





Unitils

Summary
Downloads 
Tutorial
Wiki 
Cookbook
API Javadoc
Forum 

Project info

License
Team Members 
Issue Tracking 
Source Repository
Acknowledgements



Tutorial

Unit testing should be easy and intuitively... at least in theory. Real-life projects typically span multiple layers, are data-driven and use middleware technologies such as EJB and Hibernate.

Unitils originates from an attempt to get a more pragmatic view on unit testing. It started as a set of guidelines and resulted in an open source library containing utilities that facilitate the application of these guidelines.

This tutorial will show you, using examples, what Unitils can do for your projects. If you want to learn how you can setup Unitils to get started quickly, read the [cookbook](#).

- [Assertion utilities](#)
- [Unitils modules](#)
- [Database testing](#)
- [Automatic test database maintenance](#)
- [Testing with Hibernate](#)
- [Testing with JPA](#)
- [Testing with Spring](#)
- [Testing with mock objects](#)
- [EasyMock support](#)

Assertion utilities

We start this tutorial by describing some assertion utilities that can be used independently of the Unitils core module system, which is described in the remainder of this text. No configuration is needed, just add the Unitils jar and the [core dependencies](#) to your classpath and they are ready to be used in your tests.

Using reflection for assertion

A typical unit test contains a part in which it compares test result values with expected values. Unitils offers assertion utilities to help you with this. Let's start with an example of comparing 2 *User* instances having an id, a first and a last name:

```
public class User {  
    private long id;  
    private String first;  
    private String last;  
  
    public User(long id, String first, String last) {  
        this.id = id;  
        this.first = first;  
        this.last = last;  
    }  
}  
  
User user1 = new User(1, "John", "Doe");  
User user2 = new User(1, "John", "Doe");  
assertEquals(user1, user2);
```

You could expect this assertion to be successful since both instances contain the same values. This is however not the case, because *User* does not override the *equals()* method. Checking equals of 2 *User* instances is in that case the same as checking

whether both instances are the same. In other words, the `assertEquals` actually results in `user1 == user2` being tested causing the assertion to fail.

Suppose you implemented the `equals` method as follows:

```
public boolean equals(Object object) {
    if (object instanceof User) {
        return id == ((User) object).id;
    }
    return false;
}
```

This is a totally reasonable implementation for your application logic, stating that 2 user instances are referring to the same user when they have the same id. However, this method is not useful in your unit tests. Testing whether 2 objects are equal is now reduced to the test whether they both have the same id:

```
User user1 = new User(1, "John", "Doe");
User user2 = new User(1, "Jane", "Smith");
assertEquals(user1, user2);
```

This assertion will be successful, which is probably not what you want. Best is to avoid using `equals()` altogether when comparing objects (except of course for objects with value semantics such as `java.lang.String`, ...). One approach can be to do the comparison of each of the properties one by one:

```
User user1 = new User(1, "John", "Doe");
User user2 = new User(1, "John", "Doe");
assertEquals(user1.getId(), user2.getId());
assertEquals(user1.getFirst(), user2.getFirst());
assertEquals(user1.getLast(), user2.getLast());
```

Unitils offers utilities that help you perform these checks more easily, through reflection. Using `ReflectionAssert.assertEquals`, the above example could be re-written as follows:

```
User user1 = new User(1, "John", "Doe");
User user2 = new User(1, "John", "Doe");
assertReflectionEquals(user1, user2);
```

This assertion loops over all fields in both objects and compares their values using reflection. For the above example, it will first compare both *id* field values, next both *first* field values and finally both *last* fields values.

If a field value itself is also an object, it will recursively be compared field by field using reflection. The same is true for collections, maps and arrays. Their elements will be traversed and recursively be compared using reflection. If a value is a primitive type (`int`, `long`, ...) or one of the primitive wrapper types (`Integer`, `Long`, ...) it will be compared by value (using `==`). As a result, following assertions will be successful:

```
assertReflectionEquals(1, 1L);

List<Double> myList = new ArrayList<Double>();
myList.add(1.0);
myList.add(2.0);
assertReflectionEquals(Arrays.asList(1, 2), myList);
```

Lenient assertions

For reasons of maintainability, it's important to only add assertions that are of value for

the test. Let me clarify this with an example: suppose you have a test for a calculation of an account balance. There is no need to add any assertions to this test that will check the value of the name of the bank-customer. This will only add complexity to your test making it more difficult to understand and, more importantly, more brittle against changes to the code. If you want your test code to easily survive refactorings and other code changes, make sure you limit your assertions and test data setup to the scope of the test.

To help you in writing such assertions we added some levels of leniency to the `ReflectionAssert` checks. We'll cover these leniency levels in the sections that follow.

Lenient order

A first type of leniency you can specify is ignoring the order of elements in a collection or array. When working with lists, you are often not interested in the actual order of the elements. For example, code for retrieving all bank-accounts with a negative balance will typically return them in an order that is unimportant for the actual processing.

To implement this behavior, the `ReflectionAssert.assertReflectionEquals` method can be configured to ignore ordering by supplying it the `ReflectionComparatorMode.LENIENT_ORDER` comparator mode:

```
List<Integer> myList = Arrays.asList(3, 2, 1);
assertReflectionEquals(Arrays.asList(1, 2, 3), myList,
    LENIENT_ORDER);
```

Ignoring defaults

A second type of leniency is specified by the `ReflectionComparatorMode.IGNORE_DEFAULTS` mode. When this mode is set, java default values, like `null` for objects and `0` or `false` for values are ignored. In other words, only the fields that you instantiate in your expected objects are used in the comparison.

As an example, suppose you have a user class with a first name, last name, street... field, but you only want to check that the resulting user instance has a certain first name and street value, ignoring the other field values:

```
User actualUser = new User("John", "Doe", new Address("First
street", "12", "Brussels"));
User expectedUser = new User("John", null, new Address("First
street", null, null));
assertReflectionEquals(expectedUser, actualUser,
    IGNORE_DEFAULTS);
```

You specify that you want to ignore a field by setting this value to null in the left (=expected) instance. Right-instance fields that have default values will still be compared.

```
assertReflectionEquals(null, anyObject, IGNORE_DEFAULTS); //
Succeeds
assertReflectionEquals(anyObject, null, IGNORE_DEFAULTS); //
Fails
```

Lenient dates

A third type of leniency is `ReflectionComparatorMode.LENIENT_DATES`. This will assert that the date field values in both instances are both set or both equal to null; the actual values of the dates are ignored. This can be useful if you want to do strict checking on the fields of objects (without using `ReflectionComparatorMode.IGNORE_DEFAULTS`), but there are fields in your object set to the current date or time that you want to ignore.

```
Date actualDate = new Date(44444);
Date expectedDate = new Date();
assertReflectionEquals(expectedDate, actualDate, LENIENT_DATES);
```

assertLenientEquals


The *ReflectionAssert* class also offers an assertion for which two of the leniency levels, lenient order and ignore defaults are pre-set: *ReflectionAssert.assertLenientEquals*. The above examples can therefore be simplified as follows:

```
List<Integer> myList = Arrays.asList(3, 2, 1);
assertLenientEquals(Arrays.asList(1, 2, 3), myList);

assertLenientEquals(null, "any"); // Succeeds
assertLenientEquals("any", null); // Fails
```

This len/ref thing is a general naming convention: **assertReflection...** is the version that is strict by default and for which you can manually set the leniency levels, **assertLenient...** is the version for which the lenient order and ignore defaults are pre-set.

Property assertions

The *assertLenientEquals* and *assertReflectionEquals* methods compare objects as a whole. *ReflectionAssert* also contains methods to compare a specific property of two objects: *assertPropertyLenientEquals* and *assertPropertyReflectionEquals*. The actual field that needs to be compared is specified using the **OGNL**  notation. This language supports typical bean property expressions, like `field1.innerField` and a number of more powerful constructions.

Some examples of property comparisons are:

```
assertPropertyLenientEquals("id", 1, user);
assertPropertyLenientEquals("address.street", "First street",
user);
```

You can also supply collections as arguments for this method. In that case the specified field will be compared on each element in the collection. This provides an easy way to for example check whether all users in a retrieved list have certain id:

```
assertPropertyLenientEquals("id", Arrays.asList(1, 2, 3),
users);
assertPropertyLenientEquals("address.street",
Arrays.asList("First street", "Second street", "Third street"),
users);
```

Again there are two versions of each method: **assertPropertyReflectionEquals** and **assertPropertyLenientEquals**. The ref version doesn't have leniency specified by default but provides the option to specify leniency modes, the len version has lenient order and ignore defaults set as fixed leniency modes.

Unitils modules

The rest of this tutorial describes Unitils' module system and the modules that are provided. We start with explaining how to setup a test environment to let your tests make use of these modules. The subsequent chapters dive deeper into the functionality of each

of these modules.

Configuration

As with many projects, Unitils needs some configuration of its services. By default, there are 3 levels of configuration, each level overriding the settings of the previous one:

- *unitils-defaults.properties*: default configuration that is shipped with the distribution of Unitils
- *unitils.properties*: can contain project-wide configuration
- *unitils-local.properties*: can contain user-specific configuration

The first file, **unitils-default.properties**, contains default values and is packaged in the Unitils jar. There is no need to make changes to this file, but it can be used as reference, since it contains all possible configuration settings for Unitils.

The second, *unitils.properties*, can override the defaults and is typically used to set values for configuration settings for all developers on a project. For example if your project uses an oracle database, you can create a *unitils.properties* file that overrides the driver class name and url properties as follows:

```
database.driverClassName=oracle.jdbc.driver.OracleDriver
database.url=jdbc:oracle:thin:@yourmachine:1521:YOUR_DB
```

This file is not required, but if you create one, it should be placed somewhere in the classpath of your project. To get started, you can find a sample file that contains most commonly used configuration settings in following **unitils.properties** template.

The last file, *unitils-local.properties*, is optional as well. It can contain settings that override the project settings and is used for defining developer specific settings. For example, if each user uses its own unit-test database schema, you can create a *unitils-local.properties* for each user that contains the corresponding username, password and database schema:

```
database.userName=john
database.password=secret
database.schemaNames=test_john
```

Each of these *unitils-local.properties* files should be placed in the corresponding home folders of the user's (*System.getProperty("user.home")*). A typical local configuration can be found in following sample **unitils-local.properties** template.



The name of the local file, *unitils-local.properties*, is also defined by a property. This makes it possible to use different names for each of the projects you are working on. For example, suppose you're using Unitils on a project named *projectOne* and want to start using it on a new project named *projectTwo*. Adding following property to the file *unitils.properties* of *projectTwo* will make Unitils use *projectTwo-local.properties* as the local properties file for this project:

```
unitils.configuration.localFileName=projectTwo-local.properties
```

Making your test Unitils-enabled

Unitils offers services to test classes through a test listener system. To enable Unitils to provide services to your tests, you first have to Unitils-enable them. This can be done easily by (indirectly) extending from a Unitils base test-class. Currently there are base classes for the major test frameworks:

- **JUnit3** : `org.unitils.UnitilsJUnit3`

- **JUnit4** : `org.unitils.UnitilsJUnit4`
- **TestNG** : `org.unitils.UnitilsTestNG`

As an example, suppose you have a JUnit3 test that you want to Unitils-enable:

```
import org.unitils.UnitilsJUnit3;


public class MyTest extends UnitilsJUnit3 {
}
```

Typically you would create your own base test class containing some utility behavior common for all your tests, e.g. datasource injection, and then let this base class extend from one of the Unitils base classes.

When you use JUnit4 you can also Unitils-enable a class by adding a `@RunWith` annotation instead of extending from the base class:

```
import org.junit.runner.RunWith;
import org.unitils.UnitilsJUnit4TestClassRunner;

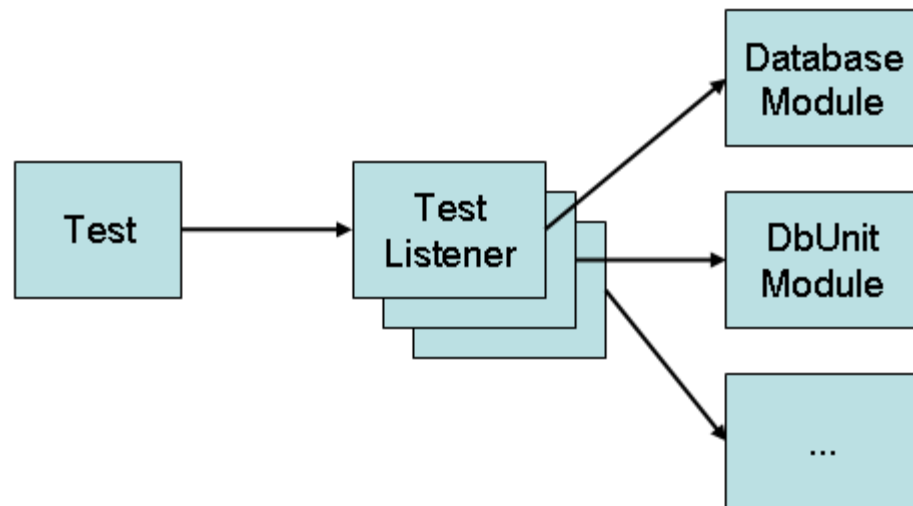
@RunWith(UnitilsJUnit4TestClassRunner.class)
public class MyTest {
}
```

Instead of extending from one of Unitils base classes you could also create a custom Unitils-enabled superclass by copying the source code of the base class (e.g. *UnitilsJUnit3*) to the custom superclass. This superclass can then still extend from another class, e.g. from **Spring** 's `AbstractDependencyInjectionSpringContextTests` which is already a subclass of JUnit3's `TestCase`.

Module system

Before starting with the examples, let's first take a look at some of the concepts used in Unitils and how it functions under the hood.

Unitils is structured as an easily extendible system of modules. Each of these modules offers some type of service to a test by listening to the execution of the test and invoking the correct service behavior when needed. The base classes that Unitils provides (*UnitilsJUnit3*, *UnitilsJUnit4*, *UnitilsTestNG*), couple the unit tests with the modules listening system.



This mechanism offers a uniform way for providing extra services to your tests and a flexible way of adding services without having to change the superclass of the test. Adding new services is as easy as adding a new module and registering this module in

one of Unitils' configuration files.


Currently, following modules are available in Unitils:

- *DatabaseModule*: unit-test database maintenance and connection pooling
- *DbUnitModule*: test data management using DbUnit
- *HibernateModule*: Hibernate configuration support and automatic database mapping checking
- *MockModule*: support for creating mocks using the Unitils mock framework
- *EasyMockModule*: support for creating mocks using EasyMock
- *InjectModule*: support for injecting (mock) objects into other objects
- *SpringModule*: support for loading application contexts and retrieving and injecting Spring beans

Database testing

Unit tests for the database layer can be extremely valuable when building enterprise applications, but are often abandoned because of their complexity. Unitils greatly reduces this complexity, making database testing easy and maintainable. The following sections describe the support that the *DatabaseModule* and *DbUnitModule* have to offer for your database tests.

Managing test data with DbUnit

Database tests should be run on a unit test database, giving you complete and fine grained control over the test data that is used. The *DbUnitModule* builds further on **DbUnit**  to provide support for working with test data sets.

Loading test data sets


Let's start with an example of a UserDao with a simple findByName method for retrieving a user based on its first and last name. A typical unit test looks as follows:

```
@DataSet
public class UserDaoTest extends UnitilsJUnit4 {

    @Test
    public void testFindByName() {
        User result = userDao.findByName("doe", "john");
        assertPropertyLenientEquals("userName", "jdoe",
result);
    }

    @Test
    public void testFindByMinimalAge() {
        List<User> result = userDao.findByMinimalAge(18);
        assertPropertyLenientEquals("firstName",
Arrays.asList("jack"), result);
    }
}
```

The *@DataSet* annotation in the test instructs Unitils to look for DbUnit data files that need to be loaded for the test. If no file name is specified, Unitils automatically looks for a data set file that is in the same directory as the test class and has following name pattern: *className.xml*

The data set file should be in DbUnit's **FlatXMLDataSet**  file format and should contain all data needed for the test. All existing content of tables in the data set will first be deleted, then all data of the data set will be inserted. Tables that are not in the data set will not be cleared. You can explicitly clear a table by adding an empty table element, e.g. `<MY_TABLE />` to the data set file. If you explicitly want to specify a null value, you can do so by using the value *[null]*.

For the *UserDAOTest* we could for example create a class level data set file named *UserDAOTest.xml* and put it in the same directory as the *UserDAOTest* class:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>

    <usergroup name="admin" />
    <user userName="jdoe" name="doe"     firstname="john"
userGroup="admin" />

    <usergroup name="sales" />
    <user userName="smith" name="smith" userGroup="sales" />

</dataset>
```

This will clear the user and usergroup tables and insert the user groups and user records. The first name of the user named 'smith' will be set to a null value.

Suppose the *testFindByMinimalAge()* method needs a specific data set instead of the class-level data set. You could create a file named *UserDAOTest.testFindByMinimalAge.xml* and put that file in the same directory as the test class:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <user userName="jack" age="18" />
    <user userName="jim" age="17" />
</dataset>
```

You can then use this data set by adding an *@DataSet* annotation to the method, overriding the default data set file of the class.

```
public class UserDAOTest extends UnitilsJUnit4 {

    @Test
    @DataSet("UserDAOTest.testFindByMinimalAge.xml")
    public void testFindByMinimalAge() {
        List<User> result = userDao.findByMinimalAge(18);
        assertPropertyLenientEquals("firstName",
Arrays.asList("jack"), result);
    }
}
```

Method-level data sets should not be overused, since having more data files means you have to do more maintenance. You should first try to reuse the data in the class level data set - in most cases a small set of test records can be reused for multiple tests. If this would result in a big and incoherent data set, it may be better to use a method specific one, or to try to split the test in 2 separate tests, each having their own data set.

By setting the *@DataSet* annotation on class or superclass level a data set is loaded for every test in the class. If a data set should only be loaded for some of the tests, you should remove the class level annotation and annotate the test methods instead. Datasets will then only be loaded for the annotated tests. If your data set file does not comply with the naming conventions as described above, you can also specify a file name explicitly by passing the name as an argument to the *@DataSet* annotation. You can also specify multiple data set file names, in case you want to use more than 1 data set:

```
@DataSet({"UserDAOTest_general.xml", "ConfigSettings.xml"})
public class UserDAOTest extends UnitilsJUnit4 {

    @Test
    public void testFindByName() {
        User result = userDao.findByName("doe", "john");
        assertPropertyLenientEquals("userName", "jdoe",
```



```

        result);
    }

    @Test
    @DataSet("UserDAOTest_ages.xml")
    public void testFindByMinimalAge() {
        List<User> result = userDao.findByMinimalAge(18);
        assertPropertyLenientEquals("firstName",
            Arrays.asList("jack"), result);
    }
}

```

Configuring the dataset load strategy

By default, datasets are loaded into the database using a *clean insert* strategy. This means that all data in the tables that are present in the dataset is deleted, after which the test data records are inserted. This behavior is configurable, it can be modified by changing the value of the property *DbUnitModule.DataSet.loadStrategy.default*. Suppose we add following in unitils.properties:

```
DbUnitModule.DataSet.loadStrategy.default=org.unitils.dbunit.datas
```

This sets the load strategy to *insert* instead of *clean insert*. The result is that data already available in the tables present in the dataset is not deleted, and test data records are simply inserted.

The loadStrategy that is used can also be configured for specific tests using an attribute of the *@DataSet* annotation. E.g.:

```
@DataSet(loadStrategy = InsertLoadStrategy.class)
```

For those familiar with DbUnit, configuring the load strategy is equivalent to using a different DatabaseOperation. Following are load strategies that are supported by default:

- CleanInsertLoadStrategy: Insert the dataset, after removal of all data currently present in the tables specified in the dataset
- InsertLoadStrategy: Simply insert the dataset into the database
- RefreshLoadStrategy: 'Refresh' the contents of the database with the contents of the dataset. This means that data of existing rows is updated and non-existing rows are inserted. Any rows that are in the database but not in the dataset stay unaffected
- UpdateLoadStrategy: Update the contents of the database with the contents of the dataset. This means that data of existing rows is updated. Fails if the dataset contains records that are not in the database (i.e. a records having the same value for the primary key column).

Configuring the dataset factory

Dataset files in Unitils have the *multischema xml* format, which is an extended version of DbUnits *FlatXmlDataSet* format. Configuration of the file format and file extension is handled by a *DataSetFactory*.

Although Unitils currently only supports one dataset format, the possibility is offered to implement a custom implementation of *DataSetFactory* to use a different file format. This can be done by specifying the value of the property *DbUnitModule.DataSet.factory.default* in unitils.properties or by using the factory attribute of the *@DataSet* annotation. Such a custom factory could e.g. be implemented to create an instance of DbUnit's *XlsDataSet*, if you want to use Excel files instead of XML.

Verifying test results

Sometimes it can be useful to use data sets for checking the contents of a database after a test was run. For example when you want to check the result of a bulk update method or a stored procedure.

Following example tests a method that disables all user accounts that haven't been used for an entire year:

```
public class UserDaoTest extends UnitilsJUnit4 {

    @Test @ExpectedDataSet
    public void testInactivateOldAccounts() {
        userDao.inactivateOldAccounts();
    }
}
```

Note that we have added the *@ExpectedDataSet* to the test method. This will instruct Unitils to look for a data set file named *UserDaoTest.testInactivateOldAccounts-result.xml* and compare the contents of the database with the contents of the data set:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <user userName="jack" active="true" />
  <user userName="jim" active="false" />
</dataset>
```

For this data set, it will check whether there are 2 different user records with corresponding values in the User table in the database. Other records and other tables are not taken into account.

As with the *@DataSet* annotation, a file name can explicitly be specified. If no name is specified, following naming pattern will be used: *className.methodName-result.xml*

The use of result data sets should be kept to a minimum. Adding new data sets means more maintenance. As an alternative, you should always try to perform the same check in the test code (e.g. by using a method *findActiveUsers()*).

Using multi-schema data sets

Some applications connect to more than one database schema. To facilitate this, Unitils extends the data set xml definition to enable it to contain data for multiple schemas. Following example loads data for tables in 2 different schemas:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset xmlns="SCHEMA_A" xmlns:b="SCHEMA_B">
  <user id="1" userName="jack" />
  <b:role id="1" roleName="admin" />
</dataset>
```

In this case we defined 2 schemas, *SCHEMA_A* and *SCHEMA_B*. The first schema, *SCHEMA_A*, is linked to the default xml namespace, the second schema, *SCHEMA_B*, is linked to xml namespace b. If a table xml element is prefixed with namespace b, the table is expected to be in schema *SCHEMA_B*, if it doesn't have a namespace prefix it is considered to be in *SCHEMA_A*. In the example, test data is defined for tables *SCHEMA_A.user* and *SCHEMA_B.role*.

If no default namespace is specified, it is by default set to the first of the list of schema names defined by the property *database.schemaNames*. So suppose you have defined following schema names:

```
database.schemaNames=SCHEMA_A, SCHEMA_B
```

This will make *SCHEMA_A* the default schema. You can then simplify the above data set example by leaving out the default namespace declaration:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset xmlns:b="SCHEMA_B">
  <user id="1" userName="jack" />
  <b:role id="1" roleName="admin" />
</dataset>
```

Connecting to the test database

In the above examples we left out 1 important thing: where is the *DataSource* for connecting to the test database coming from and how can we let our DAO classes make use of this *DataSource*?

When the first database test in your test suite is run, Unitils will create a *DataSource* instance that will connect to your unit test database using the settings defined in the properties. Subsequent database tests will then reuse this same *datasource* instance. The connection details are specified by following properties:

```
database.driverClassName=oracle.jdbc.driver.OracleDriver
database.url=jdbc:oracle:thin:@yourmachine:1521:YOUR_DB
database.userName=john
database.password=secret
database.schemaNames=test_john
```

As described in the [Configuration](#) section you would typically put the driver and url properties in the project's *unitils.properties*, defining them for the project as a whole and put the user, password and schema properties in the *unitils-local.properties*, defining them for that specific user only. This way you can make each developer connect to its own unit test database schema and run the tests without interference with others.

Before a test is set up, the *DataSource* instance will be injected into the test instance: if a field or setter method is annotated with *@TestDataSource* is found, it will be set to or called with this instance value. You still have to provide some project specific code that configures your code to use this *datasource*. Typically all this is implemented once in a project-specific superclass for all your database tests. A simple example of such a base class could be:

```
public abstract class BaseDAOTest extends UnitilsJUnit4 {

    @TestDataSource
    private DataSource dataSource;

    @Before
    public void initializeDao() {
        BaseDAO dao = getDaoUnderTest();
        dao.setDataSource(dataSource);
    }

    protected abstract BaseDAO getDaoUnderTest();

}
```

The above example uses annotations to get a reference to the *datasource*. Another way of making your code use the Unitils *DataSource* is by calling *DatabaseUnitils.getDataSource()*.

Transactions

For different reasons, it can be useful to run tests that access a test database in a

transaction. The most important reasons are the following:

- Database actions exist that only work properly when executed in a transaction, such as when using *SELECT FOR UPDATE* or triggers that execute *ON COMMIT*.
- A lot of projects like to run their test on a database that is pre-filled with some general-purpose data. During each test, data may be inserted or modified. To make sure the database is in the same known state before every test, a transaction is started before and rolled back after each test.
- When using hibernate or JPA, since these require you to run each test in a transaction to make the system work properly.

By default every test is executed in a transaction, which is committed at the end of the test.

This default behavior can be changed by by setting a property. Transaction management can for example be disabled (= auto-commit behavior) as follows:

```
DatabaseModule.Transactionnal.value.default=disabled
```

Other supported values for this property are *commit*, *rollback* and *disabled*.

The transactional behavior can also be modified at the level of a test class, by annotating the test class with *@Transactional*. For example:

```
@Transactional(TransactionMode.ROLLBACK)
public class UserDaoTest extends UnitilsJUnit4 {
```

This will roll back the transaction after each test in the test class. The *@Transactional* annotation is inheritable, so it can be moved to a shared superclass instead of specifying it for each test separately.

Under the hoods, unitils depends on spring to implement transaction management. This doesn't mean you that need to use spring in your application code for transaction management. The fact that spring is used, is completely transparent.

If you are using unitils' spring support (see [Testing with spring](#)), and you've configured a bean of type *PlatformTransactionManager* in your spring configuration, unitils will make use of this transaction manager.

Automatic test database maintenance

When writing database tests, keep in mind following guidelines:

- Use small sets of test data, containing as few data as possible. In your data files, only specify columns that are used in join columns or the where clause of the tested query.
- Make data sets test class specific. Don't reuse data sets between different test classes, for example do not use 1 big domain data set for all your test classes. Doing so will make it very difficult to make changes to your test data for a test without braking anything for another test. You are writing a unit test and such a test should be independent of other tests.
- Don't use too many data sets. The more data sets you use, the more maintenance is needed. Try to reuse the testclass data set for all tests in that testclass. Only use method data sets if it makes your tests more understandable and clear.
- Limit the use of expected result data sets. If you do use them, only include the tables and columns that are important for the test and leave out the rest.
- Use a database schema per developer. This allows developers to insert test data and run tests without interfering with each other.
- Disable all foreign key and not null constraints on the test databases. This way, the data files need to contain no more data than absolutely necessary.

Following section describes how Unitils provides support for these 2 last guidelines: automatic test database schema maintenance and constraints disabling.

Maintaining the database structure

The *DBMaintainer* can automatically maintain each developer's individual database schema. It will check whether database structure updates are available and apply them if necessary.

The database maintainer is disabled by default. If you want to use it you have to enable it by setting following property:

```
updateDataBaseSchema.enabled=true
```

If enabled, the database maintainer is invoked automatically when executing a test for which a *DataSource* must be created. This happens at most once during each test run. If no persistence layer tests (or any other tests that require a *DataSource*) are run, the database maintainer is not invoked.

The database maintainer works as follows: a project has a directory that contains all database scripts of the application. The database maintainer monitors this directory and makes sure that every new script and every change to an existing script is applied on the database.

The directory in which the scripts are located can be configured with following property:

```
dbMaintainer.script.locations=myproject/dbscripts
```

Multiple directories can be specified, separated by commas. Subdirectories are also scanned for script files. All scripts are required to follow a particular naming pattern: they start with a version number, followed by an underscore and end with '.sql' (the supported extensions are configurable). The example below shows some typical scripts names. The leading zeroes in the example are not required, they are only added for convenience, to make sure they are shown in proper sequence in a file explorer window.

```
dbscripts/ 001_initial.sql  
           002_tracking_updates.sql  
           003_auditing_updates.sql
```

Suppose you add a new script, this time with version number 4:

004_create_user_admin_tables.sql. The next time you execute a database test, the database maintainer will notice that the database structure is no longer up to date. It will update the database schema incrementally by executing all of the new scripts, in this case only *004_create_user_admin_tables.sql*.

The database maintainer will also notice if one of the existing database scripts was modified. For example suppose that *002_tracking_updates.sql* was changed. In that case, the database maintainer will first clear the database, thereby removing all tables and then recreate it from scratch, executing all scripts in sequence.

The database script folders may also contain scripts without a version number. Such scripts are executed last, after execution of all versioned scripts. If an existing version-less script is updated, this script will be applied to the database again, the database is not recreated from scratch. We call such scripts 'repeatable' scripts, since their execution can be repeated without having to update from scratch. You typically use this for stored procedure definitions, views or definition of reference data.

For performance reasons, it's a good practice to work incrementally, each time adding new update files for small database updates. This makes sure updates can be executed very quickly. If your database is small, you could of course just as well simply maintain one script and update it each time a database update is needed.

Database scripts are often organized in multiple directories, usually grouped per

application version. To define the order in which the scripts in these folders must be executed, the folder names can be prefixed by a version number, just like individual scripts. For example:

```
dbscripts/ 01_production/ 001_initial.sql
              002_auditing_updates.sql
              02_latest_dev/ 001_add_user_table.sql
                              002_rename_product_id.sql
```

In this example, the scripts in folder 01_production are executed first, the scripts in 02_latest_dev are executed afterwards.

Scripts located in a folders without a version number are executed after the scripts located in a folder with a version number. Folders may contain subfolders that in turn have a version number. The same ordering system is recursively applied to those subfolders.

The database maintainer stores the current version of a database schema in a table called *DBMAINTAIN_SCRIPTS*. This way, it can determine whether the database is up to date or not.

This table is by default not created automatically. This is to make sure unitils does not clear the wrong database by accident. Suppose for example you have configured unitils to connect to your system test database. If you would enable the database maintainer it would see that the database is not in synch with the scripts (because there is no *DBMAINTAIN_SCRIPTS* table) and as a result drop the whole database and recreate it from scratch. This is probably not the desired result. Therefore a database can only be updated automatically if it is 'unitils-enabled', i.e. when it contains a *DBMAINTAIN_SCRIPTS* table. ? The database maintainer will throw an exception when it does not find a version table. The correct DDL statements for creating the version table will be displayed in the exception message. If desired, you can make unitils create the *DBMAINTAIN_SCRIPTS* table automatically by setting following property to true.

```
dbMaintainer.autoCreateExecutedScriptsTable=true
```

Tip: We strongly advise to manage your project's database scripts in a version control system (e.g. cvs, subversion). This way, database scripts are brought into version control just like regular code. Developers can make updates to their local copy of the scripts and have them applied to their own test database. Database updates and associated java code changes are checked in together, and the rest of the team always receives these updates all at once, so that they aren't bothered with failing tests.

Oracle PL/SQL support

Unitils also supports Oracle functions and stored procedures, written using the PL/SQL syntax. If the database dialect is set to Oracle (property `database.dialect=oracle`), Unitils makes sure that function, stored procedure, trigger, type, package and library declarations are correctly parsed and sent to the JDBC driver as a whole. Blocks of PL/SQL code must always end with a line containing a single forward slash (/)!

Tip: Declarations of functions, stored procedures etc. can usually be executed multiple times (using the *CREATE OR REPLACE* syntax). If a stored procedure changes, the script that declares the procedure can simply be re-executed: the old definition is overwritten by the new one. It's therefore a good idea to put these declarations in script files containing no version number: When such a script is modified, the modified script is simply re-executed, without re-creating the database from scratch. In case of a from-scratch creation of the database, the versionless scripts are executed after the versioned scripts, which is normally also the desired behavior.

Post processing scripts

It's often useful to have one or more database post-processing scripts: e.g. a script that compiles all PL/SQL procedures, that adds any missing grants, or that re-calculates the Oracle statistics. Scripts are automatically recognized as post-processing scripts if they are located in the subdirectory `postprocessing` of the scripts root dir. To change the directory in which post processing scripts are located, you can use `dbMaintainer.postProcessingScript.directoryName`

Preserving items

It's possible to exclude certain database objects from being dropped when a `fromScratch` update occurs, or when the `clearDatabase` operation is invoked: the data in these tables is also not removed when performing an update using the `cleanDatabase` option. If you want a table to be dropped in case of a `fromScratch` update, but you want it's data to be preserved when performing the `cleanDatabase` operation, you can use one of the `preserveDataOnly` properties.

```
# Comma separated list of database items that may not be dropped
# or cleared by DbMaintain when
# updating the database from scratch.
# Schemas can also be preserved entirely. If identifiers are
# quoted (eg "" for oracle) they are considered
# case sensitive. Items may be prefixed with the schema name.
# Items that do not have a schema prefix are
# considered to be in the default schema.
dbMaintainer.preserve.schemas=
dbMaintainer.preserve.tables=
dbMaintainer.preserve.views=
dbMaintainer.preserve.materializedViews=
dbMaintainer.preserve.synonyms=
dbMaintainer.preserve.sequences=

# Comma separated list of table names. The tables listed here
# will not be emptied during a cleanDatabase operation.
# Data of the dbmaintain_scripts table is preserved
# automatically.
# Tables listed here will still be dropped before a fromScratch
# update. If this is not desirable
# you should use the property dbMaintainer.preserve.tables
# instead.
# Schemas can also be preserved entirely. If identifiers are
# quoted (eg "" for oracle) they are considered
# case sensitive. Items may be prefixed with the schema name.
# Items that do not have a schema prefix are considered
# to be in the default schema
dbMaintainer.preserveDataOnly.schemas=
dbMaintainer.preserveDataOnly.tables=
```

Note that the items in the `preserve` properties must exist! If one of them doesn't exist, the operation is aborted and an error message is given.

Note that in the ant tasks, no attributes are provided for configuring items to preserve.

Disabling constraints and updating sequences

The database maintainer does more than just update the schema structure. After each update, it performs following actions:

- *Disable foreign key and not-null constraints*
- *Update sequences to a high value*
- *Generate an XSD or DTD of the schema structure*

To be able to work with the smallest possible test data sets, all foreign key and not null constraints on the test database are disabled. This way you only need to add data that is of real value for your test. Because less data is specified in the data set, they are easier to write, indicate more clearly what is actually being tested and, more importantly, they

become much easier to maintain.

All sequences and identity columns are updated to a sufficiently high initial value (1000 by default). This way, you can use fixed primary key values when inserting test data, avoiding conflicts with for example primary key sequences.

The database maintainer is highly configurable. You can switch on or off things like constraints disabling, disable from scratch updates... Consult the [unitils-default.properties](#) for a list of possible configuration settings and their values.

Some functionality of the database maintainer, e.g. constraint disabling, uses DBMS specific functionality. This functionality will only run properly when the correct dialect (oracle, mysql...) is configured. If you are for example using an Oracle database, you should set the dialect property to *oracle*:

```
# Supported values are 'oracle', 'db2', 'mysql', 'hsqldb',
'postgresql', 'derby' and 'mssql'
database.dialect=oracle
```

Currently there is support for following databases:

- **Oracle** - Tested with version 9i, 10g, 10XE
- **Hsqldb** - Tested with version 1.8.0
- **MySQL** - Tested with version 5.0
- **PostgreSQL** - Tested with version 8.2
- **DB2** - Tested with version 9
- **Derby** - Tested with version 10.2.2.0
- **MS-SQL Server** - Tested with version 2005

Generate an XSD or DTD of the database structure

After updating the database structure, the database maintainer will also generate a number of xml schema definitions (XSD) that describe the structure of the database. These XSDs can help you to write datasets more quickly. Modern IDEs offer you code-completion and give errors when e.g. a column name is incorrectly spelled in one of your dataset files.

The database maintainer generates an XSD for each configured database schema. These XSDs have the same name as the corresponding database schema, e.g. *SCHEMA_A.xsd*. Next to these schema XSDs, a general *dataset.xsd* is generated. This XSD bundles all schema XSDs. This is the one you have to include in your XML declaration.

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="target/xsd/dataset.xsd">

    <user id="1" userName="jack" />
    <user id="1" userName="jack" />
    <role id="1" roleName="admin" />

</dataset>
```

The target location for the XSDs is configured by setting following property:

```
dataSetStructureGenerator.xsd.dirName=target/xsd
```

Unitils creates a file *dataset.xsd* and stores it in this directory. If the directory didn't exist yet, it's created automatically.

The generated XSD also supports multiple database shemas. Suppose you are using 2 schemas:


```
database.schemaNames=SCHEMA_A, SCHEMA_B
```

Since *SCHEMA_A* is listed first, it is considered to be the default database schema. The database maintainer generates 3 XSD files for this database:

```
target/xsd/ dataset.xsd
target/xsd/ SCHEMA_A.xsd
target/xsd/ SCHEMA_B.xsd
```

You can use these XSDs like demonstrated in following example:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="target/xsd/dataset.xsd"
xmlns:a="SCHEMA_A" xmlns:b="SCHEMA_B">

    <user id="1" userName="jack" />
    <a:user id="1" userName="jack" />
    <b:role id="1" roleName="admin" />

</dataset>
```

The first 2 attributes of the dataset element configure the XSD of the dataset to point to *dataset.xsd*. This enables the validation of 3 namespaces: the default namespace, the *SCHEMA_A* and the *SCHEMA_B* namespace. The last 2 attributes configure the usage of the *SCHEMA_A* and *SCHEMA_B* namespaces.

As a result, the user table element (without namespace prefix) is validated against the default database schema structure, i.e. *SCHEMA_A*. The same is true for the user element that is explicitly prefixed with the *SCHEMA_A* namespace (prefix a). The role table element is prefixed with the *SCHEMA_B* namespace and will thus be validated against the *SCHEMA_B* database structure.

If you only use 1 database schema, there is no need to use namespaces, you can simply use the default namespace.

For backward compatibility reasons, a DTD can also be generated instead of an XSD. This DTD however does not support multiple database schemas. Only the structure of the default database schema, i.e. the first one in the *database.schemaNames* list will be described.

You can switch to the DTD generator by setting following properties:

```
org.unitils.dbmaintainer.structure.DataSetStructureGenerator.implClass=
datasetStructureGenerator.dtd.filename=target/dtd/MyDatabase.dtd
```

This will generate a DTD at the specified location. You can then use this DTD in your data set by adding a doctype declaration pointing to the DTD file. As with XDS, the IDE will then offer you auto-completion and validation while you are editing the file.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "target/dtd/database.dtd">

<dataset>
    <user userName="jack" active="true" />
    <user userName="jim" active="false" />
</dataset>
```

An O/R mapping framework like Hibernate greatly simplifies the data access layer of an application. Even when using such a O/R mapping tool, unit tests remain valuable. They can ensure the mapping of the Hibernate mapped classes is always in sync with the database and can be used for easy and quick testing of HQL queries.

Testing a HQL method is done in the same way as described above in [Managing test data with DbUnit](#). The `@DataSet` annotation can be used for loading test data from a data set and the `ReflectionAssert` class or `@ExpectedDataSet` annotation can be used for checking the result.

Unitils hibernate support is based on spring's ORM abstraction layer. Spring is used under the hoods to configure a `SessionFactory` that connects to the unitils-configured database, and associate a hibernate `Session` with the current transaction. This doesn't mean you that need to use spring in your application code. The fact that spring is used, is completely transparent.

Loading Hibernate configurations

Unitils provides out-of-the box configuration support for Hibernate. You can obtain a `SessionFactory` that connects to the unit test database simply by annotating a class, field or method with the `@HibernateSessionFactory` annotation. The Hibernate configuration files that you want to have loaded can be passed as a parameter. If you annotate a field or method with this annotation, the session factory is also injected into this field or method.

Note that in order to function correctly, Hibernate tests must be run in a transaction. Unitils is configured by default to run in a transaction. Read the chapter on [Transactions](#) for more information on transaction support.

Under the hoods, hibernate uses an instance of `org.hibernate.cfg.Configuration` or `org.hibernate.cfg.AnnotationConfiguration` for loading configuration files and registering mapped classes, depending on whether you're using hibernate the classic way or whether you use hibernate with annotation. By default, Unitils uses `org.hibernate.cfg.AnnotationConfiguration`. If you're using mapping files only and you don't have the hibernate annotations extension in your classpath, you can change this by setting following property:

```
HibernateModule.configuration.implClassName=org.hibernate.cfg.Confi
```

```
public class UserDaoTest extends UnitilsJUnit4 {
    @HibernateSessionFactory({"hibernate.cfg.xml", "mapped-
classes.cfg.xml"})
    private SessionFactory sessionFactory;
}
```

This will create a Hibernate *Configuration* and load the configuration files *hibernate.cfg.xml* and *mapped-class.cfg.xml*. The *Configuration* instance is used to create a *SessionFactory* which is injected into the annotated field method of the test. Setter injection is also supported by annotating the setter method instead of the field.

Unitils also scans superclasses for `@HibernateSessionFactory` annotations. We advise to specify all configuration on a common superclass for all hibernate tests. E.g.:

```
@HibernateSessionFactory("hibernate.cfg.xml")
public class BaseDaoTest extends UnitilsJUnit4 {
}

public class UserDaoTest extends BaseDaoTest {
    @HibernateSessionFactory
```

```

    private SessionFactory sessionFactory;
}

```

If desired, a test subclass can override the configuration of the superclass.

```

@HibernateSessionFactory("hibernate.cfg.xml")
public class BaseDaoTest extends UnitilsJUnit4 {
}

@HibernateSessionFactory({"hibernate.cfg.xml", "user-mapped-classes.cfg.xml"})
public class UserDaoTest extends BaseDaoTest {

    @HibernateSessionFactory
    private SessionFactory sessionFactory;
}

```

This will create a new configuration for UserDaoTest class, first loading the *hibernate.cfg.xml* followed by the *user-mapped-classes.cfg.xml*. The configuration is then used to create a *SessionFactory* instance. Note that there is a performance implication. By adding this new configuration file in the subclass, a new session factory has to be created. In the previous example, no extra configuration is added and the session factory can be reused. *SessionFactory* creation is a heavy operation, so try to reuse them as much as possible.

Programmatic configuration

Performing programmatic configuration is also supported. You can use it by annotating a custom initialization method with *@HibernateSessionFactory*. A custom configuration method has 1 parameter of a *Configuration* (sub-)type and a void return type. It can be used to further configure the *Configuration* instance that was created by Unitils. A typical usage is to programmatically register mapped classes:

```

@HibernateSessionFactory
protected void initializeConfiguration(AnnotationConfiguration configuration) {
    configuration.addAnnotatedClass(User.class);
}

```

Hibernate Session management

The previous section showed you how to get a reference to a *SessionFactory* instance. To make your persistence layer code use this session factory, you still need to provide some project specific code. Typically this is implemented in a common superclass for all your persistence layer tests. An implementation of such a class could be:

```

public abstract class BaseDAOTest extends UnitilsJUnit4 {

    @HibernateSessionFactory
    private SessionFactory sessionFactory;

    @Before
    public void initializeDao() {
        BaseDAO dao = getDaoUnderTest();
        dao.setSessionFactory(sessionFactory);
    }

    protected abstract BaseDAO getDaoUnderTest();
}

```

Unitils manages all sessions created by these session factories. It will flush the session before checking the state of a database and close the session after a unit test was

finished.

Unitils also provides what we call a 'test-bound session context'. This means that, for the duration of a test, every call to *SessionFactory.getCurrentSession()* will return the same hibernate session.

Hibernate mapping test

Unitils is shipped with a unit test that verifies if the mapping of all your mapped classes is consistent with the database structure.

HibernateUnitils.assertMappingWithDatabaseConsistent() checks if any updates are required to the database to make it consistent with the Hibernate mapping. If yes, the test fails showing all DDL statements that should be issued to the database. This test is not automatically executed, you have to write a unit test yourself like follows:

```
@HibernateSessionFactory("hibernate.cfg.xml")
public class HibernateMappingTest extends UnitilsJUnit4 {

    @Test
    public void testMappingToDatabase() {
        HibernateUnitils.assertMappingWithDatabaseConsistent();
    }
}
```

An example error message could be:

```
Found mismatches between Java objects and database tables.
Applying following DDL statements to the database should resolve
the problem:
alter table PERSON add column lastName varchar(255);
alter table PRODUCT add column barCode varchar(255);
```

Testing with JPA

Unitils' JPA support has been built on top of spring's JPA abstraction layer on top. Therefore, it supports all JPA vendors that spring supports. The fact that spring is used under the hood, is completely transparent. It doesn't mean you that need to use spring in your application code.

To indicate which JPA vendor you are using, you need to set the property `jpa.persistenceProvider` in `unitils.properties` to one of the supported values: `hibernate` (default), `toplink` or `openjpa`.

```
jpa.persistenceProvider=toplink
```

Note that, if you use `toplink` as persistence provider, you need to run your tests with a JVM agent to enable load-time bytecode modification. We use the `spring-agent.jar` for this purpose. You enable the spring agent by adding following JVM option:

```
-javaagent:/path/to/spring-agent.jar
```

If your application uses spring IOC, you'll probably want to make unitils work with the *EntityManagerFactory* configured in spring. To read more about this, refer to [Testing with Spring](#). If you're not using spring, you can configure an *EntityManagerFactory* as follows:

```
public class UserDaoTest extends UnitilsJUnit4 {

    @JpaEntityManagerFactory(persistenceUnit = "pu")
```

```
EntityManagerFactory entityManagerFactory;
```

Unitils will initialize an *EntityManagerFactory*, load the persistence unit named 'pu', and inject it into the field annotated with *@JpaEntityManagerFactory*. By default it will look in jar files and class folders for a config file named *META-INF/persistence.xml*. This jar file or class folder is then automatically scanned for objects that are annotated as JPA entities.

It's also possible to use a different configuration file than *META-INF/persistence.xml*, by using the *configFile* attribute, e.g.:

```
public class UserDaoTest extends UnitilsJUnit4 {
    @JpaEntityManagerFactory(persistenceUnit = "pu", configFile =
    "persistence-test.xml")
    EntityManagerFactory entityManagerFactory;
```

Fields or setter methods of the test object can be annotated with *@javax.persistence.PersistenceUnit* and *@javax.persistence.PersistenceContext*. Unitils will inject an *EntityManagerFactory* or *EntityManager* on these annotated fields / setters, respectively.


Note that in order to function correctly, JPA tests must be run in a transaction. Unitils is configured by default to run in a transaction. Read the chapter on **Transactions** for more information on transaction support.

Unitils also supports using a custom configuration method. The method should be annotated with *@JpaEntityManagerFactory* and should have a single argument of type *org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean*. It will be called after loading the persistence.xml file.

You can create a test that verifies the mapping of the entities with the database. Note that this functionality only works when hibernate is used under the hoods as persistence provider. An implementation looks as follows:

```
@JpaEntityManagerFactory(persistenceUnit = "pu", configFile =
    "persistence-test.xml")
public class JpaMappingTest extends UnitilsJUnit4 {
    @Test
    public void testMappingToDatabase() {
        JpaUnitils.assertMappingWithDatabaseConsistent();
    }
}
```

➡ Testing with Spring

Unitils also offers features for unit testing when working with **Spring** . One of the basic principles of Spring is that your objects are designed in such way that they are easily testable without Spring or any other container. There are times however, when it can be useful to work with object structures that are wired by the Spring container.

The features Unitils offers for Spring are:

- Management of *ApplicationContext* configuration
- Injection of Spring beans in unit tests
- Make use of a Hibernate *SessionFactory* configured in Spring
- Reference the Unitils *DataSource* in Spring configuration

ApplicationContext configuration

Loading an application context can easily be achieved by annotating a class, method or

field with a `@SpringApplicationContext` annotation, and specifying the spring config files as attribute. The resulting application context is injected into fields or methods annotated with `@SpringApplicationContext`. For example:

```
public class UserServiceTest extends UnitilsJUnit4 {
    @SpringApplicationContext({"spring-config.xml", "spring-
test-config.xml"})
    private ApplicationContext applicationContext;
}
```

This will create a new *ApplicationContext* loading the config files *spring-config.xml* and *spring-test-config.xml* and will inject it into the annotated field. Setter injection is also supported by annotating the setter method instead of the field.

Superclasses are also scanned for the presence of `@SpringApplicationContext` annotations. If found, these configuration files are loaded before the configuration files specified in subclasses. This makes it possible to override configuration settings or to add extra configuration specific for the test. For example:

```
@SpringApplicationContext("spring-beans.xml")
public class BaseServiceTest extends UnitilsJUnit4 {
}

public class UserServiceTest extends BaseServiceTest {
    @SpringApplicationContext("extra-spring-beans.xml")
    private ApplicationContext applicationContext;
}
```

As in the previous example, this will create a new *ApplicationContext*, first loading the *spring-beans.xml* followed by the *extra-spring-beans.xml* configuration files. The application context is then injected into the annotated field.

Note that in this example, a new application context was created. This was needed because an extra configuration file was specified for the test. Unitils will try to reuse the application context whenever possible. In the following example, no extra files need to be loaded, so the same instance will be reused:

```
@SpringApplicationContext("spring-beans.xml")
public class BaseServiceTest extends UnitilsJUnit4 {
}

public class UserServiceTest extends BaseServiceTest {
    @SpringApplicationContext
    private ApplicationContext applicationContext;
}

public class UserGroupServiceTest extends BaseServiceTest {
    @SpringApplicationContext
    private ApplicationContext applicationContext;
}
```

By specifying the configuration on a common superclass *BaseServiceTest*, the *ApplicationContext* will only be created once and then reused for the *UserServiceTest* and *UserGroupServiceTest*. Since loading an application context can be a heavy operation, reusing the context will greatly improve the performance of your tests.

Programmatic configuration

Programmatic configuration is also possible. You can do this by annotating a method that takes no or a *List<String>* as parameter and returns an instance of *ConfigurableApplicationContext*. If a *List<String>* parameter is specified, all locations of

the *@SpringApplicationContext* annotations that would otherwise be used to create a new instance will be passed to the method.

The result of this method should be an instance of an application context for which the *refresh()* method was not yet invoked. This is important, since it allows Unitils to perform extra configuration such as adding some *BeanPostProcessors*, which is no longer possible once *refresh* is invoked. In case of a *ClassPathXmlApplicationContext* this can easily be achieved by passing *false* as value for the *refresh* parameter of the constructor:

```
@SpringApplicationContext
public ConfigurableApplicationContext
createApplicationContext(List<String> locations) {
    return new
    ClassPathXmlApplicationContext(Arrays.asList(locations), false);
}

@SpringApplicationContext
public ConfigurableApplicationContext createApplicationContext()
{
    return new
    ClassPathXmlApplicationContext(Arrays.asList("spring-
    beans.xml", "extra-spring-beans.xml"), false);
}
```

Injection of Spring beans

Once you've configured an *ApplicationContext*, Spring beans are injected into fields / setters annotated with *@SpringBean*, *@SpringBeanByType* or *@SpringBeanByName*. Following example shows 3 ways to get a *UserService* bean instance from the application context:

```
@SpringBean("userService")
private UserService userService;

@SpringBeanByName
private UserService userService;

@SpringBeanByType
private UserService userService;
```

With *@SpringBean*, you can get a Spring bean from the application context by explicitly specifying the name of the bean. *@SpringBeanByName*, has the same effect, but now the name of the field is used to identify the bean.

When using *@SpringBeanByType*, the application context is queried for a bean having a type assignable to the field. In this case, this is a bean of type *UserService* or one of its sub-types. If no such bean exists or if there is more than one possible candidate, an exception is thrown.

The same annotations can be used on setter methods. For example:

```
@SpringBeanByType
public void setUserService(UserService userService) {
    this.userService = userService;
}
```

Connecting to the test database

In the [Database testing](#) section we explained you how to obtain and install a datasource for your test. The [Testing with Hibernate](#) section describes how to setup

hibernate for your test. In Spring all this wiring is typically done in a Spring config file. Suppose, for example, that your application contains following *application-config.xml* file:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource"

    <property name="driverClassName"
value="org.hsqldb.jdbcDriver"/>
    <property name="url"
value="jdbc:hsqldb:hsqldb://localhost/sample"/>
    <property name="username" value="sa"/>
    <property name="password" value="" />
</bean>

<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSess

    <property name="dataSource" ref="dataSource"/>
    <property name="annotatedClasses">
        <list>
            <value>org.unitils.sample.User</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=org.hibernate.dialect.HSQLDialect
            hibernate.show_sql=true
        </value>
    </property>
</bean>
```

The first bean defines the data source that connects to the application database. The second bean configures a hibernate *SessionFactory* that connects with this *DataSource*.

In our tests we need a connection to the test database. Typically, a test-specific spring configuration file is written with for example following contents:

```
<bean id="dataSource"
class="org.unitils.database.UnitilsDataSourceFactoryBean" />
```

With this configuration, Spring will make use of the *DataSource* configured in Unitils. Refer to [the configuration chapter](#) for more information on how to configure a test *DataSource*.

Unitils automatically detects the *SessionFactory* configured in Spring: all integration features Unitils offers for Hibernate also work when using Spring. The [hibernate mapping test](#) will therefore also work as follows:

```
@SpringApplicationContext({"application-config.xml", "test-ds-
config.xml"})
public class HibernateMappingTest extends UnitilsJUnit4 {

    @Test
    public void testMappingToDatabase() {
        HibernateUnitils.assertMappingWithDatabaseConsistent();
    }
}
```

➡ Testing with mock objects

Unit tests should test code in isolation. Mock objects enable you to test a piece of code without having to care about other objects or services that it depends on. Since version 2.0, unitils provides a complete mock object solution to dynamically create mock objects. Support is also offered for mock creation and injection.

Before version 2.0, mock objects support was built upon EasyMock. This functionality will

remain available. Refer to the chapter on [EasyMock support](#) for more info.

You may wonder why we've written a complete mock module given the abundance of dynamic mock object libraries already available. The reason is that in our opinion, a lot of things can be improved in the user friendliness of existing libraries. The features that make the difference are:

- Mock behavior definition is performed before invoking the method under test. After the tested method invocation, assert statements are used to verify expected method invocations.
- A concise, easy to use and consistent syntax.
- Concrete objects, argument matchers and the null reference (meaning I-don't-care) can be mixed in behavior definitions and assert statements, whereas in other mock libraries you must use either concrete objects or argument matchers, but you cannot mix them within the same method call.
- By default, value semantics is used by default to compare expected and actual arguments: You don't need to worry about the contents of objects being changed in between the method call and the call in which you perform an invocation assert statement: a copy of the original object is used for comparison.
- The best possible feedback is provided in logs and in error messages: The observed scenario, which lists all method calls performed on the mocks in order, with links to the source code, the contents of all objects involved, and suggested assert statements that can be used as a basis for your own asserts.

Mock test example

Following example is a test for an alert service: the *sendScheduledAlerts()* method requests all scheduled alerts from an *AlertSchedulerService*, and sends them using a *MessageSenderService*.

```
public class AlertServiceTest extends UnitilsJUnit4 {

    AlertService alertService;
    Message alert1, alert2;
    List<Message> alerts;

    Mock<SchedulerService> mockSchedulerService;
    Mock<MessageService> mockMessageService;

    @Before
    public void init() {
        alertService = new
AlertService(mockSchedulerService.getMock(),
mockMessageService.getMock());
        alert1 = new Alert(...); alert2 = new Alert(...);
        alerts = Arrays.asList(alert1, alert2);
    }

    @Test
    public void testSendScheduledAlerts() {

mockSchedulerService.returns(alerts).getScheduledAlerts(null);

        alertService.sendScheduledAlerts();

        mockMessageService.assertInvoked().sendMessage(alert1);
        mockMessageService.assertInvoked().sendMessage(alert2);
    }
}
```

The example test makes use of a mock *SchedulerService* and *MessageService*. The first statement in this test specifies that subsequent invocations of *getScheduledAlerts* on the mock *SchedulerService* with any parameter (*null* means *any argument* in a behavior definition statement), will return a *List* containing *alert1* and *alert2*. The subsequent statement invokes the method under test. After this, assert statements are used to verify that *sendAlert* was called on the *mockMessageService*, with *alert1* and then *alert2* as argument.

Instantiation of mock objects

Mock objects are wrapped inside a control object, which you can use to define behavior and invoke assert statements. Simply declare the mock as a field of your test, you don't have to instantiate it yourself.

```
Mock<MyService> mockService;
```

Unitils creates the mock control object and assigns it to this field before each test. To obtain the mock object itself, simply call the `getMock()` method on the control object:

```
MyService myService = mockService.getMock();
```

Define mock behavior

Unitils offers a simple and concise syntax for defining mock behavior. To make the `getScheduledAlerts` method return *alerts* if called with *myUser* as argument, simply write:

```
mockSchedulerService.returns(alerts).getScheduledAlerts(myUser);
```

Or to make it throw an exception:

```
mockSchedulerService.raises(new
    BackEndNotAvailableException()).getScheduledAlerts(myUser);
```

You may also simply specify the exception class, an instance is created for you. Note that the exception class doesn't need to offer an empty constructor since under the hoods, Obgenesis is used to create the instance.

```
mockSchedulerService.raises(BackEndNotAvailableException.class).get
```

Or you can specify custom behavior like in this example:

```
mockSchedulerService.performs(new MockBehavior() {
    public Object execute(ProxyInvocation mockInvocation) {
        // ... (retrieve alerts logic)
        return alerts;
    }
});
```

If the same method must behave differently during subsequent calls, you can make sure each behavior definition gets only matched once, by using the methods *onceReturns*, *onceRaises* or *oncePerforms*. E.g.

```
mockSchedulerService.onceReturns(alerts).getScheduledAlerts(myUser)
mockSchedulerService.onceRaises(new
    BackEndNotAvailableException()).getScheduledAlerts(myUser);
```

If you would use *returns* and *raises* instead of *onceReturns* and *onceRaises*, the second behavior definition could never be matched: in this case, the first matching behavior definition always wins.

For sake of maintainability, we advise to avoid the usage of the *once* syntax, since

making assumptions about the order of method calls makes your tests brittle. If possible, try to exploit the fact that subsequent calls to the same method use different parameter values.

Verify expected invocations

After the execution of the tested method, you usually want to check that some expected method calls were performed on the mock objects. For example:

```
mockMessageService.assertInvoked().sendMessage(alert1);
```

This verifies that *sendMessage* was called on the mock *MessageService* with *alert1* as argument. Note that each invocation can only be verified once: if you repeat the same assert statement twice, unitils checks that the method call in question was called two times.

Note that unitils doesn't react on 'unexpected' invocations by default. You can explicitly prohibit method calls using *assertNotInvoked*. E.g. we can explicitly verify that alert number 3 wasn't sent like this:

```
mockMessageService.assertNotInvoked().sendMessage(alert3);
```

To make sure that your mocks didn't receive any other method calls, except from the ones you've stubbed or verified, you can use following static method call:

```
MockUnitils.assertNoMoreInvocations();
```

If you want to prohibit unexpected method calls in general, you can call *assertNoMoreInvocations* in an *@After* method, or in the *@After* method of you base test class.

By default, the order of the calls is not verified. If you do want to verify the order, you can use *assertInvokedInSequence*. For example we can make sure *alert1* was sent before *alert2* as follows:

```
mockMessageService.assertInvokedInSequence().sendMessage(alert1);  
mockMessageService.assertInvokedInSequence().sendMessage(alert2);
```

Argument matching

To improve maintainability and to simplify tests, it's better not to impose strong expectations on the values of arguments. Unitils offers a very simple way to ignore the value of an argument: simply pass the *null* reference. E.g. to ignore the *user* argument value in the *getScheduledAlerts* method call, we write:

```
mockSchedulerService.returns(alerts).getScheduledAlerts(null);
```

Note that, if you do pass an object reference, the expected and actual object are compared using *lenient reflection comparison*: the fields are compared using reflection, fields which are *null*, 0 or false in the expected object are ignored and the order of collections is ignored (refer to [Assertion utilities](#) for more information about reflection comparison).

If you need yet another way to match arguments, you can use an argument matcher instead. A set of argument matchers is provided in *org.unitils.mock.ArgumentMatchers*,

which contains a bunch of static methods that you'd typically import by means of a static import:

```
import static org.unitils.mock.ArgumentMatchers.*;
```

Typical use cases for argument matchers are the following:

```
mockSchedulerService.returns(alerts).getScheduledAlerts(notNull(User)
// Matches with any not-null object of type User
mockSchedulerService.returns(alerts).getScheduledAlerts(isNull(User)
// The argument must be null
mockMessageService.assertInvoked().sendMessage(same(alert1)); //
// The argument must refer to alert1 instance
```

Other argument matchers are *eq*, *refEq* and *lenEq*. When using *eq*, the *equals* method is called to see if the actual argument matches the expected one. With *refEq*, strict reflection comparison is used. *lenEq* is the default argument matcher which is also implicitly used when directly passing an argument: reflection argument matching will ignore java defaults (null, 0, false) and the order of collections.

Value / reference semantics

When passing arguments directly or using the *lenEq* / *refEq* argument matchers, a copy of the original objects is used for comparison. This way your checks are not affected by changes to values inside the objects. For example: suppose that after the alert has been sent, the *successfullySent* property is set to true and the alert is archived. We can verify that this property was correctly set as follows:

```
alert1.setSuccessfullySent(true);
mockAlertRepository.assertInvoked().archiveAlert(alert1);
```

If no copy would be taken, the assert statement would not fail if the *successfullySent* property wasn't set, because the *alert1* object would be compared with itself.

When using the *same()* or *eq()* argument matcher, the original objects are used for comparison. Following piece of code is therefore not correct:

```
alert1.setSuccessfullySent(true);
// Useless assignment
mockAlertRepository.assertInvoked().archiveAlert(eq(alert1)); //
```

Even if the *equals()* method of *Alert* would take into account the *successfullySent* property, this piece of code doesn't make any sense since the property is set in both the expected and actual object.

Note that when defining behavior, the same principle is used: changes to the arguments after the behavior definition will not change the semantics.

Custom argument matchers

Unitils currently only offers 6 argument matchers, which is a lot less than competing libraries. This is intentional: We've experienced that most of them are rarely or never used. If you do have specific needs, it's easy to implement a custom argument matcher. The implementation of a *lessThan(...)* argument matcher could be as follows:

```
public class CustomArgumentMatchers {

    @ArgumentMatcher
    public static int lessThan(Integer lessThan) {
```

```

ArgumentMatcherRepository.getMock().registerArgumentMatcher(new
LessThanArgumentMatcher(lessThan));
    return 0;
}
}

public class LessThanArgumentMatcher implements ArgumentMatcher
{
    private Integer lessThan;

    public LessThanArgumentMatcher(Integer lessThan) {
        this.lessThan = lessThan;
    }

    public boolean matches(Object argument, Object
argumentAtInvocationTime) {
        Integer argumentAsInt = (Integer) argument;
        return lessThan.compareTo(argumentAsInt) < 0;
    }
}

```

The *matches* method of *ArgumentMatcher* is passed both the original object and a copy of the object taken at invocation time. Which one you use depends on whether you want to implement reference or value semantics.

Dummy objects

In tests you often need an instance of a domain entity or value object, where it's actually not important to have it populated with test data. For example in the *AlertServiceTest* listed above, we need 2 *alert* instances. In the method under test, the alerts are retrieved from the *SchedulerService* and passed on the *MessageService*. Since no methods are called on the alert instances, the contents of the object are not important. However, the constructor typically forces you to pass in all required parameters, and the constructor may check that these parameters are not-null.

Although you could create a mock object and only store the proxy instance, you actually simply need a dummy instance. For this you can use dummy objects. You can create a dummy object by calling *MockUnitils.createDummy*, or by annotating a field with the *@Dummy* annotation:

```

@Dummy
Message alert1, alert2;

```

Scenario and suggested assert statements reports

When an invocation assert statement fails, a detailed report is added to the error message containing the following sections:

- Scenario overview: An overview of all the invocations performed on the mock objects, with links to where the invocations occurred in the source code.
- Suggested asserts: A set of invocation assert statements, which you can use as a basis for the assert statements of your test.
- Detailed scenario: All the invocations performed on the mock objects, the contents of the arguments and return values at invocation time, with a link to where the invocation occurred and where the behavior of the call was defined.

You can also have these reports logged after each test, even if the test passes. Simply set one or more of the following properties to true in *unitils.properties* or *unitils-local.properties*:

- *mockModule.logObservedScenario*
- *mockModule.logDetailedObservedScenario*
- *mockModule.logSuggestedAsserts*

- `mockModule.logFullScenarioReport`

Partial mocks

It may sometimes be desirable to obtain a mock object while keeping the original implementation, in order to mock only some method calls or simply to monitor the behavior of the object without actually mocking it. You can use partial mocks for this purpose. A partial mock can be obtained either by calling `MockUnitils.createPartialMock()`, or by declaring it as a field using the `PartialMock` interface:


```
PartialMock<MyService> mockService;
```

Mock injection

Unitils provides services that simplify mock injection in various ways. Following example shows how a `UserDao` mock is created and injected in the `userService`:

```
@InjectInto(property="userDao")
private Mock<UserDao> mockUserDao;

@TestedObject
private UserService userService;
```

Before executing a test method in the above example, but after the test setup, the `@Inject` annotation will cause the `mockUserDao` to be injected into the `userDao` property of the `userService`. The value for the attribute `property` of `@Inject` can be an arbitrary **OGNL**  expression. Getter, setter and field access may be mixed and private access is also supported.

The target object for injection is by default the field that is annotated with `@TestedObject`. If multiple fields are annotated with `@TestedObject` the object will be injected in each of them. A new instance will automatically be created if the tested object does not exist yet. If injection is not possible, for example when a property in the OGNL expression does not exist or the tested object could not be created, a `UnitilsException` will be raised and the test will fail.

If needed, the target object can also be specified explicitly, using the `target` attribute on the `Inject` annotation:

```
@InjectInto(target="userService", property="userDao")
private Mock<UserDao> mockUserDao;

private UserService userService;
```

The previous example showed how you can explicitly inject mocks by specifying the name of the target property. Unitils also supports injecting objects into the target automatically by type. E.g.

```
@InjectIntoByType
private Mock<UserDao> mockUserDao;

@TestedObject
private UserService userService;
```

The `mockUserDao` will be injected into a property of `userService` with the type `UserDao` or a superclass or interface of `UserDao`. If more than one candidate target property exists, the most specific one is chosen. If no single most specific target property can be found, a `UnitilsException` is thrown and the test fails.

Static injection

There are variants of the `@InjectInto` and `@InjectIntoByType` annotations that can be used for injection into static fields or setter methods. These are called `@InjectIntoStatic` and `@InjectIntoStaticByType`, respectively. A common use of these annotations is to inject a mock into a singleton class. For example:

```
@InjectIntoStatic(target=UserService.class property="singleton")
private Mock<UserDao> mockUserDao;
```

This will inject the created mock into the static *singleton* field of the *UserService* class.

If this would be the only action performed, this test would leave the *UserService* in an invalid state. Other tests that access the same *singleton* field will now also get the mock instead of the real user service. To resolve this, Unitils will restore the old value of the field after the test was performed. In the above example, the *UserService* instance or null value contained in the *singleton* field will be put back after the test execution.

The actual restore action can be configured by specifying it in the annotation. You can choose to restore the old value (=default), set the field to a null or 0 value or to leave the value as it is.

```
@InjectIntoStaticByType(target=UserService.class
restore=Restore.NULL_OR_0_VALUE)
private Mock<UserDao> mockUserDao;
```

The restore action can also be set project-wide, for all mocks at once, by changing the defaults in the configuration settings:

```
InjectModule.InjectIntoStatic.Restore.default=old_value
```

Mock injection when using a Service Locator

The above way of injecting mocks is very powerful and easy, but sometimes applications are not coded in such a way that this is possible. It could be for example, that an application uses a service locator instead of dependency injection.

To help with this, an `@AfterCreateMock` annotation is provided that allows intercepting a mock immediately after its creation. The annotated method takes as parameters an *Object* (the mock), a *String* (the name of the field to which the mock is assigned) and a *Class* (the type of the mock object). The mock instance can then be used to register the mock in the service locator or perform some other task of configuration. Such a method could for example be implemented as follows:

```
@AfterCreateMock
void injectMock(Object mock, String name, Class type) {
    ServiceLocator.injectService(type, mock);
}
```

The *injectService* method installs the instance, so that the mock instance is returned when the *ServiceLocator* is asked for an instance of that type. This allows for a very simple way of using mock objects in combination with such a service locator.

➔ EasyMock support

Unitils also offers support for tests that use **EasyMock** . It provides convenience

functionality that reduces the plumbing: This involves simplification of mock creation, argument matching and injection of mocks.

Mock creation

Mocks can automatically be created by annotating a field with the *@Mock* annotation, Unitils will create a mock of the same type as the field and inject the instance into that field. This mock creation and assignment is performed before the setup of your test. During the setup you can then do extra configuration and, for example, install the mock so that it is used by your code during the test. The previous section already showed you how Unitils can help you perform this mock injection more easily.

Following example shows a unit test for a method on a *UserService* that disables all user accounts that haven't been active for a certain time:

```
public class UserServiceTest extends UnitilsJUnit4 {

    @Mock
    private UserDao mockUserDao;

    private UserService userService;

    @Before
    public void setUp() {
        userService = new UserService();
        userService.setUserDao(mockUserDao);
    }

    @Test
    testDisableInactiveAccounts() {

        expect(mockUserDao.getAccountsNotAccessedAfter(null)).andReturn(acc

                mockUserDao.disableAccount(accounts.get(0));
                mockUserDao.disableAccount(accounts.get(1));
                EasyMockUnitils.replay();

                userService.disableInactiveAccounts();

    }
}
```

In this example the *UserDao* of the user service is replaced by a mock object. This mock is automatically created by Unitils and then installed in the user service during the setup of the test. During the test we then first record the expected behavior and call *EasyMockUnitils.replay()* which will call *replay* on all mock objects, in this case only *mockUserDao*. Then the actual test is performed. After the test, Unitils will automatically invoke *EasyMockUnitils.verify()* which will call *verify* on all mock objects to check the expected behavior.

The created mock objects by default use EasyMock's strict call expectations (i.e. the test fails when unexpected method calls occur) and ignore the invocation order. You can change these settings by specifying attributes on the *@Mock* annotation, e.g.:

```
@Mock(returns=Calls.LENIENT,
      invocationOrder=InvocationOrder.STRICT)
private UserDao mockUserDao;
```

You can also change the default values for all *@Mock* annotations by changing the values in the configuration settings:

```
EasyMockModule.Mock.Calls.default=lenient
EasyMockModule.Mock.InvocationOrder.default=strict
```


Reflection lenient argument matching

The mocks that are supplied by Unitils are slightly different from mocks objects that you get when directly using EasyMock: a *LenientMocksControl* is used for these mocks. This control will make sure that arguments specified in method calls are matched using reflection. The arguments of the expected and actual method calls are compared in the same way we saw in the [Assertion utilities](#) section. By default, ignore defaults and lenient order leniency levels are used. E.g., following expectation and actual method calls will match:

```
expected: dao.findById(0);
actual:   dao.findById(99999);

List<Integer> userIds = new ArrayList<Integer>();
userIds.add(3);
userIds.add(2);
userIds.add(1);
expected: dao.deleteById(Arrays.asList(1,2,3));
actual:   dao.deleteById(userIds);

expected: dao.update(0,    new User(null, "Doe"));
actual:   dao.update(9999, new User("John", "Doe"));
```

As you can see, the leniency levels not only apply to objects and their fields but also apply to the arguments themselves. For example, take a look at following expectation:

```
expect(mockUserDao.getAccountsNotAccessedAfter(null)).andReturn(acc
```

The *null* parameter in this method call actually means that we don't care what argument is passed to this method. This provides a very convenient way of setting flexible expectations! There is no need to specify *anyInt*, *notNull* ... argument matchers for each of the arguments anymore.

By default, lenient order and ignore defaults are used as leniency levels. If you want to change this, you can specify which levels to use by setting attributes on the *@Mock* annotation:

```
@Mock(order=Order.STRICT, defaults=Defaults.STRICT,
      dates=Dates.LENIENT)
private UserDao mockUserDao;
```

The levels can also be set project-wide, for all mocks at once, by changing the defaults in the configuration settings. E.g. to do the same as above for the entire project you could set following properties:

```
EasyMockModule.Mock.Order.default=strict
EasyMockModule.Mock.Dates.default=lenient
EasyMockModule.Mock.Defaults.default=strict
```

If you don't want to use reflection for lenient argument matching, you can also make Unitils inject regular EasyMock mocks into your test, using the *@RegularMock* annotation.

```
@RegularMock
private UserDao mockUserDao;
```