# COMPONENT BASED SYSTEMS TESTING WITH UML AND INCORPORATED CONTRACTS

**AVERIAN, Alexandru**
Faculty of Mathematics-Informatics,
*Spiru Haret* University,
a.averian.mi@spiruharet.ro

***Abstract***
*Production of complex systems done by reusing prefabricated software and hardware components has proved it's efficiency by accelerating the development process, significantly reducing development time and costs. Generally, the components are tested in their production environment, and the test elements are eliminated before publishing the component. The goal of testing based on incorporated contracts is to equip components with test elements which allow testing in the environment in which they are being developed and also if they are integrated in the development of other systems*.

**Key-words:** *component-based systems, UML models, test components, test interfaces, design for test*

## 1. **Introduction**

Program development based on reutilizing some existing components came into being in the field of hardware development where costs were greatly reduced by reutilizing existing hardware components. Over the years, in the field of software development, a series of functional modules were created and used to develop a series of programs in an ad hoc manner. The development of component based programs has proved difficult due to the lack of a well defined component development discipline and due to the inexistence of a large collection of high quality, standardized components.

In the traditional method of developing programs, the largest effort to integrate the parts of a system is put into the development environment where any area of the developed system can be accessed and where we have the possibility to test the compatibility and the correct connecting of the subsystems, thus ensuring that the entire system can function correctly. On the other hand, in component based development, the development environment of a system differs from the one in which a certain component was developed and most of the times a component is created without following a certain utilization profile. In this new environment, the component can be utilized in a manner that has not been anticipated by it's creator, thus leading to the invalidation of the tests that have been performed in the component's development stage.

Contract based testing refers mainly to those testing elements which are incorporated in the component's code with the purpose of facilitating the tests and checking the conditions while running in the integration environment of the component. These elements do not belong to the functional requirements of a component and don't appear in the execution code of the component after it's publication. In this article we sustain the idea that designing a component should be accomplished in a way that allows the test elements to be an integral part of the component's structure starting from the specification stage.

## 2. **Incorporated test elements**

Reutilizing certain poor quality components can lead to the creation of poor quality programs or defective ones. But reutilizing good quality components in an incorrect manner can have the same effect. In conclusion, testing and ensuring the quality of the components and programs developed by reutilizing components must be pursued at all times during development, starting with the specification stage followed by implementation, integration of components, testing and validating the system. As opposed to the traditional testing methods, testing programs that are being developed by reutilizing components can be looked at from two points of view: the creator's point of view and the user's point of view.

The creator tests the component in a certain stage of the development cycle so that he can validate the component. The user tests the component within the framework of developing a component based program, in which case testing looks at the integration of the component and it's behaviour in a certain context.

Incorporated contract based testing looks at the way in which components are equipped with test elements that are meant to allow testing the environment in which they are integrated and also to be tested by the environment in which they are running. Therefore, in the development of a new system, the components that incorporate the test elements of the contract check and validate the contractual conformity of any two components of the system, including the running environment.

### 2.1. *Assertions*

Assertions are not new elements in the field of testing. Assertion represents a boolean expression which defines the necessary conditions needed for a function, an object or a component to function correctly. Assertions follow the principle of self-testing that is found in hardware components. When executed, an assertion checks if a certain variable belongs to the expected domain.

Theoretically, assertions can be placed in a code in any arbitrary location but there are places where they can be a lot more useful. They can be placed before the appeal of a procedure, with the purpose of checking all the preconditions necessary for the appeal of that procedure. Also, an assertion can be placed after the appeal of a procedure in order to test if all the post conditions of the procedure check out after execution. Other assertions must be validated at all times, this being the case of invariations of a certain class.

An assertion consists of three components:

- a predicate, a boolean expression;
- an expression of action which should indicate the place in which the problem started as exactly as possible;
- a mechanism of activation/deactivation which controls the execution of assertions.

During the process of executing a program, an assertion can be evaluated ***true*** in which case no error is revealed. If an assertion is evaluated ***false*** then it is considered that an error has been detected and the expression of action is executed. The action launched by the assertion can consist of displaying an error message on screen or it can consist in the activity of a complex reconstruction mechanism after the error.

Assertions represent an extremely useful instrument in detecting a vast range of errors. Moreover, assertions are useful in overcoming problems that occur in testing object oriented programming due to encapsulation and information concealment.

### 2.2. *Incorporated test elements*

Incorporated test elements are represented by the procedures utilized in testing experiments and test data utilized in experiments also known as test sets. Assertions can be used in tests and give information which is useful in detecting errors that occur during test execution but we cannot say that assertions or their execution represents the testing procedure. Testing is represented by an experiment performed in certain conditions which apply a set of test data using a testing procedure and a testing context.

Components which contain incorporated testing elements are those types of components which contain test data and testing procedures through which they are able to test their own implementation. The main motive that sustains the incorporation of testing elements in the code of a component is to allow the checking of different implementations of a component based on the same specification, because a component is seen as a collection of documents that describe the interface, structure and behaviour and an implementation of this model. Each implementation consists of two parts: the implementation of the functional part of the component and the implementation of it's testing part. Implementing the self-test strategy is part of the development stage of the component and represents a method  for testing individual components and cannot be used in testing interactions between different components.

Object oriented programming languages offer an elegant possibility to implement the two parts of  component using the inheritance concept.

For example, in C++ if we have a base class which represents a component and which contains the implementation of it's functional part, then the implementation of the testing method will be implemented in a derived class.

```
class Component{
public:
//the component's interface and the functional implementation
};
class TestComponent: public Component {
public:
        //the test interface and the test data
    };
```

In order to test this type of component, it’s test variant will be initialized and additional methods brought by the test interface will be invoked. If the integration of the component in a new system is desired, the base class will be implemented. During the execution of the tests, the derived class will have access to the functional implementation of the component through inheritance as long as the base class is not using private attributes. The advantage of this method of approaching incorporated self-test element implementation is the fact that the barrier created by encapsulation can be overcome but with maintaining the functional implementation separate from the interface, implementation and testing data.

## 3. **Testing component based systems**

Testing component based systems is accomplished in two steps. The first step is accomplished by the component’s creator, who can use both the black box method and the transparent box method because he has access to the source code of the component and knows the logical structure of the program, of the algorithms and data structures that have been utilized. In this stage, testing is done by checking each service offered by the component individually and not certain combinations or successions of services.

The second stage of testing is accomplished by the component’s user who builds a new system and looks for adaptation and correct coupling with the other components. From this point of view, a component is seen as a black box and the user will use the black box method for testing. For components designed by the user, such as the adaptors between components, he will use the transparent box method. Next, we take a look at the most important problems that may occur while testing component based systems.

- Testing the component in a new context. The component’s creator can test the component in it’s development environment and will utilize tests which are specific to the way in which the component was designed to be used, but he cannot imagine all the possible scenarios in which the component may be used. The component’s user will utilize and test the component in a context that has not been foreseen by the creator and will utilize criteria and testing methods that are specific to the developed application.
- The lack of access to the internal logic of the component reduces the possibilities of control and testing. For most components, especially for those which are commercially produced, apart from the interface and the way in which the component is meant to be used, we have no other information regarding it’s internal functioning mechanism.
- Adequate testing of components is a matter that concerns how and how much the component must be tested by it’s creator and also how and how much the component must be tested by the user in order to ensure the correct functioning of the component. There are no empirical observations regarding the testing criteria which must be used or the circumstances.

In conclusion, testing component based programs implies a series of which don't appear in the process of testing programs produced from the beginning. If two correct components which were validated in their development environment and were developed separately without taking each other into consideration, are joined, they might not function correctly in their new environment and there is no guarantee that the interaction between them is semantically correct. This problem does not concern the components because if taken separately these function correctly, but the way in which they interact with the environment or with each other can produce incorrect or unexpected effects.

## 4. **Testing with incorporated contracts**

The goal of testing with incorporated contracts in the component's code is to ensure that the environment in which a component is executed meets the expected requirements of the component and the component meets the expected conditions of the environment in which the component is executed. Testing based on incorporated contracts requires each component to be equipped with it's own set of tests, which, as opposed to the incorporated tests mentioned before, which test the implementation of the component, must test the behaviour of the environment in which the component is meant to run.

An important condition which is necessary in all forms of testing is observability. A testing operation can be executed only if the test component can obtain a correct result, observable. If the invocation of a method of a test component does not obtain any result, testing cannot take place.

Next, we stress the main stages in the development of test elements based on incorporated contracts:

- identifying the tested interactions from the ensemble model of the application;
- defining and modeling the test architecture;
- specifying and building the test interfaces of the components which will play the role of the server to the identified associations;
- specifying and creating the tester components;
- integrating the components.

### 4.1. *The component contract*

Software development through reutilizing components utilizes programming concepts that are oriented toward objects, such as encapsulation and sending messages  from one object to another. The interaction between components is based on the server/client model. Within the framework of an interaction, one of the parts plays the role of the client and the other the role of the server.

The bonds between components represent a key concept together with composing the components. This concept comes from the client/server relationship within object oriented programming where it represents the way in which two objects interact. Without bonds between components there can be no interaction among these and the process of composing components cannot be accomplished.

The bonds between components requires the appeal made by a client component of the public operations performed by a server component. This interaction takes place in only one direction and requires that the client possesses information about the appealed server, usually a reference to an instance of the server. The server component must not possess any kind of information about the client to which the appeal is made. The bond between a client and a server is defined as a contract between the two, a contract which states the services that the server offers and invoking method that the client must respect.

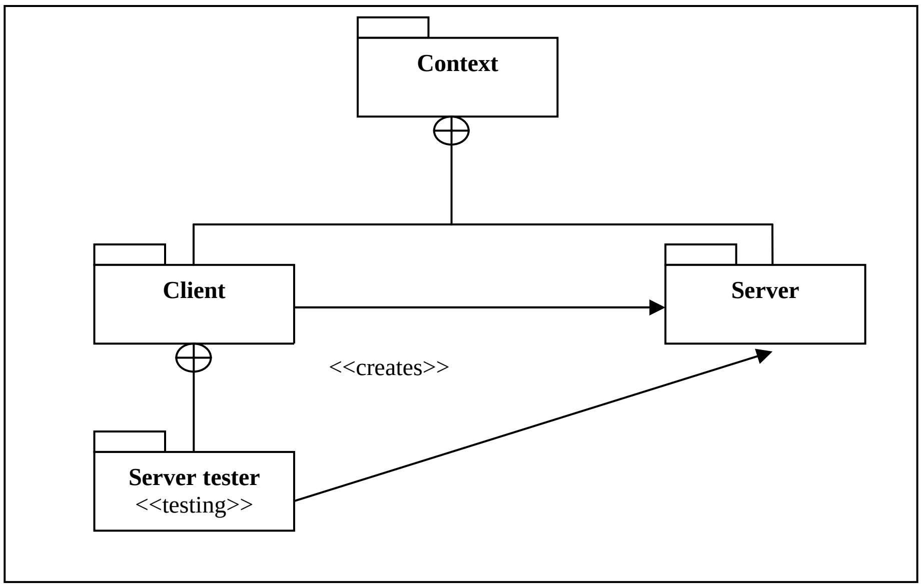

Fig. 1. The client/server relationship

Component based programming can be seen as an extension of object oriented programming where the rules that govern the interactions between two components form the contract. The contract characterizes the relationship between server and client under the form of a formal agreement that states the rights and obligations of each part.

Each client component can be connected to one or more server components which have one or more incorporated test components, depending on the number of the servers. Each test component is developed according to the model in which the test components are carried out. In other words, each tester possesses a description of the environment with which it interacts. The test component is a self standing component and it contains the sets of tests applied on the server.

### 4.2. *Test architecture and UML*

Because the test procedures associated to a client component execute a set of tests on a tested server component, it’s natural that these testers belong to the client under the form of a method. Although at first sight it seems correct, this approach does not allow the optimal reutilization of the components. It would be a lot more correct if these tests would be encapsulated into a separate component.

This allows the tests to be developed in tandem with the components which they are to test, at the same time adding a great degree of flexibility to the level of tests incorporated in the development stage and in the release version. Moreover, while running the program it is possible to choose which test instance will be executed at a certain moment depending on the complexity level chosen.

Specifying and implementing the testing interface applies to the server components in a client-server type relationship. Adding the testing interface can be accomplished only if the server is developed in-house and we have access to it's source code or in the case of an open-source component.

The testing interface will contain the setting methods of the component in a certain state and also the methods used to check the state in which the component is in. The UML behavioural model in which the states of a component are shown can represent a guide mark in starting the process of selecting states taken into consideration when building the test interface. Each state in this model will have two associated methods from the testing interface, one method sets the respective state and the other checks if the process took place correctly.

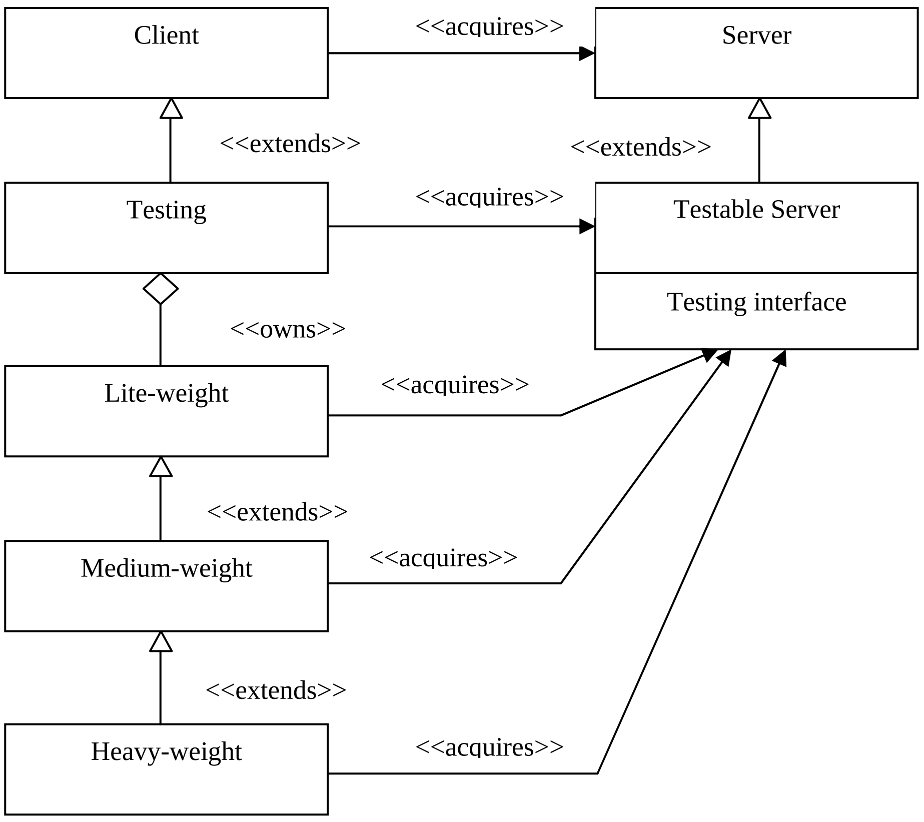

Fig. 2. Modelling the test architecture using UML

This way, the testing procedure does no longer belong to the client but to a separate component. Instantiating the test component can occur once the server has been instantiated. Heavy-weight type tests are usually an extension of certain medium-weight or light-weight tests and, therefore, aside from the defined testes, will include an appeal to the tests to which they are an extension, thus respecting the principle of reutilizing the code from object oriented programming.

## CONCLUSIONS

The development of component based programming is based on the idea of building new programs by reutilizing some already constructed parts, called components, which are prepared to be utilized in the development of new systems. The most important characteristic of a component consists of the fact that it can be reutilized in a series of different contexts that were not anticipated by the component's creator. Only the component's user can decide if the component can serve a certain purpose and can be a part of the structure of the developed product. This is the biggest difference regarding approach between traditional developing and component based developing of programs. Decisions concerning the insertion into the model of incorporated contracts will have to appear in the specification of the system that has been modeled with UML, adding these can be considered an extra effort as part of the process of software development through which testing functionalities will be added to the initial functionalities.

## REFERENCES

1. Hans-Gerhard Gross, *Component-Based Software Testing with UML*, Springer Verlag, New York, 2003
2. Tian, Jeff, *Software Quality Engineering. Testing, Quality Assurance, and Quantifiable Improvement*, John Wiley & Sons, Inc., New Jersey, 2005
3. Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, New York, 1998
4. Jerry Zeyu Gao, H.-S. Jacob Tsao, Ye Wu, *Testing and Quality Assurance for Component-Based Software*, Artech House Inc., Boston, 2003
5. Leonardo Mariani, Mauro Pezze, David Willmor, *Generation of Self-Testing Components*, International Workshop on Test and Analysis of Component Based Systems, Electronic Notes in Theoretical Computer Science, 2004
6. Beizer, Boris, *Software Testing Techniques*, 2nd Edition, New York, Van Nostrand Reinhold, 1990
7. Quirk, W. J., *Verification and Validation of Real-Time Software*, New York. Springer-Verlag, 1985
8. Benoit Baudry, Yves Le Traon, *Measuring Design Testability of a UML Class Diagram*, Information and Software Technology, 2005
9. Reiko Heckel, Leonardo Mariani, *Component Integration Testing by Graph Transformations*, 2005
10. Sami Beydeda, Volker Gruhn, *Black and White-Box Self-testing COTS Components*, Computer Software and Applications Conference, 2003
11. Hartmann, J., Imoberdorf, C., Meisinger, M., *Uml-Based Integration Testing*, Intl. Symposiumon Software Testing and Analysis, ACM Press, 2000
12. Binder, R. V., *Testing Object-Oriented System. Models, Patterns, and Tools*, Addison-Wesley, 1999