Abstract

Legacy systems with a growing code base has become an increasing burden in the IT industry the last couple of years. Often these systems has grown to become colossal systems with an unclear architecture and messy interfaces that are difficult to use. At the same time there is an ever-increasing need to integrate more and more IT systems in companies to meet new demands are increasing. As a consequence, exposing functionality from legacy systems as services is becoming common. This is of course a good thing since clients then can more easily integrate with these legacy systems. However, one should be careful so that one does not just build yet another layer on top of an already existing pile of rotten code.

The challenge

The figure shows a simplified view of the architecture for the system that we will look into for the rest of this paper. Our task is to create services on top of an already existing core layer. This layer has grown into a huge unmaintainable system over many years and has a messy and unclear interface, in short it has all the bad signs of a legacy system which pretty soon will be a bottleneck for the organization. The core API consists almost solely of concrete classes and instantiation of some of these classes results in an initialization of nearly all the different systems in the chain. Because of this inherent complexity, unit testing of the methods in the service layer is extremely difficult.

In an ideal world with unlimited resources and no deadlines, we would try to fix the real problem, which is the core layer. In a real world however, we know that this isn't always possible. Refactoring of a huge business critical system with a test coverage near zero, would be both extremely time consuming and also very risky. So what are our options? One option is to use the concrete classes of the core api in our services. Using this approach requires a minimum of effort when it comes to thinking, you just use the api as it was intended to be used. However, there are several drawbacks:

1. There will be a tight coupling between your new service layer and the core api making it very difficult to substitute the core with another api in the future
2. Unit testing the code in the service layer will be as difficult as testing the Core API because of this tight coupling and
3. Because the service layer is going to be difficult to test, it is very likely that you'll end up skipping the test part and end up with crappy code that is impossible to refactor and therefore is extremely resistant to change.

What we have done then is that we have just built yet another layer of unmaintainable code which pretty soon will add to the already growing maintenance burden.
Luckily there is a way out of this bad cycle. In the following we will see how we can improve both the design and the testability of the service layer by using sound techniques such as dependency injection, mock objects and a well-known pattern known as the adapter pattern. Before proposing a solution we will take a quick look at the named techniques.

**Dependency injection**
Dependency injection, also known as inversion of control, is a well-known design principle which is widely used to achieve loose coupling and we'll see that this very important principle is at the heart of our solution:

«Dependency injection is a way to achieve loose coupling. The technique results in highly testable objects, particularly when applying test-driven development using mock objects: Avoiding dependencies on the implementations of collaborating classes (by depending only on interfaces that those classes adhere to) makes it possible to produce controlled unit tests that focus on exercising the behavior of, and only of, the class under test. To achieve this, dependency injection is used to cause instances of the class under test to interact with mock collaborating objects, whereas, in production, dependency injection is used to set up associations with bona fide collaborating objects.» [1]

There are quite a few excellent articles on this principle, some of them are listed at the end of this article.

**Mock objects**
In order to be a true unit test, the piece of code under test needs to be tested in isolation. One way of accomplishing this is by using mock objects:

«A mock object is a an object created to stand in for an object that your code will be collaborating with.»[2]

Obviously this gives us some very nice advantages as opposed to the «normal» way of doing unit tests:

1. No need to for set up of complex environment
2. Tests are functional driven, not data driven, that is: You don't need to worry about digging up test data for your unit tests
3. Produces better tests and better code
4. Easy to simulate error situations

**Static vs. Dynamic mock objects**
«Mock tools can normally be categorized into two types, dynamic or static. Static mock tools generate mock implementations of the dependencies as extra classes. These are compiled together with the source ensuring type safety check. Dynamic mock tools generate the mock implementations at runtime without adding any additional classes. Dynamic mocking tools are normally more popular and preferred to static mocking tools as they are much easier to use and do not create any extra classes. However dynamic mocks are not type safe as compared to static mocks»[3]

Although using static mock objects definitely is an option, I believe dynamic mocks are superior for the purpose of unit testing. The reason is that the mock objects then live inside the tests and this forces you to keep the mocks up to date. Static mocks on the other hand has a tendency to get out-dated. However, if you are planning to use mock objects to act like placeholder for real code in a broader scale, static mocks can be a useful option.
**EasyMock**

The examples in this paper are based on mock objects created with the EasyMock framework (www.easymock.org). It is quite easy to set up and use, and has an easy to read documentation. The framework allows you to mock both interfaces and concrete classes, and although the latter has a few limitations, you are at least able to mock your dependencies if they are concrete classes.

From easymock.org:

«EasyMock provides Mock Objects for interfaces in JUnit tests by generating them on the fly using Java's proxy mechanism. Due to EasyMock's unique style of recording expectations, most refactorings will not affect the Mock Objects. So EasyMock is a perfect fit for Test-Driven Development.» [4]

The mock objects simulate parts of the behaviour of domain code, and are able to check whether they are used as defined[4]. The behaviour of the mock is specified by setting expectations on the mock (record state), we then activate the mock through the MockControl object (replay state) and finally verify that the mock behaved as we expected it to (verify state).

A detailed description of the mocking framework is beyond the scope of this paper. Later, before exploring how to use mock objects in unit tests, a brief description of the main concept of the framework is given. For a more thorough description please refer to the EasyMock web page[6]

For those that are lucky enough to work with Java 1.5, the latest version of the framework is recommended. Since I currently work with Java 1.4, the examples in this paper is based on EasyMock version 1.2.

---

**The Adapter pattern**

The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program. The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface [5]

In the next section we will see how we can use this pattern to improve the testability of the solution.

---

*Drawing 2: The adapter pattern*
Solution

As we have seen earlier in this paper, a mock is a stand in for a real object that our class under test collaborates with. If we can replace the Core with mocks we have at least solved the «external dependency problem». Unfortunately, mock objects is best used if the objects to mock are interfaces and that is not the case in our situation. For example, final methods cannot be mocked. So creating mock object to stand in for the core is not a very good solution either. As it turns out a new layer in our architecture may prove helpful and this is where the adapter pattern comes into play.

Figure 3 shows the architecture after introducing an adapter layer. The role of this layer is to adapt to the core layer through properly defined interfaces. So instead of the service layer interacting with the core layer, it will instead interact with the adapter layer which is now responsible for all interactions with the core. This also gives a nice separation of concerns in our architecture.

Now, how does this help us with the test part? Since the service layer now only interacts with interfaces instead of concrete classes, we can easily introduce mock objects for the adapter layer. There is one big gotcha when doing this: If we start coding the business logic into the adapters we are on a dangerous road to just another layer of indirection. It is extremely important that the adapter layer is kept as thin as possible, its only responsibility should be to adapt to the layer below, nothing else.

A brief introduction to EasyMock

We will soon explore a concrete example on how to create and unit test a service on this proposed architecture. But first we need to give a brief description of the most important mechanisms of the EasyMock framework.

MockControl

The MockControl object controls the behaviour of its associated mock object[6]. You use this object to create your dynamic mock object, when specifying behaviour and also in the replay and verify state. Suppose that we are to create a mock object for an interface called Collaborator. We can then create a MockControl object like this

```java
MockControl collaboratorControl = MockControl.createControl(Collaborator.class);
```

which we then can use to create a mock object for the interface that our class under test has a dependency to

```java
Collaborator collaborator = (Collaborator)collaboratorControl.getMock();
```
This is it, we have created a mock object and we are now ready to specify behaviour for our mock.

**Specifying behaviour**

Assume that our Collaborator has a getSomeNumber() method that returns an int value and that we call this method from our class under test. To mock this dependency all we need to add is the following two lines of code:

```java
collaborator.getSomeNumber();
collaboratorControl.setReturnValue(10);
```

expectAndReturn is a convenience method on the MockControl for specifying the same behaviour as above:

```java
collaboratorControl.expectAndReturn(collaborator.getSomeNumber(), 10);
```

Either way, what we say here is that we expect one call to the getSomeNumber() of the Collaborator interface and that the return value from this call is 10.

**Replay and verify state**

After we have specified the behaviour we are ready to replay the behaviour we specified for our mock object in the previous step

```java
collaboratorControl.replay();
```

Since we have specified behaviour we also need to check whether the mock object was used the way we specified. It is as simple as this:

```java
collaboratorControl.verify();
```

If the mock does not behave as we expected, the verify command will fail the test.

There is of course a lot more to EasyMock than our short introduction reveals, amongst others the possibility to mock exceptions from dependant objects. For a more thorough description please refer to the EasyMock website.
**Code example - Unit testing the AccountService**

Our task is to create and expose a service for creating an account, we will call the service createAccount. Since our focus is on the mocking part we don’t want to clutter the example with loads of domain details and in that respect the createAccount service is a good example to use since it is not particularly complex in our context. In fact, this service only requires us to call the core layer once.

The diagram above shows the class diagram for the classes in our design. In the following we will list the most relevant interfaces and classes in the solution.

The interface of the createAccount service looks like this:

```java
public interface AccountService {
    public String createAccount(Account account) throws BusinessException, SystemException;
}
```

*Listing 1: Interface for the AccountService*

Here is the class implementing this simple interface:

```java
public class AccountServiceImpl implements AccountService {
    /**
     * Dependency to BillingAccountAdapter
     */
    private BillingAccountAdapter billingAccountAdapter;

    /**
     * Dependency to AccountAssembler responsible for assembling DTO object
     */
    private AccountAssembler accountAssembler;
}
```

Unit testing services using mock objects and the adapter pattern
public void setBillingAccountAdapter(BillingAccountAdapter billingAccountAdapter) {
    this.billingAccountAdapter = billingAccountAdapter;
}

public void setAccountAssembler(AccountAssembler accountAssembler) {
    this.accountAssembler = accountAssembler;
}

public String createAccount(Account account) throws BusinessException, SystemException{
    BillingAccountDTO billingAccountDTO = accountAssembler.toBillingAccountDTO(account);
    return billingAccountAdapter.createBillingAccount(billingAccountDTO);
}

Listing 2: Implementation class for AccountService

To complement the picture we will also list the interfaces for the adapter and the assembler.

public interface BillingAccountAdapter {
    public String createBillingAccount(BillingAccountDTO dto)
        throws BusinessException, SystemException;
}

Listing 3: Interface for the BillingAccountAdapter

public interface AccountAssembler {
    public BillingAccountDTO toBillingAccountDTO(Account account) throws SystemException;
}

Listing 4: Interface for the AccountAssembler

The implementation classes for these interfaces are not shown since the implementation is really not important for our example. What's important is that we know that our class under test is dependent on these two external objects.

Now, let's move on to the test class for this service. It is always important to have a clear picture of what to test, in our case the class under test is the AccountServiceImpl class. Earlier in this paper we emphasized that a unit test should be a test of class in isolation. To achieve this we must do something about the two external dependencies (the adapter and the assembler). One way of doing this is to introduce mocks as placeholders for these dependencies.
Luckily, since we have created our service so that we support the dependency injection principle, doing this is fairly easy.

**Testing the happy case for the createAccount service**

Listing 5: Happy test for createAccount service

```java
import org.easymock.MockControl;
import junit.framework.TestCase;

public class AccountServiceTest extends TestCase {

    /**
     * Tests the createAccount service (happy case)
     *
     * @throws Exception If the test fails in an unexpected manner
     */
    public void testCreateAccount() throws Exception{
        BillingAccountDTO dto = new BillingAccountDTO();//1
        Account account = new Account();//2
        MockControl assemblerControl = MockControl.createControl(AccountAssembler.class);//3
        AccountAssembler assembler = (AccountAssembler)assemblerControl.getMock();//4
        assembler.toBillingAccountDTO(account);//5
        assemblerControl.setReturnValue(dto);//6
        assemblerControl.replay();//7

        MockControl adapterControl = MockControl.createControl(BillingAccountAdapter.class); // 8
        BillingAccountAdapter adapter = (BillingAccountAdapter)adapterControl.getMock(); //9
        adapter.createBillingAccount(dto);//10
        adapterControl.setReturnValue("123456789"); //11
        adapterControl.replay(); //12

        AccountServiceImpl classUnderTest = new AccountServiceImpl();//13
        classUnderTest.setAccountAssembler(assembler); //14
        classUnderTest.setBillingAccountAdapter(adapter); //15
        String accountId = classUnderTest.createAccount(account);//16
        assertEquals("123456789", accountId);//17
        assemblerControl.verify(); //18
        adapterControl.verify(); //19
    }
}
```

The test case above tests the happy case of the createAccount service in listing 1.

In line 1-2 we set up the domain objects that are consumed by the service. The Account object is part of the domain model of the service layer while the DTO is a data transfer object that is required by the legacy system.

In lines 3 – 7 we set up the mock object for the assembler we are dependent on. In line 3 we create a MockControl object while we get the actual mock object (which is a dynamic proxy) in line 4. We are now ready to specify the behaviour, also known as the record state, of the mock object. In our case we only call the AccountAssembler once, to assemble a BillingAccountDTO object from an Account object (line 5).

Since we are testing the happy case we want the method to return a BillingAccountDTO object (line 6). All we need to do know is to replay this behaviour (line 7).

After mocking the dependency to the BillingAccountAdapter (lines 8-12) we are now ready to do the actual test. So we create an instance of the class under test – namely the AccountServiceImpl class (line 13). In the next two lines we inject the mocks into the class under test using the dependency injection principle which we can see is crucial in order to use dynamic mocks.

In line 16 we do the actual call to the service and can assert that the method returns what we expect.

Finally, we verify that the mocks behaved like we wanted them to by calling the verify() method on the two
Another great thing about using mock objects is that it is really easy to test that your class handles error situations correctly since it is extremely easy to mock exceptions thrown from dependent objects. The listing below shows a unit test for handling BusinessExceptions being thrown from the adapter object:

```java
import org.easymock.MockControl;
import junit.framework.TestCase;
public class AccountServiceTest extends TestCase {
    /**
     * Tests that we can handle a business exceptions from the adapter dependency
     * @throws Exception If the test fails in an unexpected manner
     */
    public void testCreateAccount_BusinessExceptionFromAdapter() throws Exception{
        BillingAccountDTO dto = new BillingAccountDTO(); //1
        Account account = new Account(); //2
        MockControl assemblerControl = MockControl.createControl(AccountAssembler.class); //3
        AccountAssembler assembler = (AccountAssembler)assemblerControl.getMock(); //4
        assemblerControl.expectAndReturn(assembler.toBillingAccountDTO(account),dto); //5
        assemblerControl.replay(); //6
        BusinessException be = new BusinessException("Test exception"); // 7
        MockControl adapterControl = MockControl.createControl(BillingAccountAdapter.class); // 8
        BillingAccountAdapter adapter = (BillingAccountAdapter)adapterControl.getMock(); // 9
        adapter.createBillingAccount(dto); // 10
        adapterControl.setThrowable(be); //11
        adapterControl.replay(); //12
        try{
            AccountServiceImpl classUnderTest = new AccountServiceImpl(); //13
            classUnderTest.setAccountAssembler(assembler); //14
            classUnderTest.setBillingAccountAdapter(adapter); //15
            String accountId = classUnderTest.createAccount(account); //16
        } catch (BusinessException be){
            assertTrue(true);
        }
        assemblerControl.verify(); //18
        adapterControl.verify(); //19
    }
}
```

Listing 6: Unit testing exception handling
Conclusion

Working with legacy system can be challenging and daunting task and testing of these enormous system even more so. One of the most obvious challenges is that complex environment setup is neccessary to test but the simplest parts of the system. I personally believe this is the single most important reason why testing of these systems never takes off. Another challenge is the endless search for test data, not to speak of the maintenance of these data. Since few of us work with single user systems you will always risk that some other users or systems have corrupted your test data.

As we have seen, using mock objects help us deal with these challenges, and in my opinion mock objects are simply invaluable. When using mock objects for our external dependencies, we are free to construct the test data that we need for our system. Another major advantage is that we don't have to relate to the complex environment needed by the legacy system. In addition, if you decide to focus on unit tests from the start and also use healthy techniques such as mock objects and dependency injection, I guarantee that you will produce better code with less bugs than without using such techniques.

Although there are many advantages to using mock objects, there are definitely some disadvantages as well. The most obvious one is that if you have lots of external dependencies in your classes under test, you need to write a lot of set up code in order to mock these dependencies. One could argue however, that if this is the case, you really should refactor your classes. Another disadvantage is that, if you use mock objects incorrectly, you could end up testing anything but the classes you were supposed to test (see Aslak Hellesøys blog «oh no we're testing the mocks» listed in the resource list below)

References

2. Junit in Action  Vincent Massol
3. http://confluence.public.thoughtworks.org/display/NMock/Introduction+to+NMock
6. EasyMock javadoc

Useful resources

- http://www.easymock.org
  The EasyMock website
  Martin Fowler on dependency injection
- http://static.springframework.org/spring/docs/2.0.x/reference/index.html
  Spring documentation contains a bunch of good Dependency injection examples
- http://blogs.codehaus.org/people/rinkrank/archives/000551_oh_no_were_testing_the_mock.html
  Very interesting blog entry by Aslak Hellesøy about unit testing with mock objects
- http://www.pragmaticprogrammer.com/articles/may_02_mock.pdf
  The authors of «The Pragmatic Programmer» have written an article about mock objects
  Presented at XP2000, this article from Tim Mackinnon, Steve Freeman and Philip Craig pioneered the concept of mock objects and coined the term.