International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:10, 2008

Unit Testing with Déjà-Vu Objects

Sharareh Afsharian, Andrea Bei, and Marco Bianchi

***Abstract*—**In this paper we introduce a new unit test technique called déjà-vu object. Déjà-vu objects replace real objects used by classes under test, allowing the execution of isolated unit tests. A déjà-vu object is able to observe and record the behaviour of a real object during real sessions, and to replace it during unit tests, returning previously recorded results. Consequently déjà-vu object technique can be useful when a bottom-up development and testing strategy is adopted. In this case déjà-vu objects can increase test portability and test source code readability. At the same time they can reduce the time spent by programmers to develop test code and the risk of incompatibility during the switching between déjà-vu and production code.

***Keywords*—**Bottom-up testing approach, integration test, test portability, unit test.

I. Introduction

UNIT testing is a process of testing the individual classes in a program. The purpose of unit testing is to compare the function of a class to some functional or interface specification defining the class. In ideal conditions a unit test should test in isolation the class under test. Unfortunately, it is not simple to test a class in a vacuum, especially when an object oriented language is used. In fact, since an object oriented program is a set of objects that send and receive messages, object under testing uses to interact with other objects (see Fig. 1). As a consequence, the problem to face is to place the object under testing in a simulated environment, where its interactions with the environment are controlled and, possibly, verifiable.

Some testing techniques can be adopted to properly implement unit test, such as *stubs* [1], *server stubs* [2], and *mock objects* [3]. All these technique, referred as unit testing techniques in the following, have the same goal: to isolate the object under testing. With the exception of stubs, this isolation is reached replacing objects used by the tested object, *real objects* in the following; with *faker* ones (see Fig. 2). In fact, faker objects can be set to return results useful to properly test the object under testing. Furthermore, some testing techniques (e.g. mock objects) also allow to verify if fakers have been

Sharareh Afsharian is with Ericsson Lab Italy, Rome and Computer Science Department, University of L'Aquila, L'Aquila, Italy (e-mail: afsharian@ ericsson.com).

Andrea Bei is with Netlab, IASI "Antonio Ruberti" - National Research Council, Rome, Italy (e-mail: Andrea.Bei@dis.uniroma1.it).

Marco Bianchi is with IASI "Antonio Ruberti" - National Research Council, Rome, Italy (e-mail: bianchi@iasi.cnr.it).

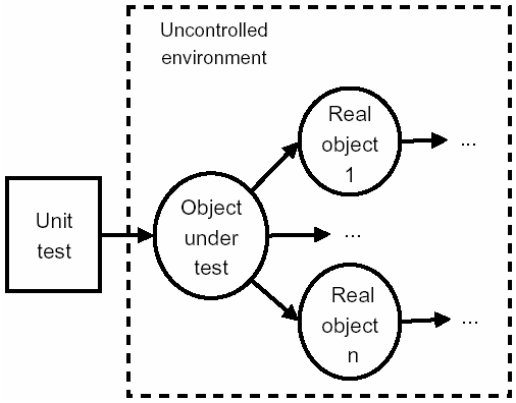

Fig. 1 An object tested in an uncontrolled environment interacts with several real objects. These possibly interact with other objects, and so on

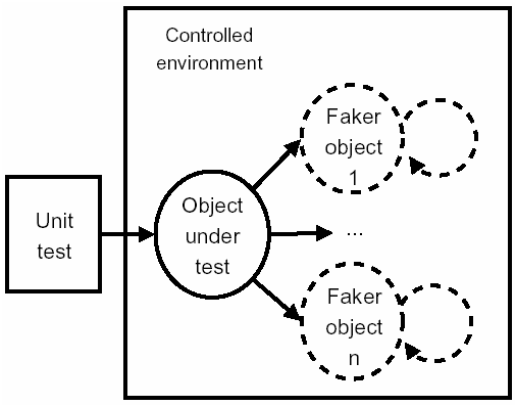

Fig. 2 An effective unit test is run in a controlled environment. Real objects are replaced by fakers set to return results useful to properly test the object under testing

properly used by the object under test during the test execution. Sometimes the adoption of unit testing techniques becomes a necessity: it happens when the behaviour of real objects is complex with respect to the test object's environment. For example, when a real object does not yet exist or may change behaviour, or supplies non-deterministic results (e.g. the current time or the current temperature), or has states that are difficult to create or reproduce (e.g. a network error) [3].

Unit testing techniques are often used also during integration testing. An integration test is a type of testing in which software and/or hardware components are combined and tested to confirm that they interact according to their

requirements. Integration testing can continue progressively until the entire system has been integrated [11]. If this is the case, a system integration plan is necessary. The system integration plan defines the order of integration, the functional capability of each version of the system, and responsibilities for producing ”scaffolding”, code that simulates the function of nonexistent components [1]. With respect to integration tests, stubs, server stubs or/and mock objects can be adopted to simulate components and/or subsystem not still integrated.

The adoption of unit testing techniques also presents some undesirable side effects, such as more code to implement and added complexity in the test source code. These drawbacks are amplified when configuration of fakers needs a lot of human effort. For example, when a faker object has to return a large data set as result of a SQL query, the initialization of the object to be returned can be a very verbose activity.

In our opinion, if you follow a bottom-up development and testing strategy, the impact of these kinds of side effects could be reduced *delegating* real objects to properly configure fakers. Starting from this idea, we developed a new technique called déjà-vu object.

A *déjà-vu object*, déjà-vu in the following, is an object able to observe and record the behaviour of a real object during real sessions, and to replace it during tests, returning previously recorded results. A déjà-vu can be configured to play the role of a faker as well as possible. In fact, it can exactly replicate the behaviour of the real object or not: in the first case, déjà-vu throws an exception, for example, if the object under test does not respect the exact order of method calls observed during the real session; in the second case, it is possible to specify default results for each method among those have been recorded, or how many times to return a certain result, and so on. A remarkable feature of a déjà-vu is the ability to play the role of events source. In fact, it can be set to reproduce a source of asynchronous message flows, simulating real timing or not.

The usage of the déjà-vu technique can be useful in several cases. For example, when a real object:

- requires a lot of time to be properly configured or dummy programmed;
- requires particular environment resources not available during the test phase (e.g. software/hardware resources);
- has states or behaviour that are difficult to create or reproduce (i.e. the development of a simulation model and its implementation is burdensome);
- would have to include information and methods exclusively for testing purposes;
- has a behaviour with an elapsed time too high (e.g. high response time, rare event generation, ..) to produce efficient unit test when the test is time independent.

and the faker object has at least one of the following feature/ requirement:

- objects passed as parameters to or returned by a faker requires too much time to be properly defined;
- faker has to reproduce an asynchronous flow of events;
- faker has to return *real* results.

Depending from the context in which it is adopted, it allows to increase the test portability, and/or the test source code readability, and/or the ”reliability” of messages exchanged by object under test and the faker one (since they are derived by the observation of a real session), and/or to decrease time spent by programmers to develop test code.

It is worth to note, déjà-vu technique is not applicable when a top-down development approach (e.g. Test Driven Development [4]) is adopted, because in this case the real object does not yet exist or may change behaviour. Furthermore, in presence of real objects supplying non-deterministic results, the creation of a déjà-vu object can be time-expensive, because it can be difficult to establish when the observation period has to be started and stopped.

This paper is organized as follows: Section II introduces the déjà-vu object technique. In this section we focus on unit test and omit integration test. This is justified considering that integration test is often not regarded as a separate testing step and, when incremental unit testing is used, it is an implicit part of the unit test. In Section III, we present two real scenarios in which déjà-vu objects have been adopted and report about gotten advantages. Since déjà-vu are related to stubs, servers stubs, and mock objects, most important difference between déjà-vu object and these unit test techniques are presented in Section IV. At last, final remarks and future works are reported in Section V.

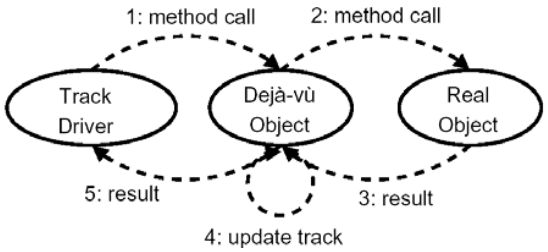

Fig. 3 A track driver calling a real object method on a simple déjà-vu during the recording phase

II. Deja-Vu Objects

As already stated in Section I, a *déjà-vu object* is an object able to observe and record the behaviour of a real object during real sessions, and to replace it during tests, returning previously recorded results.

A déjà-vu can be *simple* or *active*. A simple déjà-vu is an object used to replace a real object which methods are invoked by the class under testing. On other hand, an active déjà-vu is an object used to replace real object representing a source of events listened by the class under testing by means of some callback mechanism in accordance with observer pattern [5].

Both simple and active déjà-vu have the same life cycle consisting of two main phases: the *recording phase* and the *simulation phase*. In the following subsections these phases are detailed described and some examples are reported. In

these examples, the API of *Déjà-vu Creator* is used. Déjà-vu Creator is a prototype of an open source Java framework we developed in order to verify the feasibility of the déjà-vu adoption in several programming contexts.

A. The Recording Phase

The recording phase occurs when a déjà-vu observes the behaviour of a real object and stores all information needed to replicate this behaviour during the simulation phase. The recording phase produces one or more *tracks*, each one representing a different usage scenario for the real object. More precisely, a track is defined as a sequence of records, each one containing information about a method call invoked on the real object (i.e. method name, actual parameters, result, timestamp). All tracks are detailed described into automatically generated documentation associated to each déjà-vu object: this documentation is useful both during the recording and the simulation phase, to check the track correctness and to support software testers in the choice ofwhich tracks to use, respectively.

In order to generate a track, a *track driver* has to be defined and executed. Fig. 3 shows a track driver in action: a track

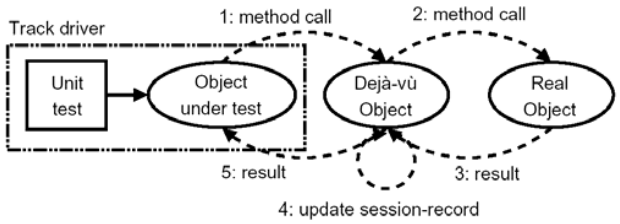

Fig. 4 When the recording phase is performed during a unit test, the role of track driver is played by the unit test class and the object under test

driver calls a real object method on a simple déjà-vu. This last one plays the role of proxy [5] between the track driver and the real object. In such way the déjà-vu is able to record all information about messages they exchange.

From the track driver developer's perspective, the recording phase consists of the following steps:

i) *Obtain an instance of a déjà-vu object.* A déjà-vu reference is obtained passing an instance of the class to be simulated. If a déjà-vu object has been already associated to the specified class, new tracks will be added to the already existing ones. In other words, it is possible to perform the recording phase every time a new usage scenario has to be simulated.

ii) *Add meta-information about the d ´ej`a-vu object (optional).* Since every déjà-vu has to be detailed described, some meta-information should be specified, such as author, date of creation, version number, etc.

iii) *Start the track definition.* To start the definition of a track the method record has to be invoked on the déjà-vu, specifying the name to be assigned to the track.

iv) *Add meta information about the track (optional)*. Also every tracks should be detailed documented. Some meta-information should be associated are: name of the author, date of creation, a brief textual description, etc.

v) *Record the track.* A track is defined invoking methods on the déjà-vu as it be the real object.

vi) *Stop the track definition.* To stop the definition of a track the method stop has to be invoked on the déjà-vu.

As an example, a piece of code of a track driver is reported:

```
// create and set (if needed) real object
RealObject realObj = new RealObject(...);

// create dejavu object
DejavuRealObject dejavu =
(DejavuRealObject)DejavuFactory.getSimpleDejavu(realObj);

dejavu.setDescription("...");

// define a session
dejavu.record("Track01");
dejavu.setTrackDescription("A track example");

dejavu.aMethodOfRealObject("aString");
dejavu.anotherMethodOfRealObject(newInteger(261204));
dejavu.aMethodOfRealObject("anotherString");

dejavu.stop();
```

In this example a déjà-vu object is associated to the `RealObject` class and a track, labelled *Track01*, is added. *Track01* consists of the following sequence of records containing something like:

Method call	Actual parameters	Result
`AMethodOfRealObject`	`"aString"`	`void`
`AnotherMethodOfRealObject`	`new Integer(10)`	`100`
`AMethodOfRealObject`	`"anotherString"`	`void`

where, the invoked methods have the following signatures:

```
void aMethodOfRealObject (String s);
int anotherMethodOfRealObject(Integer i)
            throws ExampleException;
```

The method `getSimpleDejavu` analyses the `RealObject` instance, creates and returns an instance of the class `DejavuRealObject`. This class has to be previously generated by Déjà-vu Creator. That instance is used by track driver to perform the recording phase.

A track can be also defined to terminate throwing an exception, invoking the `generateException` method provided by the déjà-vu object. For example:

```
dejavu.aMethodOfRealObject("anotherString");
dejavu.generateException(
         new ExampleException(...),
         dejavu.LAST_CALL);
```

forces the track to generate the specified `ExceptionClassName` instance during the simulation of the last method invoked (`aMethodOfRealObject`). Note that the `generateException` method checks if the `aMethodOfRealObject` can really throw an `ExceptionClassName` instance.

It is worth to note the recording phase of a simple déjà-vu can be also performed during a unit test. In fact, the role of track driver can be played by the pair: unit test class and class

under test. If this is the case, operations concerning the track definition (step *v*) are contained in the object under test and, consequently, the readability of the unit test class source is further increased.

The recording phase of an active déjà-vu is slightly different. In this case track driver has only to create the déjà-vu and to invoke the record and, after a time, the stop methods. If needed, a track driver can also subscribe itself as listener of events generated by the real object as shown in Fig. 5.

B. The Simulation Phase

The simulation phase occurs when a déjà-vu plays the role of the faker depicted in Fig. 2.

In order to perform the simulation phase a software tester has to create the déjà-vu associated to the real class to be simulated. Then, the software tester has to create a track consistent with the unit test to be performed. At last, he/she can implement a unit test in which the déjà-vu is loaded, configured, used instead of the real object and verified. More in details, from the software tester's perspective, the simulation phase consists of the following steps:

i) *Obtain an instance of a déjà-vu.* A déjà-vu object can be obtained specifying the name of the class to be simulated.
ii) *Load the track to be used.* To load the a track the load method has to be invoked on the déjà-vu, specifying the name of the track.
iii) *Configure the track (optional).* By default, the class under test has to exactly replicate the sequence of call methods defined during the recording phase (i.e. same order of invocation methods with same parameters). In order to increase re-usability of tracks, this default behaviour can be modified. For example, it should be convenient to define default results for each method among those have been recorded, or how many times to return a certain result and so on. An active déjà-vu can be configured to respect or not the recorded timing.
iv) *Replace the real object with the déjà-vu object.* Strategies aimed to replace a real object with a faker are out of the scope of this paper. Nevertheless, several strategies have already available in literature. Refer to mock object Web site [7] for more details.
v) *Run the unit test suite.*
vi) *Verify the déjà-vu consistency.* Verify that the track has been used according to rules defined in step 3. To verify the déjà-vu consistency the verify method has to be invoked on the déjà-vu.

For example, the following piece of code – extracted froma JUnit [9] test case – uses the previously recorded *Track01*.

```
// get dejavu reference
DejavuRealObject dejavu =
     (DejavuRealObject) DejavuFactory.get("RealObject");

// load a session
dejavu.loadSession("Track01");
```

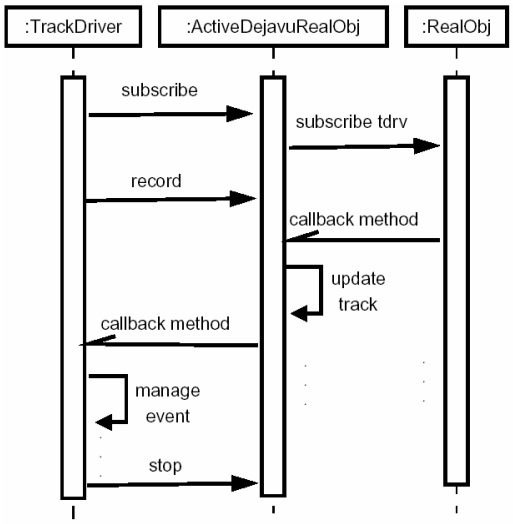

Fig. 5 Recording phase of an active déjà-vu

```
// use dejavu instead real object
TestedClass testedObject =
     new TestedClass((RealObject) dejavu);

// perform JUnit test.
assert...(..., testedObject.methodToTest(...));

// verify deja-vu consistency
dejavu.verify();
```

In this example, the déjà-vu associated to `RealObject` is used by the `testedObject` when the `methodToTest` is invoked. Since the default behaviour of *Track01* has not been modified, all the domain code behind the `methodToTest` has to exactly replicate the session defined during the recording phase. For example, the following sequence of instructions does not generate exceptions:

```
...
realObject.aMethodOfRealObject("aString");
...
Integer integer = new Integer(261204);
realObject.anotherMethodOfRealObject(integer);
...
String string = "anotherString";
realObject.aMethodOfRealObject(string);
...
```

where the `realObject` refers to the déjà-vu object. On the contrary, if you change, for example, the order of the methods call sequence, or call another method of `RealObject`, a `TrackException` will be thrown. Finally, the `verify`() method check if all methods defined in *Track01* have been called during the test execution.

III. Usage Scenarios

Déjà-vu object technique is been already usefully used in two different contexts.

The first context concerns services for information sharing in distributed environments. More precisely, the aim of a workpackage of the "FIRB Wide-scalE, Broadband, MIddleware for Network Distributed Services" (WebMINDS) project [12] was the development of a PhD-level e-learning

and course-ware system, called WebMINDS E-learning System (WES). The main functionality of WES is to allow semantic searches and retrieval of SCORM [13] compliant learning objects stored by several heterogeneous and distributed repositories. From the architectural point of view, WES is composed by four subsystems: *a Web portal*, the user front-end; *a semantic search engine*; a set of *repositories* containing leaning objects; and a *communication infrastructure*, used by the other subsystems to exchange messages. The last subsystem is induced by a non functional requirement depicting a scenario in which Web portal, semantic search engine and repositories run on different machines connected by the Internet. During the testing activity of WES, several déjà-vu objects have been created and used. For example, the communication infrastructure subsystem offers to Web portal developers a Java class exposing the following public interface:

```
public class SemanticSearchEngine {
   public LOQueryResult[] executeQuery(QueryMetadata q);
   public DownloadedLO getLO(LObject lo);
}
```

The `SemanticSearchEngine` class plays the role of façade [5], hiding to Web portal developers the presence of some Web services allowing query executions and learning object downloads. A `DejavuSemanticSearchEngine` class was created and several tracks have been recorded, one for each use case scenario. This déjà-vu has been released to Web portal developers, which used it during testing activity of their subsystem. We notice the adoption of déjà-vu objects involved several benefits both for communication infrastructure and Web portal developers. From the communication infrastructure developers point of view, each recorded track represented the ”proof” of a well-implemented use case. Furthermore, the cost of recording phase was very low, because all tracks have been recorded during integration tests involving the communication infrastructure, the semantic search engine and the repositories subsystems. On other hands, Web portal developers can immediately start unit and integration tests without configure any faker object.

The second context in which we used déjà-vu technique concerns *a telephone communication accounting system*. The main functionalities of an accounting system is gather, store and report data related to the inbound and outbound phone communication traffic related to an organisation. These functionalities are aimed to bill the calls cost to the correct organizational units and optimise carrier contracts in regard to the effective use of the service. From a technical point of view the telephone communication traffic is managed by a device called PABX [14]. Late models of PABX device send data about phone calls on the network in a standard application protocol called ECMA-CSTA [15]. For each call the PABX sends data such as caller id, called id, begin call time stamp, end call time stamp, carrier and so on. The accounting system was implemented as a layered Web-based application, containing at the bottom layer both database and network communication services. In this scenario an active déjà-vu object was used to simulate an observable object representing a PABX Java adapter [5]. This object receives packets sent by PABX and triggers a call-back method to parse their payloads and to store data in a database. Thanks to the adoption of the déjà-vu object technique it was possible to implement unit tests using a PABX faker able to simulate an asynchronous data source. In fact, at first we recorded several tracks in production environment, each one representing a different traffic condition. Then, we used them to replicate PABX activities in testing environment. It is worth to note, the development of a PABX simulation model and its implementation would have been burdensome. Infact, it is not easy create a PABX simulator because its behaviour may be very complex under certain traffic condition.

IV. Related Works

Since déjà-vu object are related to stubs, server stubs and mock objects, it is seasonable to compare our proposal with these already existing unit testing techniques. On account of exposition completeness, we also briefly summarise the main idea underling these ones.

A. Stubs

A *stub metho*d, in short *stub*, is a piece of code used to stand in for some other programming functionality. Each stub method contains code able to return an object or a value useful to run correctly tests.

The following class is an example of a stub implementation of an hypothetical `Thermometer` class:

```
class Thermometer {
   Device thermometerDevice;
   ...
   double getTemperature()throws ThermometerException {
      if (debug) {
         return debugTemp;
      } else {
         thermometerDevice.getCurrentTemp();
      }
   }
}
```

The main drawbacks affecting the stub method technique are:

- the testing code is mixed with the domain class code: this is a not good practise.
- When the behaviour to be simulated is not trivial, their code can become more and more complex and, consequently, error prone and hard to maintain (e.g. the above implementation of `Thermometer` it is simple, but it does not simulate any exception condition);
- during tests, real object method calls are not executed (e.g. `getCurrentTemp`() method). As a consequence, possible error due to a wrong usage of real objects interface is not detected by tests.

With respect to stubs, déjà-vu objects overcome the above limitations. In fact, the adoption of déjà-vu object implies:

- the code of the class under test does not contains any line of code belonging to the testing domain;

- the complexity of real object behaviour does not have impacts at coding level and bring out only a major number of tracks to record;
- the real object methods are really invoked.

B. Server Stub

A *server stub* [2] is a simulation of an object or component. It should exactly replace a component in a system for test or prototyping purposes, but remain lightweight.

Main concerns about using server stubs are: that stubs can be to hard to write, that the cost of developing and maintaining stubs can be too high and that switching between stub and production code can be risky.

With respect to server stubs, déjà-vu objects are automatically generated as result of a recording activity. This recording activity is less expensive of developing a server stub. Furthermore, déjà-vu objects do not present maintenance problems. In fact if the real object modify its behaviour all tracks can be simply updated repeating the execution of the associated tracks drivers.

At last, if déjà-vu and real objects are synchronised, the risk of problems during the switch between déjà-vu and production code is negligible.

C. Mock Objects

A *mock* [3] is a faker object that mimics the behaviour of a real object in controlled way.

The first important difference between mock and déjà-vu concerns their applicability context. Mock object was prevalently designed to be bundled in extreme programming [6] approaches, where the test is used to design the software architecture and mocks play the role of not yet implemented classes [10]. In contrast déjà-vu object technique is not applicable when a top-down development approach is adopted, because in this case the real object does not yet exist or may change behaviour.

On other hand, when real objects already exist, the training phase of mock objects could be a time consuming activity. In these cases the adoption of déjà-vu object technique reduces the time spent for setting expectations and, at the same time, allows to use *real results* during tests.

Another important difference between mock object and déjà-vu arises from the comparison of their life cycles. In general, mock objects accomplish the following pattern [8] for their usage in unit tests:

i) *Create an instance of mock object*.
ii) *Set state in the mock object*: set any parameters or attribute that could be used by the object to test).
iii) *Set expectations in the mock object*: setting expectations in mock object is where the desired or expected outcome is set. This includes the number of method call and returned value of mock object invocations.
iv) *Set invoke domain code with mock object*: when all of the expectations have been set, use the mock object within the domain code.
v) *Verify consistency in the mock object*: A common practice within mock object testing is to implement a "verify" method, which is called as a final step in the test to verify that the expected outcomes match the actual.

With respect to the mock object life cycle, déjà-vu technique splits the training and the usage phases. This has two positive consequences: at first the unit test code is more readable. Then these two activities can be accomplished in different time, even by different developers.

Finally, at best of our knowledge, mock objects can not simply replace event sources.

V. Conclusion and Further Work

In this paper the déjà-vu object unit test technique has been presented and compared with the already existing ones. Furthermore, we presented two usage scenarios in which déjà-vu technique has been successfully adopted.

Our current research activity is focusing on an interesting evolution of déjà-vu object technique, called *déjà-vu session*. A déjà-vu session is an object able to observe and record the behaviour of *several* real objects during a real session, and to replace *all* them during tests, returning previously recorded results. This new kind of objects inherits all advantages deriving by déjà-vu objects usage. In addition, they further decrease time needed to develop unit tests using several faker objects and allow the definition and verification of sessions involving several real objects.

References

[1] G.J.Myers. *The Art of Software Testing (second edition)*, John Wiley & Sons, Inc. 2004.
[2] R.Binder. *Testing object-oriented systems: models, patterns, and tools*, Reading Mass.: Addison-Wesley, 1999.
[3] T.Mackinnon, S.Freeman, P.Craig. "Endo-Testing: Unit Testing with Mock Objects", in *Proceedings of the International Conference on eXtreme Programming and Flexible Process in Software Engineering (XP2000)*, 2000.
[4] K.Beck. *Test-Driven Develpment by example*, Addison-Wesley, 2003.
[5] E.Gamma, R.Helm, R.Johnson, J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
[6] K.Beck. *Extreme programming explained: embrace change*, Addison-Wesley, 1999.
[7] Mock object Web site, http://www.mockobjects.com
[8] M.A.Brown, E.Tapolcsanyi. "Mock object patters", in *Proceedings of The 10th Conference on Pattern Languages of Programs (PLOP03)*, 2003.
[9] V.Massol JUnit in action, Manning, 2004.
[10] S.Freeman, N.Pryce, T.Mackinnon, J.Walnes "Mock roles, not objects", in *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*.
[11] The Free Dictionary, Farlex, Inc., 2006 http://computingdictionary.thefreedictionary.com
[12] The WebMINDS Project Web site, http://web-minds.consorzio-cini.it/
[13] Advanced Distributed Learning Web site, http://www.adlnet.gov/
[14] Wikipedia – The free encyclopedia 2006, PABX definition, http://en.wikipedia.org/wiki/PABX
[15] Ecma International Web site, http://www.ecma-international.org/