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

Justus Posthuma
Stellenbosch University
Stellenbosch, Cape Town
justus.posthuma@gmail.com

Fritz Solms
Stellenbosch University
Stellenbosch, Cape Town
fritz@solms.co.za

Bruce W. Watson
Stellenbosch University
Stellenbosch, Cape Town
bruce@bruce-watson.com

ABSTRACT

A contract-driven development approach requires the formalization of component requirements in the form of a component contract. The Use Case, Responsibility Driven Analysis and Design (URDAD) methodology is based on the contract-driven development approach and uses contracts to capture user requirements and perform a technology-neutral design across layers of granularity. This is achieved by taking use-case based functional requirements through an iterative design process and generating various levels of granularity iteratively.

In this project, the component contracts that were captured by utilizing the URDAD approach are used to generate test interceptors which validate whether, in the context of rendering component services, the component contracts are satisfied. To achieve this, Java classes and interfaces are annotated with pre- and postconditions to represent the contracts in code. Annotation processors are then used to automatically generate test-interceptor classes by processing the annotations. The test-interceptor classes encapsulate test-logic and are interface-compatible with their underlying component counterparts. This enable test-interceptors to intercept service requests to the underlying counterpart components in order to verify contract adherence. The generated test interceptors can be used for unit testing as well as real-time component monitoring. This development approach, utilized within the URDAD methodology would then result in unit and integration tests across levels of granularity.

Empirical data from actual software development projects will be used to assess the impact of introducing such a development approach in real software development projects. In particular, the study assesses the impact on the quality attributes of the software development *process*, as well as the qualities of the *software* produced by the process.

Process qualities measured include development productivity (the rate at which software is produced), correctness (the rate at which the produced software meets the clients requirements) and the certifiability of the software development process (which certifiability requirements are fully or partially addressed by the URDAD development approach). Software qualities measured include reusability (empirical and qualitative), simplicity (the inverse of the complexity measure) and bug density (number of defects in a module).

The study aims to show conclusively how the approach impacts the creation of correct software which meets the client requirements, how productivity is affected and if the approach enhances or hinders certifiability. The study also aims to determine if test-interceptors are a viable mechanism to produce high-quality tests that contribute to the creation of correct software. Furthermore, the study aims to determine if the software produced by applying this approach yield improved reusability or not, if the software becomes more or less complex and if more or less bugs are induced.

CCS CONCEPTS

• **Software and its engineering → Software development methods**; *Reusability*; *Requirements analysis*; *Software implementation planning*; *Error handling and recovery*;

KEYWORDS

Contract Driven Development, Test-Interceptors, URDAD

ACM Reference Format:
Justus Posthuma, Fritz Solms, and Bruce W. Watson. 2018. The impact of using a contract-driven, test-interceptor based software development approach. In *Proceedings of Technology for Change (SAICSIT 2018).* ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Contract Driven Development is a design methodology that aims to produce correct software through formalized requirements in the form of component contracts which include the specification of preconditions, postconditions and invariants, also referred to as contracts [6]. The benefits of Contract Driven Development have been demonstrated [4] and it consequently forms the basis of the Use Case, Responsibility Driven Analysis and Design (URDAD) methodology [9, 10]. The URDAD methodology uses contracts to capture user requirements and perform a technology-neutral design across layers of granularity [10]. These component contracts can be used to generate annotated interfaces (for example Java interfaces with pre- and post-condition annotations) which represent the contracts in code. Using these encoded component contracts, test-interceptors are generated. The test-interceptors contain test logic and are used for unit and integration tests, as well as for self-testing-components. In the case of unit-tests, the unit-test-developer only needs to specify the test data and call the relevant component via its test-interceptor.

*Masters project

SAICSIT 2018, September 2018, Port Elizabeth, South Africa

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

This study assesses the impact on the quality attributes of the software development *process*, as well as the qualities of the *software* produced by the process. Process qualities measured include development productivity (the rate at which software is produced), correctness (the rate at which the produced software meets the clients requirements) and the certifiability of the software development process. Software qualities measured include reusability, simplicity (the inverse of the complexity measure) and bug density.

2 THE QUALITIES OF THE SOFTWARE DEVELOPMENT PROCESS

2.1 Development Productivity

There are a number of varying metrics that are used to determine productivity. In the context of this project, the following metrics are considered:

* The rate at which software is produced.
* Costs to implement.
* Time to implement.

2.2 Correctness of the produced software

This refers to the rate at which the produced software meets the client requirements and entails producing correct results and handling exceptions properly. It can be measured by counting defects over a period of time with bug-tracking software.

2.3 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), DO-178B (airborne) or EN 50128 (rail transportation)? The industry standard that is relevant to the test environment must be examined to determine which requirements are partially or fully addressed by utilizing a contract-driven development approach.

2.4 Test Quality

This entails the quality of the testing itself, which include the number of tests and the level of detail of tests that can be automatically generated. A high number of test failures may be indicative of good test quality: 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.

3 THE QUALITIES OF THE SOFTWARE PRODUCED BY UTILIZING THE PROCESS

3.1 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. [8]. 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.

3.2 Simplicity

This quality is the inverse of the complexity measure. Although many methods exist to measure software complexity, most use the following metrics: **Cyclomatic Complexity** which measures how much control flow exists in a program, **Halstead Volume** which measures how much information is in the source code (how many variables are used and how often they are used), and **Maintainability Index** which formulates an overall score of how maintainable a program is [2, 3].

3.3 Bug Density

Bug density is 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.

4 IMPLEMENTATION AND MEASUREMENTS

The Use-Case, Responsibility Driven Analysis and Design (URDAD) 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. [9]

Java interfaces are annotated with pre- and postconditions (and invariants) which represent component contracts. Current annotation frameworks are assessed and selected based on the functionality they provide in order to develop annotation processors that can automatically generate test-interceptor classes by using the annotated pre-and postconditions. Either existing frameworks can be extended, or a new contract annotation framework can be defined if no existing frameworks offer the required functionality.

The test-interceptor classes encapsulate test-logic and are interface-compatible with their underlying component counterparts. This enable 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 precondition on service request.
(2) Delegating the request to the underlying component for processing.
(3) Upon return the test interceptor verifies:
   (a) if an exception was generated by the underlying component, verify that the associated precondition did not hold via a contract violation exception which is an error.
   (b) if the service was provided and returned (no exception) verify that all preconditions were met.
   (c) if the service was provided (no exceptions) verify that all post-conditions hold true.

The following code illustrates a Java interface with a method that is annotated with preconditions and postconditions:

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

    double returnValue = 0;
    try
    {
      /*
      Call the wrapped method of the counterpart
      component to be tested
      */
      returnValue = counterpart.getDiscountPercentage(ID,
      RewardsPartner, itemPrice);

      /*
      If method was implemented correctly, it would have
      thrown exceptions if any of the preconditions were
      false
      */

      if (!_pre_1)
      {
        /*
        The method above did not throw an exception but
        precondition 1 is false.  This means there is a
        problem with the implementation of the method -
        it does not properly enforce precondition 1
        */
        throw new PreconditionNotEnforcedException("
    isIDValid(ID) == true", "getDiscountPercentage()");
      }
      if (!_pre_2)
      {
        /*
        The method above did not throw an exception but
        precondition 2 is false.  This means there is a
        problem with the implementation of the method -
        it does not properly enforce precondition 2
        */
        throw new PreconditionNotEnforcedException("
    isSupportedPartner(RewardsPartner) == true", "
    getDiscountPercentage()");
      }
      if (!_pre_3)
      {
        /*
        The method above did not throw an exception but
        precondition 3 is false.  This means there is a
        problem with the implementation of the method -
        it does not properly enforce precondition 3
        */
        throw new PreconditionNotEnforcedException("
    isSupportedPartner(RewardsPartner) == true", "
    getDiscountPercentage()");
      }

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

      if (!(returnValue > 0))
      {
        throw new PostconditionNotMetException("
    returnValue > 0", "getDiscountPercentage()");
      }

      if (!(getCounter() == _preAs_1))
      {
        throw new PostconditionNotMetException("
    getCounter() == //getInstanceCounter() + 1//", "
    getDiscountPercentage()");
      }
    }
    /*
    The wrapped method threw an exception
    */
    catch (InvalidArgumentException iae)
    {
      if (!_pre_1)
      {
        /*
        Precondition 1 is false, so the method was
        supposed to throw this exception.  This
        means it implemented the check for this
        precondition correctly.  Rethrow the valid
        exception.
        */
        throw iae;
      }
      if (!_pre_2)
      {
        /*
        Precondition 2 is false, so the method was
        supposed to throw this exception.  This
        means it implemented the check for this
        precondition correctly.  Rethrow the valid
        exception.
        */
        throw iae;
      }

      /*
      If this point is reached, it means that
      the preconditions are valid, but the
      method still threw an exception.  So
      there is a problem with the implementation
      of the method.
      */
```

```
        throw new
      PreconditionsHoldButServiceRefusedException("
      InvalidArgumentException", "getDiscountPercentage()"
      );
    }
    catch (InvalidStateException ise)
    {
      if (!_pre_3)
      {
        /*
        Precondition 3 is false, so the method was
        supposed to throw this exception. This
        means it implemented the check for this
        precondition correctly. Rethrow the valid
        exception.
        */
        throw ise;
      }
      /*
      If this point is reached, it means that the
      preconditions are valid, but the method still
      threw an exception. So there is a problem with
      the implementation of the method
      */
      throw new
      PreconditionsHoldButServiceRefusedException("
      InvalidStateException", "getDiscountPercentage()");
    }
    /*
    All preconditions and postconditions are true
    */
    return returnValue;
  }
}
```

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) Unit tests would dependency inject mock objects for any dependencies of the component under test and specify test data based on an analysis of equivalence partitions and boundary values.
(3) Integration tests would dependency inject actual objects used for their dependencies and potentially use the same test data as the unit tests to the component.

This development approach, utilized within the URDAD methodology would then result in unit and integration tests across levels of granularity.

4.1 Empirical measurements of the qualities of the software development process

Development productivity can potentially be measured by comparing planned productivity to actual productivity when using Design By Contract with Test Interceptors. Project management tools usually contain enough information to be able to calculate time taken, resources assigned to activities and time taken to complete activities.

Bug-tracking tools (for example BugZilla) can potentially be used to measure turn-around time as well as frequency and quantity of defects over certain time periods. Tools like this can also be used to determine bug density in components.

An industry standard (for example IEC 61508) can be examined to determine if the introduction of a contracts based approach enhances certifiability, and which aspects may actually impede it.

Automated testing tools like JUnit can be used to create tests to quantitatively determine the errors that were not captured by the auto-generated interceptors. This collection of errors will then consist of errors due to incorrect requirements, and errors in the code.

4.2 Empirical measurements of the qualities of the software produced

To empirically measure code reuse, a static code analyzer like PMD could potentially be used. It includes built-in rule-sets (for example to detect duplicate code) and also allows the creation of custom rule sets.

A number of automated tools exist to measure cyclomatic complexity (SourceMeter, Sonarqube), Halstead Volume (Halstead Metrics Tool, JHawk) and the maintainability index (Testwell CMTJava, JHawk) which could potentially be used to measure code simplicity.

5 RELATED WORK

Nebut et al describes a requirement-based testing technique that leverages use-cases in order to generate functional test objectives [7]. Their technique uses declarative formalisms in the form of pre- and post-conditions (contracts) to express dependencies between use-cases, from which a labeled transition system is built to capture all possible valid sequences of use-cases from an initial configuration. The main disadvantage of their technique is that it does not cover integration or unit testing of specific aspects if they are not described at a very high level. They use the pre-and post-conditions of the use-cases to express the system properties that make it applicable, and the properties acquired by the system after its application. The proposed project differs from this approach in that it uses pre-and postconditions throughout levels of granularity in order to determine contract adherence of components, not use-cases. It also has a strong focus on unit and integration testing on all levels. The URDAD methodology with test-interceptors could potentially be used to extend this approach by being applied to the development process used to create the components that would realize the different use-cases.

Jazequel et al have demonstrated that interfaces between software modules should be governed by precise specifications [4], and concludes that Design by Contract is imperative for effective component reuse. This view supports the utilization of Contract Driven Development which this project is based on.

The Use-case, Responsibility Driven Analysis and Design (URDAD) methodology has been formulated to encourage sound design principles and is architecture and technology neutral [9, 10]. This makes it suitable for any test environment.

Belhaouari et al created an experimental platform known as Tamago-Test for software analysis and automated testing [1]. 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. Their approach is very similar to the Contract Driven Design with Test Interceptors approach, but differs in that they 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.

Leitner et al recognizes that unit testing is a resource-intensive and time-consuming activity. They created a tool called Cdd which is integrated into EiffelStudio (an Eiffel IDE) that uses contracts in code to extract test cases and run them as background activities during development. This is achieved by exploiting actions that developers perform anyway while writing code [5]. Their approach is interesting in that it uses failures (purposely induced or by mistake) in order to extract test-cases and so doing, build up a comprehensive unit testing suite.

6 CONCLUSION

The study aims to show conclusively how the approach impacts the creation of correct software which meets the client requirements, how productivity is affected and if the approach enhances or hinders certifiability. The study also aims to determine if test-interceptors are a viable mechanism to produce high-quality tests that contribute to the creation of correct software. Furthermore, the study aims to determine if the software produced by applying this approach yield improved reusability or not, if the software becomes more or less complex and if more or less bugs are induced.

REFERENCES

[1] 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

[2] Jonathan Bloom. 2018. Five Reasons You MUST Measure Software Complexity. (May 2018). https://www.castsoftware.com/blog/five-reasons-to-measure-software-complexity [Online; accessed 10. May 2018].

[3] M Clark, B Salesky, and C Urmson. 2008. Measuring Software Complexity to Target Risky Modules in Autonomous Vehicle Systems. In *AUVSI North America Conference* (2008-06-11). http://www.mccabe.com/pdf/MeasuringSoftwareComplexityUAV.pdf [Online; accessed 10. May 2018].

[4] 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

[5] 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 the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 425–434. https://doi.org/10.1145/1287624.1287685

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

[7] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. 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.

[8] Jeffrey Poulin. 2001. Measuring Software Reusability. (01 2001).

[9] Fritz Solms. [n. d.]. *URDAD for System Design*.

[10] Fritz Solms and Dawid Loubser. 2010. URDAD as a semi-formal approach to analysis and design. In *Innovations Syst Softw Eng*.