the information required for service requests and the information provided with responses.
2. An **Application adapter layer**, which maps user requests from the user interface layer onto the infrastructure layer and service responses back onto response domain objects provided to the user interface layer.
3. An **Infrastructure layer**, which implements the application services API (also implemented on the server side) and maps the Java requests onto REST requests using data transfer objects (DTOs) and responses back onto Java responses. This layer is responsible for communication with external systems and persistent storage.
4. A **Domain layer**, containing objects from the user domain as objects encapsulating the request and response data. This layer does not have technical details like database connections and should be understandable to those who do not have technical knowledge.

Normally, contract validation should happen on the back-end (in this case, the AWS Lambdas), since it is the service provider for three different types of clients. But due to the following factors, this wasn't possible:

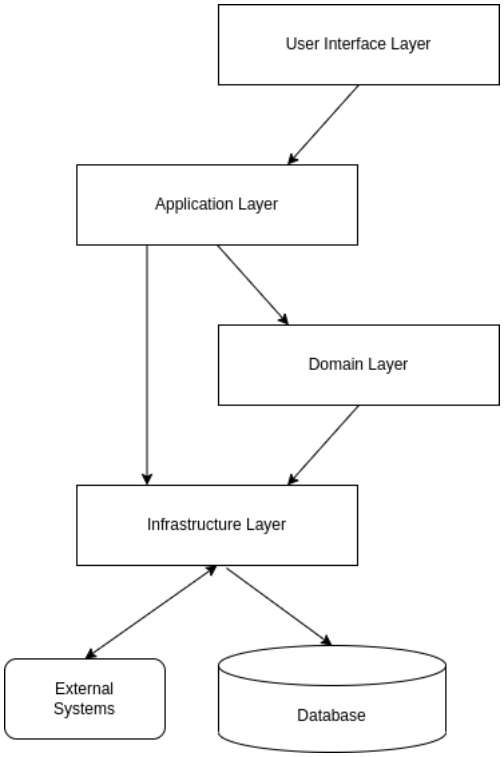


Figure 6.2: Layered architecture

- The AWS Lambdas are written in TypeScript. Our interceptor-generator is written in Java.
- Only the Android application was written in Java.

However, implementing contract validation via interceptors in the infrastructure layer of the Android application that communicates with the AWS Lambdas, does, in theory, have the same effect as if it were implemented on the back-end itself because the contracts are the same. The only difference is that the validation happens before the data leaves the client, instead of when it arrives at the server.

Unit testing was done using JUnit (junit.org) which is one of the most popular testing frameworks for Java, and Mockito (https://github.com/mockito/mockito) which is a popular mocking framework used to create mock objects ("stubs") of external dependencies in order to ensure that a component interacts with the dependency in an expected way. This allowed us to test each component in isolation. The unit tests were part of the build-process - each time the project was built, all the unit tests were run and had to pass successfully in order for the project to finish

building. Contracts were tested by calling components via their test interceptors by using valid and invalid data and ensuring that contract validation exceptions were thrown when invalid data was used, and no exceptions were thrown when valid data was used.

Integration testing was done by utilizing development environment, and QA environment which mirrors the production environment. This allowed us to do end-to-end testing to ensure that individual components behaved correctly in an integrated environment. Interceptors were switched on by default, which made it very easy to determine which components were problematic: since the unit tests already ensured that components work in isolation with valid data, most often contract validation errors occurred when users entered invalid data via the UI, where after efforts were taken to validate user input.

Figure 6.3 shows a screen-grab of a contract violation exception that occurred during integration testing.

```
Justus.masters.annotationprocessor.exceptions.PreconditionNotEnforcedException: request != null|postSurvey
  at com.client.mobile.domain.covidl9.Covidl9ApiInterceptor.postSurvey(Covid19ApiInterceptor.java: 170)
  at com.client.mobile.application.ApiService.initFromSettings(ApiService.java:90}
  at com.client.mobile.application.ApiService.<init>(ApiService.java:31)
  at com.client.mobile.presentation.ui.MainActivity.checkPushNotificationAction(MainActivity.java:145}
  at com.client.mobile.presentation.ui.MainActivity.onCreate(MainActivity.java : 32)
  at android.app.Activity.performCreate(Activity.java:8000)
  at android.app.Activity.performCreate(Activity.java:7984)
  at android.app.Instrumentation.callActivityOnCreatednstrumentation.java:1309)
  at android.app.ActivityThread.perfonnLaunchActivity(ActivityThread.java:3422)
  at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3601)
  at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:85)
  at android.app.servertransaction.TransactionExecutor.executeCallbacks(TransactionExecutor.java:135)
  at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:95)
  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:2066)
  at android.os.Handler.dispatchMessage(Handler.java:106) at android.os.Looper.loop(Looper.java:223)
  at android.app.ActivityThread.main (ActivityThread.java:7656) <1 internal call>
  at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(Runtimelnit.java:592)
  at com.android.internal.os.Zygotelnit.main(Zygotelnit.java:947)
```

Figure 6.3: Example Contract Violation

Since interceptors are automatically generated when the project is built, they are included in the production code that gets deployed. In production, we can then choose to use them or not. This is done by using a feature-flag: when the app starts up, it uses a REST call to get a list of features that are defined in the database on the back-end. These features are used to enable or disable certain functionality in the application, which allows us to do phased roll-outs of new functionality or only enable certain functions for users with certain rights. We used the same mechanism to enable/disable interceptors in production - when the new functionality was first rolled out, interceptors were switched on for a certain time to ensure everything worked properly.

	Experiment	Reference
Activities (screens)	11	8
REST API calls	30	26
Planned scope (in days)	30	30

Table 6.1: Experimental project and reference project

Thereafter, they were switched off.

To determine the impact of our approach, we compared the Covid-19 project (using a CDD approach) to a different project called Travel (which was developed using the traditional Agile approach). The Travel project allows users to book personal or business trips, upload digital copies of their passports and visas, notify the user if the destination is high-risk (Covid-19 infections or other reasons like unrest) and provide digital waivers for business trips. Although the Travel project is functionally completely different, it was developed on the exact same layered architecture depicted in Figure 6.2. We analyzed the requirements specification, UI interfaces, number of REST calls and project plans to ensure that the projects were as similar as possible in scope:

In both projects, the principles of a good design were followed:

- Responsibility localization. Each component is responsible for a single piece of functionality.
- Clean layers of granularity. The higher level workflow is understandable without knowing the details of the lower levels.
- Decoupling. The client-server relationship improves maintainability and flexibility. Clients are not dependent on a single service provider.
- Simplicity. Domain Driven Design is easy to understand and maintain.
- Architecture and technology neutral. Front-end applications can be written in any language and be able to communicate with the back-end systems via the REST protocol. Similarly, the back-end systems can be written in any server language and use any cloud provider service, and still be able to communicate with the front-end applications via the REST protocol.

The data used for analysis was obtained from Trello (https://trello.com), the bug-tracking software that the client uses for projects. Trello has a function to export the data related to a project into a comma-delimited text file. We then used a simple program to parse the file, and insert the data into a local MySQL database. This allowed us to filter the data using SQL statements. The filtered data was then pulled into Excel, which was used to do statistical analysis and render graphs (see Figure 6.4 and 6.5).

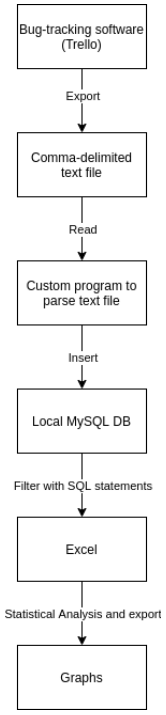
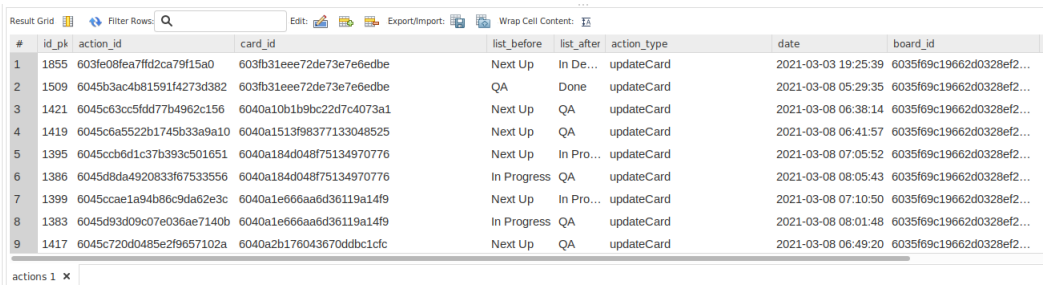


Figure 6.4: Analysis Process



Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

#	id_pk	action_id	card_id	list_before	list_after	action_type	date	board_id
1	1855	603fe08fea7ffd2ca79f15a0	603fb31eee72de73e7e6edbe	Next Up	In De…	updateCard	2021-03-03 19:25:39	6035f69c19662d0328ef2…
2	1509	6045b3ac4b81591f4273d382	603fb31eee72de73e7e6edbe	QA	Done	updateCard	2021-03-08 05:29:35	6035f69c19662d0328ef2…
3	1421	6045c63cc5fdd77b4962c156	6040a10b1b9bc22d7c4073a1	Next Up	QA	updateCard	2021-03-08 06:38:14	6035f69c19662d0328ef2…
4	1419	6045c6a5522b1745b33a9a10	6040a1513f98377133048525	Next Up	QA	updateCard	2021-03-08 06:41:57	6035f69c19662d0328ef2…
5	1395	6045ccb6d1c37b393c501651	6040a184d048f75134970776	Next Up	In Pro…	updateCard	2021-03-08 07:05:52	6035f69c19662d0328ef2…
6	1386	6045d8da4920833f67533556	6040a184d048f75134970776	In Progress	QA	updateCard	2021-03-08 08:05:43	6035f69c19662d0328ef2…
7	1399	6045ccae1a94b86c9da62e3c	6040a1e666aa6d36119a14f9	Next Up	In Pro…	updateCard	2021-03-08 07:10:50	6035f69c19662d0328ef2…
8	1383	6045d93d09c07e036ae7140b	6040a1e666aa6d36119a14f9	In Progress	QA	updateCard	2021-03-08 08:01:48	6035f69c19662d0328ef2…
9	1417	6045c720d0485e2f9657102a	6040a2b176043670ddbc1cfc	Next Up	QA	updateCard	2021-03-08 06:49:20	6035f69c19662d0328ef2…

actions 1 ×

Figure 6.5: Data in DB example

A simple SQL query to find all bugs related to the Android application, looks as follows:

```
select * from actions where action_type in ('updateCard', 'createCard')
and list_before is not null and list_after is not null
and actions.card_id in (
select card_id from cards
where (label like '%android%' or description like '%android%')
) order by card_id, date;
```

From here, we can just expand the where-clause to refine the filters and get the data we need, for example to find the number of bugs for a specific day, we just add:

```
... and date(date) >= date('2021-03-03') and date(date) <= date('2021-03-03')
```

(The date column contains timestamps, which include the time, so we need to convert the values to dates using the SQL *date()* function).

Similarly, we can use SQL statements to determine the first and last dates of specific activities to calculate their duration, for example:

```
select min(date) from actions where card_name = ....
select max(date) from actions where card_name = ....
```

and also when the first activity started and the last activity ended.

The filtered result set was then exported into a comma-delimited (CSV) file, and opened in Microsoft Excel which has built-in formulas that we used to do statistical analysis, and generate the graphs which are discussed in the next section.

6.2 Measurements

The measures listed in Chapter 3 were used to quantify the impact of our approach, versus the 'normal' approach:

6.2.1 Number of man-hours to develop

To determine this measure, we calculated the number of days between the date of project kick-off, and the date that the final build was deployed to production.

The number of hours were very similar between the project with contracts and interceptors, versus the project without contracts and interceptors. This result shows that the introduction of contracts does not significantly impact the duration of a project.

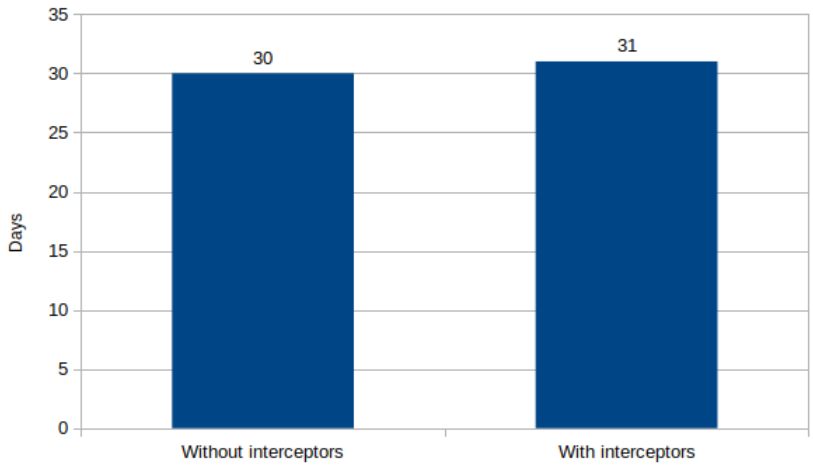


Figure 6.6: Man-hours to develop

6.2.2 Bug Density

To determine this measure, we had to divide the size of the module by the number of bugs. In order to gain more insights, we decided to also split the types of bugs into three categories:

- Logic - these are errors that occurred in the lower-level Java components.

- Integration - errors that were discovered when testing the user-interface, as well as external components.
- Cosmetic - All errors relating to non-functional items (look-and-feel, like layout, design, spacing, colors).

To determine the size of a module, the following Linux command was used to determine the total number of lines of code from the specified root folder, including all sub-folders:

```
find /path/to/root/folder -name '*.java' -type f -exec cat {} + | wc -l
```

We then divided this number by the total numbers of logic, integration and cosmetic bugs.

There was a significant improvement in the bug density of the project that utilized contracts and interceptors, compared to the project without them. This was especially true for integration and logic bugs. Cosmetic bugs are high in both projects due to the fact that the client is notorious for always requesting layout, label and graphical changes, and these changes are logged as bugs. Cosmetic bugs are shown here for completeness, but they are not used in the measurements since unit and integration testing (and hence test-interceptors) would only detect functional errors.

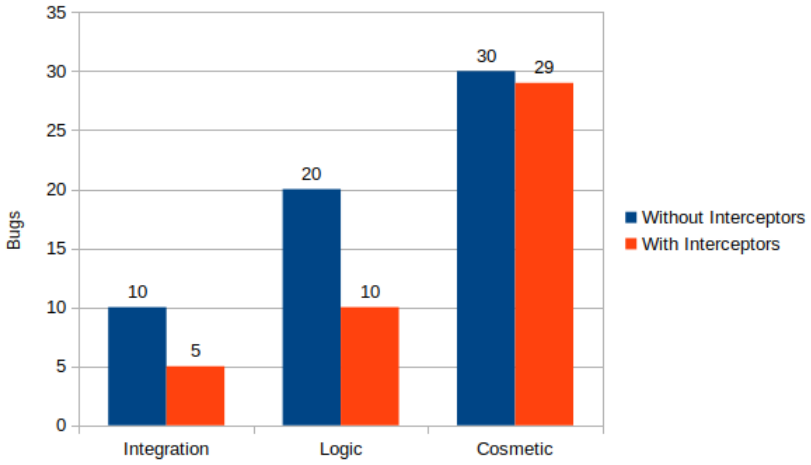


Figure 6.7: Bug Density

6.2.3 Correctness of the produced software

To determine this measure, we calculated the number of bugs from the date that the first bug was logged, until the date that the last bug was logged, and divided the total by the number of days between those two dates.

The average bugs per day was less in our CDD approach versus the project using the traditional approach.

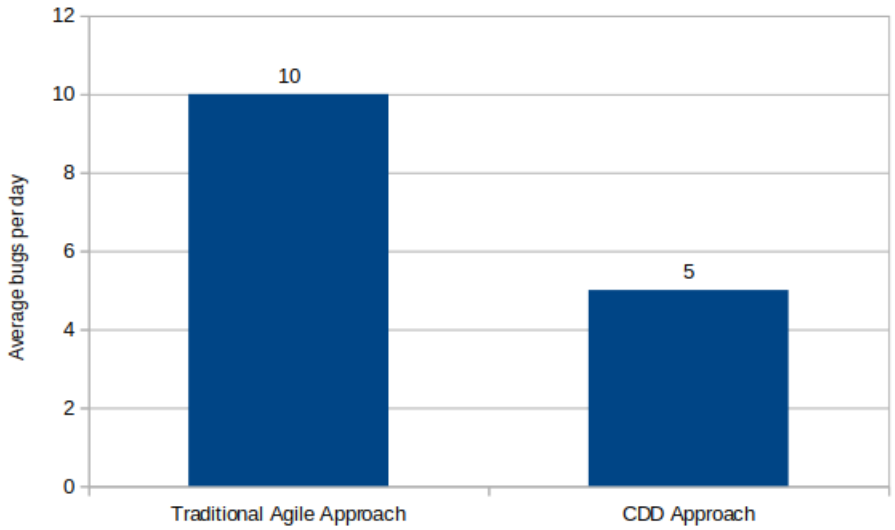


Figure 6.8: Correctness of the produced software

6.2.4 Reusability

Adding contracts to components greatly improved reusability due to the following factors:

- Developers can easily determine what kind of values a method expects, and what kind of values it returns by looking at the annotations.
- As long as the formalized requirements (contract) stays the same, developers know with confidence what functionality a component provides for them.

6.2.5 Simplicity

To determine if the software is more or less complex as a result of utilizing contracts, we examine and compare the Cyclomatic Complexity, Halstead

Volume and Maintainability Index of the software from this project against the project without contracts and interceptors. To calculate the metrics, we used Source Meter (https://www.sourcemeter.com) which is a popular free and open-source tool for source code analysis. It was invoked using the following command:

```
./SourceMeterJava -resultsDir=<dir_to_save_results> -projectName=<P1...Pn>
-runAndroidHunter=false -runVulnerabilityHunter=false -runFaultHunter=false
-runRTEHunter=false -runDCF=false -projectBaseDir=<root_of_project_to_analyze>
-runFB=false -runPMD=false
```

This command produces a number of result output files, among which is a comma-delimited file containing the Cyclomatic Complexity, Halstead Volume and Maintainability Index (among others), calculated per method.

	A	B	C	D	E	F	G	H	I	J
1	Project	Method	Complexity	Halstead Difficulty	Halstead Effort	Halstead Length	Halstead Vocabulary	Halstead Volume	Maintainability Index	
2	Android	void reset()	1	1	4.75489	3	3	4.75489	162.662	
3	Android	void reset()	1	1	4.75489	3	3	4.75489	162.662	
4	Android	String stringFromJNI()	1	1.5	12	4	4	8	159.957	
5	Android	void disconnectCamera()	1	1.5	12	4	4	8	159.957	
6	Android	void closeCamera()	1	1.5	12	4	4	8	159.957	
7	Android	String getLibraryList()	1	2	23.2193	5	5	11.6096	158.02	
8	Android	long create_0()	1	2	23.2193	5	5	11.6096	158.02	
9	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	
10	Android	String getBuildInformation_0()	1	2	23.2193	5	5	11.6096	158.02	
11	Android	double getTickFrequency_0()	1	2	23.2193	5	5	11.6096	158.02	
12	Android	int getNumThreads_0()	1	2	23.2193	5	5	11.6096	158.02	
13	Android	int getNumberOfCPUs_0()	1	2	23.2193	5	5	11.6096	158.02	
14	Android	int getThreadNum_0()	1	2	23.2193	5	5	11.6096	158.02	
15	Android	long getCPUTickCount_0()	1	2	23.2193	5	5	11.6096	158.02	
16	Android	long getTickCount_0()	1	2	23.2193	5	5	11.6096	158.02	
17	Android	long n_Mat()	1	2	23.2193	5	5	11.6096	158.02	
18	Android	long TickMeter_0()	1	2	23.2193	5	5	11.6096	158.02	
19	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	
20	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	
21	Android	long BFMatcher_1()	1	2	23.2193	5	5	11.6096	158.02	
22	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	
23	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	
24	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	
25	Android	long FlannBasedMatcher_0()	1	2	23.2193	5	5	11.6096	158.02	
26	Android	long create_0()	1	2	23.2193	5	5	11.6096	158.02	
27	Android	long create_1()	1	2	23.2193	5	5	11.6096	158.02	

Figure 6.9: Source Meter example output

We then calculated the average value of each metric over all methods for each of the two projects. We used the built-in Excel function (which adds up all the values and divides the total by the number of values) to achieve this. The results show that the metrics are the same between the two projects, which indicate that utilizing contracts and interceptors do not influence code complexity.

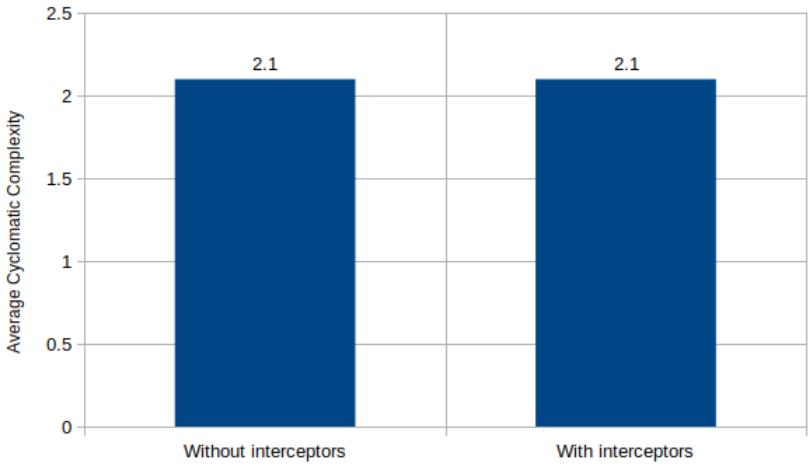


Figure 6.10: Cyclomatic Complexity

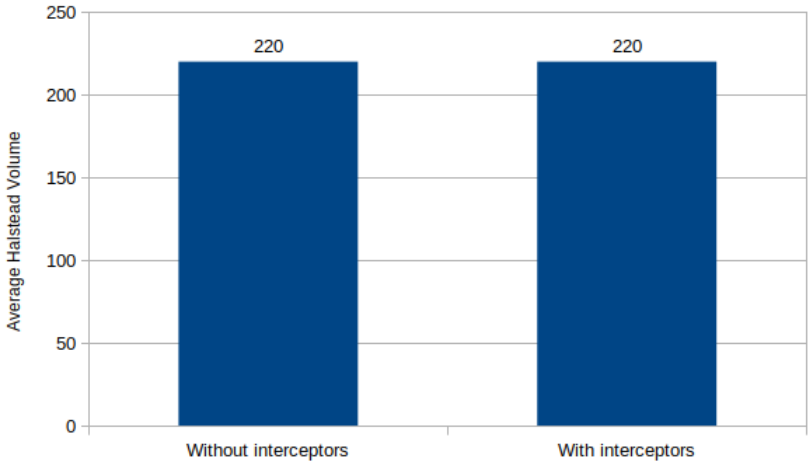


Figure 6.11: Halstead Volume

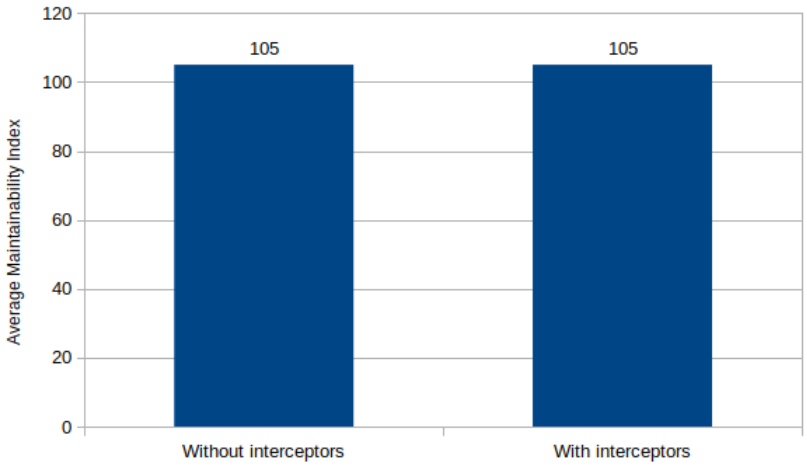


Figure 6.12: Maintainability Index

6.2.6 Certifiability

Although this project does not have to be certified against any official standards, it is important to note that our contract driven approach does address the requirements for certifiability as stipulated in formal standards like DO-178:

- **Precise** - Our approach allows for precise control and monitoring of test inputs and code execution and such testing can be performed on components that are functionally isolated by utilizing interceptors.
- **Formalized Requirements** - Requirements are formalized in annotations.
- **Traceability** - Any pre-condition, post-condition or invariant violation can be traced back to the specific requirement stipulated in the annotation of the relevant component.

6.2.7 Test Quality

In Chapter 3, we discussed the formulas to use for calculating test quality. To calculate the average complexity of the projects, we used the cyclomatic complexity values, rounded to the nearest decimal (shown in Table 6.2):

	CDD Approach	Traditional Approach
Complexity	2	2

Table 6.2: Complexity values of the experiment and reference project

	CDD Approach	Traditional Approach
Total Errors	15	30
Errors captured by tests	0	0
Incorrect requirements	5	10
Relative undetected errors	10	20

Table 6.3: Errors logged for the experiment and reference project

This resulted in a value of 2 for the average complexity of the two projects. For the rest of the parameters of the formula, we scrutinized the bugs that were logged, and aggregated them according to total errors logged (excluding cosmetic errors), errors captured by tests, and errors due to incorrect requirements, shown in Table 6.3. We found that there were no errors logged that were captured by tests, because any such errors were fixed immediately by the developers when the tests were run and a test failed.

The results obtained for testing quality (see Figure 6.13)show that there are still a number of errors in the project utilizing our framework. After careful investigation, we have found that this is due to the test data not being complete: a manual process for generating test data was used, which did not cover all equivalence partitions and boundary values. Testing is normally a tedious activity which is often neglected by developers, and is usually the first activity to suffer when deadlines are tight. One of the most time-consuming problems is generating proper test data that covers most scenarios. The team was not trained to systematically generate test data and still used the "old way" which takes random samples for which the outcomes are known. We purposefully did not alter this aspect of the process (by introducing a systematic approach to test data generation) due to time constraints.

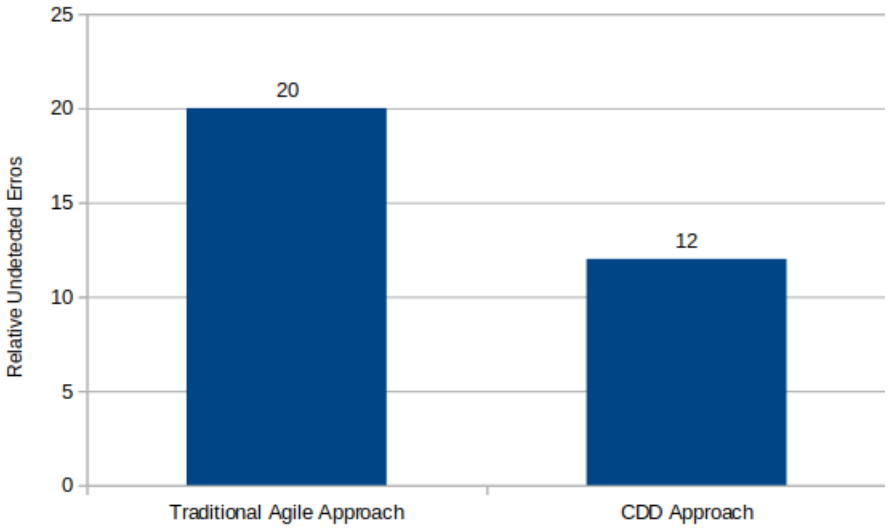


Figure 6.13: Test Quality

Tests can be improved by using a systematic way of, either manually or via an automated approach, generate complete test data in the sense that all equivalence partitions are covered, and boundary values are covered. With a contract driven approach where contracts are formalized upfront and tests and test data are automatically generated from that, testing will not be impacted by short deadlines. Automated test data generation has been proven to be effective in this regard. [33, 9] This is further discussed in Conclusions and Future Work.

The results did show a decrease in errors due to requirements in the project utilizing CDD, which can be attributed to the test interceptors. The errors that were logged for incorrect requirements in the project using interceptors, can be ascribed to inexperience with the CDD approach and utilizing annotations - in these cases a requirement was either forgotten, or a contract was incorrectly captured. However, the test interceptors were functioning correctly, since they were able to detect errors in all the cases where contracts were formulated correctly. The table below shows how the errors are attributed:

During Integration testing, where components were tested in their integrated environments and real data is used, interceptors made a big difference with fault-finding and troubleshooting thanks to the information contained in the contract violation exceptions.
The ability to easily switch the interceptors on or off, helped during oper-

Errors due to incomplete test data	10
Errors due to incomplete contract specification	5
Errors due to incorrect test interceptor generation	0

Table 6.4: Reasons for errors

ational testing where components were tested in the context of normal system operation. Components were deployed onto a development environment which closely resembles the production environment in terms of hardware and other components. Interceptors were always switched on since new components were deployed regularly, and contract violation exceptions immediately showed when a new component was deployed or updated that did not adhere to the contracts. The exact same code was then deployed to the production environment, only with different configuration settings to switch interceptors off in order to maximize the speed of the processes. However, should a production issue occur, it would be trivial to switch interceptors on again in order to determine if the issue is because of a contract violation.

6.2.8 Usability

Our light-weight CDD approach has proven very effective, as can be seen most notably from the significant reduction of integration bugs, as well as no significant increase in the man-hours of the project. By not altering the traditional approach too much, the comfort zone of the developers was not affected.
The tool-set that we have developed was also very effective. Other than the initial set-up required, everything happened automatically without any intervention or extra steps required from the developers.

This chapter is the culmination of the experimental design, CDD methodology and use of the tools we designed. We compared the results obtained by using our approach versus the traditional company software development approach. Specifically, we looked at the number of man-hours to develop, bug density, correctness of the produced software, reusability, simplicity, certifiability, test quality and usability of the toolset.
In the next chapter, we look at conclusions and future work.

Chapter 7

Conclusions and future work

In this chapter, we discuss the conclusions that we have reached from the study. We also discuss points that we discovered that might prove useful to explore in future work.

7.1 Conclusions

This empirical study assesses the impact of embedding tool-supported Contract Driven Development (CDD) within a basic agile software development process, the quality attributes of the outputs of the process and those of the development process itself. The core of the tool support is a test-interceptor[1] generator which processes contract annotations to generate a contract specific test interceptor validating whether a wrapped application component which implements the contract-annotated interface meets its contractual obligations in the context of processing its service requests.

A team was asked to develop two applications with similar complexity, with the one using the traditional agile software development process used by the company, and the other following a minimally modified process which incorporated component contract specification and test-interceptor generation. This study assesses whether the following propositions hold true:

1. Requiring developers to formalize functional requirements prior to implementation improves code quality including:
    - reduced bug density, and
    - improved reusability.

[1] The test-interceptor is really a contract validation interceptor.

	Our project	Reference project
Man-hours to develop	31	30
Average bugs per day	5	10
Integration errors	5	10
Logic errors	10	20

Table 7.1: Summary of results

	Our project	Reference project
Cyclomatic Complexity	2.1	2.1
Halstead Volume	220	220
Maintainability Index	105	105

Table 7.2: Summary of code metrics

2. The extra effort and cost incurred by formalizing functional requirements into services contracts can be set off against the reduction in work due to test logic (test-interceptor) generation and reduced efforts around bug fixes.

3. Requiring software developers to specify pre- and post-conditions in code using a constraints specification language based on the programming language they are using does not increase the skills requirements for developers significantly.

This study found that:

- The number of both, logic and integration errors were halved.
- The code complexity as measured by *cyclomatic complexity*, *Halstead Volume* and a *Maintainability Index* were unaffected.
- The impact on cost (number of man-hours required to develop the solution) was insignificant (around 3%).

One can also reasonably expect reusability to improve as developers can inspect the component contracts in order to assess whether they meet their needs. This requires a more long-term study in order to assess whether the reuse by subsequent projects of code developed in the CDD project was higher than that of the project which followed the traditional agile software development process.

We have found that it is easier to get buy-in from the parties involved to try a new approach, if the software development process is not too radically

different from what they are used to. When people are used to a specific methodology and deadlines are tight, the introduction of a completely different methodology causes stress and negativity. One of the major benefits of our approach is that it integrates very well with the standard company Agile methodology. Since at its core, the Agile methodology embraces continuous incremental improvement through small and frequent releases, contracts were updated regularly and thanks to the contracts being encoded as annotations, they were immediately validated during unit and integration tests. This resulted in more thorough validations (where previously it was easy to miss or forget a validation when they were implemented manually).

Positive outcomes of the experiment are:

- The benefits of thinking in terms of contracts - even the developers working on other parts of the project benefited from knowing exactly what a component expects and the results it returns;
- Boolean expressions and annotations are familiar concepts - utilizing these features to express contracts is an intuitive process;
- Setting up the tool-set that generates interceptors is easy and fast - it involves updating a configuration file and importing the relevant libraries. This is only necessary once when the development environment is set up;
- The automatic generation of the test-interceptors requires NO extra effort - it is a pre-compilation step that is executed automatically each time the project gets built;

To convince managers, as well as the development team for our study to be performed on actual software development projects, the following strategy has proven effective:

- Divide and conquer - engage with small groups or individual team member to discuss the merits of Contract Driven Development informally (for example during coffee breaks). It is easier to convince small groups or individuals during relaxed discussions than trying to convince a large group during a formal design session where peer pressure and ego are involved.
- Get the buy-in of the main technical authority, like the chief architect. Such a person normally has the final say about the technologies used in the projects, and it is important to address any concerns they might have about introducing new concepts into a project.

- Use examples from the actual domain, and include frameworks and architectural designs (like Domain Driven Design) that are being used in current projects. It helps developers understand new concepts if it is being implemented with familiar concepts. Also note the suggestions and comments from the other developers, often times it might be valuable.

- Start small with simple examples. This instills the confidence that contracts are not that complex to implement.

- Do a proof of concept with an existing component, by discussing and adding contracts to it.

- Minimize the impact on existing components and systems, for example if dependencies for contract processing are missing, the component should still function normally.

We have found the Java programming language to be very suitable for contract driven development - language features like annotations and reflection were extensively used to define contracts and generate interceptors. The fact that interceptors are generated as a pre-compilation step also makes the process easy and seamless from the perspective of a developer.

A negative outcome was that the unit tests still had to be written manually - the point was made that if interceptors can be generated automatically, it is technically possible to generate unit tests as well. This will be considered for future work.

7.2 Future Work

1. It is expected that the specification of services contracts improves reusability as it is the contract can be used to verify whether the component fulfills one's requirements. Whether Contract Driven Development (CDD) actually does result in a higher level of service reuse was, however, not empirically assessed in this study.

2. In this project, only the test interceptors were generated from the contract specifications, but the unit tests creating and populating the test data structures for the different test scenarios were hand coded. The efficiency of CDD can be further improved by generating templates

unit tests which contain the boilerplate code of preparing up the test-interceptor wrapped application component under test and creating the data structure specifications for the test data

3. The majority of cases where the test-interceptor did not catch a functional error were found to be due to software developers not using a systematic approach to test data generation. Future work could look at augmenting CDD with automated test data generation. This could further improve both, the process efficiency and code quality.

4. To reduce the probability of incorrect or incomplete contract specification, one can incorporate peer reviews of contracts in the development process.

5. This project studied only the use of test interceptors for unit testing where lower level service providers are mocked out with mock objects in order to test the component in isolation. Future projects can additionally use the generated test interceptors for

   (a) integration testing across levels of granularity where components are tested using the actual lower level service provider components instead of mock objects,

   (b) operational testing where components in the system are wrapped with test interceptors which verify that each of the wrapped components addresses its functional requirements in the context of normal system usage.

   (c) external service provider oversight where contracts are specified for external service providers and the adapters are wrapped with test interceptors in order to assess that these external service providers provide their services as per contract.

Bibliography

[1] QA and testing budget allocation 2012-2019 | Statistic, May 2018. [Online; accessed 7. May 2018].

[2] agilealliance.org. Acceptance test driven development (atdd), 2016.

[3] H. Akhtar. What is bdd? (behavior-driven development).

[4] K. Beck, M. Fowler, R. C. Martin, et al. The agile manifesto. Technical report, 2001.

[5] H. Belhaouari and F. Peschanski. Automated Generation of Test Cases from Contract-Oriented Specifications: A CSP-Based Approach. In *HASE '08: Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 219–228, Washington, DC, USA, 2008. IEEE Computer Society.

[6] J. Bloom. Five Reasons You MUST Measure Software Complexity, May 2018. [Online; accessed 10. May 2018].

[7] K. Brush and V. Silverthorne. Agile software development.

[8] R. Butler. What's next for test-driven software development.

[9] G. Candea and P. Godefroid. *Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances*, pages 505–531. Springer International Publishing, Cham, 2019.

[10] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. 2002.

[11] M. Clark, B. Salesky, and C. Urmson. Measuring software complexity to target risky modules in autonomous vehicle systems. In *AUVSI North America Conference*, Dec 2008. [Online; accessed 10. May 2018].

[12] M. Donat. Automating formal specification based testing. 1997.

[13] N. El-Fouladi, A. Abran, and Y. Ettki. Automatic test case generation: A systematic review. *Journal of Software Engineering and Applications*, 7(6):345–357, 2014.

[14] S. J. Galler, M. Weiglhofer, and F. Wotawa. Synthesize it: From design by contract to meaningful test input data. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 286–295, Sep. 2010.

[15] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[16] M. Henning. The rise and fall of corba.

[17] T.-W. Huang, C.-W. Chen, Y.-H. Chen, and R.-H. Huang. Real-time middleware for distributed collaborative virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):714–723, 2018.

[18] M. Huo, J. Verner, M. Ali Babar, and L. Zhu. How does agility ensure quality? 06 2004.

[19] InfoQ. Contract-driven development: A technical introduction.

[20] ISO25000. Software and data quality.

[21] A. Jain and V. Panchal. A comparative study of manual testing and automation testing. *International Journal of Advanced Research in Computer Science and Software Engineering*, 5(7):978–981, 2015.

[22] T. Janca. Contract-first api development: A closer look.

[23] J.-M. Jezequel and B. Meyer. Design by contract: the lessons of ariane. *Computer*, 30(1):129–130, Jan. 1997.

[24] R. Klopper, S. Gruner, and D. Kourie. Assessment of a framework to compare software development methodologies. In *SAICSIT*, 2007.

[25] R. Kramer. icontract - the java(tm) design by contract(tm) tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, pages 295–, Washington, DC, USA, 1998. IEEE Computer Society.

[26] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. 1992.

[27] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 425–434, New York, NY, USA, 2007. ACM.

[28] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[29] B. Meyer. Applying "design by contract". *Interactive Software Engineering*, 1992.

[30] C. Nebut, F. Fleurey, Y. L. Traon, and J. Jézéquel. Requirements by Contracts allow Automated System Testing. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 85, Washington, DC, USA, 2003. IEEE Computer Society.

[31] R. Pandey and A. Mishra. Automated vs. manual testing: Approaches and challenges. *International Journal of Engineering Research and Applications*, 3(4):1364–1369, 2013.

[32] J. Poulin. Measuring software reusability. 01 2001.

[33] C. Ramamoorthy, S.-B. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, 1976.

[34] I. Robinson. Consumer-driven contracts: A service evolution pattern, 2006.

[35] B. Skurrie. Pact: Getting started, 2020.

[36] slashdot.org. Why is design by contract not more popular, 2007.

[37] F. Solms. *URDAD for System Design*, 2018.

[38] F. Solms and D. Loubser. Urdad as a semi-formal approach to analysis and design. In *Innovations Syst Softw Eng*, 2010.

[39] spring.io. Spring cloud contract, 2020.

[40] P. Srivastava and V. Sehgal. A comparative analysis of manual testing vs. automated testing. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(11):212–216, 2013.

[41] Stackoverflow.com. Why is design-by-contract not so popular compared to test-driven development?, 2018.

[42] Swagger. Contract-first development with openapi.

[43] C. Technologies. Challenges of microservices: Contract management.

[44] ThoughtWorks. Contract testing: An overview.

[45] T. Tulka. The benefits and challenges of contract-driven development.

[46] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 16(2):201–229, 2011.

[47] K. D. Welker, P. W. Oman, and G. G. Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9(3):127–159, 1997.