POJO programming approach and the development of Java applications

Vitaly Boguslavsky
Bachelor of Science Thesis
Department of Information Technology
Instructor: Ivan Porres
Åbo Akademi 2012

Abstract

Sometimes it is difficult to maintain a Java application because it is often depended on infrastructure frameworks that can evolve quite rapidly. The problem may arise when the new framework becomes incompatible with the existing code. In this case the application has to be modified which can be challenging and time consuming.
In this study I will discuss the POJO(Plain Old Java Object) approach and how it can improve the maintainability and design of Java applications.

Contents

Abstract

1 Introduction

2 Design decisions

3 What is POJO

4 EJB 2 application architecture(or classic EJB approach)
4.1 The problems with EJB 2

5 Simplifying EJB development with EJB3
5.1 Using POJOs and regular interfaces
5.2 Removing need for Callback methods

6 Developing with Spring
6.1 Dependency Injection
6.2 Wiring POJOs in Spring

7 Conclusion

References

Design desicions

There are two quite different ways to design an enterprise Java application. One option is to use the classic EJB 2 approach, which refers to as the heavyweight approach. When using the heavyweight approach, you use session beans and message-driven beans to implement the business logic. You use either DAOs (data access objects) or entity beans to access the business logic.

The other option is to use POJOs and lightweight frameworks, which I'll refer to as the POJO approach. When using the POJO approach, your business logic consists entirely of POJOs. You use a persistence framework (a.k.a. object/relational mapping framework) such as Hibernate or JDO (Java Data Objects) to access the database, and you use Spring AOP (aspect-oriented programming) to provide enterprise services such as transaction management and security.

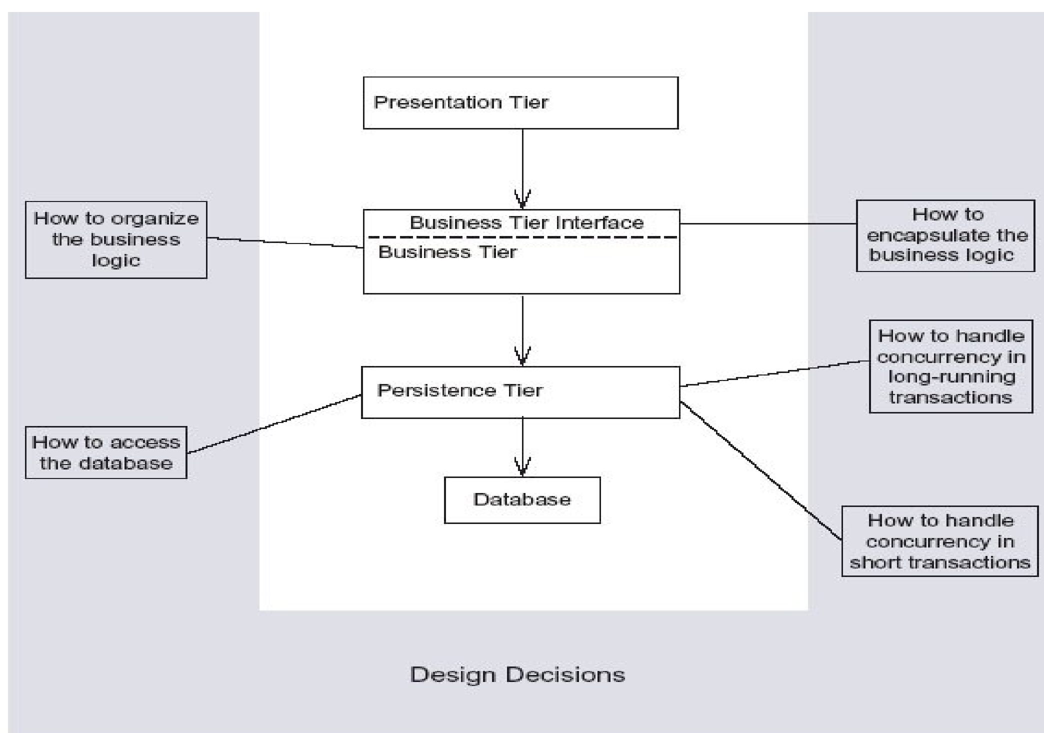


Figure 1. A typical application architecture and the key business logic and database access design decisions.

The application consists of the Web-based presentation tier, the business tier, and the persistence tier. The Web-based presentation tier handles HTTP requests and generates HTML for regular browser clients and XML, and other content for rich Internet clients, such as Ajax-based clients (asynchronous JavaScript and XML). The business tier, which is invoked by the presentation tier, implements the application's business logic. The persistence tier is used by the business tier to access external data sources such as databases and other applications. [4]

3 What is POJO?

POJO is an acronym for Plain Old Java Object. The name is used to emphasize that a given object is an ordinary Java Object, which does not follow any of the major Java object models, conventions, or frameworks.[1][2]

The following code is an example of a simple POJO. Note that there are no references to any interfaces. [3]

```
public class Test {

    String name;

                /**
    *  This is a constructor for a Test Object.
    **/
    public Test(){

        name = "Jane";

    }

}
```

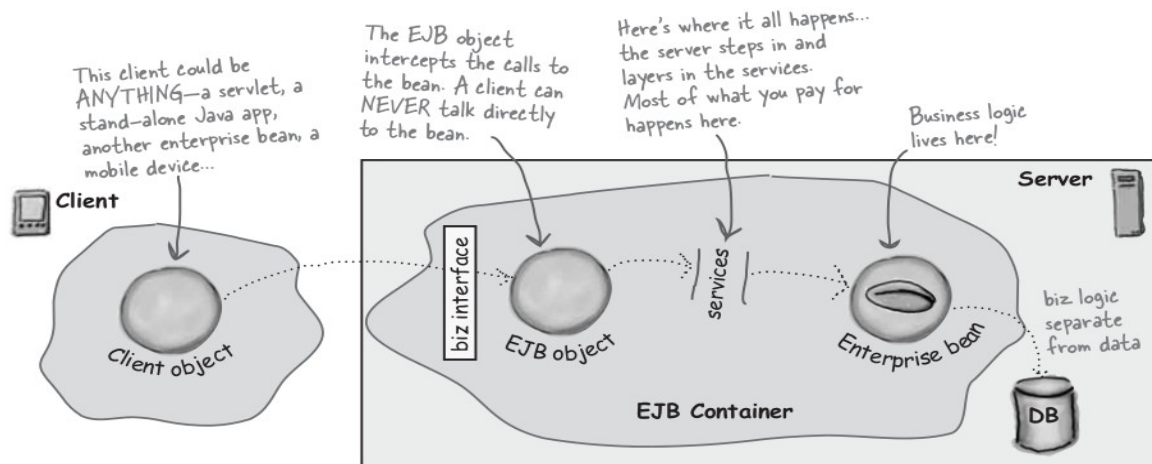
4 EJB 2 application architecture(or classic EJB approach)

EJB version 1.0 was released in 1998. It provided two types of enterprise beans: session beans and entity beans. Session beans can be marked as either stateless or stateful. [4]

A stateful bean can remember conversational state between method calls, while a stateless bean won't remember anything about the client between method invocations. [5]

Entity beans represent data in a database and were originally intended to implement business objects. EJB 2 refined the EJB programming model. It added message-driven beans (which process Java Message Service, or JMS, messages) as well as enhanced entity beans to support relationships managed by the container. [4]

The beans live and run in the server, and the server does virtually everything to manage transaction, security, persistence, and even the life and death of the objects. [5] The high-level view of EJB architecture is illustrated in the figure below.



A ridiculously high-level view of EJB architecture

A high-level view of EJB architecture [5]

4.1 The problems with EJB2 architecture

To understand the limitations of EJB2 architecture lets consider an *Entity Bean* for a persistent Bank class. An entity bean is the in-memory representation of relational data, in other words, a table row.

At first one had to define a local or remote interface, which clients would use. A Java code for a possible local interface and an Entity Bean Implementation for a bank can be found in Appendix 1. [6]

The problem with this design is that each bean is a mixture of business logic and EJB2 infrastructure code. [1]
The business logic is tightly coupled to the EJB2 application "container" and you must provide many lifecycle methods that are required by the container.

Furthermore because of coupling to the container, isolated unit testing is difficult and the reuse outside of the EJB2 architecture is effectively impossible. Finally, even object-oriented programming is undermined. One bean cannot inherit from another bean.

5 Simplifying EJB development with POJOs

The evolution continues in EJB3 which simplifies the programming model considerably by enabling POJOs to be EJBs.

EJB 3 address the problems related to EJB2 model by:

1. Using POJOs as EJBs and regular business interfaces for EJBs

2. Removing need for unnecessary interfaces and implementation of lifecycle methods.

3. Using metadata annotations instead of deployment descriptors

[4] [7]

5.1 Using POJOs and regular interfaces

The EJB3 beans are POJOs that do not have to extend or implement any EJB-specific interfaces. The session bean would consist of plain old java interface, which defines its public methods and a POJO bean class. Instead, the annotations such as *Stateless, Stateful, MessageDriven* or *Entity* are used to indicate the bean class, while the annotations such as *Remote or Local* determine the type of the interface. For example, the stateless bean as HelloWorld can be defined as follows:

```
import javax.ejb.Remote;

@Remote

public interface HelloWorld

{
  public void sayHello(String name);
}
```

```
import javax.ejb.Stateless;

@Stateless
public class HelloWorldBean implements HelloWorld
{
  public void sayHello(String name)
  {
    System.out.println("Hello "+name +" from your first EJB 3.0 component ...");
  }
}
```

[7]

5.2 Removing need for Lifecycle methods

In EJB2 the implementation of such lyfecycle methods as *ejbPassivate, ejbActivate, ejbLoad and ejbStore* was necessary. However not every EJB needed these methods. For example, *ejbPassivate* is not required for a stateless session bean. As bean classes now resemble POJO, implementation of lyfecycle methods has been made optional. But they can

be defined either in the bean class or as an external class. [7]

Because of these changes in EJB3 you have to write a lot less code. Another important change in comparison with EJB2 is that entity beans can be used both inside and out of the application server. Thus it is possible to test the entity beans without deploying the in the EJB container. [4]

5.3 Using metadata annotations instead of deployment descriptors

Let's look at the bank example rewritten in EJB3 which can be found in Appendix 2. This code looks much cleaner because all the information related to framework is contained in the annotations. Hence, it is much to test drive and maintain the code. [6]

It is also possible to use XML deployment descriptors instead of annotations in EJB3. Whether you use one or another is largely a matter of personal preference. However there are situations in which the deployment descriptors are more useful. I will come back to this matter later in my work.

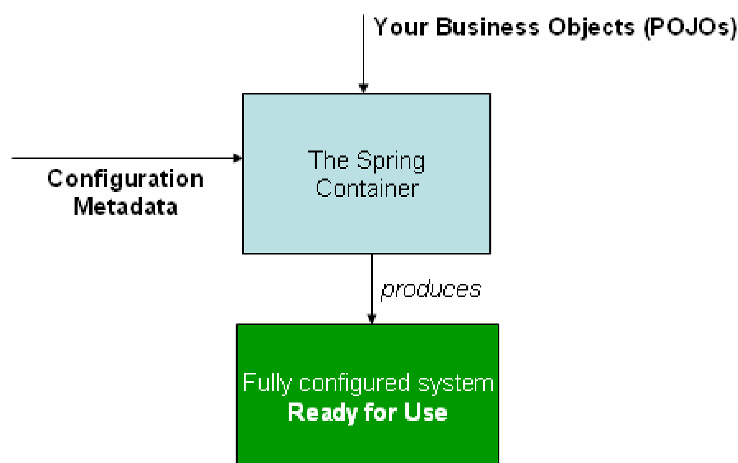
6. Developing with Spring

By the time the EJB3 specification had appeared, other POJO-based frameworks had already established themselves in the Java community. One of them is Spring which is an open source framework, originally created by Rod Johnson. While Spring was created to address the complexity of enterprise application development, any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.[8]

As EJB3, Spring employs POJOs as beans to simplify the development of Java applications. Despite the fact that the concept of POJO is very simple it can be very powerful. One of the ways Spring unleashes this power is by assembling POJOs using dependency injection. In this chapter I would like to discuss this idea more in detail. [8]

6.1 Dependency injection

In Spring the dependency injection is accomplished with the help of a Spring core container. The container allows to inject required objects into other objects. The metadata such as XML file or annotations is used by the container to handle the configuration. This mechanism results in loose coupling between objects or classes. [10] The image below represents a high view of the Spring container.



The Spring container [9]

Most of the Java applications are made of two or more classes that

collaborate with each other to do some business logic. [8] To understand the benefits of dependency injection lets first consider an example of highly coupled code.

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
  private RescueDamselQuest quest;

  public DamselRescuingKnight() {
    quest = new RescueDamselQuest();
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

[8]

From the example above we can notice that `DamselRescuingKnight` creates its own quest, a `RescueDamselQuest`. Thus it results in tight coupling between the `DamselRescuingKnight` and the `RescueDamselQuest`. Which means that this code is difficult to test, reuse and understand. On the other hand the classes need to know about each other or another words coupled in some way to do anything useful. The following example demonstrates how it can be achieved with the dependency injection. [8]

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
  private Quest quest;

  public BraveKnight(Quest quest) {          //Quest is injected//
    this.quest = quest;
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

[8]

Unlike in the previous example the quest is not created within the constructor but is given as an argument for the `BraveKnight` constructor. This is called a *constructor injection* and it is one of the types of the dependency injection. This allows the `BraveNight` not to be depended to any specific implementation of quest. Instead, it s dependency on quest is provided through `Quest` interface. This is how loose coupling is achieved with the dependency injection. [8]

6.2 Wiring POJOs with XML

Spring is a container-based framework and therefore it needs to be configured. We need to tell Spring what beans it should contain and how they are associated or using another words wired with each other. There are many choices for wiring the beans in Spring but the most common way is to do it with XML. [8]

Lets come back to `BraveKnight` class in order to see how the bean wiring works in practice. The `BraveKnight` is written in such a way that it can accept any required quest. Now we can use XML to specify exactly which quest we want him to give. The following example demonstrates the XML configuration file that gives a `BraveKnight` a `SlayDragonQuest.` [8]

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

 <bean id="knight" class="com.springinaction.knights.BraveKnight">
  <constructor-arg ref="quest" />       //Inject quest bean//

 </bean>

 <bean id="quest"
                                        //Create SlayDragonQuest//
       class="com.springinaction.knights.SlayDragonQuest" />

</beans>
```

[8]

Now, the relationship between `BraveKnight` and a `Quest` is defined. In Spring an *application context* is responsible for the loading of definitions from one or more XML files. There are several implementations of *application context* in Spring. The following example shows how the bean definitions are loaded by `ClassPathXmlAppliocationContext` from XML file located in the application's classpath. [8]

```
package com.springinaction.knights;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class KnightMain {
  public static void main(String[] args) {
    ApplicationContext context =
        new ClassPathXmlApplicationContext("knights.xml"); //Load Spring//
                                                           //context//

    Knight knight = (Knight) context.getBean("knight");    //Get knight//
                                                          //bean //

    knight.embarkOnQuest();                                //Use knight//

  }
}
```

[8]

It is important to point out the fact this class does not know about the type of `Quest`and the fact it is dealing with `BraveKnight`. Only the knights.xml file is aware of what the implementations are. [8]

References

[1] Christopher Richardson (February 2006), "Cover Story: What Is POJO Programming?" http://java.sys-con.com/node/180374 (visited 20.03.2012)

[2] http://en.wikipedia.org/wiki/Plain_Old_Java_Object

[3] Eclipse documentation, http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.jst.ejb.doc.user%2Ftopics%2Fcpojosandee5.html (visited on 23.03.2012)

[4]Chris Richadson (2006). "POJOs in action Developing Enterprise Java Applications with Lightweight Frameworks"

[5] Kathy Sierra, Bert Bates (2003). "Head First EJB"

[6] Robert C. Martin (2009). "Clean Code"

[7] Debu Panda. "Simplifying EJB Development with EJB 3.0" http://www.oracle.com/technetwork/topics/simplifying-ejb3-093046.html (visited: 23.04.2012)

[8] Craig Walls (2011). "Spring in Action"

[9] Spring documentation, http://static.springsource.org/spring/docs/2.5.x/reference/beans.html (visited: 1.04.2012)

[10] Lars Vogel (30.08.2009). Dependency Injection with the Spring. http://www.vogella.de/articles/SpringDependencyInjection/article.html

Appendix 1

An EJB2 local interface for a Bank EJB

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject{
  String getStreetAddr1() throws EJBException;
  String getStreetAddr2() throws EJBException;
  String getCity() throws EJBException;
  String getState() throws EJBException;
  String getZipCode() throws EJBException;
  void setStreetAddr1(String street1) throws EJBException;
  void setStreetAddr2(String street2) throws EJBException;
  void setCity(String city) throws EJBException;
  void setState(String state) throws EJBException;
  void setZipCode(String zip) throws EJBException;
  Collection getAccounts() throws EJBException;
  void setAccounts(Collection accounts) throws EJBException;
  void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

[6]

EJB2 Entity Bean Implementation

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
  // Business logic...
  public abstract String getStreetAddr1();
  public abstract String getStreetAddr2();
  public abstract String getCity();
  public abstract String getState();
  public abstract String getZipCode();
  public abstract void setStreetAddr1(String street1);
  public abstract void setStreetAddr2(String street2);
  public abstract void setCity(String city);
  public abstract void setState(String state);
  public abstract void setZipCode(String zip);
  public abstract Collection getAccounts();
  public abstract void setAccounts(Collection accounts);
  public void addAccount(AccountDTO accountDTO) {
  InitialContext context = new InitialContext();
  AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
  AccountLocal account = accountHome.create(accountDTO);
  Collection accounts = getAccounts();
  accounts.add(account);
  }

  // EJB container logic
  public abstract void setId(Integer id);
  public abstract Integer getId();
  public Integer ejbCreate(Integer id) { ... }
  public void ejbPostCreate(Integer id) { ... }
  // The rest had to be implemented but were usually empty:
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void ejbRemove() {}
}
```

[6]

Appendix 2

An EBJ3 Bank EJB

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
   @Id @GeneratedValue(strategy=GenerationType.AUTO)
   private int id;
   @Embeddable // An object “inlined” in Bank’s DB row
   public class Address {
      protected String streetAddr1;
      protected String streetAddr2;
      protected String city;
      protected String state;
      protected String zipCode;
   }

   @Embedded
   private Address address;

   @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
              mappedBy="bank")
   private Collection<Account> accounts = new ArrayList<Account>();

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public void addAccount(Account account) {
      account.setBank(this);
      accounts.add(account);
   }

   public Collection<Account> getAccounts() {
       return accounts;
   }
```

[6]