Using Aspects to Develop Built-In Tests for Components

Jean-Michel Bruel[1] João Araújo[2] Ana Moreira[2] Albert Royer[1]

[1]Computer Science Research Laboratory
LIUPPA, University of Pau, France
Phone: (+33)559.40.76.37
{bruel,royer}@univ-pau.fr

[2]Departamentode Informática
FCT, Universidade Nova de Lisboa, Portugal
Phone: (+351) 21.294.85.36
{ja,amm}@di.fct.unl.pt

ABSTRACT
The quality of component-based software critically depends on how effectively testing is carried out. To address this problem, we have incorporated built-in tests into components as a way to facilitate their validation in different execution environments. However, the modularization of crosscutting concerns is ignored, leading to redundancy of code spread among several components. Aspect-orientation can improve component-based software development as it provides the appropriate mechanisms to keep in different modules concerns that cut across other concerns. Our goal is to define an UML compliant approach to incorporate testing features into components in an aspect-oriented software development. This paper is a stepping stone towards this goal. Here we explore how that can be achieved at the detailed design and implementation levels.

Keywords
Aspects, Built-in Tests, UML, Testability.

1. INTRODUCTION
Despite the progresses made in component development, especially platform-specific models (PSMs, such as CORBA, .Net, etc.), the main problems that remain in component-based software development (CBSD) are composition and certification. This has been pointed out by number of surveys (e.g., [17]). In this paper we are focusing on testability that we see as a key issue in certification.

During the past few years we have developed a framework to support the implementation of testability features built in components [1, 21]. Components with such feature are called BIT-components. The main goal of this project was to be able to incorporate components in a new environment with the ability to check that they behave as expected.

The main limitation of BIT-technology is that testability features are attached to single components, without taking into consideration the implication of composition. This can lead to duplication of testing code spread among several different components. This results in tangled implementations that are difficult to maintain and evolve. These drawbacks are at the core of the problems that aspect-oriented software development (AOSD) techniques aim to solve [13].

Our goal is to define an approach, covering the whole development lifecycle, to incorporate testing functionality into components developed accordingly to AOSD principles. Aspects will then be used to develop aspectual components, implement interactions between components and implement testability features in a component. In this paper we explore how testability features can be expressed as aspects in BIT-components.

This paper is organized as follows. Section 2 gives some background on BIT-technology. Section 3 introduces some initial steps towards an approach to aspect-oriented and component-based development and illustrates the use of aspects at the detailed design and implementation levels. Section 4 discusses some related work and, finally, section 5 draws some conclusions and points directions for future work.

2. BACKGROUND
Component-based software engineering has been adopted in order to improve software development as well as maintenance efficiency and quality. It also aims to increase reuse rate of existing software in multiple applications. Viewing software architectures as being composed of components is helpful for enabling software development, test, and maintenance to be carried out at a higher level than that of language statements. Nevertheless, despite the progresses made in component development, especially platform-specific models (PSMs, such as CORBA, .Net, etc.), the main problems that remain in CBSD are composability and testability. This has been pointed out by number of surveys (e.g., [17]). The European COMPONENT+ project[1], sponsored by the European 5th Framework Program and by a number of leading industrial partners in component-based software engineering, has developed a technology to provide a set of testability features to components, called BIT – *Built-In Test* [21]. This section starts by introducing the concepts developed in this project, then presents in more detail the "Contract-testing" part of the technology, in which we have been particularly

Submitted in July 03 to the *AO Modeling with UML* Workshop at UML'03, San Francisco, USA.

[1] European IST-1999-20162 project. See http://www.component-plus.org for more details.

involved and which we are focusing here, it then follows by describing the implementation of the technology by presenting our BIT Java library, and finally, illustrates this library with an example.

2.1 Concepts and definitions

In BIT [21], a *component* is defined by a number of provided and required interfaces, through which its functionality is understood. In general, a system developer is unable to look inside a component. However, problems can arise in a system composed of components, especially with COTS, when there is no provision for testing. The developer must be able to verify that the components used will function correctly when deployed in the target system, and when exposed to a specific usage profile. It must also be verified that components interact correctly in order to meet the overall system requirements. To address these issues, the notion of a built-in-test-component (BIT-component) has been introduced. A *BIT-component* is composed of its functional interface(s), augmented by one or more test interfaces. The purpose of these *BIT interfaces* is to enable detection of errors, which, in a component-based system, can be classified in two levels:

1. those that are confined to a specific component (and can be detected within that component), and
2. those at system level arising from incorrect component interaction (which cannot be detected within the components involved).

The *BIT architecture* is based on several architectural elements such as *BIT-components* (components that provide a number of built-in test services), *Testers* (components that use the test services of BIT-components to determine whether a system-level error condition exists), *Handlers* (components that handle errors detected by BIT-components or test components), and *System constructor* (a conceptual element, nominally responsible for the instantiation of BIT-components, testers, and handlers, and their interconnection).

2.2 Contract testing

The correct functioning of a system of components at run time is dependant on the correct interaction of individual pairs of components according to the client-server model. Component-based development can be viewed as an extension of the object paradigm in which, following Meyer [15], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a *contract*. This characterizes the relationship between a component and its clients as a formal agreement, expressing each party's rights and obligations.

The testing approach we are focusing here is based on the notion of building contract tests [8] into components so that they can validate that the servers to which they are "plugged" dynamically at deployment time will fulfill their contract. Although built-in contract testing is primarily intended for validation activities at deployment and configuration-time, the approach also has important implications on the development phases of the overall software lifecycle. In the overall BIT project, some partners focused in a more real-time oriented type of testing, closer in fact to monitoring than testing. These particular testing activities were called in the project *Quality of Service* testing. In this paper we focus on contract testing, since that was the focus of our team at University of Pau. In this context, and to demonstrate the feasibility of the technology we have developed the library described in the next subsection.

2.3 BIT/J library

In this paper we describe the development of a pragmatic Java library (BIT/J) that supports the BIT technology [1]. The architecture of the library described here is partially illustrated in Figure 1. Built-in contract testing can initially be carried out using three primary concepts: an *<<interface>>*, that implement the testability contract; and two *classes,* that implement the test cases and the tester. These three implementations, shown in Figure 1, are: *IBITQuery*, *BIT test case* and *BIT tester*. These implementations mainly support the assessment of test results, control of the execution environment, and actions to be taken if faults are encountered. Additionally, the library provides state-based testing support that is essential to built-in contract testing. The state-based concepts abide by the principles of Harel's state machines, whose mechanisms are fully available in the library (definition of states, combination of these states, definition of transitions, etc.) [11]. This is the reason for the three implementations we have: *State-based IBITQuery*, *State-based BIT test case*, and *State-based BIT tester*.

The Java library is bounded to the original Java (even a COTS one) component either through an extension mechanism (inheritance) or through a containment relationship. The access to COTS components can be realized through Java's Reflection mechanism inside the library. The behavioral model that is required for the contract-testing interface must be defined according to the generic behavioral facilities that the library is providing and it is completely incorporated in the newly created wrapper. Through this technique, the wrapper component represents an executable behavioral model of the original COTS component. In addition to this library, a full generation code environment, described in Figure 1, is provided. The tool, based on the Sun's JMX tool, allows component and testers management. For a complete description of the tool, see the user manual [2].

The strength of the tool is the ability to support COTS (through the Java introspection mechanism) and to ease the job of the (human) tester that manipulates the components for testing purposes by generating as much code as possible. The problem is that the tool remains dependent on the Java and JMX technology . In addition, it is only dedicated to the generation of a BIT component out of a component without consideration on potential subcomponents or existing sub-BIT-components. We hope that the use of aspect-oriented technology will help us overcome those drawbacks.

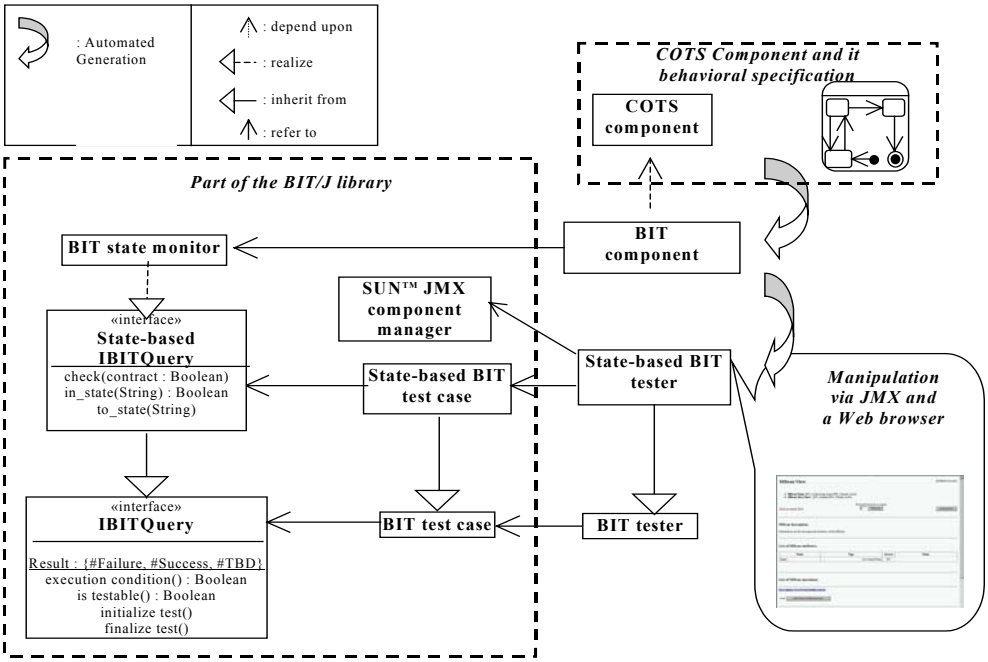

Fig. 1 – Use of the BIT/J library

2.4 An example of the use of the library

In order to illustrate our ideas, we have used very simple stack/queue java classes. These classes have been considered as components and the existing BIT/J tool has been used to generate the corresponding BIT code. Some manual information has been added to the testers to check some particular behavior of the component and the component implementation has been manually altered to illustrate that we were able to detect some specification violations (bad component implementation).

As mentioned before, the BIT/J library is a suite of programs supporting the Built-In Test software technology. The BIT/J library also provides monitoring, configuration and remote testing capabilities based on JMX: a specific agent, which is called a JMX agent, launches a BIT component in its environment. Operations of the BIT component testing interface are accessible via a Web browser from the network. The BIT/J library is supplied with a code generator. It generates a skeleton for the BIT component code as well as the BIT tester code and the JMX code. The use of JMX was not discussed in the previous section because it adds no particular testing capabilities. The parts of the code generated by the tool and that are directly related to JMX, do not need any additional human intervention, as they are mainly wrappings of components into manageable beans (the entities manipulated by JMX).

We reuse in this paper the step-by-step example taken from our user guide[2] which consists in making a BIT version of the Java Stack component. Java Stack is offered by Java SDK without source code. Thus we consider the Java Stack as a COTS component.

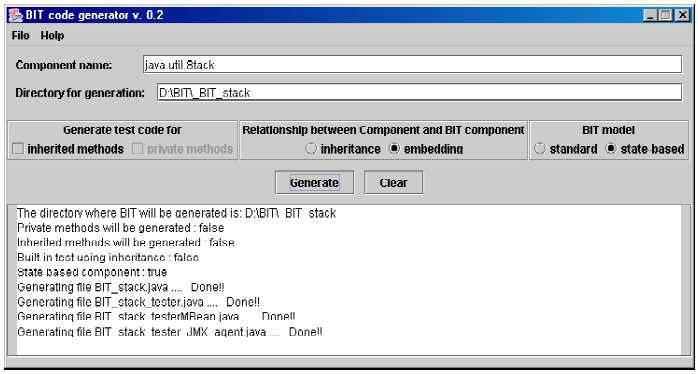

Fig. 2 – The BIT/J generator tool

After the BIT Generator (illustrated in Figure 2) has been launched, and all the fields answered and selected (name of the component, generation directory, use of inheritance or not, etc.), the *BIT component*, the *BIT tester* and the *JMX components* are generated. In our Stack example, four Java files have been generated. The result of the generation process (which is visible on the text area of Figure 2) are: *BIT_stack.java* (the BIT component), *BIT_stack_tester.java* (the tester), *BIT_stack_tester_JMX_agent.java* (the JMX agent, needed for remote testing executions), and *BIT_stack_testeMBean.java* (the JMX interface specifying what are the operations that can be tested through the browser). The JMX files are then modifiable by the user (e.g., removing some operations from the methods list, etc.).

The user then needs to add some specific code into the generated files. Indeed, the BIT Generator supplies a skeleton of the BIT component. Users must, for example, initialize it before using it.

[2] Available at http://liuppa.univ-pau.fr/themes/aoc/aoc/BITJ

First of all, in the *state-based BIT model*, users must implement the BIT component statechart (we have used in our example the one illustrated in Figure 3). To do so, they must modify the `init_behavior()` operation. This is part of the generated code of the BIT Stack:

```
protected void init_behavior()
{
/* state defs and formal relationships here */
}
```

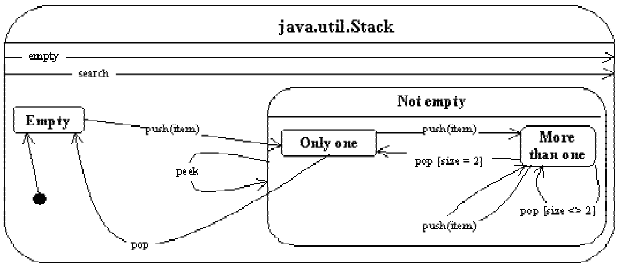

Fig. 3 – The Stack statechart

And here is the modified one:

```
protected void init_behavior(){
/* state defs and formal relationships here */
_Empty = new BIT_state("Empty");
_Only_one = new BIT_state("Only one");
_More_than_one = new BIT_state("More than one");
_Not_empty = (BIT_state)
        (_Only_one.xor(_More_than_one)).name("Not
        empty");
_BIT_stack=new BIT_state_monitor
        (_Empty.xor(_Not_empty), "BIT stack");
_Empty.inputState();
}
```

As can be easily understood in the code, four states were declared (*Empty*, *Only one*, *More than one* and *Not empty*). *Not empty* is a composite state composed from *Only one* and *More than one*. The *BIT Stack state monitor* contains the *Not empty* and the *Empty* states. The testing interface has been generated automatically and no modification is required.

The third part of the skeleton is composed of operations corresponding to public operations within the original component. Each generated operation contains a call to an original operation. Moreover, in the state-based model, users must also code the state transitions appearing in the statechart according to a precise process. For example, below there is (a simplified version of) the final push operation after modification by the user (generated code is in regular font and added one is in *italic*):

```
public Object push(java.lang.Object o1) {
java.lang.Object result = _stack.push(o1);
/* state transitions here */
_BIT_stack.fires(_Empty, _Only_one);
_BIT_stack.fires(_Only_one, _More_than_one);
_BIT_stack.fires(_More_than_one,
_More_than_one);
_BIT_stack.used_up();
return result;
}
```

For this particular operation the user has defined three mandatory state transitions (*Empty* to *Only one*, *Only one* to *More than one* and *More than one* to *More than one*). For the peek operation, a single state transition is necessary which keeps the BIT component in the same state. Lastly, the user could add manually specific testing operations in the BIT component. For the BIT tester, configuration operations may be added to the tester to set the component in a certain state. The JMX tester interface can be modified using the *BIT_component_testerMBean.java* file. The user can set up the accessible operations from the remote testing interface. S/he can, for example, forbid access to some operations by just removing them from this file or add new ones. To be consistent, s/he has to remove or add the associated implementation in the BIT component tester class. The JMX agent, by default, is launched on the port number 8082. If needed this can be changed by modifying the port number into the JMX agent code.

After the modifications, and the compilation of all of the files, the JMX agent and the browser can be launched. The user can manipulate the component through the JMX Web-based interface (see Figure 4).

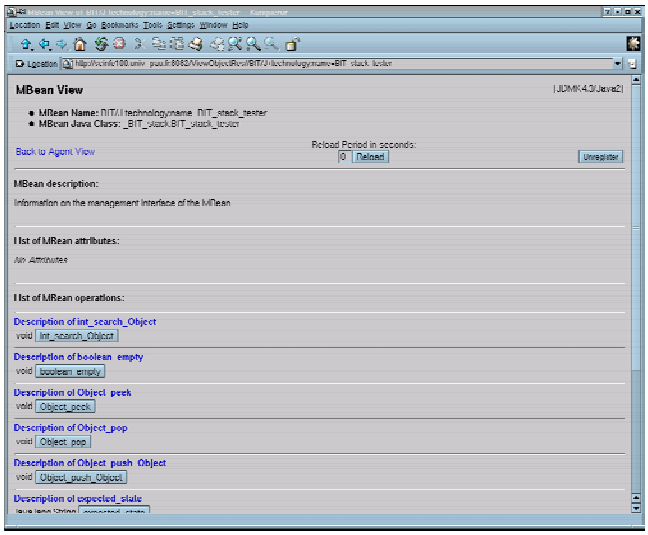

Fig. 4 – The JMX Interface

3. TOWARDS AN ASPECT-ORIENTED APPROACH FOR BIT-COMPONENTS

In this section we discuss our views of how aspects can contribute to help handling CBSD problems, more specifically testability.

3.1 Overview

Figure 5 depicts our general idea, where aspect-orientation should be used during the full software life cycle.

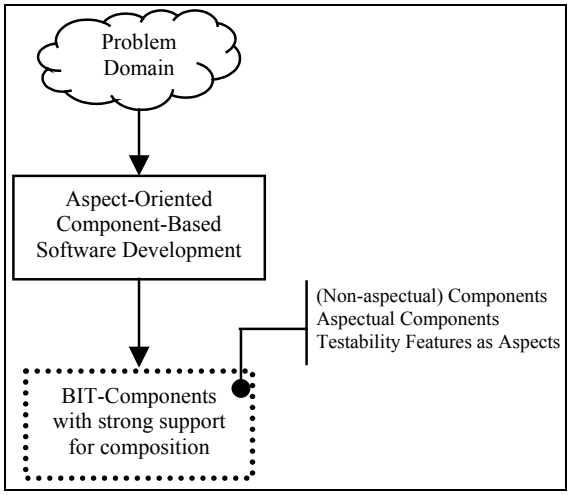

Fig. 5. BIT-Components by way of aspects

Aspects can play a major role in a component-based software development, as they can be used to:

1. develop aspectual components;
2. implement interactions between components;
3. plug and unplug testability features in a component.

The first point is related to the *aspectization* of the component-based software development, i.e., develop components according to the AOSD principles, by applying advanced separation of concerns to their development. To accomplish this we need to borrow from the AOSD its main concepts and techniques. The justification is that components, besides having to satisfy a given functionality, also need to conform to certain design restrictions that affect all (or a subset of its) sub-components. Examples are the so-called non-functional requirements, such as performance, accuracy and security (e.g., [3]). Therefore, it makes sense, in an object-oriented development, for example, to have in a component some sub-components implemented as classes, called here *non-aspectual components*, and others implemented as aspects, the *aspectual components*. The resulting global behavior of that component will be obtained by weaving those aspects with the classes they cut across.

The second point is probably the most interesting one in the context of CBSD. As agreed by many authors [17], the specification and implementation of interactions between components is a difficult task. We believe that aspects can facilitate this job. The composition of components will then be accomplished by composing aspects, which implement interactions between components, and the components themselves.

Finally, the third point proposes the use of aspects to implement built-in test cases. This is not surprising as testability can be seen as a non-functional requirement and therefore better suited to be implemented as an aspect [16, 18]. Such a solution will aim at externalizing from each sub-component, within a component, all the code that implements a test case, by keeping it in a module separated from those that implement both aspectual and non-aspectual functionalities of the component.

3.2 Develop Aspect-Oriented and Component-Based Software

In this paper we take the work already developed for BIT components and show how aspects can be used within that framework at both detailed design and implementation levels.

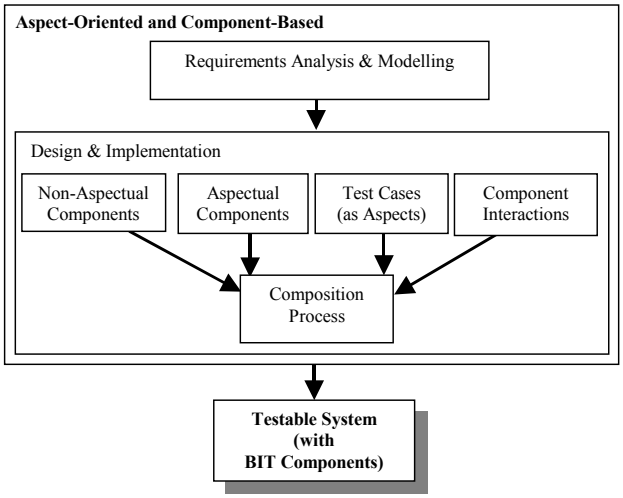

Figure 6. A model for AOCBSD

There are four major activities to achieve before we can think of composing all the elements to form a testable system (see Figure 6). The first and the second ones are dedicated to the implementation of the components, both aspectual and non-aspectual. The third activity aims at defining the test cases necessary to fully test a component by itself. Test cases generation is an interesting area of research. Here we have used state transition diagrams can help testing by adapting path testing. This is a testing strategy whose aim is to exercise all independent execution paths through a component or program [20]. If every independent path is executed then all statements in the component must have been executed at least once. After discovering the number of independent paths the next step is to design test cases to execute each of these paths.

In our approach, test cases will be designed for both aspectual and non-aspectual components. We propose each test case to be implemented as an aspect.

The fourth activity is aimed at specifying and implementing interactions between components. As we said above, this is a very interesting theme in the area of CBSD. Here we propose that interactions between components are implemented as aspects. Having accomplished these four activities we can then weave all the obtained elements and composing them together. The composition process uses the weaving mechanisms available in AspectJ and should first take into consideration the interactions between components already defined.

3.3 The example using AspectJ

We have used the approach described in the previous section to define testability features as aspects. For comparison purposes we will use the stack example discussed in section 2. This section

describes how we have reached the same testability provided by the BIT/J tool, by defining aspects in AspectJ [14]. We also discuss the limitations of the example, and provide some ideas on the generalization of our approach, only illustrated in this paper in the Java world.

The full description of the case studies is available in a detailed technical report [4].

3.3.1 The component

We have worked with a "home-made" implementation of the stack component for two reasons. First, we wanted to highlight the ability to detect bad implementations (e.g. functionally tested, but not conforming to its statechart). Second, we wanted to simplify our work in the use of AspectJ as we had doubts about its introspection mechanisms. Figure 7 depicts a simple class diagram of the simple experimental environment we have developed in Java.

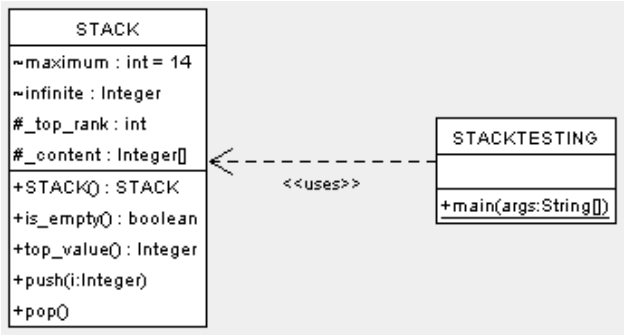

Figure 7. Component implementation

The *StackTesting* program illustrates the typical use and test of the STACK component (a class in our example), when no particular testing functionality is available.

3.3.2 The aspect

According to what we discussed above, the testability features associated with our component should be implemented separately as an aspect. In this paper we are not introducing the reader to the basic concepts of AspectJ. Good tutorials on AspectJ are available [13] and a lot of information can be found on the AOP web site[3].

The aspect, a *modular unit of crosscutting implementation*, is going to be used in order to:

1. add some attributes to the STACK class, and also new methods,
2. declare some *pointcuts* (execution, events, etc. in which we are particularly interested).

The result of the weaving of the class and its aspect will be a testable class. We have chosen in our example to write only one aspect for the component. This is of course not the only possible way to use aspect technology possibilities. It is only an illustration. In fact we believe that it is more readable and consistent to have each test case described by an aspect and have a composition of aspects attached to a particular component.

3.3.2.1 New attributes

The introduction of new attributes (and new methods) illustrates the fact that it does not matter if the component has not been developed to be tested (absence of methods such as *IsEmpty* for example). AspectJ allows us to add whatever is needed. Here is the beginning of the definition of the aspect with the two new attributes *_history* and *_current_state*:

```
aspect AspectedStack{

… STACK._history = new java.util.LinkedList();
… STACK._current_state = "Empty";
```

The two attributes are used to keep track of what is going on in the Stack component, and to implement its state (here initialized to "Empty"). Notice that for readability reasons we have removed some parts from the AspectJ code (e.g. static).

3.3.2.2 New methods

The new added methods implement the test cases. They can complement the interface of the component if needed, and they link the regular functional interface of the component with the way its statechart is going to be described. Our aspect includes: (i) some generic testing functions, such as the one used in the BIT approach, and (ii) some specific methods such as some particular values for parameters the user wants to test, etc. For example, in the case study we add the following *void_push_Integer* method that simply add the integer 999 into the stack:

```
public void STACK.void_push_Integer() {
    Object[] inputs = new Object[1];
    Integer I = new Integer(999);
            push(I);
            _history.add("push"+I.toString());
        }
```

We then define three predicates (*_Empty, _Only_one, _More_than_one*) to check the three possible states (cf. statechart in Figure 3):

```
public boolean STACK._Empty () {
        return _top_rank == -1;}
public boolean STACK._Only_one () {
        return _top_rank == 0; }
public boolean STACK._More_than_one () {
        return _top_rank > 0; }
```

3.3.2.3 Transitions

Transitions are "captured" using *pointcuts* and *advices*. Let us just briefly cite the AspectJ documentation for those not familiar with these concepts: "*A join point is a well-defined point in the program flow. Pointcuts select certain joint points and values at those points. Advice defines code that is executed when a pointcut is reached.*" In our simple example we only consider if the number of items in the stack is increasing or decreasing. We then define two pointcuts:

```
pointcut increasing(STACK s) :
 target(s)&&call(STACK.push());
pointcut decreasing(STACK s) :
 target(s)&&call(STACK.pop());
```

Transitions are treated by *advices* executions. We have used here the *after* mechanism to adjust the Stack state after the event (we only give the code for increasing as an illustration):

[3] http://www.eclipse.org/aspectj

```
after(STACK s): increasing(s){
  if (s._current_state.equals("Empty"))
        s._current_state = "Only_one";
  else if (s._current_state.equals("Only_one" ))
        s._current_state = "More_than_one";
}
```

The overall architecture of the new environment is illustrated in Figure 8 (where we are using a UML notation for describing aspect proposed by [19]). The *TestableStack* is representing in this UML class diagram the conceptual weaving result of STACK and its aspect. We modified the testing program according to the new testability features and made the same kind of manipulation of the component that we did in the BIT version.

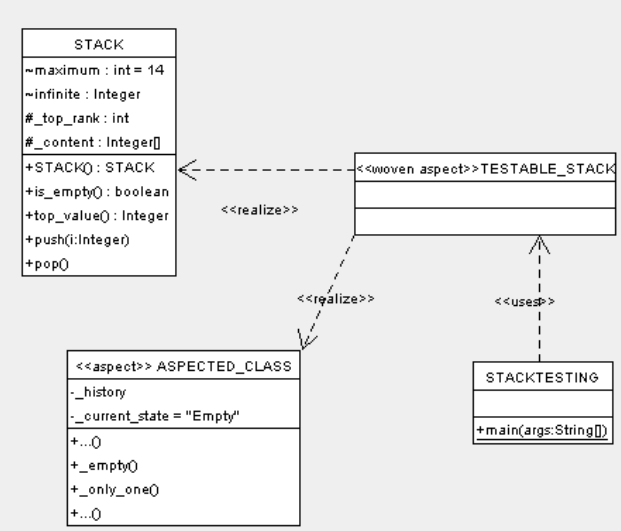

Figure 8. Component + Aspect

The example can be seen as too simple to draw interesting conclusions. We have taken a minimalist approach to show that we could get the same testability power than the one obtained using the BIT/J library. This is why we have started by applying our approach to only one component. We have repeated the experiment with other components, and used different aspect definitions to explore the potential use of aspects. These experiments (described in [4]) have led to promising conclusions. Not only could we repeat the same kind of testability potential we got from the BIT/J, but there are far more possibilities of testing and monitoring components using aspects. Despite potential limitations from AspectJ in comparison with theoretical possibilities of aspects, we could, for example, substitute method calls by others, according to the current state of the component. Here is a short illustration of what it can look like:

```
void around(Stack s): decreasing(s){
   if (s.is_empty())
        System.out.println("Empty Stack!");
   else
        proceed(q);
}
```

This advice, called *around*, is used to substitute the execution of the called method. This is a powerful tool for debugging or implementing an aspect related to non-functional requirement (for example, when memory is low, then use this method instead of this one, etc.). If the method preconditions are verified, then the regular method is executed (through the command *proceed(...)*) otherwise a specific action is executed (in our simple example it is only a message).

But this is only one part of the benefits we found in using aspects. The other, which is only expected because it has not yet been experimented, but it surely will have the most interesting improvements, is when we deal several components linked together by dependencies and client-server collaborations.

4. RELATED WORK

The approach by Grundy [9] is also committed to component based software development. In his method AOCE (*Aspect-Oriented Component Engineering*) he categorizes various aspects of a system that each component provides to end users or other components. The strength of this approach is that it addresses the whole development lifecycle, from requirements to implementation. However, the approach does not contemplate testability to be incorporated in components.

In distributed systems based on CORBA, some interesting ideas can be found in [7] where they extend the existing CORBA Component Model (CCM) with some aspectual features (the overall model being called AspectCCM). They illustrate the need to differentiate between the required interfaces of a component, the ones that "are vital to make the component functional", from the ones that "depend on the context in which the component is developed". They call the former "intrinsic dependencies", and apply the current "uses" CCM artifact, and they call the latter "non-intrinsic dependencies" and then define a new "aspect_uses" artifact, which applies the AOP approach to separate the dependencies from the component itself. Our approach aims to be platform-independent while they are mainly addressing interface definitions in a CORBA-related environment, which is not yet suitable for our purpose.

The closest approach to ours is described in [12], where they use aspects as a way to implement quality of service contracts. They are also convinced that "AOP is an appropriate solution for separating the implementation of a contract from the rest of the model". Their focus is on providing a new and organized way to express non-functional requirements, and aspects comes as an implementation of these contracts, the idea being to reuse AOP weaving approach. As illustrated in this paper, we have started to implement BITs as aspects also in order to reuse weaving capacities of languages such as AspectJ. But our goal is really to have, as soon as at the requirements level, aspects describing features such as testability.

One important issue in our work is the way abstract aspects (the definition of abstract and common testability features) will be implemented into concrete aspects (testability features of one particular component). There are some works that deal with this problem. In [12] for example, they benefit from their existing model transformation tool, UMLAUT, to define abstract aspects. Then, before the weaving, there is an "adaptation" transformation, to first match the aspect with its particular targets, using a kind of parameterized approach. In [10], they try to avoid the "vanishing"

of design patterns throughout the implementation code by implementing these patterns as aspects so that instead of being spread out through the whole application, the pattern remains as an entity. In [5, 6] they strengthen the use of aspects by using composition patterns. They have the merit of illustrating the way aspects at design level can be mapped to implementation level. Similarly, we would like to use extension mechanisms where abstract aspects would be seen as interfaces and concrete aspects have implementation of these interfaces.

5. CONCLUSION AND FUTURE WORK

In this paper we have addressed how aspect-oriented concepts and techniques can be used to improve BIT-technology. In order to illustrate our ideas, we took an existing example of BIT-components and redeveloped it using aspects. We started by building a UML class diagram, extended with some aspectual stereotypes, to represent the new structure of a BIT-component. The next step was to implement the classes and aspects. As the available BIT-library was implemented in Java, we used AspectJ to express testability features with aspects. The results of this experiment were that we not only gained the same testability power but also we could go a lot further in terms of integrating the BIT part within the component.

Thanks to the advantages of aspect-orientation, we believe that enhanced modularization, evolvability and, therefore overall quality of components, will be less difficult to achieve.

For future work we want to contemplate the testability features inserted in a full aspect-oriented software development from requirements to implementation. In particular, we are interested in (i) deriving test cases from requirements models using aspect-orientation, (ii) investigate how aspects can help specifying and implementing composition and interactions between components, (iii) provide an aspect-based composition model for testability features.

6. ACKOWLEDGEMENTS

The work described in this paper is part of the "Agile Requirements Analysis" project, partially funded by the joint Portuguese-French agreement n° F-23/03. The authors from Pau would like to thank Nicolas Belloir for his help in the BIT/J example and for solving our Java problems in general.

7. REFERENCES

[1] Franck Barbier, Nicolas Belloir, Jean-Michel Bruel. *Incorporation of test functionality into software components*. 2nd International Conference on COTS-Based Software Systems, Ottawa, Canada, ISBN 3-540-00562-5, pp. 25-35, 2003.

[2] Nicolas Belloir, Jean-Michel Bruel and Franck Barbier. *BIT/J Library – user's guide*. Available online at: http://liuppa.univ-pau.fr/themes/aoc/aoc/bitj.php.

[3] Jean-Michel Bruel, editor. *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*. Cépadues Edition, Toulouse, France, 2003. ISBN XX.

[4] Bruel, J.-M., and Royer, A. Aspects and BIT: a comparative case study. UPPA Technical Report 2003-07-01, 2003.

[5] Siobhán Clarke and Robert J. Walker. *Composition Patterns: An Approach to Designing Reusable Aspects*. In Proceedings of the 23rd International Conference in Software Engineering (ICSE'2001), IEEE Computer Society Press, 2001, pp. 5-14.

[6] Siobhán Clarke, and Robert J. Walker. *Mapping Composition Patterns to AspectJ and Hyper/J*. In Workshop of Advanced Separation of concerns in Software Engineering, 2001.

[7] Pedro J. Clemente, Juan Hernández, Juan M. Murillo, Miguel A. Pérez, Fernando Sánchez. AspectCCM: An Aspect-Oriented Extension of the Corba Component Model. Proceedings of the 28th Euromicro Conference (EUROMICRO'02). IEEE Computer Press. September 04-06, 2002.

[8] Hans-Gerd Gross, Colin Atkinson, Franck Barbier, Nicolas Belloir, Jean-Michel Bruel. *Buit-In Contract Testing for Component-Based Development*, Chap. 4 in Business Component-Based Software Engineering, Kluwer, vol. 705, ISBN 1-4020-7207-4, pp. 65-82, 2002. http://www.wkap.nl/prod/b/1-4020-7207-4.

[9] John Grundy. *Multi-Perspective Secification, Design and Implementation of Software Components using Aspects*. International Journal of Software Engineering and Knowledge Engineering, vol. 10, No. 6, December 2000.

[10] Ouafa Hachani, Daniel Bardou. *« Aspectisation » des patrons de conception*. In Proceedings of INFORSID'03, Hermès. 2003.

[11] David Harel. *Statecharts: A visual formalism for complex systems*. Science of Computer Programming, 8(3):231--274, June 1987.

[12] Jean-Marc Jézéquel, Noël Plouzeau, Torben Weis, and Kurt Geihs. *From Contracts to Aspects in UML Designs*. In Proceedings of the Workshop on Aspect-Oriented Modeling with UML at AOSD'2002.

[13] Gregor Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, *Aspect-oriented Programming*, in M. Aksit and S. Matsuoka, editors, Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP), Finland (June 1997), Springer-Verlag, LNCS 1241, pp. 220-242.

[14] Gregor Kiczales, et al. *An Overview of AspectJ*, in Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP), Springer. Budapest, Hungary, 2001.

[15] Bertrand Meyer. *Applying Design by Contract*, IEEE Computer Special Issue on Inheritance and Classification, 25(10): pp. 40-52, 1992.

[16] Ana Moreira, João Arujo, and I. Brito. *Crosscutting Quality Attributes for Requirements Engineering*. In Proceedings of the Software Engineering and Knowledge Engineering Conference. ACM Press, pp. 167-174. 2002.

[17] National Coordination Office for Information Technology Research and Development. *High Confidence Software and Systems Research Needs*. January 2001.

[18] Awais Rachid, Ana Moreira, and João Arujo. *Modularisation and Composition of Aspectual Requirements*. In Proceedings of the 2nd Internatinal Conference on Aspect-Oriented Software Development. ACM Press, pp. 11-20. 2003

[19] Junichi Suzuki and Yoshikazu Yamamoto. *Extending UML with Aspects: Aspect Support in the Design Phase*. In Proceedings of the 3rd Aspect-Oriented Programming Workshop at ECOOP'99.

[20] Ian Sommerville. *Sotware Engineering*. 6th Edition, Addidon-Wesley, 2001.

J. Vincent. *Built-In-Test Vade Mecum*, November 2002. Available at http://www.component-plus.org.