THE IMPACT OF USING A CONTRACT-DRIVEN, TEST-INTERCEPTOR BASED SOFTWARE DEVELOPMENT APPROACH

Justus Posthuma[1], Fritz Solms[2] and Bruce W. Watson[3]

[1]Centre for AI Research, School for Data-Science & Computational Thinking,
Stellenbosch University, Stellenbosch, South Africa
[2]Department of Information Science,
Stellenbosch University, Stellenbosch, South Africa
[3]Centre for AI Research, School for Data-Science & Computational Thinking,
Stellenbosch University, Stellenbosch, South Africa

ABSTRACT

Contract Driven Development formalizes functional requirements within component contracts. The process aims to produce higher quality software, reduce quality assurance costs and improve reusability. However, the perceived complexity and cost of requirements formalization has limited the adoption of this approach in industry.

In this article, we consider the extent to which the overheads of requirements formalization can be netted off against the reduced quality assurance costs arising from being able to auto-generate functional test interceptors from component contracts. Test-interceptors are used during testing to verify that component contracts are satisfied. In particular, we investigate the impact of contract-driven development on both the quality attributes of the software development process and the quality of the software produced by the process.

Empirical data obtained from an actual software project using contract-driven development with test interceptor generation is compared to that obtained from similar projects that used a traditional software development process with informal requirements and manually written functional tests.

KEYWORDS

Software and its engineering, Software development methods, Software implementation.

1. INTRODUCTION

Contract driven development (CDD), also known as Design by Contract (DbC), Contract Programming and Programming by Contract, is a development methodology that aims to improve the quality of the produced software by formalizing functional requirements in the form of component contracts [10]. The benefits of CDD have been demonstrated [6], yet the methodology has never really gained significant traction. One of the main reasons for this is the perception that the added effort and complexity of formalizing component requirements in component contracts is tedious and does not deliver "enough bang for your buck" [14, 17].

David C. Wyld et al. (Eds): DMML, SEAS, ADCO, NLPI, SP, BDBS, CMCA, CSITEC - 2022
pp. 39-58, 2022. CS & IT - CSCP 2022 DOI: 10.5121/csit.2022.120704

There are a number of modern software development methodologies that are used by companies in their efforts to produce quality software. 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 Behaviour 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 behaviour by using a domain specific language (DSL). The DSL uses natural-language constructs that describe the required behaviour and expected outcomes, and is converted into executable tests by specialised software. BDD is also sometimes referred to as Story Test Driven Development (STDD), Acceptance Test Driven Development (ATTD) or Example Driven Development (EDD). [19]

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 specialised 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.

In this study we investigate whether the cost benefits of generating functional test logic from component contracts is sufficient to nett off the cost of requirements formalization, whilst preserving the benefit of improving the quality of the produced software. In particular, we consider an approach where the process of formalizing requirements is simplified by encoding component contracts within the programming language used for the project (in this study we used the *Java* programming language) and generate test interceptors, which monitor contract compliance of wrapped components. These test interceptors are used for unit testing, integration testing and operational testing.

Empirical data is obtained from an actual software development project by using a contract-driven development methodology to specify component contracts. We empirically assess 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), *development reliability* (the ability of the process to reliably produce software at a given rate and quality), *process usability* (the ease with which CDD can be introduced) and *process scalability* (the ability of introducing CDD to large development teams).

We also assess 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 that are more reusable), and simplicity are considered.

Section 2 examines related work and other frameworks that utilizes CDD. In Section 3 we discuss the details of our solution and exactly how test interceptors are implemented and how they function. Section 4 describes the results we obtained by using our solution in a real-world software developments project, and in Section 5 we discuss the conclusions we have drawn from the results. Finally, in Section 6 we look at future work.

2. RELATED WORK

Nebut et al describe a requirement-based testing technique that leverage use-cases in order to create functional and robustness test objectives [11]. 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, the authors 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.

The authors 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. The authors found that selecting a set of test paths (test objectives) that ensures that each use case is instantiated by at least one path and that 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 that both test processes (test functions) and test scenarios (test data) are generated and that the tests are generated from a semi-formal 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:

- Lack of 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 paper aims to manage scalability through automated contract-based test function generation for services across levels of granularity.

- Incompleteness: The authors 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, the authors 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 that need to be added to achieve completeness.

- Limited 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 approach analysed in this study 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.

Evolving service providers and consumers present a number of challenges, particularly when they change their document schemas (contracts), Robinson has found. He identifies two strategies for mitigating such issues: performing "just enough" validation of received messages and adding schema extension points [12]. Robinson 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 preserve 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.

By observing contracts between providers and consumers, Robinson expresses the following insights: The business function capabilities of a service provider are expressed by a provider contract in terms of the collection of exportable elements that are necessary to support that functionality. The main characteristics of provider contracts are:

- Closed and complete: Provider contracts show the business function capabilities of a service as a complete set of exportable elements available to consumers. As such they are complete and closed regarding the functionality of the system.

- Singular and authoritative: Provider contracts are singular and authoritative in their expression of the business functionality of the system.

- Bounded stability and immutability: Provider contracts are stable and immutable for a bounded period and/or locale. They typically use 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. The characteristics of such a contract are:

- Incomplete and open: A consumer contract is incomplete and open regarding the business functionality of the system. It shows a subset of the capabilities of the system in terms of the expectations that the consumer has of the contract of the provider.

- Non-authoritative and multiple: Each consumer contract is non-authoritative regarding the total set of contractual obligations placed on the provider, and they are multiple in relation to the number of service consumers.

- Bounded stability and immutability: Similar to provider contracts, consumer contracts are valid for a particular location and/or period of time.

Robinson concludes that consumer contracts, by showing and asserting expectations of a provider contract, allow us to know precisely which parts of the provider contract presently support business value by the system, and which parts do not. He further suggests that services might benefit from being specified in terms of consumer contracts from the start. In this way, provider contracts emerge to meet the demands and expectations of consumers - consumer-driven contracts or derived contracts.

The characteristics of Consumer-driven contracts are:

- Complete and closed: A consumer-driven contract is complete and closed with respect to the complete set of functionalities required of it by its existing consumers. The mandatory set of exportable elements is represented by the contract, which is required to support consumer expectations for the period in which those expectations are required by their parent applications.

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

- Bounded stability and immutability: In respect of a particular set of consumer contracts, a consumer-driven contract is stable and immutable. That means the validity of a consumer-driven contract can be determined according to a specified set of consumer contracts, thereby binding the backwards- and forwards-compatible nature of the contract in space and time. The compatibility of a contract remains immutable and stable for a specific set of consumer contracts and expectations, but it is subject to change as expectations change.

Robinson identifies two significant benefits of consumer-driven contracts in terms of evolving services. Firstly, the focus is on the delivery and specification of functionality around key business value drivers - the value if a service is determined by the extent to which 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: in the context of a closed community or a single enterprise (an environment in which providers can exert influence over how consumers establish contracts with them), the consumer-driven contract pattern is applicable. Consumers and providers must accept, adopt and know about an agreed-upon set of channels and conventions. This adds a layer of protocol dependence and complexity 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 the specification of a service provider is driven by consumer contracts, the conceptual integrity of that service provider could be compromised. Services are identifiable, discreet and reusable business functions whose integrity should not be undermined by demands that fall outside their mandate.

Pact [13] and Spring Cloud Contract [16] are popular testing frameworks that utilise Consumer Driven Contracts.

Belhaouari et al created an experimental platform known as Tamago-Test for software analysis and automated testing [2]. 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 pre-conditions are used to infer the relevant values to provide as input parameters to methods, and the post-conditions are then used as natural oracles (an oracle determines whether the test results are correct [3]). The creation of values for method input parameters which are correct with regards to the specification, corresponds to a constraint-satisfaction problem (CSP) [8]. 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 [4] of assertions and is included in the pre-conditions. When the CSP must instantiate a variable, it draws a random value from its reduced domain. Finally, the CSP architecture the authors 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 authors, 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 realise the contract between these two parties.

The specification language is similar to the grammar of assertions in Java Modelling Language (JML) and the authors 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 (pre-conditions, post-conditions and invariants). It also includes descriptions of service behaviours based on finite-state automata with conditional transitions.

The authors 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 analyse 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 behaviour. Using the effective pre-conditions, invariants and the guard of the predecessor transition in the behaviour automaton, the constraints to be enforced are partially evaluated. 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.

Leitner et al recognizes that unit testing is a resource-intensive and time-consuming activity [9]. 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 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 monitors program executions and, when a failure occurs, Cdd detects the last safe state and takes a snapshot of this state. It then recreates this snapshot, which serves as the starting point of the extracted test case. Cdd determines the time to take this snapshot so that it is early enough for the state not to be infected but also late enough to reduce execution time. Furthermore, in order to make the test case more robust relating to system change, the snapshot excludes that part of the state that is not relevant for reproducing the failure.

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.

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 [7]. 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 formalized functional requirements in the form of three specific comment paragraph tags:

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

iContract is run as a pre-processor over annotated source code files. It instruments the methods in these files with assertion check code that enforces the specified functional requirements. In the background, a repository of contract related information about the classes, interfaces and methods is built up. This information is used to support subcontracting, which propagates functional requirements along interface-implementation, multiple interface-extension, class-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 functional requirements are being enforced by means of automatically generated specification checks. Despite being annotated with functional requirements, the original files remain fully compliant to standard Java due to the annotations being a part of the (optional) comment paragraphs.

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.

3. IMPLEMENTATION

A common approach to requirements analysis and design is incremental refinement of requirements and design across levels of refinement or granularity [15]. At each level of refinement, one identifies the services required to assess the pre-conditions and address the post-conditions, and categorize them to responsibilities, which, in turn are assigned to components representing responsibility domains.

We represent component contracts in Java as Java interfaces, which are annotated with the component's functional requirements, i.e., the pre- and post-conditions and, in the case of stateful components, invariance constraints specifying enforced symmetries. A class implementing the Java interface is required to meet the component contracts and will be tested against the functional test logic generated from the component contract.
From component contracts we generate test interceptors, which are interface compatible with the component and which, in the context of service provision, verify that the wrapped component fulfils the functional requirements specified in the component contract. In particular, test-interceptors intercept service requests in order to verify that:

1. if all pre-conditions are met, the service is provided (no exception is raised) and all post-conditions hold after service provision;

2. if one or more pre-conditions is not met, that an exception specified for one of the pre-conditions that are not met is raised;

3. that all invariant constraints hold when the service is requested (if not, then the system was in a broken state already); and

4. that all invariance constraints hold after service provision.

In order to achieve this, the generated test interceptors perform the following steps:

1. assess any invariance constraints and raise an invalid state exception (which is not a component error) when any invariance constraint is violated;
2. assess and store the truth value of each pre-condition on service request;

3. delegate, within a try block, the request to the underlying component for processing;

4. upon catching an exception:

    a. verify that the pre-condition associated in the component contract with the caught exception did not hold upon service request;

5. if no exception has been caught, verify that:
    a. all pre-conditions were met, and that
    b. all post-conditions hold.
6. in either case that all invariance constraints hold.

Note that invariance constraints fall away if one follows a service-oriented analysis and design method [15]. In this case components are stateless and serve solely to package services within responsibility domains.

Some programming languages like Eiffel natively support the concepts required for contract specification, i.e., pre- and post-conditions and invariance constraints. Other languages like Java, C# and Python natively support language extensions via mechanisms like annotations and attributes. For programming languages that do not provide a syntax for extending the language one may need to either use an external framework for this purpose, or embed contract annotations within comments. In Java, we define the concepts and syntax for specifying functional requirements as pre-condition, post-condition, and invariance annotations. These annotations are processed using custom annotation processors that are bound into the compilation step.

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

```
@Invariant(constraint="invariant boolean expr")
public interface the_interface
{
  @Precondition(constraint = "boolean expr //pre-assessment //",  raises =
Exception.class)
  ... repeat for multiple pre-conditions
  @Postcondition(constraint = "boolean expr"}
  ... repeat for multiple post-conditions
  public returntype methodName(param1, param2, ...) throws Exception;
}
```

The pseudo-code of the interceptor class that is generated from the annotations, look as follows:

```
public class className_TestInterceptor implements the_interface {
  private the_interface counterpart = null;

  public className_TestInterceptor(the_interface counterpart){
    this.counterpart = counterpart;
  }

  /*
  Test the invariants
  */
  private validateConsistency() throws ExceptionRaisedDuringInvariantCheck,
InvariantViolatedError {

    boolean invariantHolds = false;
    try {
```

```
            invariantHolds = invariant boolean expr;
    }
    catch (Exception e){
            throw ExceptionRaisedDuringInvariantCheck("Message","<invariant boolean
expr>");
    }
    if (!invariantHolds) {
            throw InvariantViolatedError("Message","<invariant boolean expr>");
    }

  }

  @Override
  public returntype methodName(param1, param2, ...) throws Exception {
    /*
    Evaluate the pre-conditions
    */
    boolean _pre_1 = boolean expr == true;
    ... repeat for every pre-condition

    /*
    Evaluate the pre-assessments
    */
    int _preAs_1 = pre-assessment expression;
    ... repeat for every pre-assessment

    /*
    Validate the invariants before the method call
    */
    validateConsistency();

    returnValue = 0;
    try {
      /*
      Call the wrapped method of the counterpart component to be tested
      */
      returnValue = counterpart.methodName(param1, param2, ...);

      /*
      If method was implemented correctly, it would have thrown exceptions if any of the
pre-conditions were false.
      */
      if (!_pre_1){
        /*
        The method above did not throw an exception but pre-condition 1 is false.  This
means there is a problem with the implementation of the method.
        */
        throw new PreconditionNotEnforcedException("boolean expr", "methodName");
      }
      ... Repeat check for all pre-conditions.

      /*
```

```
    Evaluate post conditions and their pre-assessments
    */
    if (!(post-condition boolean expr)){
      throw new PostconditionNotMetException("post-condition boolean expr",
"methodName");
    }

    if (!(post-condition expr == _preAs_1)){
      throw new PostconditionNotMetException("post-condition expr == //pre-assessment
expr//", "methodName");
    }

  /*
  Validate the invariants after the method call
  */
  validateConsistency();
  }
  /*
  The wrapped method threw an exception, catch different types of exceptions
  */
  catch (InvalidArgumentException iae) {
    if (!_pre_1) {
      /*
      Pre-condition 1 is false, so the method was supposed to throw this exception.  This
means it implemented the check for this pre-condition correctly.  Rethrow the valid
exception.
      */
      throw iae;
    }
    ... repeat for all preconditions with this type of exception

    /*
    If this point is reached, it means that the pre-conditions are valid, but the method still
threw an exception.  So there is a problem with the implementation of the method.
    */
    throw new
PreconditionsHoldButServiceRefusedException("InvalidArgumentException",
"methodName");
  }
  catch (InvalidStateException ise){
    if (!_pre_3){
      /*
      Pre-condition 3 is false, so the method was supposed to throw this exception.  This
means it implemented the check for this pre-condition correctly.  Rethrow the valid
exception.
      */
      throw ise;
    }
          ... repeat for all preconditions with this type of exception
    /*
    If this point is reached, it means that the pre-conditions are valid, but the method still
threw an exception.  So there is a problem with the implementation of the method.
```

```
      */
     throw new PreconditionsHoldButServiceRefusedException("InvalidStateException",
"methodName");
    }
    /*
    All pre-conditions and post-conditions are true.
    */
    return returnValue;
  }
}
```

A working example is available on GitHub: https://github.com/JustusPosthuma/example

The test-interceptor wrapped components are used in the following scenarios:

Unit testing:

In the case of unit testing, the component is meant to be tested in isolation, assuming that any components it depends on perform their tasks correctly. To this end, one substitutes any dependencies of the component under test with mock objects for these dependencies, which (under the scenarios for the given test data) behave as would be expected from the components they are mocking. Dependency injection [18] can be used to inject mock objects instead of actual components for the component dependencies.

Integration testing:

In the case of integration testing the component is tested within its actual environment. In this case, one injects real components for component dependencies.

Operational testing:

For operational testing or testing on the live system, one can dependency-inject test-interceptor wrapped components for those aspects of the system one wants to monitor for contract compliance.

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 required. 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 equivalence partitions for the problem. Future work will look at augmenting the CDD tool suite with tools that automatically generate test data sets [5]. 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.

4. RESULTS

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 components that enable the client to:

- gauge the interest of students and staff to get the vaccine via a short survey;
- enable students and staff to make appointments to receive the vaccine;
- sign digital waivers;
- push-notifications with reminders for vaccine appointments and other important information;
- 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) “serverless” infrastructure for the back-end. This infrastructure hosted a MySQL relational database and provided AWS Lambdas, (event-driven computing services that run 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 1).

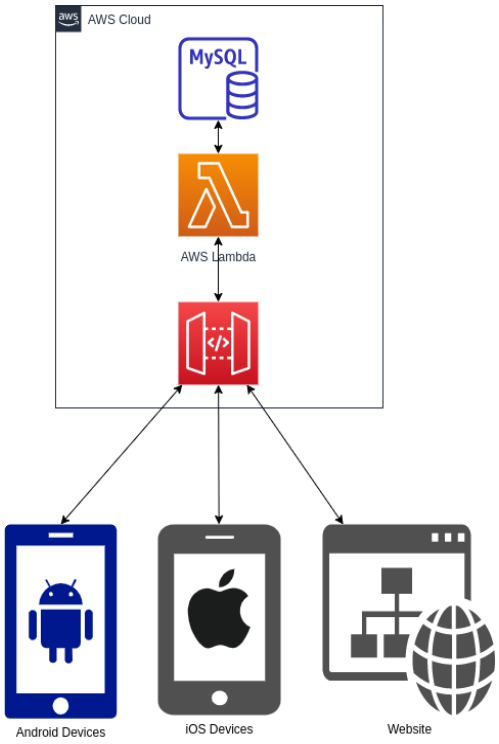

Figure 1. System Architecture

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. An informal Agile development 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 organised;
- 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.

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 with Domain Driven Design (DDD) in mind. The "domain" in this context is the sphere of knowledge and activity around which the application logic revolves [1]; in this case, all the activities around the Covid-19 vaccine roll-out. And keeping to the principles of DDD, the application was architected using four layers:

1. A **User Interface layer**, which contains both, the user views (forms) as well as 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 was not possible:

- The AWS Lambdas are written in TypeScript. Our interceptor-generator is written in Java;

- The author was a member of the team that focused on Android.

However, implementing contract validation via interceptors in the infrastructure layer of the Android application that communicates with the AWS Lambdas, does in theory, have, from the perspective of the Android application, 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 on the server.

To determine the impact of our approach, we compare the Covid-19 project (with contracts and interceptors) to a different project called Travel (with no contracts or interceptors). 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 architecture as well as the same four layers relating to DDD. We analysed the requirements specification, UI interfaces, number of REST calls and project plans to ensure that the projects are as similar as possible in scope:

Table 1. Project properties

	Covid-19	**Travel**
Activities (screens)	11	8
REST API calls	30	26
Planned scope (in days)	30	30

The following measures were used to quantify the impact:

- *Number of man-hours to develop*. This includes requirements specification, requirements refinement and design, test and bug fixing.
- *Correctness of the produced software*. This is the rate at which the produced software meets the client requirements. It entails producing correct results and handling exceptions properly, and can be measured by counting defects over a period of time with bug-tracking software.
- *Bug Density*. This measure is defined as defects per thousand lines of code (KLOC) and measured by dividing the size of the module by the number of confirmed defects.

Number of man-hours to develop

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 2).

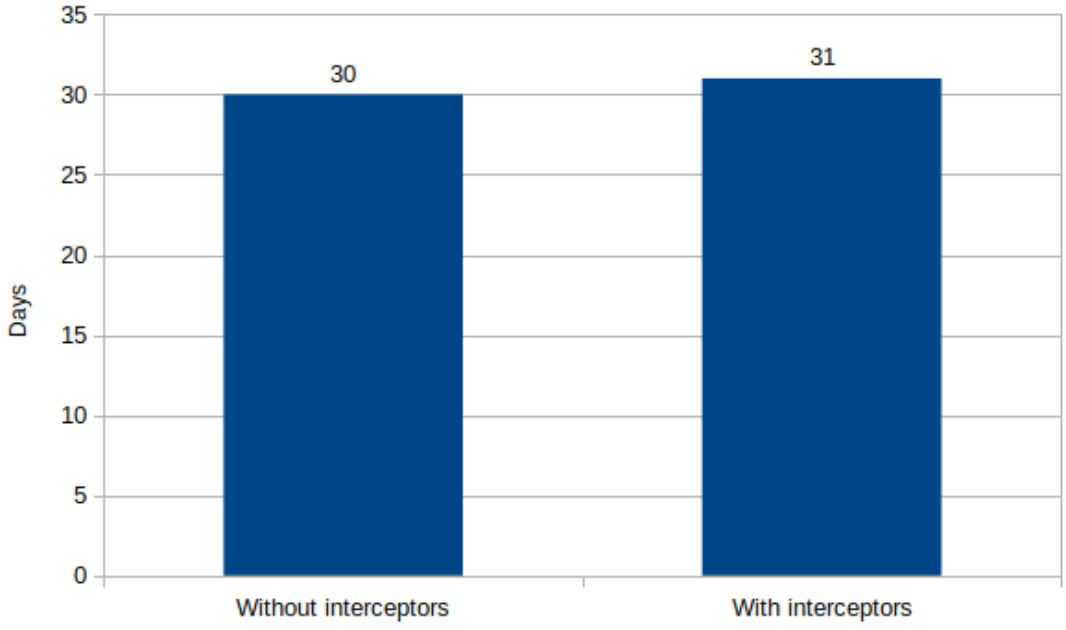

Figure 2. Man-hours to develop

Correctness of the produced software

The average bugs per day was significantly less in our project with contracts and interceptors versus a project without contracts and interceptors. We did not include cosmetic bugs in this measurement (Figure 3).

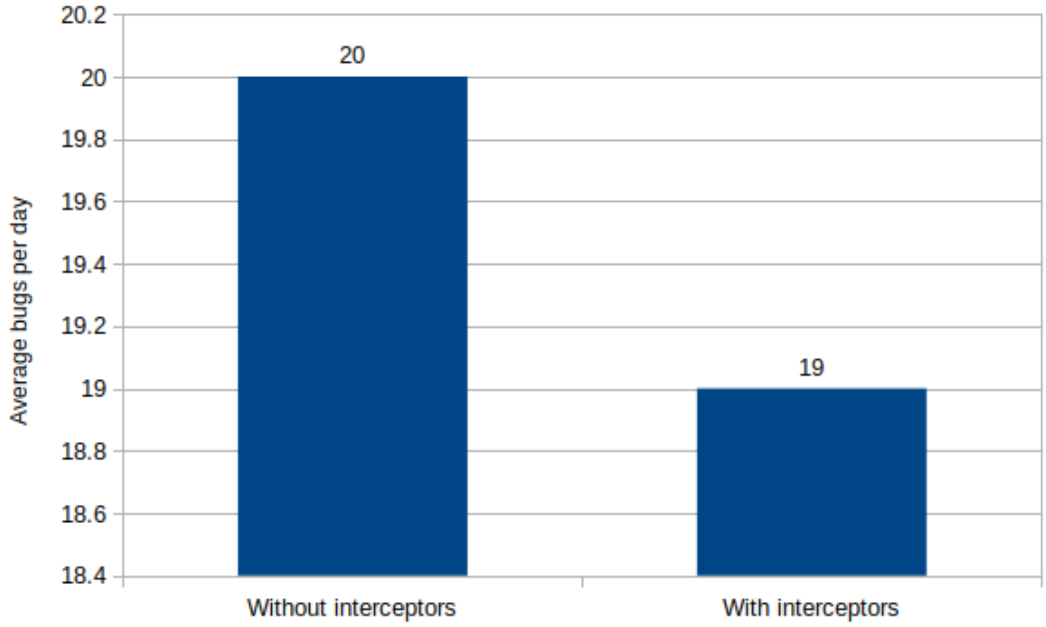

Figure 3. Correctness of the produced software

Bug density

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 (Figure 4).

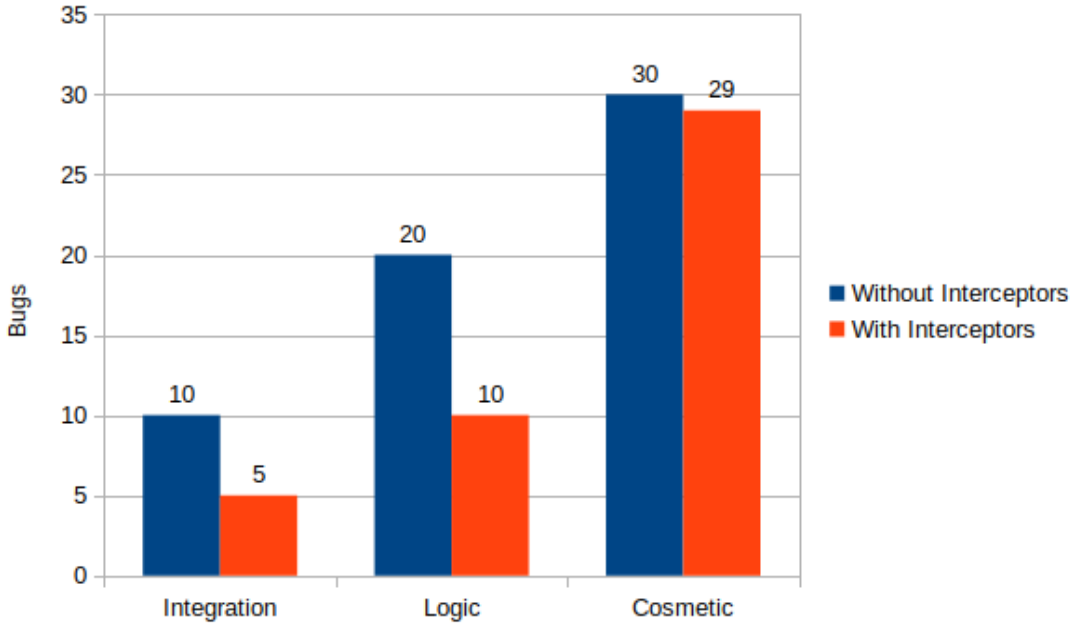

Figure 4. Bug density

4.1. Limitations and Deficiencies

Contracts are defined as Boolean expressions in annotations. These Boolean expressions are written as strings in the annotations, for example:

@Precondition(constraint = "**itemPrice > 0**", raises = InvalidStateException.class)

Since the expressions are strings, the compiler will not evaluate the syntax of the Boolean expression and return an error if it is incorrect – the compiler will simply evaluate the string as any other string. Only once the interceptor has been generated and the Boolean expression has been converted into code, will any syntax errors be discovered by the compiler, and will an error be reported in the interceptor class. Unfortunately, due to how Java annotations and Reflection work, the Boolean expressions (contracts) have to be written as strings.

Modern Integrated Development Environments (IDE) show syntax errors in code as you are typing, and it would have been ideal if any syntax errors related to the Boolean expressions (contracts) could have been identified similarly.

An important function that the interceptors cannot do (yet) is to test for errors or exceptions as part of a contract - 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. CONCLUSION

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.

Developers were able to easily implement contracts since the learning curve was not steep at all – everyone was familiar with annotations already (since it is an existing Java language feature) and everything else (interceptor generation, evaluation of contracts) happens automatically. It was easy to convince the team to try our implementation of CDD since it had so little impact on the normal development routine.

Our empirical measurements show that providing a tool suite for CDD enabling developers to specify syntax-checked component contracts and generate test interceptors used for unit, integration and operational testing, significantly improved the quality of the software produced by the software development process without increasing the cost significantly. In particular, logic and integration errors were reduced by approximately 50%, whilst the reduction of cosmetic errors and the increase in number of man-hours were both less than 5%. Further efficiency benefits can be obtained by automating the generation of test data for unit testing.

We expect that having components that perform rigorously specified functionality (in the form of component contracts) would improve re-usability. The scope of the current study is, however, too small to assess this quality attribute and this is left for future work.

Based on the results obtained so far in this study, we are confident that we are on the right track to rekindle a renewed interest in CDD in industry and we feel that this study also demonstrates that a developer-centric approach and tool suite for CDD does deliver "enough bang for your buck!"

6. FUTURE WORK

Further efficiency benefits can be obtained by automating the generation of test data from the contracts for unit testing, in particular generating unit test templates which require only the population of data structures with test data.

We also feel that these results warrant a CDD approach in other programming languages, especially Javascript which is very popular on the server side. One would have to scrutinize the specific language features to decide how best to implement contracts, because not all languages have built-in annotations. But in the case of Javascript, early investigations show that comment blocks can be used to define pre-conditions and postconditions for methods, and interceptors can be generated by launching an external process that parses the comment blocks and generate interceptors.

REFERENCES

[1] airbrake.io. 2017. Domain-Driven Design: What is it and how do you use it? https://airbrake.io/blog/software-design/domaindriven-design

[2] Hakim Belhaouari and Frederic Peschanski. 2008. 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. IEEE Computer Society, Washington, DC, USA, 219–228. https://doi.org/10.1109/HASE.2008.15

[3] Yoonsik Cheon and Gary T. Leavens. 2002. A simple and practical approach to unit testing: The jml and junit way. (2002).

[4] M. Donat. 1997. Automating Formal Specification Based Testing.(1997).

[5] Stefan J. Galler, Martin Weiglhofer, and Franz Wotawa. 2010. Synthesize It: From Design by Contract to Meaningful Test Input Data. In 2010 8th IEEE International Conference on Software Engineering and Formal Methods. 286–295. https://doi.org/10.1109/SEFM.2010.33

[6] J.-M. Jazequel and B. Meyer. 1997. Design by contract: the lessons of Ariane. Computer 30, 1 (Jan. 1997), 129–130. https://doi.org/10.1109/2.562936

[7] R. Kramer. 1998. iContract - The Java(Tm) Design by Contract(Tm) Tool. In Proceedings of the Technology of ObjectOriented Languages and Systems (TOOLS '98). IEEE Computer Society, Washington, DC, USA, 295–. http://dl.acm.org/citation.cfm?id=832254.832856

[8] Vipin Kumar. 1992. Algorithms for constraint-satisfaction problems: A survey. (1992).

[9] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. 2007. 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 (Dubrovnik, Croatia) (ESEC-FSE '07). ACM, New York, NY, USA, 425–434. https://doi.org/10.1145/1287624.1287685

[10] Bertrand Meyer. 1992. Applying "Design by Contract". Interactive Software Engineering (1992).

[11] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. 2003. Requirements by Contracts allow Automated System Testing. In ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering. IEEE Computer Society, Washington, DC, USA, 85.

[12] Ian Robinson. 2006. Consumer-Driven Contracts: A Service Evolution Pattern. https://www.martinfowler.com/articles/consumerDrivenContracts.html

[13] Beth Skurrie. 2020. Pact: Getting Started. https://docs.pact.io/

[14] slashdot.org. 2007. Why is design by contract not more popular. https://ask.slashdot.org/story/07/03/10/009237/why-isdesign-by-contract-not-more-popular

[15] Fritz Solms. 2018. URDAD for System Design.

[16] spring.io. 2020. Spring Cloud Contract. https://spring.io/projects/spring-cloud-contract

[17] Stackoverflow.com. 2018. Why is design-by-contract not so popular compared to test-driven development? https://stackoverflow.com/questions/481312/why-is-design-bycontract-not-so-popular-compared-to-test-driven-development

[18] Hong Yul Yang, E. Tempero, and H. Melton. [n.d.]. An Empirical Study into Use of Dependency Injection in Java. In Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on (2008-03). 239–247. https://doi.org/10.1109/ASWEC.2008.4483212

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