A Framework for Component Deployment Testing

Antonia Bertolino, Andrea Polini
Istituto di Scienza e Tecnologie dell'Informazione - "Alessandro Faedo"
Area della Ricerca del CNR di Pisa, Italy
{*bertolino, a.polini*}*@iei.pi.cnr.it*

Abstract

Component-based development is the emerging paradigm in software production, though several challenges still slow down its full taking up. In particular, the "component trust problem" refers to how adequate guarantees and documentation about a component's behaviour can be transferred from the component developer to its potential users. The capability to test a component when deployed within the target application environment can help establish the compliance of a candidate component to the customer's expectations and certainly contributes to "increase trust". To this purpose, we propose the CDT framework for Component Deployment Testing. CDT provides the customer with both a technique to early specify a deployment test suite and an environment for running and reusing the specified tests on any component implementation. The framework can also be used to deliver the component developer's test suite and to later re-execute it. The central feature of CDT is the complete decoupling between the specification of the tests and the component implementation.

1. Introduction

Component Based (CB) development is the emerging paradigm in Software Engineering. It is receiving lot of interest from both academy and industry, as testified by the spreading of devoted events (e.g., [27], [28]) and journal articles (e.g., [29], [30]), and by the launch on the market of component-oriented technological products and platforms (e.g., CCM [31], EJB [32], COM+/.Net [33]).

The captivating feature of CB development is the promise of drastically reducing the high costs of software production. A CB system is built by the assembly of already existing components, thus making true the long lasting myth of developing once for repeated deployment. Despite this very idea is older than thirty years [14], its high potential just starts being exploited.

In our opinion the major obstacles to the taking up of the CB paradigm are not technological, but relate to the process [2]. In traditional development, the process is under the control of one organization (at least in principle) and extends over a definite life cycle period. On the contrary, the process for CB software production is non-deterministically distributed across several organizations that do not necessarily coordinate or communicate with each other and the life cycle is correspondingly scattered over separate time periods. In such a distributed process we identify (among several others not of interest here) the figures of the *component developer* (in a wide sense the organization that develops and releases a component), and the *component customer* (the organization that acquires the component to assemble it within a larger system).

A new challenge that arises from the inherently distributed nature of CB production is the so-called "component trust problem", which refers to the difficulty to understand what a component really does. The trust problem is strongly related to the customer capability to validate the adequacy of a candidate component to his/her specific purposes. Obviously this question is especially hard for components built by third parties and for COTS (Component-Off-The-Shelf), which are generally delivered without the source code. However, also in the case of components reused internally to an organization, the difficulties of communication between teams and the lack of a clear documentation can produce to some extent similar effects.

Several approaches have been proposed to address the trust problem. Some [22][23] advocate a component certification strategy, with the establishment of independent certification laboratories that perform extensive testing of the components, and then publish the results. The approach relies on the conjecture that the customer puts more trust on the results provided by an independent agency than on those provided by the developer himself/herself. Moreover, in order to increase the customer confidence, the developer should adopt certified production standards.

Others authors have argued the paucity of valid accompanying information, often reduced to just the component interfaces, which anyhow provide syntactic information, insufficient for many analysis purposes. Therefore, they [18] [20] suggest approaches to augment the components with additional information (or metadata) aimed at increasing the customer understanding and analysis capability of the component behaviour. A related approach is to automatically extract models of the interfaces of a class, e.g., as Finite State Machines [26]; this information also goes in the direction of increasing the customer understanding of a component usage and scope. In the same approach, the usage of a component is assessed to identify, for instance, wrong call sequences of the component methods.

To deal with the trust problem, a focussed, coordinated initiative [16], has been launched, acknowledging that the solution cannot be univocal, instead a mix of formal and informal approaches should be applied, including formal validation, Design-by-Contract, testing techniques and others.

In this paper we focus on CB testing. As for the other development phases, the testing stage as well needs a rethinking to address the peculiar characteristics of CB development [2]. An important requirement, in our opinion, is that the customer, on the basis of what he/she expects from a searched component, and with reference to the system specification/architecture, develops a test suite and can then routinely (re)execute these tests -without too much effort- to evaluate the potential candidate components.

In [19], Rosenblum outlines a new conceptual basis for CB software testing, introducing the complementary notions of *C-adequate-for-P* and of *C-adequate-on-M* for adequate unit testing of a component and adequate integration testing of a CB system, respectively (where C is a criterion, P is a program and M is a component). A direct application of this model is in the formalization of the problem of adequate testing (with reference to a particular criterion) of a component released by a developer and deployed by one or more customers.

As also recognized by Weyuker [25], the tests established and run by the developer lose much of their power in the realm of the assembled system. In fact, the developer cannot account for all the possible deployment environments. On the other hand, it is illusory to hope that reuse diminishes the need for testing [25][5].

As a consequence of these considerations, a good practice could be to provide the component customer with the test cases that the component already underwent. This additional information can increase *per se* the customer's trust on the component. Moreover, the documented test cases can also be used by the customer in several manners, for instance, to better understand how to use a component, or, by means of adequate mechanisms, to re-execute them in the target environment. This practice can be particularly effective, since the retesting stage performed by the customers can take benefit by the usage of the actual implementations of the required interfaces, in place of the stubs possibly used by the developer.

In view of all the needs depicted above, we have developed the Component Deployment Testing (CDT) framework. CDT supports the functional testing of a to-be-assembled component with respect to the customer's specifications, which we refer to as *deployment testing*. CDT is both a reference framework for test development and codification and an environment for executing the tests over a selected candidate component. In addition, CDT can also provide a simple means to enclose with a component the developer's test suite, which can then be easily re-executed by the customer. The key idea at the basis of the framework is the *complete decoupling* between what concerns deriving and documenting the test specifications and what concerns the execution of the tests over the implementation of the component.

Technically, to achieve such a separation, the framework requires the capability of retrieving at run-time information on the component, mainly relative to the methods signature. In other words, the component to be tested must enable reflection mechanisms [12], allowing for the component runtime introspection. As known, this is provided by the reflection API of Java [11].

Before going ahead with the description, it is necessary to clarify what we intend by a "component". In fact, the term has not a univocal interpretation in the literature. A clarifying and often reported definition is due to Szyperski [21]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* For the sake of generality, we adopt a more simplified view, identifying a component as in [18] with a system or subsystem developed by one organization, deployed by one or more different organizations, and possibly provided without the source code. According to this definition, we will consider also a class or a set of class as a particular example of a component.

In the next section we expand our speculations on the needs for a revised process for CB development. In Section 3 we present in detail the CDT framework. In Section 4 we discuss the usage of the framework, its benefits and how it could be improved. We further discuss related work in Section 5, and briefly draw the conclusions in Section 6.

2. New process for CB software testing

In this section we briefly review how the software testing process is modified in the context of CB production; we discuss in particular what we mean by "Component Deployment Testing".

Traditionally, the development of complex systems involves three main testing phases. The basic phase is unit testing and concerns a relatively small executable program; for instance, in Object-Oriented systems a unit can consist of a class or a group of logically related classes. A second phase is generally referred to as integration testing. In this phase a system or subsystem obtained from the integration of several (already tested) units is tested with the purpose to evaluate their mutual interfaces and cooperation. Finally, the phase of system testing focuses on the various "ilities", or quality attributes, and on the functionalities that characterise the entire system [5]. These three phases are performed along a defined process, keeping the pace with the proceeding of system construction.

In CB development, the three traditional testing phases have to be reconsidered and extended (see Fig.1). The smallest test unit becomes here the component. **Component testing** is performed by the component developer and is aimed at establishing the proper functioning of the component and at early detecting possible failures. The tests established by the developer can rely not only on a complete documentation and knowledge of the component, but also on the availability of the source code, and thus in general pursue some kind of coverage. However, such testing cannot address the functional correspondence of the component behaviour to the specifications of the system in which it will be later assembled.

The phase of integration testing corresponds to the stage we denote by **deployment testing**, however conceptually the two tasks are very different. Even though devising a new development process for building a system by the assembly of software components is outside the scope of this paper, we can certainly assume as a minimum within such a process the following three phases:

1. a specification phase, in which the features of the required components are identified;
2. a searching phase, in which some candidate components are identified, within or outside the organization;
3. a validation phase, in which the identified components are validated, in particular by deployment testing.

With regard to point 3, we need to figure out a testing methodology that can allow for the effective testing of a component by someone who has not developed it, and within an application context that was completely unknown when the component was developed.

Performed by the component customer, the purpose of deployment testing is thus the validation of the implementation of the components that will constitute the final system. The real components identified at this stage are placed in the specific application environment and integrated with the already present components to form the logic structure of the system. It is worth noting that potential mismatches discovered by the customer during deployment testing are not in general "bugs" in the implementation. Rather they evidence the non conformance between the expected component and the tested one (and hence the need to look for other components).

Also for deployment testing (as usual for integration testing) we can consider to adopt an incremental strategy, allowing for the progressive integration of components into larger subsystems. In the presentation of CDT we will always speak in terms of a single component, however the framework could be identically applied to the deployment testing of a subsystem (we return on this in Section 4.1).

A particular case of deployment testing is when a real component comes equipped with the developer's test suite, and the customer re-executes those tests in his/her environment. These tests guarantee that the "intentions" of the developer are respected in the final environment and their execution generally lead to a more comprehensive evaluation. They can possibly include test cases not relevant for the customer's specific purposes, but that can be however useful to evaluate the behaviour of the component under customer's unexpected entries.

Finally, **system test** does not show major conceptual differences with respect to the traditional process (at this level of the analysis) and is performed by the customer when all the various components are integrated and the entire system is ready to run.

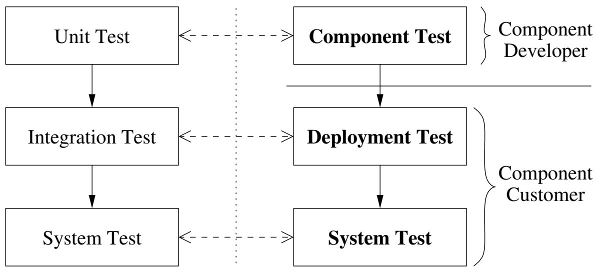

Figure 1. Adapting the test process

3. The CDT framework

In this section we introduce the CDT framework, describing its elements and how it can be used for deployment testing. To help comprehension we provide examples from the simple scenario of a component for managing bank accounts.

In our investigation we suppose that the modelling of a CB system is done by UML and specified following the guidelines expressed in [7] for CB specification (or a similar iterative process) according to which the relevant components are described in term of their provided and required interfaces. In this process we can identify two different levels of components: the *virtual components* that are the di-

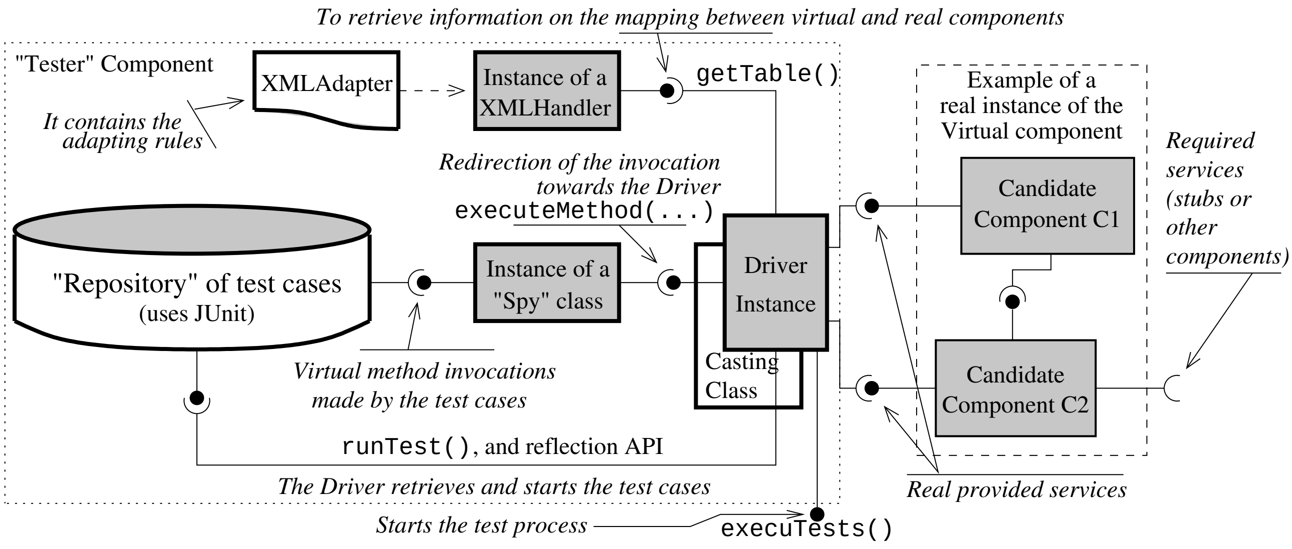

Figure 2. Schema of the framework showing the most important interactions

rect result of the analysis phase, and the *real components*, that will realize the virtual components once assmbled the final system.

3.1. Framework goals

The CDT framework has been designed to suit the needs for component deployment testing, as discussed in Section 2. CDT serves various goals:

- it allows the component customer **to early specify and document the test cases** for a searched component (independently from any possible component implementation);
- it supplies an **environment for the execution** of the specified test cases, once a candidate component is identified;
- it facilitates the **reuse of the specified test cases** for choosing among several candidate components or for substituting an existing one, making possible also the selective reuse of test cases for regression testing purposes;
- as a by-product, it can be used by the component developers **to deliver, together with a component, the test suite** used to test it, thus making easy the re-execution of those tests in the customer's application environment;
- it is a standardised, systematic means to provide a component with **additional behavioural documentation**.

A complete separation of the test derivation and documentation from the implementation of the components was instrumental to achieve these goals. The decoupling is expressly imposed by the same framework architecture, as described below.

3.2. Structure and elements of the framework

The architecture shown in Figure 2 provides an overview of the elements in the CDT framework. For explanation purposes, we can split its elements into three groups:

A. The classes "Spy" and "`TestCase_i`";
B. the XML file "XMLAdapter", and the class "`CastingClass`";
C. the package "`it.cnr.testing`" and the class "`Driver`".

The first group allows for the early codification of the test cases to be later executed on a candidate component. The tests are referred to a virtual component model specified within the "Spy" class. The elements in the second group are used to put in relation the actual component with the customer specifications. The package in the last group drives the testing by opportunely applying the various test cases to the actual component.

Although the three groups are strongly related, different teams are responsible for building and managing them, as we explain in Section 4. Besides, they are built at different stages of development.

3.2.1. The classes "Spy" and "`TestCase_i`". As already highlighted, the main feature of CDT is the decoupling between test cases specification and the actual implementation of a component candidate for assembly. A key role is played here by the classes "Spy". In short, they realize an abstraction of the components, or, in other words, they specify the functionalities expected for the components syntactically expressed by the methods of these classes.

For instance, if we consider a component for the management of bank accounts, without referring to any specific implementation, we can expect for it as a minimum the functionalities expressed by the three methods of the `Spy` class in Figure 3: `deposit`, `withdraw`, and `balance`. In reality, when we look in a repository or on the market for a bank account manager, we will find components much more complex than this, and including multiple classes. Indeed, the correspondence between the "Spy" abstract models and the actual components might result quite complicated; in general, the basic functionalities specified in the "Spy" class might be provided across several objects. We will discuss how we handle this by means of the "XMLAdapter" in the next section.

The implementation of the `Spy` methods does not have to be very elaborate (in Figure 3 only the implementation of the method `deposit` is shown). The methods act as a form of delegation towards the class `Driver` (presented in Section 3.2.3.). In fact, each "Spy" class only constitutes a syntactic reference; at run-time, the `Driver` redirects the virtual method invocations towards the corresponding ones in the real component. The name "Spy" of this kind of class was inspired by this behaviour, in that the role of "Spy" is to provide the `Driver` with the needed information about what a test case does.

After the "Spy" class has been coded, we can develop the deployment test cases taking as a reference the virtual component. Each test case is codified in a class `TestCase_i` that contains the implementation of the method `runTest`. The invocations included in the method `runTest` refer to the methods defined in the corresponding "Spy" class precisely as if it were the component under test. In Figure 4 a simple test case is shown, that invokes the methods of the class `Spy` shown in Figure 3.

We observe that the only external dependence of `Spy` and of `TestCase1` is the package `it.cnr.testing`. The latter is analysed in detail in Section 3.2.3., for the moment it is important to notice that this package is a fixed element of the approach, independent from any specific application context or system. It provides a support tool for test execution, which is developed once and for all. Therefore the classes "Spy" and `TestCase_i` can be built independently from how any candidate real component is implemented.

Finally, as shown in Fig. 2, we also use the well-known framework JUnit [9]. This is a framework developed inside the eXtreme Programming (XP) comunity. Scope of the framework is to give a means to the developer for the easy codification and execution of the test cases concurrently with the development, following the motto "*code a little, test a little, code a little, test a little...*" The framework permits also the simple storing of test cases that can be successively reused. Since we also need to store test cases, we have integrated a part of JUnit in CDT; hence we follow the same rules of JUnit to codify the test cases (see the example in Fig. 4). In this manner we have provided the developed framework with the capability of splitting the test cases in "*target equivalence classes*". This feature is particularly important when we need to modify a virtual component substituting some internal pieces, to avoide to have to re-execute too many tests.

```
package it.cnr.isti.test.bank;
import it.cnr.isti.testing.*;
public class Spy extends InformationSwap {
  /* performs a deposit and returns the
     resulting balance */
  public int deposit(String accNum, int amount) {
    Object[] parameters = new Object[]
                 {accNum, new Integer(amount) };
    Object ris = driver.executeMethod(
                 "deposit", parameters);
    return ((Integer)ris).intValue();
  }
  /* performs a withdraw and returns the
     resulting balance */
  public int withdraw(String accNum, int amount)
  {...}
  /* returns the account balance */
  public int balance(String accountNumber)
  {...}
}
```

Figure 3. A "Spy" class for a bank account manager

```
package it.cnr.isti.test.bank;
import it.cnr.isti.testing.*;

 /* Test codified following the JUnit schema */
import junit.framework.*;
public class BankTest extends TestCase {
  public BankTest() {}
  public void testCase_A() {
    int init=((Spy)spy).balance("123");
    int afterDeposit=((Spy)spy).deposit("123",500);

     /* Control element as specified by JUnit */
    assertEquals((init+500),afterDeposit);
    int afterWithdraw=((Spy)spy).withdraw("123",
                            init+500);
    assertEquals(0,afterWithdraw);
  }
}
```

Figure 4. A possible test-case

3.2.2. "XMLAdapter" and the classes "`CastingClass`".

As said, a class "Spy" does not represent a real component and its purpose is only to codify the desired functionality for a component, so to permit the early establishment of test cases. In this sense, we can say that the role of the "Spy" class is not

```
<MataHari>
      <testPackage name="it.cnr.isti.test.bank"/>
      <testClass name="BankTest"/>
      <realPackage name="real"/>
      <createObject class="REAL_PACKAGE.RealComp" objectName="bank"/>
      <VirtualMethod name="deposit" parameters="accNum␣amount">
            <execMethod class="TEST_PACKAGE.CastingClass" name="par2Couple" putResultIn="c">
              <parameter value="accNum"/>
              <parameter value="amount"/>
            </execMethod>
            <execMethod object="bank" name="put">
                  <parameter value="c"/>
            </execMethod>
            <execMethod object="bank" name="howMuch" putResultIn="aCouple">
                  <parameter value="accNum"/>
            </execMethod>
            <execMethod class="TEST_PACKAGE.CastingClass" name="Couple2Int" putResultIn="output">
              <parameter value="aCouple"/>
            </execMethod>
      </VirtualMethod>
      <VirtualMethod name="withdraw" parameters="accNum␣amount">
            . . .
      <VirtualMethod name="balance" parameters="accNum">
            . . .
      </VirtualMethod>
</MataHari>
```

Figure 5. An excerpt of the “XMLAdapter” file

different from that of an interface. At a certain stage of development, the component modelled by the “Spy” class must be instantiated: at this point it will be necessary to evaluate potential candidate components, using the developed test cases. Although the searching of the candidate components is driven by the same system specification used to build the classes “Spy”, we think it reasonable to only focus on the behavioural aspects of the specification. In other words, we search for components that provide the desired functionality, but in this search we neglect possible syntactic differences.

As a result, several differences at various levels can exist between the virtual component (codified in a “Spy“ class) and a real instance of it. The purpose of the “XMLAdapter” and of the `CastingClass` is to overcome these differences and to permit anyway the execution of the specified test cases. Clearly this can be done only after a real implementation for the virtual component has been identified. To establish the correspondence between the virtual component and the real instance, a customer can rely on his/her intuition of what the methods of the candidate components likely do. This intuition can be based on the signatures of the methods, and on any additional documentation that accompanies the components. Obviously this process (which is always the most delicate part of a CB development) is subject to misinterpretation (especially if the components are not adequately documented). However, deployment test execution should highlight possible misunderstandings. For instance, in our case study we have analysed the following levels of possible differences:

1. differences in the methods names and signatures:
   a. the methods have different names;
   b. the methods have the same number and types of parameters, but they are declared in different order;
   c. the parameters have different types, but we can make them compatible, through suitable transformations. It can be also necessary to set some default parameters;
2. one method in the “Spy” class corresponds to the execution of more than one method in the real implementation of the component.

Obviously the instances listed above are not mutually exclusive, and for instance it is possible to have different name methods with different signatures. It may be worth notice that the symmetric case to 2 (more methods in the “Spy” class correspond to one method in the real implementation) is not generally relevant. In fact, the “Spy” class is kept simple and typically contains a minimal number of necessary methods. If, say, two methods in the “Spy” class correspond to one method in the real implementation, then either the real implementation is not compatible, or we do not need to invoke the two methods alone, but always together and in the same sequence. If so, then it would be more intuitive to indicate only one method in the specification of the class “Spy”.

Item 2 also includes the case in the list enclose also the case in which the real implementation of a virtual component is achieved combining together more candidate real

```
package real;
public class RealComp {
  private int sum;
  public RealComp() { this.sum = 0; }
  public RealComp(int sum) { this.sum = sum; }
  public void put(Couple c) {
    sum = sum + c.getSum();
  }
  public int get(Couple c) {
    int imp = c.getSum();
    if ( (sum-imp) > 0) {
      sum = sum - imp;
    } else {
      System.out.println("Not␣enough␣deposit");
    }
    return sum;
  }
  public Couple howMuch(String accNumber) {
    return new Couple(accNumber,sum);
  }
}

package real;
public class Couple {
  int sum; String name;
  public Couple(String name,int sum) {
    this.sum = sum;
    this.name = name;
  }
  public int getSum() { return sum; }
  public String getName() { return name; }
}
```

Figure 6. A real component

```
package it.cnr.isti.test.bank;
import real.*;
public class CastingClass {
  public static int Couple2Int(Couple c) {
      return c.getSum();
  }
  public static Couple par2Couple(String name,
                              Integer amount) {
      return (new Couple(name,amount.intValue()));
  }
}
```

Figure 7. The `CastingClass`

components. In this case the methods executed in a test case could belong to different real components. However we think that a good design made in a well established CB environment generally should lead to a one-to-one correspondence between virtual and real components.

In Figure 5 we have reported an excerpt of the "XMLAdapter" for the candidate component in Figure 6 and the "Spy" class in Figure 3. The file is structured in several sections delimited by specific tags. The first half of the files reports information regarding the elements to be tested and that can mostly be derived automatically. In detail, first we put the specification of the packages containing the test case classes. It can be in fact useful to group the various test cases in more packages, according to suitable criteria, so to permit a separate reuse of groups of tests. Afterwards we specified the names of the packages containing the real components (those to be tested), and then we have specified the specific classes to instantiate within those packages.

The second half of the file is more complex and reports the information on the correspondences among the various methods in the considered "Spy" class and those in the real candidate components. As an example, we have reported the specified correspondence for the method `deposit` (see Fig. 3) with regard to the real component in Figure 6. If we observe the methods in Figure 6, we note how there is no method in this class directly corresponding to the method `deposit` as specified in the relative class "Spy". However (e.g., by further analysing the class interface in the real component or from the accompanying documentation), we suspect that a compatible behaviour can be obtained by the consecutive execution of the methods `put` and `howmuch`. This case illustrates one of the trickiest case of correspondence, where nothing seems to fit at first sight. Moreover the returned type of the method `howMuch` is not the same of that expected in `deposit`, and then, for compatibility purposes, we need to perform a type conversion.

In general, type conversions can be also required for the input parameters. It is possible in principle to perform these conversions in declarative way via XML. However in some cases the conversion can be very sophisticated, disabling this possibility in practice. For this reason we have introduced in CDT the other element of this group, called the `CastingClass`, in which the "casting" between objects is specified via class methods. In Figure 7 the `CastingClass` for the example discussed is shown. It contains two methods that permit the conversion of the basic parameters in the `Spy` class to the type `Couple` in the candidate component and vice versa.

As a final consideration, the drawing up of the "XMLAdapter" is certainly not an easy task. Nevertheless this task can be partially automated and alleviated with the implementation of suitable tools and graphic interfaces.

3.2.3. The package "`it.cnr.testing`" and the class "`Driver`". The classes contained in this package form the core of the approach. Differently from the elements described so far, that have to be implemented and customized for each new CB system, the elements contained in this package are completely independent from the application context.

There are two main elements in this package, the class `Driver` and the XML parser (named `XMLHandler`). The only duty of the latter is to retrieve the data from the XML file and organize them into suitable data structures. The information so organized can be used by the `Driver` to redirect, at run-time, the invocations made by the test cases to the associated class “Spy”, towards the real methods of the selected candidate components. More precisely, in the implementation of the class `Driver` we have used the introspection mechanisms provided by the component model (in our case the reflection API of Java), to identify and invoke at run-time the methods of the candidate implementation as specified in the “XMLAdapter”.

The class `Driver` is quite large and for space reasons we cannot report it entirely here, but present its most important features. There are only two public methods in the class `Driver`:

1. `executests()`: the invocation of this method starts the deployment test. Hence as first thing the `Driver` retrieves the information reported in the “XMLAdapter” via an instance of the XML parser (the class `XMLHandler` in Fig. 2). Obtained this information the `Driver` sets up an instance of the class “Spy” that specifies the Virtual component under test, and then instantiates the elements of the real implementation. At this point we need to identify the test cases to execute. Hence, the `Driver`, using the provided reflection mechanisms, retrieves, from the packages containing the test cases, the tests to execute, as specified in the “XMLAdapter”, and provides them with the reference to the pertinent “Spy” class. This last step is mainly performed reusing the mechanisms defined in the JUnit framework. Now the `Driver` can start the testing invoking on each suite retrieved the appropriate method that as a result provides a “report” concerning the test execution.
2. `executeMethod (String name Object[] par)`: this method is invoked by the instance of the “Spy” class “to inform” the associated `Driver` object of the method executed by the test case. On the basis of this information and of the data retrieved from the “XMLAdapter”, `executeMethod` invokes the corresponding method/methods in the real implementation of the component.

3.3. CDT described in terms of design patterns

Now that all the main elements of the framework have been illustrated, to help comprehension it may be useful to briefly look at it in terms of design patterns [8] (with the warning that an accurate study of a framework, even a relatively small one, in terms of design patterns would be too long to fit into one paper, see, e.g., [9]).

Among the patterns presented in [8], the CDT framework shares the intent of the *Adapter* pattern described as: *Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.*

In fact, the CDT structure is slightly different from the one in [8], as a consequence of the high dynamic nature required to set a clear distinction between tests and component implementations, enabled by the use of introspection mechanisms, and to easily permit the integration in the framework of additional functionality. Therefore more precisely we can say that CDT realizes a *dynamic run-time adapter* that is reconfigured each time a new component is deployed (by using the “XMLAdapter” as an input). Figure 8 shows how the adapter structure can be recognized in the framework. The test case classes represent the clients of the target object constituted by an instance of a “Spy” class. The invocations made by the methods of this instance are directed to an object within `Driver` that, on the basis of the information contained in the XML file, “adapts” them to an invocation, or a sequence of invocations, of the methods of the candidate implementation. This structure is not static, but is visible only at run-time, after the instance of the class `Driver` has acquired a reference to the candidate components as consequence of the information contained in the XML file.

Moreover, it should be now clear that the duties of each instance of the class `Driver` are not limited to the adaptation of the invocations in a class “Spy”, but also include the control of the test process, in particular retrieving and instancing the test cases classes and invoking on these instances the opportune method.

The choice of this kind of adaptation, i.e., the use of the XML file instead of a more static one realized implementing a class adapter, was mainly driven by the necessity of obtaining an easily extensible architecture. In particular, we want to avoid the need of building, each time, complex adapters that have to contain additional logic to perform duties as contract verification or traceability. In our solution, all this logic in fact is implemented once and for all in the `Driver` class, while in the “XMLAdapter” we only have to explain the methods correspondences.

4. Considerations on the Framework

This section explains how the CDT framework can be useful, in different regards, both to the component customer and to the component developer. We illustrate which are the main benefits gained by using the framework, and in which directions our research will continue from here.

4.1. The good news for the component customer

With reference to the phases of specification, searching and testing of a component (see Section 2), CDT can be fruitfully used to support and distribute the relative tasks among different teams. The “Spy” classes can be developed as a result of the specification phase: as said, the func-

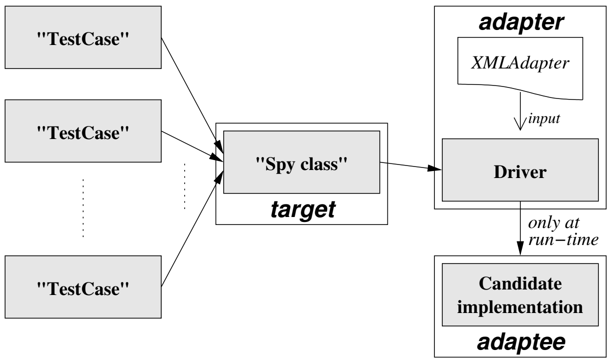

Figure 8. The CDT framework as a run-time adapter

tionalities required for a virtual component are specified as methods of the class. After that, two different kinds of team are involved: the *searching teams* and the *testing teams*. The former, on the basis of the specifications for the virtual component, looks for suitable implementation. If one seemingly good candidate implementation is found, they develop an appropriate "XMLAdapter" (the implementation, as already highlighted in paragraph 3.2.2., can consist of more real components). In parallel, the testing teams can start developing functional deployment tests with reference to the virtual component represented by the "Spy" class. When a candidate implementation is identified, and at least a test suite is ready, the deployment test phase can start.

So far, in the presentation of the CDT framework we have mainly analyzed the validation of a single virtual component, however the CDT framework can be used for incremental deployment testing without any modification. In fact, using an appropriate instance of a class "Spy" and a suitable "XMLAdapter", a set of interconnected components can be viewed as a single component that exports the necessary interfaces. More in general, we can figure out a deployment test phase that is organized in incremental steps (precisely as in traditional integration testing). In such a scenario, a component may be tested first alone, and then as part of a larger subsystem. In the latter case, the functional tests are not related to the behaviour of the component alone, but of the subsystem in which the component is integrated.

In order to simplify the integration of more virtual components, as well as the integration of more real components to form a virtual one, we have presented in [3] a particular wrapper named *WCT* (Wrapper for Component Testing) that permits the easy configuration of a system according to the specified system architecture. WCT has a similar structure of the "Tester" component in Figure 2, and its main duty is to adapt the services required by the contained component to those provided by other virtual components. A WCT foresees also mechanisms for execution tracing and contract verification. This last feature results particularly beneficial since it permits the early identification of a problem during test execution through the detection of a contract violation in the interaction between the implementations of the virtual components (this technique was presented in [6] for the testing of OO systems).

4.2. The good news for the component developer

We have discussed in the introduction the benefits that the component customer could gain from getting, together with a component, the related functional test cases. As a consequence, the opportunity to obtain the developer's test cases, established in a white-box context (i.e., with a full knowledge and access to the source code), and packaged in a way that makes them easily re-executable, can make a big difference in the component customer's choices within a competitive market. This fact could convince the developers of the advantages to identify effective methods for transferring their test suite to the customer.

In this respect, the CDT framework can also be usefully applied by the component developer to transfer executable test to the customers. The developer might define the test cases on the basis of his/her own developed "Spy" class, that in this case will obviously not show substantial differences from the actual component, and should also codify the "XMLAdapter", which will be trivial. The customer, obtained these artefacts, will be immediately able to test the component in the target environment, using the same mechanisms employed to execute the internally developed test cases. If the major advantages in the described scenario will be for the component customers, also the developer can gain benefits in codifing the test cases following the rules established in the CDT framework, in order to test successive versions of a component in which the interfaces definition could be modified.

4.3. Major benefits and further developments

In developing the framework, the main purpose was to provide the component customer with a means for the early codification of test cases in a manner completely independent from a specific real component and for easily executing them to evaluate any candidate component. We can ascribe several good qualities to the resulting framework:

1. the complete decoupling of tests and components, thus facilitating the separate reuse of these two kinds of artifacts (inside the same organization);
2. the test suite flexibility, e.g., it results extremely simple to add new test cases;
3. the grouping the test cases in different packages on the basis of relevant criteria; this feature can simplify subsequent regression test phases;

4. no specific interface implementation is requested on the candidate components.

With regard to item 4, for this same reason the framework does not permit the codification of tests that access the private elements of the components, capability that other methodologies, like Built-In Testing [24], foresee. However, by adding a simple requirement to the implementation interfaces we could obtain, using the framework, the same power of the built-in tests. Precisely, to obtain this, we have to impose that each component implements a public method that using the introspection mechanisms of the component model, redirects the invocation on the private elements, similarly to what is also presented in [10].

For instance, in Java we could impose the implementation of the method:

```
handlePrivateElement(
    String Kind,String Name,
    Object[] parameters,String op);
```

in which `Kind` $\in$ {CONSTRUCTOR,METHOD,FIELD}, `Name` is the name of the private element, the vector `parameters` contains the parameters, and `op` $\in$ {SET,GET} and specifies the type of the operation that must be performed when accessing a field element.

We have already noted that the framework is easily extensible with new control functionalities. An extension currently under development is the integration in the framework of mechanisms for contract verification. Design by Contract (DbC) [15] seems one of the most promising means in order to alleviate the "component trust problem". It can be exploited all along CB systems construction to evaluate the conformance of the real component to the searched one. In the area of testing, DbC can be used as a further means to validate the execution of a test case. The integration in CDT results quite simple and we think that, thanks to the fine grained control of the `Driver` on the methods invocation, the result can be effective. So far we have already integrated in the framework the possibility of instrumenting the virtual components with pre- and post-conditions that are checked at run-time. A further development on which we are working is the adding of traceability features to the framework in order to simplify the result analysis. Besides, another important piece of work that we plan is the realization of a graphical tool to support the compilation of the "XMLAdapter". Such a tool would obviously reduce the effort spent in test development and moreover it would drastically reduce the probability of errors.

5. Related work

The CDT framework is proposed as a practical contribution to alleviate the component trust problem. In the Introduction we have already extensively discussed it and have overviewed several approaches related to ours.

The special needs posed by CB development are widely discussed in [13], and a prototype environment called WREN is proposed. A strict relation clearly exists between this and our approach, in that both studies call for a process revision and start from similar requirements. However, while WREN applies in general to component composition issues, we specifically focus on the testing task. A combination of the two frameworks is not only feasible, but also desirable.

With regard to our technical solution, we are not aware of other projects specifically aimed at supporting the component customer in developing test cases for the deployment testing phase, without assuming any specific real implementation. In this respect we believe that our work is original and hopefully pioneer of an important research direction. Instead, some interesting approaches have been proposed, that the developer can use to transfer the test cases to the customer. In this direction, the authors of [17] present a tool to allow the component developer to design and run test cases. The test specification and the test results are stored into a XML document to be successively packaged with the component, thus providing to the component user a means to verify the component compliance to the specifications in the target environment.

Another well-known approach consists in enclosing the test cases as public methods of the component (the already cited Built-In Test approach [24]). The disadvantage of this methodology is the growth of component complexity, which soon makes the component hard to manage.

An interesting approach presented by Gao et al. [10] introduces the concept of a "testable bean" as a component with well defined built-in test interfaces. Goal of these interfaces is to enhance the component testability and facilitate component testing. With a mechanism that uses the introspection mechanisms of the component model, similar to what we have described above, they obtain the capability of built-in testing, but without enclosing a lot of code in the component itself.

As CB testing is a very though and expensive task, we believe that all the above approaches constitute important contributions, but that the field is far from being settled, and much more research is still needed.

6. Conclusions and future work

We introduced the CDT framework, to facilitate the testing within a customer's target environment of a component independently developed.

The central feature of the framework is the complete decoupling between the tests and the component implementation, allowing for the early codification of deployment tests, and for their reuse over different components.

This research goes in the direction of providing means to increase the trust in a component. We believe that the success of CB development passes through the adoption of a rigorous, systematic test validation process.

We have implemented a trial version of the CDT framework in Java and played it on a simple case study of a bank account manager. The empirical work demonstrated the feasibility of the idea and at the same time indicated several interesting additions to the framework structure that we plan to implement in the next future.

We plan also to investigate, in the next future, how to derive useful test cases from specifications codified in UML diagrams. In particular in a recent paper [4] we sketch some ideas on adapting Cow_Suite [1], a tool to derive test cases for OO systems, to CB development, and on how this could be integrated with the CDT framework.

7. Acknowledgements

Andrea Polini's PhD grant is supported by Ericsson Lab Italy in the framework of the Pisatel initiative. http://www.iei.pi.cnr.it/ERI

References

[1] F. Basanieri, A. Bertolino, E. Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects", in *Proc. ≪UML≫ 2002*, Dresden, Germany, Sept. 30-Oct. 4, 2002, LNCS 2460.

[2] A. Bertolino, and A. Polini, "Re-thinking the Development Process of Component-Based Software" in [28].

[3] A. Bertolino, and A. Polini, "WCT: a Wrapper for Component Testing", in *Proceedings of Fidji'2002*, Luxembourg, November 28-29, 2002, to appear in LNCS.

[4] A. Bertolino, E. Marchetti, and A. Polini, "Integration of "Components" to Test Software Components", to appear in *Proceedings of TACoS 2003* workshop at ETAPS 2003, Warsaw, Poland, April 13^{th}, 2003.

[5] R.V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 2000.

[6] L.C. Briand, Y. Labiche, and H. Sun, "Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code", *Proc. ACM ISSTA 2002*, Roma, Italy, July 22-24, 2002, pp. 70-80.

[7] J. Cheesman and J. Daniels, *UML Components - a Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Rusable Object-Oriented Software* Addison-Wesley, 1995.

[9] E. Gamma, and K. Beck, "JUnit a Cook's Tour" available at: http://www.junit.org

[10] J. Gao, K. Gupta, S. Gupta, and S. Shim "On Building Testable Software Components", in J. Dean and A. Gravel (Eds) *Proc. ICCBSS 2002*, LNCS 2255, pp.108-121.

[11] Java Reflection Documentation, available at: http://java.sun.com/products/jdk/1.1/docs/guide/reflection/

[12] G. Kiczales, J. des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[13] C. Lüer, and D.S. Rosenblum, "WREN - An Environment for Component-Based Development", in *Proc. of ACM ESEC/FSE 2001*, Vienna, Austria, September 10-14, 2001.

[14] D. McIllroy, "Mass Produced Software Components", in P. Naur and B. Randall Eds, Software Engineering: Report on a Conference by the NATO Science Committee, Brussels, 1968, pp. 138-155.

[15] B. Meyer, "Applying Design by Contract", *IEEE Computer*, vol. 25, no. 10, October 1992, pages 40-51.

[16] B. Meyer, C. Mingins, and Heinz Schmidt, "Trusted Components for the Software Industry" available at: http://trusted-components.org/documents/tc_original_paper.html

[17] J. Morris, G. Lee, K. Parker, G.A. Bundell, and C.P. Lam, "Software Component Certification", in *IEEE Computer*, September 2001, pp. 30-36.

[18] A. Orso, M.J. Harrold, and D. Rosenblum "Component Metadata for Software Engineering Tasks", in W. Emmerich and S. Tai (Eds) EDO2000, LNCS 1999, pp. 129-144.

[19] D. Rosenblum, "Adequate Testing of Component-Based Software", Univ. California, Irvine, T.R. UCI-ICS-97-34, (1997)

[20] 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.

[21] C. Szyperski, *Component Software, Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

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

[23] J. Voas, "Certifying Off-the-Shelf Software Components", *IEEE Computer*, June 1998, pp. 53-59.

[24] Y. Wang, G. King, and H. Wickburg, "A Method for Built-in Tests in Component-based Software Maintenance", *Proc. of the 3rd ECSMR*, 1999.

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

[26] J. Whaley, M.C. Martin, and M.S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces", *Proc. ACM ISSTA 2002*, Roma, Italy, July 22-24, 2002, pp.221-231.

[27] 5th ICSE Workshop on CBSE, Benchmarks for Predictable Assembly, Orlando, Florida, USA, May 19-20, 2002.

[28] ECBS 2002 Workshop on CBSE, Composing System From Components, April 10-11, 2002, Lund, Sweden.

[29] *IEEE Computer*, July 1999, 32 (7).

[30] *The Journal of Systems and Software*, Special Issue on CBSE, 2003, to appear.

[31] CORBA Component Model specifications: http://www.omg.org/technology/documents/formal/components.htm

[32] Enterprise Java Bean Technology: http://java.sun.com/products/ejb/

[33] .Net resources available at: http://www.microsoft.com/net/