

Better unit tests using correct test fixtures

Isto Niemi

Abstract—When the implementation of the unit tests is made easier, developers are likely to write more and better tests. There are lots of instructions on how to write good unit tests but not much talking about test fixtures. In this article I am focus on finding out how to setup external resources for the unit tests. Mock Object can be used to eliminate the external resource. ObjectMother and data upload tools can be used to populate the test object to the external resource. And when talking about databases in-memory one can be used to make the test run shorter and the maintenance easier. All of the solutions have their pros and cons. In this article I list up the reasons when to use and what.

Index Terms— unit tests, fixtures, Mock Objects, ObjectMother, in-memory database

1. INTRODUCTION

Unit testing is white box testing done by the programmer. It can be done manually or automatically and it is focused on one class or one method.

XP utilizes unit tests heavily. Its style is to write test code for every class in the system. Test code is written with same programming language as the production code and put in the revision control. XP has many roles for automatic unit tests other than just test the correctness: they are part of design process; they work as code documentation; they can be used to verify bug corrections and they make the refactoring easier by giving direct impact about the changes.

It is quite an easy task to write a unit test for a single isolated component but when the component is accessing other components or external resources the task is not trivial anymore. Almost all unit test tutorials are written for isolated components and there you can find clean and clear solutions. But when they are talking about non-isolated components the main advice is to do better design and avoid the isolation. The second advice is to setup the testing environment before every test and tier down it afterwards. But there is no clear answer for the question what kind of strategies or patterns you should use.

The setting-up operations are for creating all needed objects and allocating external resources such as database or files. The tier-down operations are for releasing the created object and closing the database connections. All these are called test fixtures.

For instance if you would like to test purchase order functionality, the order should exist in the external resource. Before the actual test you run the setup-operation that creates the order. After that you run the test and check that the Order handled correctly. At the end you remove the Order and return the database to the state in which it was before the test. If some other test has created other Orders and left them to the database, it is possible that your test fails. The failure is not due to missing functionality but wrong expectations – if the test finds out that the customer has two orders but the expected value is one, the test fails.

1.1 Research problem

As it is pointed out test fixtures have important role in unit tests. In this paper we study how the test fixtures should be used when testing a component accessing external resources. A database is used as an example of an external resource.

The research problem is to find out best practices and solutions for the test fixture implementation. The result is a table presenting benefits for the found solutions. The benefits are taken from the found best practices.

1.2 Research method

The best practices and solutions are searched from the literature. Three different type of papers where found relevant for this paper: the papers that list best practices or code smells for extreme programming, the papers that describe patterns to solve fixture related problems and the papers talking about database management in agile development process.

1.3 Scope of the study

The focus is in the java architecture. An important criterion for the solutions is the JUnit support. Every solution listed in this paper should work with JUnit. That makes it possible to combine different solutions without big architectural changes. JUnit is a popular unit test framework for java [Noonan, 2002].

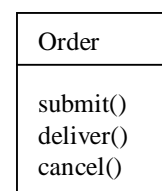


Figure 1. Purchase Order is used as an example of the business object

The solutions are presented in architectural level and actual product comparisons are left out from this study. The product names are mentioned as a pointer for the future interest.

Quite relaxed definition is used for the architecture. In this paper architecture is seen as a set of instructions on how to build up testing framework.

In this paper the goodness of the test case means how easy it is to implement and maintain. The efficiency in defect finding is mainly left out.

The business object called Purchases Order or shortly Order is used in all examples. Order-class has three methods: submit, deliver and cancel (figure 1). Submit-method creates the purchase order to the database; deliver-method set the status of the order to be *delivered* and cancel-method set the status to be *cancelled*.

1.4 Structure of the study

Chapter 2 lists four best practices for the test fixtures. These are inline resources, setup external resources, make the resource unique and reduce data.

Chapter 3 presents four architectural solutions that can be used with JUnit. The listed ones are Mock Object, ObjectMother, in-memory database and uploading tools.

Chapter 5 contains some discussion about the solutions.

Chapter 6 concludes the paper.

2. BEST PRACTICES

2.1 Inline Resource

Using external resources introduces hidden dependencies: if some force changes or deletes such a resource, tests start failing. When the same resource is used by multiple tests the possibility for failing even increases.

To remove the dependency between a test method and an external resource, the resource should be included in the test code. This is done by setting up a fixture holding the same contents as the resource. This fixture is then used instead of the resource when running the test. [Deursen, 2001]

TABLE 1. ARCHITECTURAL NEEDS FOR INLINE RESOURCES

Classes should be as isolated as possible. Simple classes are easier to test.
Separate construction from the use. Configuration manager can help to switch between test and production implementations.
Remove complex static initializers. For instance pass JDBC connection as parameter instead.
Weaken dependencies between layers. User interface and persistence code should be separated from the business logic.
Replace classes with interfaces in method signatures. Otherwise it is hard to use dummy implementations of the classes.

A simple example of this practice is when putting the content of a file into a string. This is possible only if the testable code is isolated from the file reading. Instead of reading the file content is given by argument when the object is initialized.

As seen in the previous example special kind of design is needed for the utilization of the inline resources. See Table 1 for other architectural needs listed by Freeman et al [Freeman, 2002].

2.2 Setup External Resource

It is not always possible to eliminate the need of external resources such as directories, databases, or files when performing unit test. External resource might be so complex that it is too time consuming to implement the inline replication or the used framework might be designed so that external resources cannot be left out.

If the test has to use an external resource it should be explicitly created or allocated before testing and released afterwards. Otherwise any changes in other tests can ruin this one [Deursen, 2001].

The creation operation for an external resource is not trivial. Many times in setting up proper test cases you need a business objects at a specific point in their lifetime. This means that the external resource should be in a specific state also. [Elssamadisy, 2001]

If you are testing purchase orders and want to test the object delivery, you first need an active order with correct customer information. You have two possibilities, you can create the order and bring it up to the needed state or you might have a snapshot of the external resource that contains the order and all other needed components.

2.3 Make Resource Unique

A lot of problems originate from the use of overlapping resource names, either between different test runs done by the same user or between simultaneous test runs done by different users.

Problem occurs if a test allocates or locks a resource needed also by another test. Solution is to use unique resource identifiers so that every test has its own resources. For instance for shared file repository the timestamp and developer name can be added to the file names [Deursen, 2001].

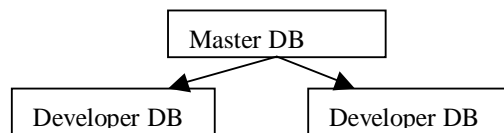


Figure 2. The master database is copied to the developer ones.

A common problematic situation is shared database instance [Hassan, 2000]. When two or more developers are running unit tests simultaneously and all tests use the same database, it is possible that one test instance corrupts the others. One

test is setting up the database correctly but before the actual testing, other test is setting its data generating some sort of conflict situation.

One practice is to have one database instance per developer and one master database from where the other ones are copied [Schuh, 2001]. See figure 2.

2.4 Reduce Data.

Minimize the data that is setup in fixtures to the bare essentials. This will have two advantages: it makes them more suitable as documentation, and your tests will be less sensitive to changes [Deursen, 2001].

Also the data in method signature should be minimized. If the method to be tested gets the purchase order as an input but only calculates the age of the order, the signature can be refactored to contain only the creation date. This kind of minimization can be used to avoid the need of inline and external resources [Burke, 2003]. In our simple example only the creation date is needed and the order loading from the database can be left out.

The problem of unminimized data can lead to *general fixture* [Deursen, 2001] that means that the fixture set-up is too general and different tests only access part of the fixture. It is also possible that if one test initializes very much data, it is testing too much and should be divided.

3. SOLUTIONS

3.1 Mock Objects

Mock Object is a dummy or stub implementation that replaces the real code. These Mock Objects are passed to the target domain code which they test from inside. There are two different definition of the term Mock Object [Burke, 2003]:

- The generic definition states that a Mock Object is any dummy object that stands in for a real object that is not accessible or is difficult to use in a test case.
- A more specific interpretation states that a Mock Object must have the ability to set up expectations and provide a self-validation mechanism.

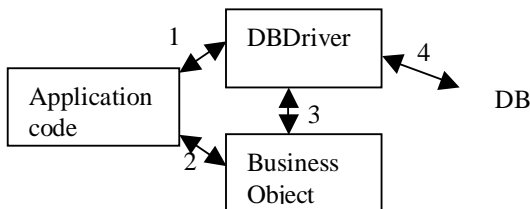


Figure 3. Application uses a business object to access database

Mock Object is used to eliminate external resource because it provides a simple way to set up fake testing data. A complex system state can be hard to set up. It is easier to replace it with a simple Mock Object. Mock Objects also offer a nice way to test error situations that are hard to test using other methodologies [Mackinnon, 2000].

Mock Object should be written as minimal as possible. Only the behavior needed for the test should be implemented. If Mock Object calls another Mock Object, it is too complex [Mackinnon, 2000].

Figure 3 and 4 illustrate how the database driver can be replaced with mock implementation. Figure 3 shows how the application uses business object to access database. First (1) the application code opens the database connection, and then (2) the database connection is passed to the business object. After that the business object uses the driver (3) and finally the database driver accesses (4) the database and does the wanted query.

It is possible to eliminate the arrow one (1) by letting the business object open the database connection. Later in figure 4 we can see that it is important to keep the creation of the database connection outside the business object. Otherwise it is not possible to replace the database driver with mock implementation.

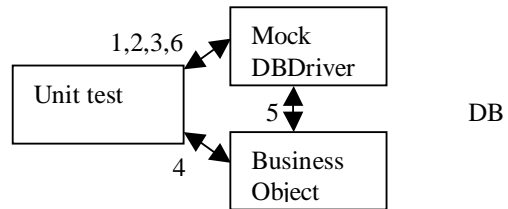


Figure 4. Unit test tests business object

In figure 4 and 5 it is presented how to perform unit tests using the mock implementation of the database driver. Instead of real database connection the unit test creates (1) the mock instance of the connection. After that unit test initializes the Mock Object by telling (2) what should be returned to the business object and what to expect (3) from the business object. After initialization the business object is invoked (4) with Mock Object as a parameter. Business object uses the mock implementation like a real database driver and asks (5) for the result. Instead of database query the Mock Object returns the value specified in the initialization phase (2). Business object returns the result to the test code and the result is validated. Then the test code checks from the Mock Object (6) that it is called with correct parameters. Also the result returned by business object is compared to the value specified in initialization phase (2).

Good place for validation logic is inside the Mock Object. This way, every test that uses the Mock Object will reuse the validation logic automatically.

As seen in previous example Mock Objects need a special kind of software architecture. Classes should be designed so that the handlers are passed to them as parameters rather than coded statically in the domain code. Interface names should be used in method signature instead of class names. This kind of architecture isolates individual classes and makes it

possible to replace handlers with mock implementations [Mackinnon, 2000].

See Appendix A for code examples on how to use Mock Objects.

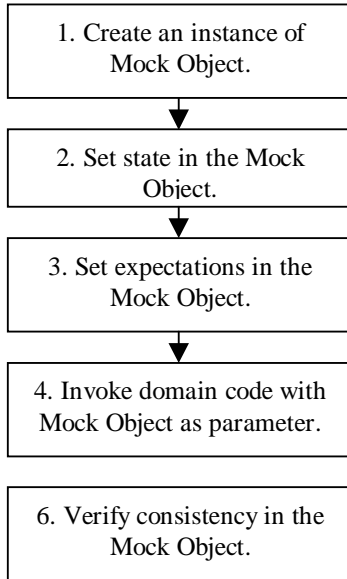


Figure 5. Generalized procedure to use Mock Objects

3.2 *ObjectMother* –pattern

Many times when setting up a test case you need a business object at a specific point in its lifetime. If you are testing the cancellation of the order you need as real and functional order as possible. Otherwise you can miss something when doing the test.

One solution is to chain test cases so that first you test the order creation and after that you try to cancel that order. The drawback is that if the first test fails you never know if the second one is working or not. A better solution is to create the order for every test. But if you implement the creation logic inside the test case you have to update multiple tests when the creation logic is changed.

ObjectMother is a special case of the factory pattern. It returns objects in any valid state whereas factory returns objects usually in initial state [Elssamadisy, 2002]. ObjectMother manages the lifecycle of test objects, including creation, customization, and, when necessary, deletion. It offers a standardized way to setup external resources and helps the maintenance because the test object creation is implemented in separated classes.

ObjectMother is an object or set of objects that [Schuh, 2001]:

1. Provides a fully-formed business object along with all of its required attribute objects,
2. Returns the requested object at any point in its lifecycle,

3. Facilitates customization of the delivered objects,
4. Allows for updating of the object during the testing process, and
5. When required terminates the object and all its related objects at the end of the test process.

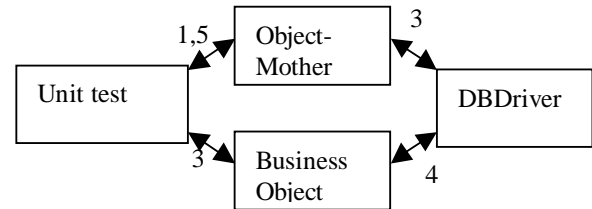


Figure 6. Unit test uses ObjectMother to setup the database

Figure 6 illustrates the usage of ObjectMother. First the unit test calls (1) the ObjectMother to create the needed order as described above. ObjectMother uses (2) the database driver to insert the needed rows to the database. Then the test calls (3) the business object to cancel the order. Business object updates (4) the database setting the state of the order to be cancelled. After that the test checks that the cancellation is done correctly and uses (5) ObjectMother to remove the order from the database.

ObjectMother implementations rely on two different types of methods: creation methods and attachment methods that assist in tailoring the created objects. Both methods are public and can be used by test cases.

Creation methods can be nested so that it is like building with legos when writing an ObjectMother implementation. [Schuh, 2001]. First you might have an ObjectMother for user management. Then you can implement ObjectMother for orders and utilize the previous ObjectMother for the person making the order.

It is important to ensure that the ObjectMother is implemented using the same programming language used in the tests and maintained in the same version control system [Schuh, 2001]. When using separate tools for managing the external resources the maintenance is not as easy any more. Test objects might be in separate configuration files and test code does not have direct control over them.

When comparing ObjectMother and Mock Objects, it can be noticed that ObjectMother uses quite a different approach from Mock Objects. Whereas a Mock Object is used to eliminate the external resource, ObjectMother offers a centralized way to manage it. To utilize Mock Objects you need to have special kind of software architecture. ObjectMother has no special architectural needs and that's why it can also be used when testing legacy code.

ObjectMother is not a factory that creates Mock Objects because its creations should be as ready business objects as

possible. In contradictory Mock Object should be implemented as minimally as possible.

See Appendix B for code examples on how to use ObjectMother.

3.3 Upload tool for relational database.

Although effective white-box tests isolate an object by controlling outside dependencies, it is not always possible. There are situations where Mock Objects cannot be utilized. Some application frameworks hide the database access so that the separation of business logic and persistence is not possible, for example EJBs with container-managed persistence (CMP) or Java Data Objects (JDO) [Glover, 2004].

When doing agile development it is quite helpful to use frameworks that hide the database layer. Iterations can be shorter when you are not forced to write code for persistence. A drawback of this is the lack of control: the developer cannot control the database schema and database optimization.

ObjectMother is not suitable for all situations because it can be too time consuming to implement. If you are doing test for a large system, the ObjectMother implementation for every object is not trivial job to do. Also if you are doing very short iterations and want to avoid over-engineering, the separate management classes for test object can sound a little bit over-kill.

A fast and easy solution is to take snap shots from the database and upload the correct one before every test and remove it afterwards. There are many helper tools for the upload procedure. DbUnit is one of them; an open source solution for easier database setup and tear down for unit tests [Glover, 2004]. It allows updating small datasets to database and removing them after the tests. See Appendix C for DbUnit examples.

Figure 7 shows how to use a data upload tool from the unit test. The procedure is similar as is used for ObjectMother but instead of Java implementation the test objects are loaded from the configuration file.

First (1) the test code starts the data uploader with correct configuration files: in our example the configuration file contains data for one order. The data uploader uses (2) DBDriver for accessing database. After the test objects are created the unit test accesses (3) the business object for doing the wanted order cancellation operation. The Business object uses (4) DBDriver for storing the cancellation to the database. After returning the unit test uses (5) data uploader to verify that the right data is found from the database, and finally uses (6) data uploader to remove the test objects from the database.

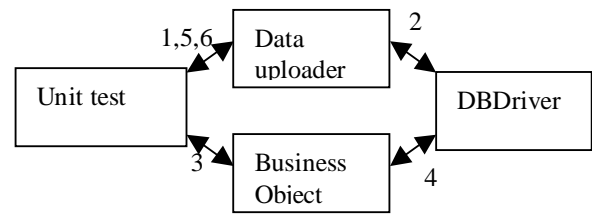


Figure 7. Unit test uses data upload tool to set up the database

The data uploader tools have three main functionalities for easier unit test implementation.

1. They support small datasets so that the developer is not forced to rebuild the whole database for every test. Small datasets are easier to maintain.
2. They offer comparison methods that allow comparing the database content to the configuration file. That makes it possible to verify that the database is in valid state after the test. It should be noticed that the configuration uploaded to the database should be different than the configuration used for verification.
3. They offer methods for removing uploaded datasets from the database. After removal the database should be in its initial state and this is usually faster operation than recreating the whole database from the scratch.

3.4 In-memory database

Smith et al [Smith, 2001] have found that running a large number of unit tests against a relational database is normally too slow to provide good enough feedback to the developers. Because in-memory database is fast, they promote it for the unit test. The idea can be generalized to other kind of external resources like external files that can be replaced by in-memory file system.

In-memory database is a lightweight object or relational database that is easy and fast to deploy and run. It is often fast to run by removing the latency of disc I/O. It usually supports no persistence nor uses any network resources. Many database products have variations from in-memory model to the full-featured database daemon. See Appendix D for examples of the database variations.

One big benefit for a in-memory database is that it allows developers to have their own database instances and solve the uniqueness problems of the single database instance. The problem is that usually there is no real data in it (Schuh, 2002). There is minimal set of data and everything is lost when the database is shut down.

In-memory database solve also most of the problems caused by the fact that code and database are not in synch. When the database is built up from scratch every time it is run, the synchronization is done more often than in normal database.

The synchronization problem can happen in constraints, tables and columns [Hassan, 2002]. For instance in application code we relaxed the constraint of our orders table so that every people can have multiple open orders. But in the database the constraints are left as is. This causes an error when the user tries to make the second order.

4. DISCUSSION

All four solution presented are quite different and have their weaknesses. In table 2 the methodologies are summarized based on sections 2 and 3.

TABLE 2. METHODOLOGY COMPARISON

	Mock Object	Object Mother	Data up-loader	In-memory database
Inline resource	yes	no	no	yes
Setup external resource	no	yes	yes	no
Make resource unique	yes	yes	no	yes
Reduce Data	yes	no	yes	yes
Architectural needs	yes	no	no	no

Mock Objects offer a nice way to eliminate the external resources by inline implementation. All test cases have their own instances of Mock Objects so the uniqueness is not a problem. The usage of Mock Objects also reduces amount of data because the mock implementation should be as minimal as possible. The only drawback is that the Mock Objects need a special kind of architecture and are not applicable to all situations.

ObjectMother can be used to set up external resources. Because it controls all creations it can easily check the uniqueness problems too. ObjectMother can be used with all kind of architectures. It does not offer any clear solution to reduce data. Its main philosophy is to offer test objects as ready and full of data as possible.

Like ObjectMother data uploader is suitable for setting up external resources – especially databases. It does not offer any help for making the resources unique; it just upload the data specified in the configuration and if the data already exists in the database on error is generated. Data uploader does not have any special architectural needs.

In-memory database can be seen as an inline copy of relational database and it makes the database resource unique for all developers. In-memory database also reduces the data objects because without persistence the size of the database is not growing all the time. It does not solve the problem of the database setup. Usually in-memory databases use standard interfaces so it is possible to replace the in-memory one with

a real one. Replacement is not a trivial task because of the vendor specific differences in SQL-syntax.

Mock Object should be used if possible. For components that needs database, in-memory database is a good solution. Test objects can be managed either with ObjectMother or with data uploader. Data uploader can be utilized faster than ObjectMother-implementations but in the long run ObjectMother offers an efficient way to reuse the object creation methods.

It looks like the different fixtures fit together but more study is needed to find out a good mixture. An interesting topic is to design a testing framework based on the results. It would also be useful to have more evidence for the approaches. Does the Mock Object make the unit test generally better or should it be used in special cases only? Is it better to maintain the test objects in the code than in the configuration file? And after all, how to define the goodness of the unit tests?

5. CONCLUSION

The best way to handle fixtures is to combine different strategies. No silver bullet exists but the best ones should be chosen depending on the architecture, available time and testing needs.

Best practices found from the literature can be used when setting up the fixtures. In this paper four practices are presented. External resources should be replaced with inline one. If the replacement is not possible then special care should be taken for the setup of the external resources. Also the resource should be made unique and the used data reduced to the minimum.

There are four different solutions presented. Mock Object eliminates external resource by inline dummy object. ObjectMother is a special case of a factory pattern. It returns object in any valid state. In-memory database simplifies the database management by replacing the relation database with an in-memory one. Data uploader tools are a fast way to manage the state of the database.

REFERENCES

- Andrea, J., Meszaros, G., Smith, S. Catalog of XP Project 'Smells'. The Third International Conference on eXtreme Programming and Agile Processes in Software Engineering- XP2002. May 2002, Alghero, Sardinia, Italy. See <<http://www.clrstream.com/Resources/xp2002/CatalogofXPProjectSmells.pdf>>
- Burke, E., Coyner, B. Java Extreme Programming Cookbook. March 2003. O'Reilly, 288 p. ISBN 0-596-00387-0
- David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, Elaine J. Weber, A framework for testing database applications, Proceedings of the International Symposium on Software Testing and Analysis, p.147-157, August 2000, Portland, Oregon, United States
- Deursen, A., Moonen, L., Bergh, A., Kok, G. Refactoring Test Code. The Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering. May 2001, Cagliari, Sardinia, Italy. See <<http://homepages.cwi.nl/~leon/papers/xp2001/xp2001.pdf>>

Elssamadisy, A., Schalliol, G. "Recognizing and responding to 'bad smells' in extreme programming." In Proceedings of the 24th international conference on Software engineering, Orlando, Florida, 2002, New York, NY: ACM Press. See <
<http://www.thoughtworks.com/library/Recognizing%20and%20Responding%20to%20Bad%20Smells%20in%20Extreme%20Programming.pdf>>

Freeman, S., Simmons, P. Retrofitting Unit Tests. The Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, May 2002, Alghero, Italy. See <
<http://www.xp2003.org/xp2002/atti/Freeman-Simmons-Retrofittingunittests.pdf>>

Glover, A. Effective Unit Testing with DbUnit. OnJava.com, 2004.

Hassan, M., Elssamadisy, A. Extreme Programming and Database Administration: Problems, Solutions, and Issues. The Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, May 2002, Alghero, Italy. See <
<http://people.umass.edu/amr/xpAndDB.pdf>>

Mackinnon, T. Freeman, S., Craig P. Endo-Testing: Unit Testing With Mock Objects. eXtreme Programming and Flexible Processes in Software Engineering – XP2000, June 2000, Cagliari, Sardinia, Italy. See <
<http://www.connextra.com/aboutUs/mockobjects.pdf>>

Robert E. Noonan, Richard H. Prosl, Unit testing frameworks, Proceedings of the 33rd SIGCSE technical symposium on Computer science education, March 2002, Cincinnati, Kentucky

Roock, S. Frameworks and Testing. The Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering, 2002.

Schuh, P., and Punke, S. ObjectMother: Easing Test Object Creation In XP. XP Universe 2001 Conference, July 2001, North Carolina. See <
<http://www.xpuniverse.com/Testing03.pdf>>

Schuh, P. Agility and the Database. The Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, May 2002, Alghero, Italy. See <
<http://www.agilealliance.com/articles/articles/PeterSchuh-AgilityandtheDatabase.pdf>>

Simpson, B. HSQLDB Beginner's Guide. Online material. See <
<http://hsqldb.sourceforge.net/doc/hsqldbBeginnersGuide.html>>

Smith, S., Meszaros, G. Increasing the Effectiveness of Automated Testing. The Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering. May 2001, Cagliari, Sardinia, Italy. See <
<http://www.clrstream.com/Resources/ImprovingTesting-Final1.PDF>>

APPENDIX A

First example below shows how to test order handling using Mock Objects. MyOrder contains information about one order and it uses handler to connect to the commerce system. Order is made using address and product name. In the example the handler is replaced with mock implementation that gets the product name and address as an expected value. After the test run the Mock Object is asked to verify that Order class has offered right product name and address.

```
void testOrderHandling() {
    Order myOrder =
        new Order(ADDRESS, PRODUCT);
    MockHandler myMockHandler =
        new MockHandler();
    myMockHandler
        .setResponseCode(OK);
    myMockHandler
        .setExpectedAddress(ADDRESS);
    myMockHandler
        .setExpectedProduct(PRODUCT);
```

```
myOrder.submit(myMockHandler);
myMockHandler.verify();
}
```

The next example shows how to test exceptions. Mock Object is instructed to throw ServerFailedException. If the exception occurs everything is ok otherwise test case fails.

```
public void testOrderSubmitFailure() {
    // initialization removed
    myMockHandler
        .setFailure(DATABASE_FAILURE);
    try {
        myOrder.submit(myMockHandler);
        fail("Database should have failed");
    } catch(ServerFailedException e){
        assert(true);
    }
    myMockServer.verify();
}
```

APPENDIX B

The example below shows how to use ObjectMother-pattern. createAddress() –method returns an address object in its initial state. The second method createInactiveAddress() creates the initial address and set it to the inactive state.

```
public static Address createAddress() {
    return createAddress("Helsinki",
        "Mannerheimint. 5",
        "00100");
}

public static Address
createInactiveAddress(){
    Address address = createAddress();
    address
        .setStatus(AddressStatus.INACTIVE);
    return address;
}
```

Method attachAddress() is an example of attachment-method. It adds address to the purchase order. By separating the attachment method from the Order and the Address creation it is possible to change the attachment logic later.

```
public static void attachAddress(
    Order order,
    Address address) {
    order.addAddress(address);
}
```

An example of test code using ObjectMother:

```
public void testOrder()
throws Exception {
    Address address =
        ObjectMother.createAddress();
```

```

Order order
  ObjectMother.createOrder();
ObjectMother.attachAddress(
  order, address);

  //testing code follows
}

```

APPENDIX C

DbUnit represents data using xml-syntax. These xml-files can be made manually or automatically from existing database. [Glover, 2004]

Below is an example of the data set of DbUnit. ORDER is the name of the table and the attributes are the table columns.

```

<data-set>
  <ORDER    order_id="11"
            status="new"
            creation_date="20.4.2004" />
  <ORDER    order_id="12"
            status="new"
            creation_date="18.4.2004" />
</data-set>

```

EMPLOYEE	order_id	status	Creation_date
#1	11	new	20.4.2004
#2	12	new	18.4.2004

DbUnit offers java API and ant tasks for developers. When using java API two interface methods need to be implemented. getConnection() returns JDBC Connection to the database. getDataSet() returns the input stream to the dataset.

```

protected IdatabaseConnection
getConnection()
  Class driverClass=
    Class.forName(MYSQL_DRIVER);
  Connection jdbcConnection =
    DriverManager.getConnection(...)
  return new DatabaseConnection(
    jdbcConnection);
}

protected IDataset getDataSet()
  throws Exception {
  return new FlatXmlDataSet(
    new FileInputStream("orders.xml"));
}

```

Ant tasks are arranged so that first dataset is inserted to the database and then the unit tests are run. Afterwards dataset is removed away. Ant tasks are used for initial setup and tear down. Test cases might have their own setup-methods.

"taskdef"-tag defines new tag called "dbunit". It is used for DbUnit related tasks and it has attribute for database connection related thing. Inside dbunit-tag there is operation tag.

```

<taskdef name="dbunit"
  classname="org.dbunit.ant.DbUnitTask" />

<dbunit driver="org.gjt.mn.mysql.Driver"
  url="jdbc:mysql://127.0.0.1/hr"
  userid="hr"
  password="hr"
  <operation type="insert"
    src="employee.xml" />
</dbunit>

```

APPENDIX C

HSQldb is an open source database product supporting three different modes to run: in-memory, standalone engine and server engine. All these modes can be used in one application.

In-memory mode supports no persistence. There is no daemon running and database uses no network resources. It is not possible to use external tools to access the data-store. It is slightly faster as no logging takes place.

Standalone engine uses files for persistence but there is no daemon running. The database is running in the same Java Virtual Machine as the application. The application and the database engine communicate through normal JDBC calls but these calls are handled internally without a network connection. In this mode, only one application can access a database at a time.

Server engine mode is similar than in full database products. Database server is running in its own process and multiple clients can access it simultaneously.

In-memory mode is specified by using a dot "." as the database file path.

```

Connection c =
  DriverManager.getConnection(
    "jdbc:hsqldb:.",
    "sa",
    "");

```

Giving the filename for database on the database file path specifies Standalone mode.

```

Connection c =
  DriverManager.getConnection(
    "jdbc:hsqldb:c:\db\test",
    "sa",
    "");

```