Testing software based on design by contract

Diana Berberova, Boyan Bontchev
Department of Software Engineering,
Faculty of Mathematics and Informatics,
Sofia University, 5, James Bourchier blvd., 1164 Sofia, Bulgaria
e-mail: didana@yahoo.com; bbontchev@fmi.uni-sofia.bg

***Abstract**: In the last decade, several proposals have been done for construction of aspect-oriented system for testing software products. The article presents results from design and development of a new aspect-oriented system for testing software created in Java 5 as an open source project named CodeContract, by using Design by Contract methodology. The system provides means of describing contracts and conditions during the design of software systems that should be satisfied during the system work process. Various conditions used in the contracts are able to be defined by means of preconditions, post-conditions and invariants. In order to describe these conditions in contracts, Java annotations are used. When using the CodeContract system, contracts should be created during the development of software application. Then runtime checks of contracts are executed during the testing of the software application.*

***Keywords:** design by contract, testing, aspect-oriented programming.*

1. INTRODUCTION

In the last twenty years, the software industry has grown tremendously. The software products become more and more complex, as well as more and more critical to everyday life. The public needs for quality software products of all kind of areas are constantly increasing. But despite the economic growth and productivity gains enabled by software, persistent complaints about the quality of software remain [5].

The paper presents results of design and development of an aspect-oriented system for testing software products, which has been created on the base of Design by Contract [6] methodology. The system is called CodeContract[1] and is composed by several components created using aspect-oriented programming and Java 5 features like annotations. It provides means of describing contracts and conditions during the design of software systems that should be satisfied during the work of the systems.

The article goes through several of the most important system design issues, such as the definition of the different conditions used in the contracts – preconditions, post-conditions and invariants. It explains the usage of Java annotations to describe these conditions in contracts and behaviour of components evaluating the conditions, described in annotations, and checking at run time the described conditions, during the work of the system.

[1] CodeContract is developed as an open source project hosted on http://code.google.com/p/codecontract/

2. ASSURING SOFTWARE QUALITY

When thinking of new software development methods and tools, it is very usual to view productivity as the major expected benefit. In object-oriented programming productivity benefits follow not just from the immediate benefits of the approach but from its emphasis on quality. One major component of quality in software is reliability. System's reliability is the ability of the system to perform its job according to the specification (correctness) and to handle abnormal situations (robustness). Every developer wants the systems to be also readable, modular, structured and easy to maintain. Obviously these adjectives describe two different sorts of qualities [8].

The other qualities applicable to a software product, such as being modular, or readable, are internal factors, perceptible only to software developers who have access to the actual software code and are concerned with its implementation and maintenance. These other qualities are called *internal factors*.

In the end - when defining software quality, only *external factors* matter. But the key to achieving these external factors is in the internal ones. For the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities. Although it is possible in many cases to find a solution that reconciles apparently conflicting factors, often it is needed to make tradeoffs.

Traditional style of testing software checks whether a software product meets its specifications; based on black and white box methods. No matter what kind of testing is done, black-box or white-box, it usually follows the scenario: first the tester studies the software system, then writes individual test scenarios and finally executes the tests on the system. The test scenarios are individually crafted and can be executed either manually or by some form test tool. But manual testing and is a labor-intensive and inefficient way to test modern software.

Unit testing [3] provides a strict, written contract that the piece of code must satisfy. It is more time consuming to write code and tests than only code, but on the long run it pays back with several benefits. Regression testing includes writing test cases for all functions and methods so that whenever a change causes a regression, it can be quickly identified and fixed. Good unit test design produces test cases that cover all paths through the unit with attention paid to loop conditions. Test Driven Development, referred as TDD [4], is a core methodology used in Extreme Programming. TDD is also known as "Test First". In TDD first test cases are written, then the code is implemented and finally the test cases are executed. The biggest benefit of Test Driven Development is verification. Defensive programming uses different techniques to avoid creating security problems and software bugs [9]. One technique is reducing source code complexity. A programmer should never make code more complex than necessary as the complexity leads to bugs.

Finally, Design by Contract (DbC) methodology prescribes that software designers should define precise checkable interface specifications for software components based upon the theory of abstract data types and the conceptual metaphor of a business contract. It complements several Extreme Programming practices, particularly unit testing and refactoring. The term Design by Contract was coined by Bertrand Meyer [6] in connection with his design of the Eiffel programming language. The first language that has provided support to Design by Contract was Eiffel, but the idea is gaining popularity and there are several tools have emerged recently for using this methodology when programming in Java, C++ and other programming languages.

DbC involves a partnership between the producer and consumer of a class and between the features promised in the class and the responsibility of using these features correctly. If both parties adhere to this contract, the resulting software has the potential to be more understandable and reliable. The user requirements are given by a set of preconditions and the producer requirements are given by a set of post-conditions. In other words, DbC defines and manages the responsibilities between a class and its users. The methodology is very useful and clear concept for software development, although it is not very popular. So it is chosen for basis of the testing system developed in this work and is described in details in the following chapter.

3. ASPECT-ORIENTED PROGRAMMING OF DESIGN BY CONTRACT

The central idea of DbC is a metaphor on how elements of a software system collaborate with each other, on the basis of mutual obligations and benefits. The metaphor comes from business life, where a "producer" also called “client” and a "consumer" also called “supplier” agree on a "contract". The producer has to provide a certain product, which is producer’s obligation and is entitled to expect that the consumer has paid its fee, which is producer’s benefit. The consumer must pay the fee, which is consumer’s obligation and is entitled to get the product, which is consumers benefit. Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts.

In the means of DbC and object-oriented programming, the method’s precondition is an obligation for the client and a benefit for the supplier. It frees the supplier from having to handle cases outside of the precondition. The method's post-condition is an obligation for the supplier, and obviously a benefit for the client. The class invariants are certain properties, assumed on entry and guaranteed on exit of every call to a method.

So preconditions and post-conditions are two fundamental elements of contracts. The third fundamental element - which is really useful mostly in an object-oriented context, is invariants. A class invariant is a condition that applies to an entire class. It describes a consistency property that every instance of the class must satisfy whenever it's observable from the outside. That means that this property, the class invariant, must be satisfied whenever an instance of the class is created.

The contracts are expressed as Boolean expressions, intended precisely to express runtime true or false properties - properties that at any point in the execution may hold or may not hold. There are rules for proper behaviour of inherited contracts, based on the Liskov Substitution principle (LSP), which is a minimal definition of inheritance [1]. In the context of DbC methodology if class B is considered a child class of class A (where A can be another class or interface) then B must obey A's contract, including all the class, method, and field checks, i.e. B inherits A’s contract. However, there is one special feature that affects derived preconditions and post-conditions described in [7]. When using an object through its base class interface, the user knows only the preconditions and post-conditions of the base class. Thus, derived objects must not expect such users to obey preconditions that are stronger then those required by the base class. They must accept anything that the base class could accept. So if the method preconditions are overridden, they should get looser, not stricter. In the terms of client and suppliers: the precondition tests are obligations that the client must meet; if a client already meets a strict test defined by the overridden test, then it will also satisfy a looser derived test transparently.

Aspect-Oriented Programming restores modularity by developing the cross-cutting concerns, or aspects, in isolation and then combining them with other modules using declarative or programmatic mechanisms that are modular [7]. Thus, the AOP paradigm appears to be very suitable for realisation of the Design by Contract methodology. The points of intersection are defined once, in one place, which makes them easy to understand and maintain. The other modules require no modifications to be advised by the aspects. This "intersection" process, sometimes called weaving, can occur at build or run time. Aspect-oriented software development weaving is a key innovation that provides very fine grained query and composition semantics. Where traditional code linking has the ability to resolve method and variable names, weaving adds the ability to replace method bodies with new implementations, insert code before and after method calls, instrument variable reads and writes, and even associate new state and behaviour with existing classes, typically for adding special behaviours. Specially, for allowing adding metadata available to the programmer at run-time to Java source code, Java annotations [2] are used.

4. REALISATION OF THE CODECONTRACT SYSTEM

Functional requirements of the system CodeContract define that it should provide:

- DbC support for Java 5 applications
- Runtime checks for class invariants and pre- and post-conditions that are associated with methods in classes and interfaces.
- If the runtime check of a contract fails, the program execution should be terminated, providing proper error message.
- Contract propagation via the Java type extension mechanisms - class extension, interface implementation and interface extension
- Support for inheritance of contracts for public methods according to Liskov Substitution principle and Design by Contract inheritance principle
- Additional support for inheritance of contract definitions on non-public methods, static methods, and constructors
- No modification of the source code of tested application
- No modification of the build process of the tested applications

The currently developed system has two types of actors - application developers (create Java 5 applications using DbC for testing) and testers (validate Java 5 applications written with DbC support). Each method call from a class declared with @Contract annotation is intercepted and the following actions are executed:

- preconditions are checked
- method is executed
- postconditions are checked

In case that preconditions or post-conditions are evaluated to false, then runtime exception is thrown and program is terminated. The use case Check Post-conditions (fig. 1) represents the case for a single check of post-condition contracts defined for a specified method. This use case is extended by four use cases, representing the different cases of checking post-conditions - public, non-public, static methods and constructors.

All the use cases that represent checks for preconditions and post-conditions include some of the following sub use cases according to the algorithm for gathering preconditions/post-conditions:

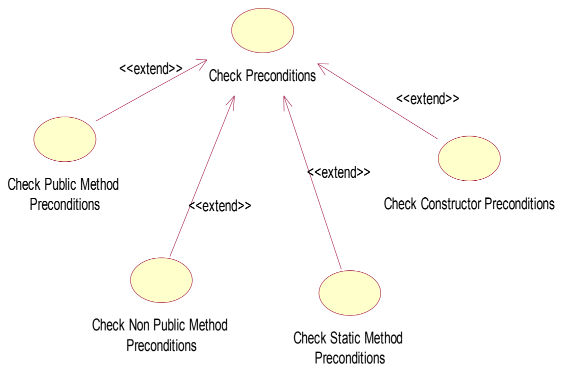
Check Preconditions
<<extend>>
<<extend>>
<<extend>>
<<extend>>
<<extend>>
Check Public Method Preconditions
Check Constructor Preconditions
Check Non Public Method Preconditions
Check Static Method Preconditions

Fig. 1: Use case diagram for Check Preconditions.

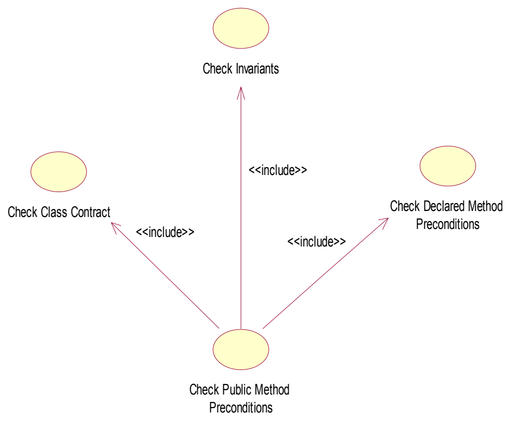

Fig. 2: Use case diagram for Check Public Method Preconditions.

- Check whether the class of the method has annotation @Contract;
- Check Declared Method Preconditions use case - check preconditions specified for the method in the current class (fig. 2);
- Check All Method Post-conditions use case - check all post-conditions specified for the method in the hierarchy of classes;
- Check Declared Method Post-conditions use case - check post-conditions specified for the method in the current class;
- Check Class Invariants use case - check all invariants contracts specified for the class in the hierarchy of classes.

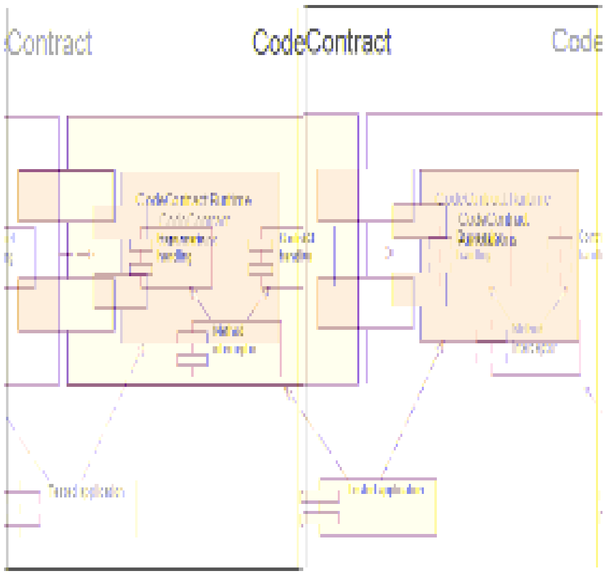

Fig. 3: Software architecture of the CodeContract system.

```
@Contract
public class Student {
    public String name;
    private String address;

    @Invar ("id > 0")
    public int id;

    @Post ("name != null")
    public void setName (String newName) {
    name = newName;
    }

    @Post ("$return != "")
    public String getName () { return name; }

    @Pre ("$args0.startsWith("Sofia"))
    public String setAddress(String newAddress)
    {
      address = newAddress;
    }
}
```

Fig. 4: A sample contract

Fig. 3 represents the architecture of the system CodeContract. The CodeContract system is composed by an annotations component and runtime part. The runtime part contains method interceptor (handles of calls to methods of annotated classed during the work of the tested application), contract handling (extracts the active contracts), and expression handling (evaluates the expressions of the contracts).

All classes that specify with contracts should have class annotation @Contract. The contracts are defined using "@Pre", "@Post", and "@Invar" annotations, for precondition, post-condition, and invariant respectively. The contract is described with the value of the annotation. In the example of fig. 4, the field "id" has an invariant test that it cannot be less or equal to 0. The method setName() has post-condition that the field name cannot be null. The method getName() has a post-condition that the return value cannot be the empty string “”.The method setAddress() has a precondition that the first argument value should start with the string “Sofia”.

5. CONCLUSIONS

The CodeContract system is based on a methodology that ensures system correctness - Design by Contract, and provides means for designing and creating contracts between different software components. Compared to other implementations of DbC such as IContract, JContrcator and Barter, it offers many advantages:

- all scopes of methods can be contracted – public, private, protected and package, although different types of check politics are used
- CodeContract supports contract inheritance and contract propagation via all four mechanisms – class extension, interface implementation and interface extension. The contracts are checked according to the LSP and the preconditions check is optimized
- the contracts are described via Java annotations and JEXL
- supports instrumentation of the code at runtime time. The source code is not needed. Neither the source code nor the byte code is modified
- CodeContract uses the load time weaving provided by AspectJ to modify the classes during the loading in JVM. In this way, CodeContract does not change the development and compilation process of the tested applications.

6. REFERENCES

[1] Briand, L., Dzidek, W., Labiche, Y. 2005. Software Maintenance, In *Proc. of the 21st IEEE Int. Conf. ICSMapos*, pp. 687-690.

[2] Friesen J. 2007. Beginning Java™ SE 6 Platform: From Novice to Professional, Apress.

[3] Hunt A., Thomas D. 2003. *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 1st edition.

[4] Kosleka L. 2007. *Test Driven: TDD and Acceptance TDD for Java Developers*, Manning Publications, 1st edition.

[5] McConnel S. 2004. *Code Complete*, Microsoft Press, 2nd edition.

[6] Meyer B. 1992. Applying "Design by Contract. In *Computer (IEEE)*, vol. 25, no. 10, pp. 40-51.

[7] Piattini M., Garzas J. 2004. *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices*, IGI Global, 2nd edition.

[8] Tian J. 2005. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, Wiley-IEEE Computer Society Press; 1st edition.

[9] Qie X., Pang R., Peterson L. 2002. Defensive programming: using an annotation toolkit to build DoS-resistant software, In *ACM SIGOPS Operating Systems Review archive*, Vol. 36, Issue SI: Robustness, pp. 45-60.