# The Layered Information System Test Pattern

Roberta Coelho  Uirá Kulesza  Arndt von Staa  Carlos Lucena

Software Engineering Laboratory
Computer Science Department
Pontifical Catholic University of Rio de Janeiro - PUC-Rio - Brazil
{roberta, uira, arndt, lucena}@inf.puc-rio.br

## Abstract

The object oriented application layer architecture [11, 4] allows the distribution of classes into well defined layers, according to different purposes (business, communication, data access, etc.). Elements from different layers communicate only through interfaces.

While this architecture helps to address requirements of many applications, it also creates many new challenges to software testing [2]. Developers must look around for some techniques that help isolate bugs more quickly in this architecture.

Test pattern is a technique that can improve the efficiency of the testing process, since, it provides a means to share test construction experience. While design patterns describe interactions between classes and determine the specification of the classes that participate in the solution of a specific design problem, a test pattern defines a configuration of objects needed to test the interactions between classes. Both are intended to guide the construction of a piece of software.

The Layered Information System Test Pattern documents a systematic way of testing a layered information system which is based on exercising only the interface defined by each layer.

**Keywords** Test Pattern, Layer Architecture.

**Example** This section presents an illustrative example of an information system that supports the management of bank accounts. Figure 1 presents the object-oriented architecture of this information system following the Layer architectural pattern [11, 4]. According to this pattern, the elements from each layer should communicate only through well defined layers` interfaces. The purpose of a layer interface is to define the set of available operations - from the perspective of interacting client layers - and to coordinate the layer response to each operation.

Several design patterns have been proposed to refine each layer of this architecture. Some of them are: the Service Layer Pattern[6], the Data Access Object Pattern[8] and the Persistent Data Collections (PDC) [9].

The example, presented in the Figure 1, focuses on the Business and Data layers which are defined according to PDC pattern. Nevertheless, different design patterns [8, 9] could be adopted to refine the information system layers, according to the system requirements and the platform used by the application. The Persistent Data Collections (PDC) design pattern [6] refines each layer by filling them with specific classes and interfaces related to business and data access concerns.

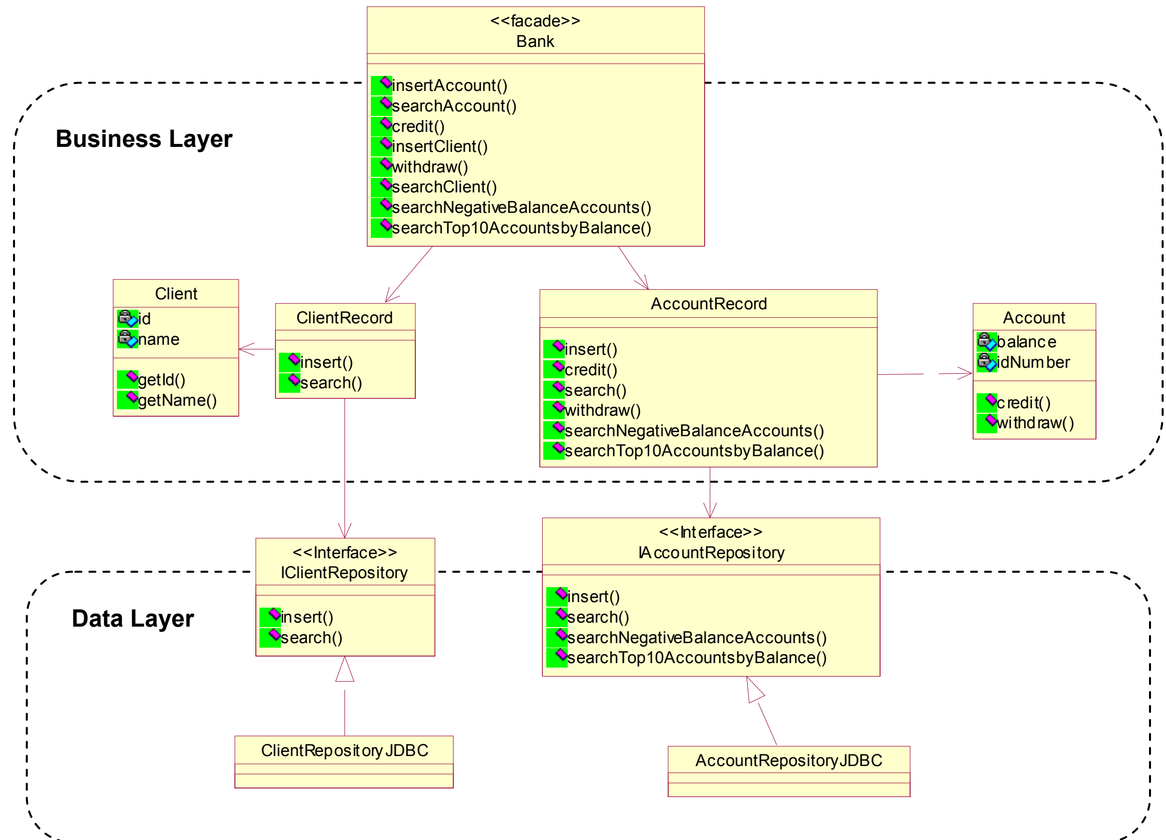


**Figure 1.** Object-Oriented Design for the Bank information system.

Following the guidelines defined in the PDC pattern, the Business layer should provide a Facade [7] to the system functionality, a unique interface for its services. In this example the Facade role is played by the `Bank` class. The Business layer also specifies a set of business collection classes (`ClientRecord`, `AccountRecord`) which defines business rules related to each entity classes (`Client`, `Account`). The business collection classes are also responsible for accessing the services of the Data layer in order to execute persistence operations, such as, insertions, searches, updates, deletions.

The Data layer interface can be structured in one or more classes. In Figure 1, the Data layer interface is structured in two modules one to each main business entity defined in the business layer (`IClientRepository`, `IAccountRepository`). These interfaces are implemented according to a specific persistence platform, in Figure 1, `ClientRepositoryJDBC` and

`AccountRepositoryJDBC` classes implement data access operations related to a specific Entity class using the Java Database Connection (JDBC) API.

Due to the lack of well defined test patterns, developers and test engineers have applied adhoc or not well defined test strategies during system testing. Some examples of common test strategies which have been adopted during system testing are the following:

(i) execution of adhoc manual tests in the user interface layer;
(ii) specification of unit tests to some classes of the system which are chosen using no systematic strategy;
(iii) implementation of one test class to every class of the system (Test Driven Development – Extreme Programming practice [1]).

Although, these test strategies can eventually help system debugging, there are many disadvantages associated to such strategies, such as:

(i) the difficulty of finding the exact localization of the fault that causes a system error; sometimes, during manual tests complex sequence of actions are performed, which can not be repeated.
(ii) high cost and effort necessary to reexecute manual tests;
(iii) a great amount of resources and effort can be wasted due to the codification of many unit tests that will not be effective during system testing;
(iv) since the classes to be tested are chosen without good selection criteria, important system functionalities may be forgotten during testing.

There are many different kinds of test: unit tests, integration tests, performance tests, stress tests, and so forth. This pattern will focus on functional unit and integration tests.

**Context**

Many information systems developed nowadays, define their architecture based on the Layer architectural pattern [11, 4]. This architectural pattern allows the distribution of classes into well defined layers according to different concerns, such as user interface, communication, business and data access. Also, several design patterns [8, 9, 6] have been proposed to refine each layer of this architecture.
Despite those patterns have been widely used, the model for testing systems structured according to this architectural pattern has received few attention and has been few explored. Most tests are limited to test suites and test cases using simple strategies [1,3]. Although those tests are useful, they fall short in the role of a general organizational for automated testing. What is required is a higher level of abstraction, a test pattern that can be reused wherever a layered architecture is adopted.

**Problem**

The development of an information system typically addresses different concerns, such as, user interface, distribution, business and data access. The lack of well defined strategies to test an information system can bring several problems to system quality and additional costs to the software development.

A recurring problem in the context of layered information systems is how to

define automatic functional tests in order to verify system services. The following forces emerges from this problem:

• *Separation of Concerns*: Developers should focus on each specific layer when testing the system. Besides, they should be able to test each layer independent from the others.

• *Test Class Modularity*: Each test class should verify a well defined set of functionalities provided by one specific layer.

• *Test Robustness:* The test classes should be resilient to internal changes in the implementation of the layer classes.

• *Cost Reduction:* The solution should reduce the cost associated to test activities without decreasing test quality.

• *Proximity between Fail and Fault*: Automatic tests should make it easier to come across system failures as well as to localize the faults that had caused them.

**Solution**

Create unit tests to exercise only the interface defined by each layer. Each test class focuses on the test of specific concerns/features implemented by a layer. Furthermore, the test code responsible for verifying all the services provided by a layer can be modularized in one or more test classes.

To allow the test of one layer at a time, this pattern adopts auxiliary classes, called mock objects [14]. A Mock Object is used by a test to replace a real component (or a set of components) on which the system under test depends. Typically, mock objects fakes the implementation either by returning hard coded results or results that were pre-loaded by the test [14].

Since the tests defined by the Layered Information System Test Pattern exercises only the interface of each layer, and there is not a one-to-one relationship between the classes that comprises the interface and the test classes, this pattern can be used to test any layered information system no matter the design pattern or design strategy used to refine the layers.

**Structure**

Figure 2 illustrates the structure of the Layered Information System Test Pattern. It has three participants:

- **BusinessTest:** this class contains all methods that test a set of functionalities provided by the Business Layer Interface and are related according to one specific criterion. This criterion can be a set of operations related to a business entity or to a business service.

- **BusinessRepositoryTest:** implements test methods to all methods provided by a Repository interface. The implementation of these test classes focus on the testing of specific data repository functionality related to insertion, searching, update and database operations. Each test method implements a test case which verifies a successful or an error condition from a specific repository method.

- **MockRepository:** this class fakes the implementation of a specific BusinessRepository. Thus, this auxiliary class enables the unit test of the

business layer.

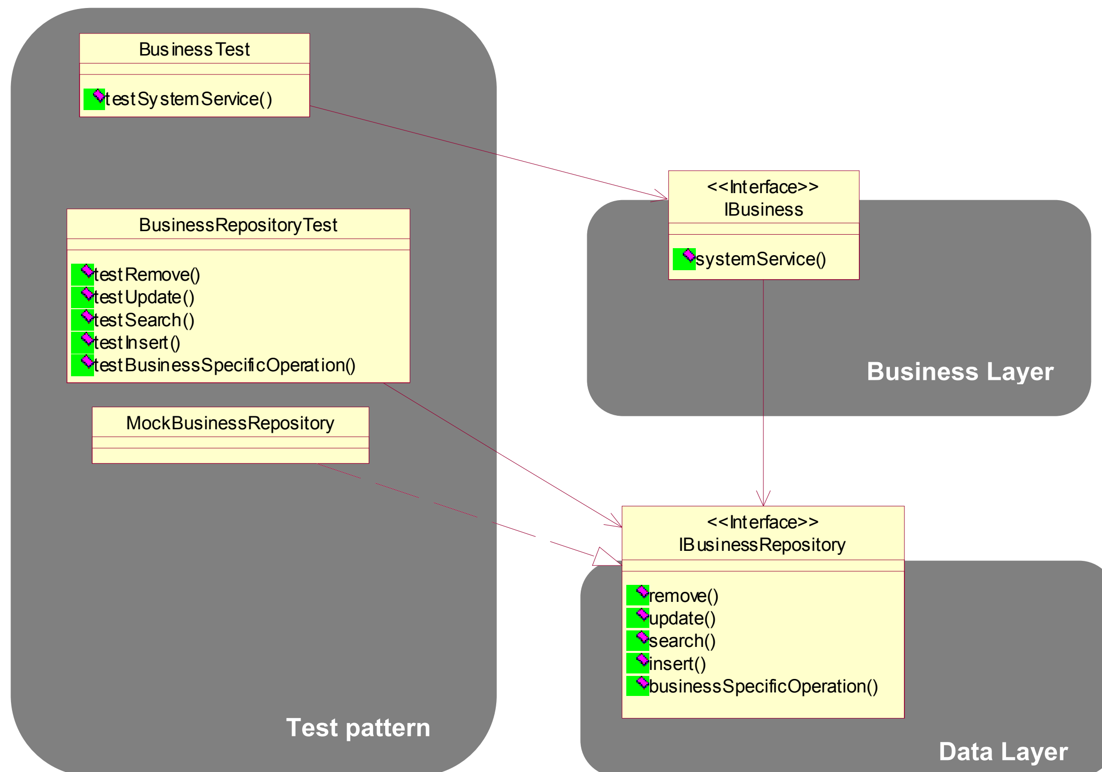


**Figure 2**. The Static View of the Layered Information System Test Pattern.

All Business Layer`s operations can be structured in one single interface or a set of interfaces [7]. The purpose of the BusinessTest classes is to modularize the Business Layer tests according to each business entity manipulated by its operations or according to each business services implemented by such operations. For example, there can be one BusinessTest class to exercise the set of operations related to a business entity or a business service.

The BusinessTest classes contain a test method to each successful and error condition of each method from the Business Layer. Most of the time developers focus on testing successful conditions and forget the error ones, which are as important as the former. If we define only one class to test all successful and error conditions of Business Layer methods, the resulting test class will probably contain too many lines of code which can impact on test maintainability.

The Data Layer will also be tested through a set of classes which exercises its interface. Each Data Layer test class concerns with one specific business repository accessible through the Data Layer`s interface. The BusinessRepositoryTest classes, illustrated in Figure 2, are the ones responsible for testing the each business repository.

Since each layer delegates services to the lower layer the only way to test Business Layer without the passing through Data Layer is to delegate data services to the MockBusinessRepository class, which fakes the implementation of a real BusinessRepository class either by returning hard

coded results or results that were pre-loaded by the test.

The MockBusinessRepository classes allow the BusinessTest classes to concentrate on testing Business Layer own code. Therefore, the integration test of those two layers is performed when Business Layer delegates services to the real repositories instead to the mock classes.

**Dynamics** This pattern allows the execution of three types of tests: Business Layer unit test, Data Layer unit test, and integration test of Data and Business Layers.

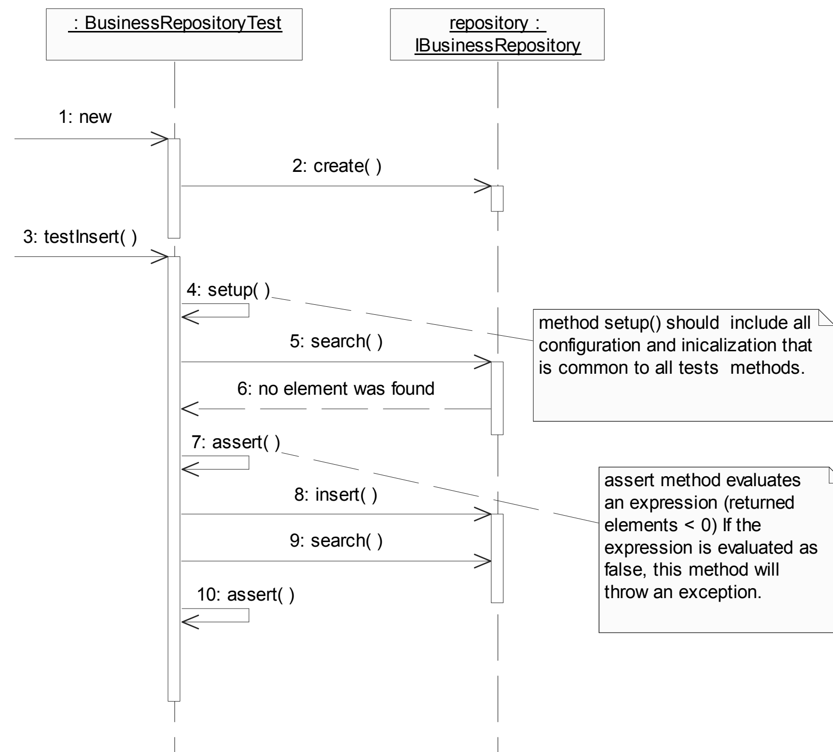


**Figure 3.** Dynamic View of the Data Layer unit test.

Figure 3 illustrates the sequence of method calls performed during Data Layer unit test. Firstly, an instance of BusinessRepository class is created during the initialization of `BusinessRepositoryTest` class (steps 1 and 2), Secondly, a test method is called, for example, `testInsert()` (step 3), then, `setup()` method is called – a private method responsible for any configuration and initialization common to all test methods (step 4). Finally, BusinessRepository methods are called (steps 5, 8 and 9) and assert operations are executed to compare expected results with returned results (steps 7 and 10).

Figure 4 represents an integration test comprising the Business Layer and the Data Layer. It illustrates the sequence of method calls performed when the Business Layer is tested in collaboration with the Data Layer. Firstly, the `BusinessTest` class creates the classes that implement the Business and Data layers. In the Figure 4, this is illustrated through the instantiation of classes that implement the `IBusiness` and `IBusinessRepository` interfaces (steps 2 and 3). After that, different test methods can be executed in order to exercise the functionalities implemented by the Business Layer. Figure 4, for example, shows the execution of the `testSystemService()` method, which calls a business method (step 6) and uses an `assert()` method (step 7) to to compare returned results with expected results.

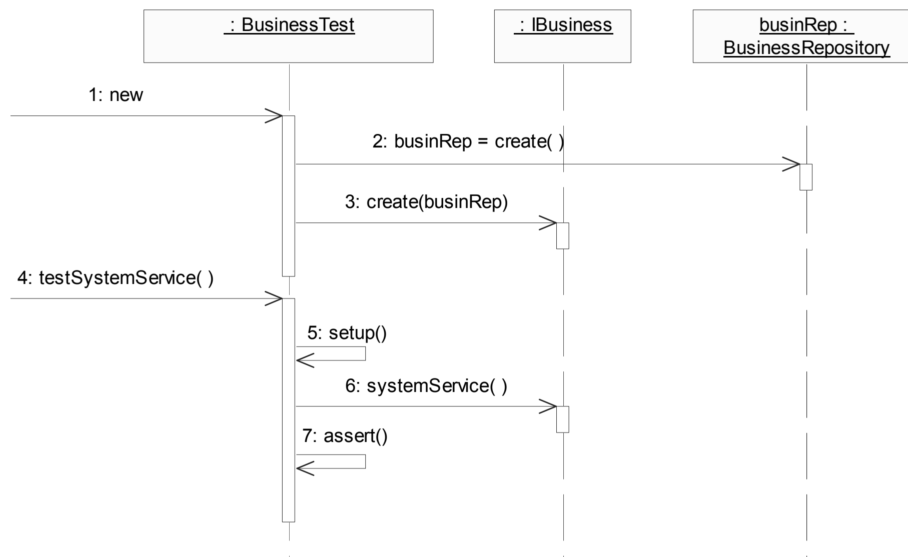


**Figure 4.** Dynamic View of the integration test of the Data Layer and the Business Layer

Figure 5 illustrates the sequence of method calls performed when testing the piece of functionality embedded in the Business Layer. This type of test, as distinct to the integration test described previously, exercises a single layer. Since Business layer depends on the services provided by Data Layer, those services should be amulated by a fake implementation of such layer, a mock object. In the Figure 5, the instantiation of Business Layer is represented by the creation of the `IBusiness` object (step 3). As we can see, this `IBusiness object` is configured with an instance of `MockBusinessRepository` (step 2), which will be used to simulate the Data layer. Finally, the test methods are executed the same way as described in Figure 5.

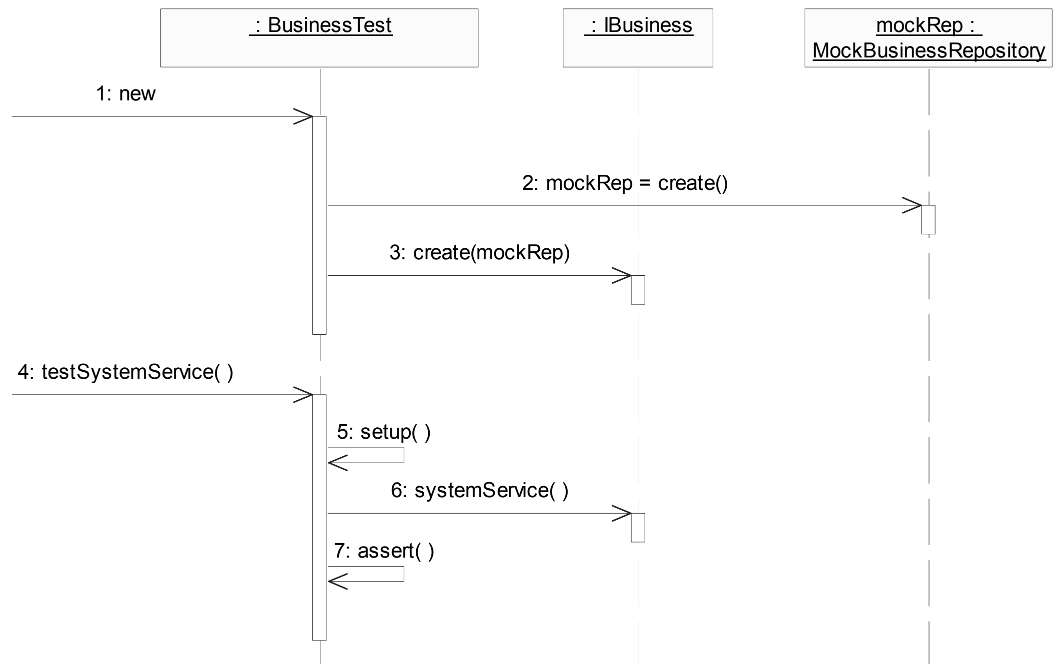


**Figure 5.** Dynamic View of Business Layer unit test.

**Solved Example**

Figure 6 presents the use of the Layered Information System Test Pattern for the bank information system illustrated previously. Two classes, `AccountRepositoryTest` and `ClientRepositoryTest`, are specified to enable the testing of the data access classes. These classes are implemented based on the method signatures defined in the `IAccountRepository` and `IClienteRepository`, respectively. This allows to reuse them in case the system developers need to provide new data access classes to a different persistency platform.

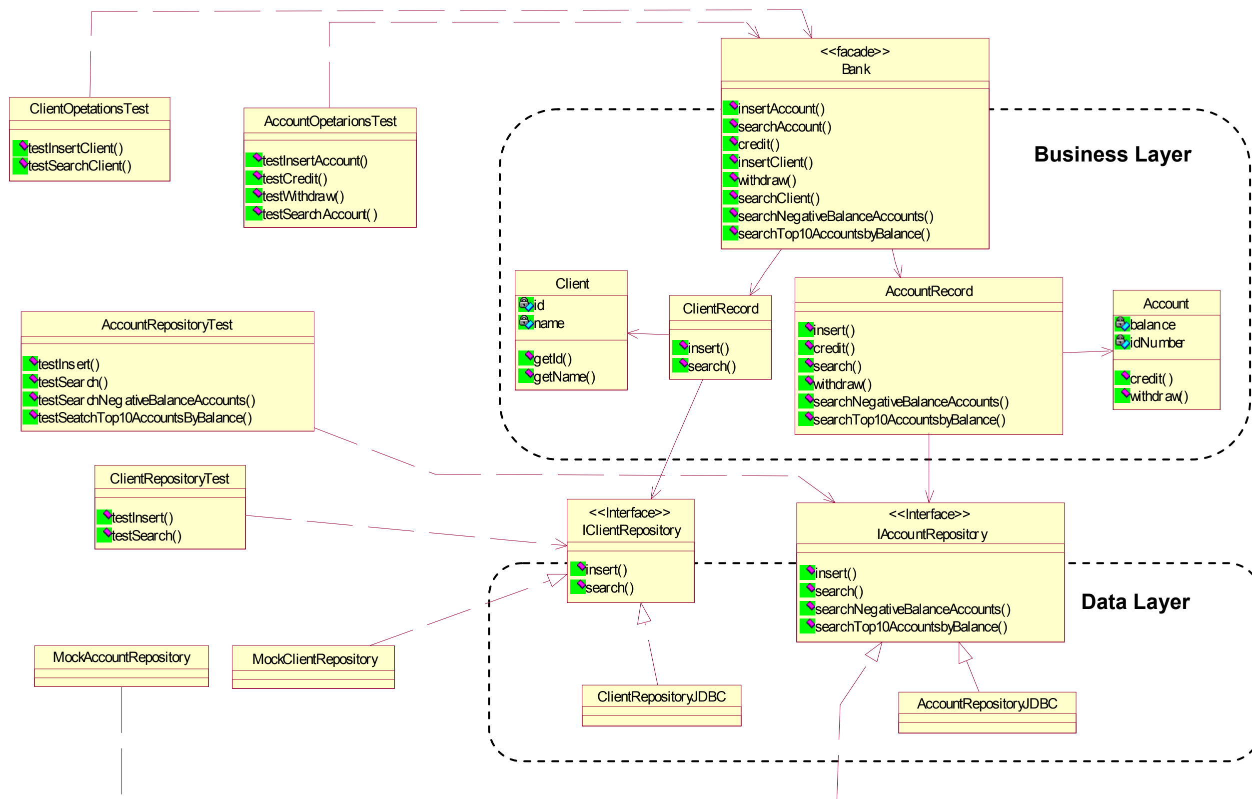


**Figure 6:** An information system and its corresponding test classes.

The test of the Business Layer for the example of the bank information system is supported by the `AccountOperationsTest` and `ClientOperationsTest` classes. Each of these classes implements a set of test methods related to a specific entity class. Also, as we can see in the Figure 6, these classes are codified based only on the business methods provided by the `Bank` facade class. Thus, internal changes in the implementation of these services do not affect the test classes.

Finally, two mock auxiliary classes, `MockAccountRepository` and `ClientAccountRepository`, are presented in the Figure 6. They represent alternative implementations of the data access classes. They are used when it is required to test the Business layer functionality individually.

**Consequences** The Layered Information System (LIS) Test Pattern maintains the following consequences:

- *Separation of Concerns*. The pattern defines an individual test to each

layer of an information system. LIS test pattern focus on the testing of individual services.

- *Test Class Modularity*. The testing code is modularized using different test classes. Each test class focus on the verification of a well defined and limited set of functionalities provided by a specific layer. It improves the readability and maintainability of the test classes.
- *Test Robustness*. Since test classes depend only on the layer interface, they are no effected due to implementation changes inside a layer.
- *Cost Reduction*. Although there is a cost associated to the implementation of the layered information system test pattern, the systematization of the test activity can reduce its cost if compared with other approaches, such as, adhoc tests and unit test of every class. Code generation tools can even reduce test costs since they can generate the overall structure of many test classes.
- *Proximity between Failure and Fault*. LIS test pattern defines individual test to each layer which make it easier to find system errors as well as to localize the faults that caused them.
- *Increase in the number of classes.* A negative consequence of this testing solution is the increase in the number of classes to be maintained. However, this Test Pattern allows the execution of automated tests along the iterations which would require high cost and effort to be reexecuted manually. Although this pattern suggests fewer test classes than Test Driven Development (TDD) agile practice (one unit test per class) it is as effective as TDD. Since the classes to be tested are chosen according to a specific criterion, important system functionalities is not forgotten during testing.

**Known Uses** The Layered Information System Test Pattern has been used during the development of two Java information systems in Recife, Brazil. A general description of these systems is given below.

- A system for managing real estate. This system allows the register of real estate and the management of tax charging related to them. It was implemented in the J2EE platform.
- A system that supports the management of market activities. The system allows the register of market activities and the management of tax charging related. It was also implemented in the J2EE platform, including the use of the Enterprise Java Bean technology.

**See Also** Just a few test patterns have already been proposed. Gerard Meszaros [12, 13] has proposed two Test Pattern languages, one for setting up XUnit test features - which describes key techniques for addressing the issues around test fixture management, and the other for automating testing of indirect inputs and outputs using XUnit.

Some design patterns for using Mock Objects have been proposed as well, some of them are the following:

- Mock Object: a basic mock pattern that allows for testing a unit in isolation by "faking" the communication between collaborating objects.

- Mock Object Factory: a way of creating mock objects using existing factory methods.

- Mock Object via Delegator: a pattern that creates a mock implementation of a collaborating interface in the test class or mock object.

**Implementation** We describe below some guidelines for implementing the Layered Information System (LIS) Test Pattern. The following code examples are related to an information system for managing bank accounts presented in previous sections. They are written using the Java programming language and the JUnit test framework [10]. However, the LIS Test Pattern can implemented in other platforms, since the guidelines presented here are followed.

**Step 1: How to prepare the Entity classes to help the codification of test classes?**

Every test method needs to evaluate the data sent or received from the methods being tested. In the context of information systems, the information manipulated are, typically, the content embedded in entity classes. Thus, before starting the implementation of test classes, it is important to define a way to compare two instances of the same Entity class. A well known way to compare two instances of a class is through the a method `equals()` that receives an instance of the same class and returns true if the argument contains the same attributes values as the class being called or false otherwise.

In the information system for the management of bank accounts, for example, the `Account` class must define its `equals()` method in order to compare its attributes `idNumber` and `balance` with the same attributes of other instance.

```
public class Account {
       private long idNumber;
       private double balance;

       public Account(long idNumber, double balance){
              this.idNumber = idNumber;
              this.balance = balance;
       }

       ...

       public boolean equals(Object anotherInstance){
          Account anotherAccount = (Account) anotherInstance;

          if ( this.idNumber == anotherAccount.idNumber &&
                this.balance == anotherAccount.balance){
            return true;
          }else {
            return false;
          }

       }
}
```

**Step 2: How to define a BusinessRepositoryTest class?**

A `BusinessRepositoryTest` class must define test methods to verify the functionality provided by a data access class (or data repository class) which are specified in the business repositories interfaces.

As mentioned in the Structure Section, a `BusinessRepositoryTest` class has many responsibilities, such as: (i) to create an instance of a data access class to be tested; (ii) to define a method that performs every configuration and initialization necessary to run the test; and (iii) to specify different test methods to each method provided by the data access class to be tested.

Each `BusinessRepositoryTest` class must define different test methods to each existent method of the data access classes. These test methods must verify the successful and error conditions, using different argument types and values and handling different types of exceptions.

In order to minimize effort, the search methods - of the data access classes - can be used to support the test of the other methods. For example, the test method of insert operations can, previously, search the object be inserted to verify if it does not already exist in the repository. Also, the test methods of delete and update operations should use the search method whenever they need.

Below, we present the partial code of a `BusinessRepositoryTest` class in the context of the banking system, responsible to test the functionality of an `IAccountRepository` instance.

```
import junit.framework.TestCase;

public class AccountRepositoryTest extends TestCase {
       private IAccountRepository accountRepository;

       public AccountRepositoryTest(String name){
          this.accountRepository = new AccountRepositoryJDBC();

          // Additional common configurations before to execute all the test
          // methods
          ...

       }

       // JUnit standard method to be executed before every test method
       protected void setUp() {
          ...
       }

       public void testInsertAccount() {
             try {
                Account account = new Account(123, 500);
                accountRepository.inserir(account);

                Account accountSearched = accountRepository.search(123);
                assertEquals(account, accountSearched);

              } catch (Exception e) {
                fail("Exception not expected:" + e);
             }
        }
        public void testInsertAlreadyExistentAccount() {
              try {
                Account account = new Account(123, 500);
                accountRepository.inserir(account);
                fail("System did not throw exception!!!");

                Account accountSearched = accountRepository.search(123);
                assertEquals(account, accountSearched);

              } catch (AlreadyExistsObjectException e) {
                System.out.println("OK: Exception expected!!!");

              } catch (Exception e) {
                fail("Exception not expected:" + e);
              }
        }
       ...
}
```

### Step 3: How to define a MockBusinessRepository class?

The `MockBusinessRepository` classes simulate the behavior of BusinessRepository classes in order to allow the unit test of the Business Layer.

In order to fake the behavior of a real repository the `MockBusinessRepository` classes can use an internal data structure (like a hash table or a vector) that is able to store the business objects. The Mock classes must implement the data access interfaces. Each method described in these interfaces uses the internal data structure.

A partial code of the MockAccountRepository class is presented below. It uses a hash table to store the business objects manipulated by the mock.

```
public class MockAccountRepository implements IAccountRepository {

    private Map accounts;

    public MockAccountRepository(){
       this.accounts = new Hashtable();
    }

   public void insert(Account account)
                    throws AlreadyExistentObjectException, ... {

       if (this.accounts.containsKey(new Long(account.getIdNumber()))){
           throw new AlreadyExistsObjectException ("Object already exists");
       }else {
          this.accounts.put(new Long(account.getIdNumber()), account);
       }
    }

   public Account search(long idNumber) throws InexistentObjectException {
      Account account = null;

      if (this.accounts.containsKey(new Long(idNumber))){
         account = (Account) this.accounts.get(new Long(idNumber));
      }else {
         throw new InexistentObjectException "Object does not exist");
      }

       return account;

     }
     ...
}
```

**Step 4: How to define a BusinessTest class?**

A BusinessTest class verifies the functionality provided by Business Layer. Different BusinessTest classes should be defined for each system.

This class contains all methods related to a business entity or to a business service. In this example each test class must focus on the testing of all Facade operations related to a business entity. Moreover, each test method defined must verify different execution conditions of the method under test, such as: (i) the correct execution of business rules; and (ii) the incorrect execution which throws business exceptions.

In the example presented in *Solved Example* Section two different BusinessTest classes were be specified: one responsible for testing the functionalities related to the Account class and the other responsible for testing the functionalities related to Client class. Below we present the AccountOperationsTest class, responsible for testing the methods in the Bank facade class related to the Account business class. We can also observe that the AccountOperationTest class constructor allows two different configurations depending on the kind of test that will be executed: (i) in case we want to perform integration tests, the Data layer will use the

system data access classes; and (ii) in case we want to perform unit tests in the Business Layer, the Data layer should be replaced by a mock object in the test method. In a more realistic implementation of BusinessRepositoryTest classes, the parameter *integrationTest* should be loaded from a configuration file.

```
import junit.framework.TestCase;

public class AccountOperationsTest extends TestCase {
       private Bank bank;
       private boolean integrationTest = true;

       public AccountOperationTest(String name){
          this.bank = Bank.getInstance();

          AccountRecord accountRecord = null;
          ClientRecord = clientRecord = null;
          if (integrationTest){
             accountRecord = new AccountRecord(
                                       new AccountRepositoryJDBC());
             ...
          } else {
            accountRecord = new AccountRecord(
                                       new MockAccountRepository());
            ...
          }
          this.bank.setAcccountRecord(accountRecord);
          ...
       }

       // JUnit standard method to be executed before every test method
       protected void setUp() {
          ...
       }

       public void testCreditAccount() {
            try {
               Account account = new Account(123, 500);
               bank.insertAccount(account);

               bank.credit(123, 200);

               Account accountSearched = bank.searchAccount(123);
               assertEquals(new Account(123, 700), accountSearched);

             } catch (Exception e) {
               fail("Exception not expected:" + e);
            }
       }
       public void testWithdrawAccount() {
             try {
               Account account = new Account(456, 500);
               bank. insertAccount(account);

               bank.withdraw(456, 200);

               Account accountSearched = bank.searchAccount(456);
               assertEquals(new Account(456, 300), accountSearched);

             } catch (Exception e) {
               fail("Exception not expected:" + e);
             }
       }
       ...
```

}

**Acknowledgments** We would like to give special thanks to Carlo Giovano, our shepherd, for his important comments, helping us to improve our pattern. This work has been partially supported by CNPq under grant No. 150678/2004-7 for Roberta de Souza Coelho and grant No. 140252/2003-7 for Uirá Kulesza. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0 and by the art. 1st of Decree number 3.800, of 04.20.2001.

## References

[1] K. Beck, Extreme Programming Explained, Addison-Wesley, 2000
[2] M. Donat, Debugging in an Asynchronous World, ACM Queue 1(6), 2003, pp. 23-30.
[3] M. Fowler, A UML Testing Framework. Software Development Magazine. April, 1999
[4] S. Ambler. Building Object Applications that Work. Cambridge University Press and Sigs Books, 1998.
[5] S. Ambler. The Object Primer. Cambridge University Press, 2001.
[6] T. Massoni, Vander Alves, Sergio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In First Latin American Conference on Pattern Languages Programming SugarLoafPLoP, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.
[7] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995..
[8] M. Fowler, et al. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.
[9] D. Alur, D. Malks, J. Crupi. Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, 2nd edition, 2003.
[10] JUnit Framework, http://www.junit.org.
[11] F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.
[12] G. Meszaros. A Pattern Language for Setting up XUnit Test Fixtures. Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
[13] G. Meszaros. A Pattern Language for Automated Testing of Indirect Inputs and Outputs using XUnit, Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
[14] M. Brown and E. Tapolcsanyi, Mock Object Patterns, Proceeding of the PLOP 2003, September 2003, Monticello, USA.
[15] C. Szyperski, Component Software: Beyond Object-Oriented Programming. New York, NY: ACM Press and Addison-Wesley, 1998.