Re-thinking the Development Process of Component-based Software

Antonia Bertolino, Andrea Polini
IEI-CNR, Area della Ricerca di Pisa, Italy
{*bertolino, a.polini*}*@iei.pi.cnr.it*

Abstract

This paper contribution to the ECBS workshop is a position statement that a wide gap exists between the technologies for Component-based Software Engineering and the scientific foundations on which this technology relies. What is mostly lacking is a revised model for the development process. We very quickly outline a skeleton for re-thinking the models that have shaped the software production in the last decades, and we start to make some speculations, in particular for what concerns the testing stages. As a working example, we take in consideration the Enterprise Java Beans framework. However, our research goal is to draw generally valid conclusions and insights.

1. Position statement

Since its early moves in the 60's, the history of software engineering has seen on the user's side the progressive growth of expectancies and reliance placed on the software services, and on the producer's side a strenuous attempt to master the consequent escalation of products dimensions and complexity.

To make software production more predictable and less expensive the research efforts have been driven by the two keywords of "discipline", in the form of process models to control the development cycle, and of "re-use", favouring the adoption of OO paradigms. However, despite the efforts, the implementation of a new system "from scratch" involves each time long development times, high production costs and difficulties in achieving further evolution and adaptations to new demands.

A component-based approach to software engineering, similarly to what is routine practice in any traditional engineering domain, seems to provide finally "the solution" to all problems inherent in traditional methods, and we assist today to a sort of revolution in the ways software is produced and marketed.

In Component-based Software Engineering (CBSE), a complex system is accomplished by assembling simpler pieces obtained in various manners. In principle, CBSE perfectly combines the two leading SE principles of "discipline" and "re-use": in fact, only forcing a rigorous discipline on how components are on one side developed, and on the opposite side utilized, a component-based system can be successfully obtained. Moreover, in CBSE re-use of components is one of the leading concerns, and is pursued since the early inception phases.

Ideally, by adopting a component-oriented approach, production times can be reduced, more manageable systems can be obtained, and, above all, such assembled systems can be easily updated by substituting one or more elements in the likely event that future market offerings provide functionalities deemed better than those of the components currently implemented in the system.

It is our concern, though, that current results are not sufficient: the rapid technology advances (e.g., .Net, EJB) are not backed by adequate parallel progress on the theoretical side. In the absence of a reference scientific framework, the proposed technological solutions appear fragmented and unrelated, and their adoption remain difficult and expensive. A software developer is provided with technologies to use and combine components, but is puzzled by the proliferation of partial solutions: a paradigm in which to use them, and criteria to follow in the selection of components and frameworks, are lacking. Paradoxically, the technologies are there, but the conceptual foundations to employ them must still be built.

It is clear that component-based software production requires a major and urgent revision of both the processes and the methods to be adopted in the development of software products. The classical life cycle models are no longer adequate, and also the professional figures that are involved in the software production and business change.

To see why, and what need to be done on the research side, we make some speculations in the following sections. To make the discussion more concrete we specifically focus this position paper on the testing stage and on the EJB framework. However, it is our future research aim to revisit the various stages of the traditional development process, and to develop concrete example within EJB as well as in other popular frameworks.

2. Considerations on the development process for the component-based age

The “standard way” in software production is a phased model in which essentially a phase starts where the previous one finishes. Let us sort out for instance what is typically found in the Table of Content of a traditional textbook in software engineering. There will certainly be a chapter dealing with the requirement analysis stage, a following chapter dealing with design, a chapter dealing with verification and testing, and finally a chapter dealing with maintenance, plus a part putting all these pieces together within a coherent process model. How well and how much does this base structure, that came out from decades of progress, fits within CBSE? The answer is obviously not so well and not so much.

The point is that, even though iterations and concurrent activities may be foreseen among the phases, a “partial order” is always imposed or assumed between the various stages above mentioned. Considering the opportunity of using components requires a totally different process that permits to manage the “non-determinism” introduced by the new approach. We bring in this notion of a non-deterministic process to highlight that, in this context, the various development activities are no longer carried out in any necessary sequence. In fact in the early phases of the development you cannot know if you will find the components already implemented or will have to develop them internally. Also, the specification of the overall architecture may depend on the adoption of certain components. Then, in a certain sense, we need generic process models that can account for the different consequences induced by the use of components produced externally or internally and that establish some “synchronization” points among all the involved stakeholders.

Besides, it is generally recognized that a condition to increase the adoption of components is to design components “for reuse”, and therefore to produce adjustable components not too much shaped to fit within a specific context. That is right, but it guarantees only a part: the possibility. For successfully achieving reuse in practice, it is necessary not to early commit to a fixed system architecture independently from its constituent components, but to consider the components features as well since the early specification and design stages. In this sense we think to an incremental process, whose various phases are concurrent activities focused on recovering and tailoring components or groups of components.

More specifically, in the development of a component-based application we must initially focus on identifying that or those components that provide the basic functionalities. That is to say, we must elicit the functional and non functional requirements for these “basic” components. When candidate components are found, we can test them, against the specified requirements, and choose the best for our objectives. After having identified the first basic components we can go towards the expansion of the application functionalities, in several directions, and look for new components. The specifications for the new searched components must now derive from considerations that include the features of the components already acquired. This cycle is repeated until all the application functionalities are covered.

Perhaps sometimes the search task, for a component, can fail. In this case you can choose to implement the component or you can reduce the required functionalities and retry the search.

Obviously this iterative search-and-refine process is a preliminary idea yet, and it does not want to be complete or definitive, it wants only to illustrate a possible path. In the next section we concentrate the attention on a particular point of the picture showed above, and explain in more detail the testing phase as we imagine it might be expanded in the component development model.

As said, we focus our investigation within the EJB architecture, which has been conceived as a component-based technology to develop server-side applications, particularly in the commercial domain. The EJB platform specification was defined by Sun [1], which has also implemented a reference realization that is freely available for download from the Sun web site [2].

The EJB architecture relies on a complex middleware that manages all the aspects relative to concurrency, security, persistence, and distribution. The management of this complex task by the middleware permit the implementation of simpler components and reduce the risk of error, then the amount of testing.

3. Revising the testing process: a proposal

The distributed component approach makes many traditional testing techniques inadequate or inappropriate, and thereby calls for defining new processes, methods and tools to support testing activities. Weyuker [3] claims that in a component approach the testing performed by the component developers is insufficient to guarantee the component behaviour in new contexts and then underlines the necessity of a retesting made by the component user.

Regarding the costs of production, the advent of true CBSE presupposes the creation of a components market that can make it economically viable to develop software pieces for subsequent assembly. The success of the component approach to development requires therefore thinking in terms of system families, rather than single systems. Consequently, testing procedures must also be refocused: rather than on the definition and maintenance of test suites for single applications, attention must be directed to the development of test patterns for product families. The need for Software Architecture models in the development of component systems is widely recognized [4]. In the stages of testing, such formal models can also be used to generate test cases, either automatically or assisted in some way.

One further complicating factor of the testing activities is represented by components whose source code is unavailable. Such components, in fact, require verification, not only that the features declared by the producer are fulfilled as expected, but also that no undeclared hazardous features are present.

The practical approach that we are going to illustrate seems to be well shaped to the component-based production, and maybe it can reduce the problems mentioned above. It originates from the considerations made in the previous section and is strongly based on the use of the *reflection* feature [5] of the selected language; for this reason the easy choice for us was the Java language.

In accordance with the process model sketched in the previous section, we suppose to have a first phase in which we establish the features that a certain component must have. In our framework this specification must be given in the form of a "virtual component" codified as a class, henceforth named `Spy`, whose required interfaces are established (so the methods and relative signatures). The only duty of every method of this class is to pack the parameters and invoke the method `executeMethod(String name, Object[] param)` of a `Driver` object (that we will illustrate afterwards), passing also to the latter its own name.

From this specification, we can put at work several teams with two different targets:

1. Developing test cases from the specification. If there are more than one team on this target, each of them can focus its attention on a particular feature;
2. Searching suitable components in the organization repository or on the market.

The test cases will be developed on the basis of the methods defined in the class `Spy`, and in a preliminary version the test cases are progressively numbered, for example, `TestCase7`, and each will form a class. All the test cases classes must be collected in a package together with the `Spy` class. Obviously the generated tests are functional/black-box and independent from a real implementation.

The test case and the `Spy` classes must extend respectively the abstract class `TestCase` and `InformationSwap`, both contained in the package `it.sssup.testing`. These classes contain methods that permit to set objects for the re-addressing of method invocation.

The searching of a suitable component is not a trivial task, in fact a real component can look very differently from that defined by the `Spy` class. In particular, we can list five different levels of accordance that, anyhow, guarantee the possible usefulness of a component in the particular application:

1. the methods name are different, but the related names have equal signatures.
2. as above, but with different parameters order
3. virtual methods have less parameter (we must set default values for the real parameter)
4. the parameters have different types, but we can make them compatible, through suitable transformations
5. the functionality of one virtual method is provided collectively by more than one method.

It is however indispensable that these differences are overtaken and for this reason we require that the searching team draw up an XML file to be used by the `Driver` object to drive the testing. In fact after the test packages are developed and at least one component is identified, a team can start the testing of it to verify that it is really compliant with the specifications.

To clarify we can provide a simple example on how we think the approach could work. The example is only declarative and obviously trivial, but we think it can be useful for the purpose.

Suppose that an Italian software house needs a simple software component to manage a bank account, and for this purpose it codifies the following `Spy` class:

```
package bankaccount.test;
import it.sssup.testing.*;
public class Spy extends InformationSwap{
    …
    public void versamento(String cod,int sum){}
    public void prelievo(String cod,int sum){}
    public int bilancio(String cod){}
}
```

From this `Spy` class, the testing teams can produce the test case class as below:

```
package bankaccount.test;
import it.sssup.testing.*;
public class TestCase6 extends TestCase{
   public runTest(){
       int before=spy.bilancio(“123”);
       spy.versamento(“123”,500);
       spy.prelievo(“123”,300);
       if (spy.bilancio(“123”)!=(before+200)){
           System.out.println(“KO”);
       } else { System.out.println(“OK”); }
   }  }
```

In the meantime let us assume that the searching team has found a suitable component, but with different method names (deposit, withdrawal, balance) and also with different parameters order. This team produces the corresponding XML file that specifies the mapping from the virtual object to the real object.

Within the EJB framework, then, we can run the following client, passing to it the name of the package containing the test and the name of the XML file.

```
import it.sssup.testing.*;
public class ClientEJB {
  public static void main(String[] args) {
    try {
      Context initial = new InitialContext();
      Object objref =initial.lookup(
               "java:comp/env/ejb/TrivAcc");
        AccHome home =
          (AccHome)PortableRemoteObject.narrow(
                         objref, AccHome.class);
        Driver dr =
               new Driver(args[0],args[1],home);
        dr.execuTests();
      } catch (Exception e) {}
}
```

Obviously the core of the approach is the package `it.sssup.testing` that contains the specifications of the class `Driver` and of the two abstract classes `InformationSwap` and `TestCase`, that must be extended by, respectively, the `Spy` and the test case classes. The scope of `Driver` is to re-direct the invocation of the virtual methods in `Spy` to the real methods in the component, based on the information contained in the XML file. It is important to note that, in our framework, the implementation of `Spy`, of test cases `and` of test client classes is sufficiently simple and must follows the various specification above outlined.

This model is particularly suited to the context of a complex middleware, such as EJB, because it might solve many questions relative to component integration. In the EJB framework the testing can be performed running a simple tester client. EJB advantage is a strong standardization, or, said in other terms, the “discipline” that we mentioned above, which is the basic philosophy of EJB. Each user-developed bean must comply to the “bean-container contract”, which imposes the realization of precise interfaces.

4. Research directions

The component-based approach opens up several new areas for research. Before all, to permit the growth of CBSE it is necessary to realize more suitable development environments. A first effort in this sense can be found in [6], where seven principal features that a development environment must satisfy are also identified.

A component-oriented world then calls for determining methodologies that can allow component builders and users to agree on the tasks to be carried out by a given component. Research in this field suggests that a component must be endowed with a series of additional information (apart from that making up its interface) that allows it, in a certain sense, to be framed semantically. This information can be used by the customer in the different phases of a development cycle [7], [8]. This line of investigation is particularly important in relation with our approach, mainly regarding the searching task. We have already outlined the difficulties concerning this task; it is desirable, then, to identify information that must reside in the specifications and in the component definition, and that can aid the searching team.

Also in the perspective of establishing an agreement between the customer and the seller, it has been investigated the opportunity that a “certification authority” is established [9]. The goal of this organization is to certify components submitted by the developers. Perhaps, also in this context the approach above depicted can be useful. In fact, the SCL (Software Certification Laboratories [9]) can define “virtual standard components” and provide, for them, benchmarks for several contexts in the form of a package containing the `Spy` and the test cases classes. The developers can then verify their components against these tests, after downloading the package and compiling the XML file. Perhaps this “modus operandi” can simplify the standardization in the production of components. In fact the SCL could define classes of components in the form of the functionality that they must provide.

Regarding more specifically the approach depicted, two directions mainly emerge as possible lines of investigation. The first is a more conceptual work, and is referred to the necessity to develop and clarify in more detail the various phases of the incremental approach. In particular we need to establish methods for extracting test case from the specifications. Besides, by way of real case studies, we want to value the real benefits that the proposed approach can produce in the component-based production.

The second line of investigation, instead, is more practical and concerns the development of tools that assist the different teams implied in the testing activities above mentioned. We refer to the development of tools to aid the drawing up of the XML file, for the searching phase and for test cases extraction.

5. References

[1] B. Shannon, “Java™ 2 Platform Enterprise Edition Specification” http://java.sun.com/j2ee/download.html

[2] J2EE reference implementation. http://java.sun.com/j2ee/sdk_1.3/index.html

[3] E.J. Weyuker, “Testing Component-Based Software: A Cautionary Tale”, *IEEE Software*, Sept./Oct. 1998, pp. 54-59.

[4] D. Garlan, “Software Architecture: a Roadmap”, in A.Finkelstein (Ed.) *The Future of Soft. Eng.*, ICSE 2000.

[5] The Java Tutorial, Reflection, http://java.sun.com/docs/books/tutorial/reflect/index.html

[6] C. Lüer and D. Rosemblum, “WREN – An Environment for Component-Based Development”, in Proc. ESEC/FSE 2001, ACM Sigsoft Vol. 26, N.5, September 2001, pp. 207-217

[7] A. Orso, M.J. Harrold, and D. Rosenblum, “Component Metadata for Software Engineering Tasks”, *EDO2000*, LNCS 1999, pp. 129-144.

[8] J.A. Stafford and A.L. Wolf, “Annotating Components to Support Component-Based Static Analyses of Software Systems”, Proc. the Grace Hopper Celeb. of Women in Computing 2001.

[9] J. Voas, “Developing a Usage-Based Software Certification Process”, *IEEE Computer*, August 2000, pp. 32-37.