Simple Design and Unit Testing with Enterprise JavaBeans™: The Box Metaphor

Vera Peeters
Tryx bvba
Colomastraat 28
B-2800 Mechelen
Belgium
+ 32 15 41 80 49
vera.peeters@tryx.com

ABSTRACT
Introducing Enterprise JavaBeans™ (EJBs) in your system adds a level of complexity. Whether or not using EJBs has its advantages is not in the scope of this paper. But when you do decide to use them, for whatever reason, there are some simple rules you can take into account. The Box Metaphor will help you keep the business logic in your system simple and clean and, most of all, independent from the technology layer (EJB). This will make it easier to develop all your code in a test harness [1], especially the business logic, which is most important to you.

Keywords
Enterprise JavaBeans, EJB, Simple Design, Unit Tests, JUnit, DIP, Boxes, Metaphor, DTSTTCPW,

1 INTRODUCTION

Enterprise JavaBeans (EJBs) are components that live in a Container inside an Application Server. The Container is responsible for managing the EJB's lifetime, transactions, persistence, load balancing…. If you use EJBs, you get some pre-defined functionality. If you happen to need exactly these functionalities, the EJB framework may help you develop software.

However, using EJBs will always be more complex than not using EJBs. You need to understand how the framework works, what kind of services the container offers you and how it does that. You have to write your components in a very specific way, and you have to pre-compile them and deploy them explicitly. There are of course environments where this entire process can be run very smoothly. But even when that's the case, it can still be useful to be able to take the EJB layer out any time and replace it by some other technology that might be better suited for your new situation.

That's the point: there's no problem in using EJBs, but they should be no more than a layer, a tool you use to make certain things easier for you.

2 SIMPLE DESIGN AND EJB

Using EJB is definitely not the simplest thing to do. So how do EJBs fit with the XP process and with the "Do The Simplest Thing That Could Possibly Work" principle?

Well, to be honest, I think they don't. If you start developing a new system, chances aren't very high that you have a good reason to start using EJBs.

After some cycles, it may turn out you need complex technical features like clustering, complex transactional behaviours, maybe across multiple databases, or security.

The most important reason you can have to start using EJBs is probably a strategic one: your customer asks for it. EJB is hot. I've seen it happen several times that a customer demands you to use EJBs. So in that case, it's a Story.

Another possibility is that you introduce XP in an environment where there's legacy code that already uses EJBs.

So, even if EJBs are not the simplest thing to do, it's still possible that you end up in a situation where you have XP and EJB simultaneously.

3 UNIT TESTING AND EJB

Testing Frameworks for Server Side Java Code

There are several Xunit testing frameworks that are designed for Unit Testing server side java code. Currently, three extensions are available for JUnit: J2EEUnit, JunitEE, HttpUnit [7].

J2EEUnit is an extension of JUnit that can test code called by a Servlet or JSP and that need valid HTTP request, HTTP response and HTTP session objects. In many architectures, the Servlets and JSPs call EJBs [6], and that makes J2EEUnit especially fit to test the integration between the presentation layer (Servlets/JSPs) and the business logic layer (implemented in the session EJBs).

JunitEE is basically a TestRunner that outputs HTML and a servlet that can be used as an entry point.

HttpUnit is an extension of JUnit that accesses websites. It can play the role of a browser. I find it particularly useful to test the java code in a JSP page. It can be used in the same way as J2EEUnit for some kind of integration test between presentation and business layers.

Requirements of a Unit Test Environment

The things that are important in a Unit Test environment

are:

- It must be easy to write the tests, because otherwise you will 'forget' to write them
- It must be easy to run the tests, because otherwise you won't run them often
- It must be easy to set up the tests. Otherwise, you won't run them often.
- It must be possible to develop the code in a Test-First-Design way: write a few lines of test-code, make the test work, refactor, in cycles of a few minutes.

Problems for Unit Testing EJBs

In most environments several steps have to be executed to deploy the EJBs. Some application servers and IDE's even require you to restart the server if you make changes in the code of the EJB or in any code that's used by the EJB in an indirect way. And that can be time-consuming.

There is a certain ironical contradiction in the marketing message, which says it's an advantage that the deployment of the J2EE components can be done in a later phase, independent of the development and by another person. It is precisely this separation of development and deployment that makes it difficult to swiftly run through the test/code cycle. If it takes several minutes to deploy the EJBs, it's impossible to run the Unit Tests every ten minutes.

In some IDE's there are some problems debugging the EJBs.

If you use one of the Server-Side-Java Xunit Testing environments, you still have these problems.

Unit Testing EJBs

In my opinion, Unit Tests should concentrate on another level of the code.

The above-mentioned XUnit frameworks are mostly dedicated to the server side java aspect of the code under test. In my opinion, they are effective, but more for some kind of Functional Tests.

It should be possible to develop and test the functional layer in a simpler way, even if you're stuck with EJBs.

Unit tests should be simple, so that you run them often.

4 THE BOX METAPHOR

Ejb: only a technology

EJB is nothing more than a technology. If you choose to use EJBs, you should treat them as a technology layer. You should never entangle the EJBs in your system.

The most important part of your system is the business logic. It's important that your business logic layer is reusable. Business logic layers should never depend upon any technology layers. You would never want to rewrite your business logic because you want to switch to another technology.

Therefore, *Business Logic should never be implemented inside EJBs.*

It is necessary to test the technical layer to a certain level. It is also necessary to test the integration of your functional layer with the technical layer. But when you write unit tests for the business logic layer, it's much easier if you can stub out the technical layer. The same thing has been described for the persistence layer in [3]. I think you should stub out the EJB layer in a similar way.

Thin EJBs

If you don't want to implement business logic inside the EJBs, but you do want to use EJBs anyway, for whatever reason, what can you do?

You can use the EJB as a facade into the business logic layer.

This means you implement all the important business logic in simple java classes. You never implement something important in the EJB classes.

Generally, the only thing an EJB class would implement is a redirection to the business logic layer.

If you want to implement your business logic independent from the EJB technology layer, there are some rules you should follow. An easy way to remember those rules is the Box metaphor.

The Box

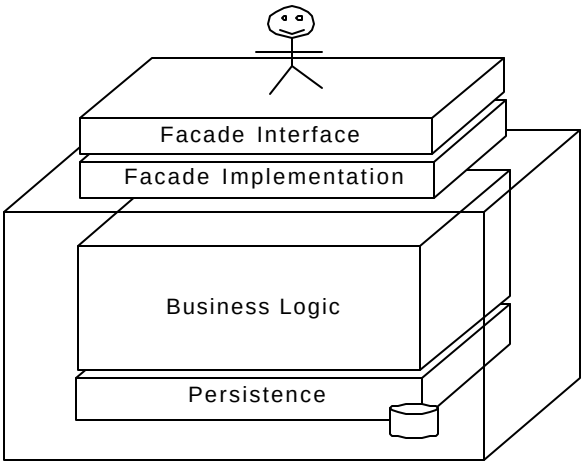
A Box is a vertical slice in the system.

A system contains several Boxes. Each Box is responsible for a certain aspect of the functionality of the system. You could say that each Box is responsible for a more or less independent part of business logic.

Calling it a Box helps to remember the rules that tell you where to put a particular part of code. That's why I think you could say it's a technical metaphor that reminds you of the way the different classes in the system collaborate.

Contents of the Box

A Box contains several parts. We usually implement the different parts in different java packages. The java packages can be thought of as different layers. The accessibility of the layers is top down.



The Business Logic (+ Unit Tests)
This is the package that is important to you. This package should be implemented in a Test-First-Design way, so that the Test Harness covers each path through the code. This package should be reusable if you ever decide to use some other technology.

The Persistence Layer (data source classes + Unit Tests)
The Persistence layer is used by the Business Logic layer, but in such a way that it's easy to stub it out.

Facade into the Box
The Facade [2] is not considered as part of the box. The Facade lies on top of the Box. The client can access the Facade, but it can never directly access the contents of the Box.

The Facade-Interface Layer (Interface + raw-data classes)
The Facade provides a way for the client of the Box to access the functionalities that the Box delivers. We divide the actual Facade in two parts: an abstract interface, and an implementation. The clients don't know anything about the way the Facade is implemented.

Raw-Data Objects
The Facade Interface layer also defines some Raw-Data Objects. These are simple objects, with almost no behaviour. The clients of the Box will use these objects. They will be acting as parameters of the functions in the Facade-Interface.

The Facade-Implementation Layer (EJBs + Unit Tests)
The EJB implements the Facade into the Box.

This must be done in such a way that the clients don't have to know anything about EJB technology in order to use the Box-Interface. We define a factory that the client will call.

5 THE RULES

Box contents are private.
The Boxes contain the classes that implement the Business Logic, and some classes that are responsible for persistence. Each Box owns one or more tables in the system.

The contents of a Box should never be accessed from someone outside the Box. A client of the Box that wants to use the functionalities delivered by the Box, must use the Facade.

Facade can use the contents of its own Box
The implementation of the Facade can use the contents of its Box. The implementation of the functions in the EJB should not contain any business logic. It should redirect the request to the code in the Business Layer.

Box content is locked up in the box
The contents of a Box should not access anything outside the Box. The façade should pass in any information that the contents of the Box need from outside its own Box.

The Facade can talk to other Boxes
If the contents of a Box need some information from other Boxes, the Facade can talk to the Facades of the other Boxes, retrieve the necessary information, and pass it into the contents of the Box.

Box content shouldn't know anything about the Facade
The contents of a Box should never call any function from its own facade. The facade can be used from outside the Box only.

Box content doesn't know anything about EJB
The Box content should not import anything from technology specific packages. It should work with interfaces only.

Box client doesn't know anything about EJB
The Box client talks to the Facade interface, but the Facade interface should be implemented in such a way that the client doesn't need to know by which technology it's implemented.

6 CONSEQUENCES FOR UNIT TESTING

Test the contents of the Box
The contents of the Box implement the business logic, which is of most value for you. This means it is vital that the contents of the box are unit tested *in a very complete way*.

The contents of the Box are simple java classes, that don't depend on EJBs. Therefore, it's easy to develop them Test-First.

Test the facade
The EJB is only a facade into the box. It acts as glue code between the technical layer (all the things that the EJB technology provides) and the business logic code.

Testing glue code is necessary. However, the tests don't need to be very detailed. You already know that the underlying classes work well: the contents of the Box are developed in a test harness. You only want to prove that it's possible to glue all the pieces together. If the separate pieces do what you expect, you can be pretty sure that the whole will work too. At least that's what you can expect

if you integrate continuously.

7 OTHER CONSEQUENCES

Bipolar EJBs

If you follow the Box metaphor, you can define bipolar EJBs [4]. A Bipolar EJB is an EJB that can be instantiated as normal EJB or as a local object that lives on the client side. The clients don't have to be aware that the EJBs can be local or remote. This can be achieved by defining the EJB's Remote Interface as an extension of the Facade-Interface.

The clients ask a factory to deliver an instance of the Facade Implementation. The factory will decide what kind of implementation it will return.

This can be of great help if you have to work in an environment where it is difficult to deploy or debug the EJBs.

This technique is very easy for stateless session beans. I don't think it's worth to apply it for entity beans, because in that case the Container has a lot of responsibilities that should be mimicked by the local bean, which makes things too complicated to be of any use.

It is possible to define Mock Entity EJBs, so that it becomes easier to write tests for the local Session EJBs.

Dependency Inversion Principle

If you want to replace the EJB with another technology, or if you want to refactor the EJBs out, it's not very difficult to do so. The only place you have to change is the Facade-implementation layer. The rules that are expressed by the Box metaphor reduce the EJBs to being a technology.

Box metaphor with other technologies

The Box metaphor can be used with similar technologies like COM Components or CORBA components.

8 CONCLUSION

In some situations it is possible that you choose to use EJBs, although it's not the simplest thing that could possibly work.

If you decide to use EJBs, it's important that you keep the EJB technology separate from the business logic of your system. The Box metaphor is a simple way of structuring your system in a way that ensures you that the business logic won't depend upon the technological layer. It helps you to keep the design of the system as simple as possible, despite the use of the complex EJBs.

If you structure your system using the Box metaphor, it will be much easier to Unit Test your business logic classes.

ACKNOWLEDGEMENTS

Thanks to Pascal Van Cauwenberghe for reviewing this paper.

REFERENCES

[1] Fowler, Martin et al, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] Gamma, Erich et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison–Wesley, 1995.

[3] Mackinnon, Tim, Freeman, Steve, Craig, Philip, Endo-testing: Unit Testing with Mock Objects, 2000. (http://www.sidewize.com)

[4] Mak, Ronald, Bipolar CORBA Objects in Java, Java Report 9/1999.

[5] Martin, Robert C, The Dependency Inversion Principle, C++ Report, May 1996.

[6] Matena, Vlada & Hapner, Mark, Enterprise JavaBeans™ Specification, v1.1, 1999 (http://java.sun.com/).

[7] JUnit extensions, (www.junit.org)