Lecture 4
Comparison of component models

1. Domain
2. Connectors
3. Composition
4. Deployment
5. Use of standard services and non-localizable concerns
6. Code generation
Java Beans class (component) diagram

Component Y

XEvent
attribute1
attribute2
getAttribute1()
getAttribute2(n)

Component X

X
property 1
property 2
private transient Vector listeners;

getProperty1()
getproperty2()
setProperty1()
setIProperty2()
addEventListener()
removeEventListener()

EJB2.1 class (component) diagram
Tendencies (1)

1. Coexistence components, objects and non-localizable concerns

- JBs coexist with objects presenting events
- EJBs coexist with POJOs, non-localizable services provided by the EJB container and implemented as interceptors
Tendencies (2)

2. Increasing the role of container for binding abstractions
   JB-model:
   1. An interface with a notification method
   2. An event with data and methods of access to data
   3. Container plays no role in binding (only for registration of listeners).
   EJB-model:
   1. An interface is a method with an illusion of direct message passing.
   2. Methods are never called directly, the calls are intercepted by the EJB container

Plan for lessons 4 and 5

• Understand the relations between objects, components and non-localizable concerns
• Verify our conclusions about the tendencies in CBD using other component models (.Net, web services)
Non-localizable concerns

Tyranny of the dominant decomposition

Existence of non-localizable concerns is a function of both the particular decomposition of a system and the underlying support environment.

Tyranny of the dominant decomposition means that the chosen decomposition may cause crosscutting concerns. If we change the decomposition, some crosscutting may disappear, but not necessarily.

A particular environment might invisibly support a concern (for example, transaction synchronization) that needs to be explicitly addressed in another environment.

Separation of Non-localizable concerns

Chosen decomposition gives birth to specific crosscutting concerns. Crosscutting is scattering the implementation or specification of a concern throughout the implementation or specification.
Separation of Non-localizable concerns

Other examples of crosscutting concerns

**High-level, user visible requirements**
- Viewing functionality is a crosscutting concern of all CRUD functionalities in an Enterprise Information System.
- Security procedures are crosscutting concerns for any bank application.

**Middleware-level**
- Operations performed with pixels in image processing applications.
- Serialization is a crosscutting concern for all remote method calls
- Logging

**Low-level implementation issues**
- Concurrency control
- Caching ...

---

Separation of Non-localizable concerns

**AOSD terminology**

- scattering – spread around
- tangling – code in one region addresses multiple concerns
- scattering and tangling (S&T) tend to appear together; they describe different facets of the same problem
Separation of Non-localizable concerns

Aspects vs Objects

A concern is a requirement or an implementation issue that is considered in the software engineering process.

An object is a unity of
- encapsulated data,
- methods to access the data,
- state and
- behaviour defining the possible change of the state.
Objects can read state of other objects but not every object reads state of other objects.

An aspect is an object that has to read state and method calls of other objects to fulfill its behaviour. Moreover, an aspect has a new abstraction ability: it can treat state of different objects or different methods calls as the same trigger of its behaviour.

Non-localizable concerns

AOSD terminology:

Aspect

An aspect - is a modular unit designed to implement a crosscutting concern.
An aspect contains a code (an advice) and the instructions on where, when and how to invoke it (pointcut designators).
An aspect is “responsible” for its own call.
Separation of Non-localizable concerns

**AOSD terminology:**

**Advice**

An advice – is the behaviour that corresponds to an aspect.

An advice is **oblivious** if the original base code “does not know” about the advice.

---

Non-localizable concerns

**AOSD terminology:**

**Join points. Pointcut designator**

**Join points** are well defined places in the structure of a program where an advice should be attached.

A join point model defines the kinds of join points allowed. The most common elements of a join point model are method calls.

A **pointcut designator** describes a set of join points. It provides a quantification mechanism – a way to specify that something happens when each of join points is reached.
Non-localizable concerns
Aspect Display.updating()

pointcut
designator

after(): call(void FigureElement+.set(*,..))
    || call(void FigureElement.moveBy(int, int)) {
    Display.update();
}

advice

Separation of Non-localizable concerns
How to specify?
Aspect Display.updating()

<<Aspect>>
Display.updating

Join point specification

Relations of aspects and targets should not be presented
if the diagrams are designed for tools
Separation of Non-localizable concerns

AOSD terminology:

Weaving

**Weaving** is a process of composing aspects in correspondence with pointcut designators.

Weaving can be done statically compiling the advice together with the base code (AspectJ).

Weaving can be done dynamically inserting aspects when loading and executing code (Interceptors in EJB3.0).

**Before, after** and wrapping, are the most common AO techniques to provide method calls at join points and to allow and advice to run **before, after** and around a method call. This techniques need a container around a component, which mediates communications to the component and enforces the desired aspects.

---

**EJB3**

Interceptors. Logger Interceptor

```java
package ejb;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.interceptor.Interceptors;
@Stateless
@Interceptors(LoggerInterceptor.class)
public class ProductFacade implements ProductFacadeRemote {
    @PersistenceContext
    private EntityManager em; // injection of entity manager
    ...
    public String productInfo(int id){...} // business method
    ...
}
```
The interceptor class contains an advice of the aspect and the join point specification. A business method is a join point.

```java
package interceptors;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class LoggerInterceptor {
    @AroundInvoke
    public Object logger(InvocationContext ctx) throws Exception{
        String nam=ctx.getMethod().getName();
        if (ctx.getMethod().getName().startsWith("product")){
            Object[] params=ctx.getParameters();
            for (int i=0;i<params.length;i++){
                System.out.println("\tnam " + nam + "tparam "+params[i]);
            }
        }
        return ctx.proceed();
    }
}
```

The Application server plays the role of the weaver. The sequence diagrams cannot be seen in the code, it is defined in the EJB3.0 standard JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements (Chapter 12. Interceptors)

---

**EJB3**

**Interceptors. Performance Interceptor (1)**

```java
@Interceptors({PerformanceInterceptor.class})
public String productInfo(int id)
{   System.out.println("-- ProductInfo()");
    Product product = find(new Integer(id));
    context.lookup(ManufactureFacadeRemote.class.getName());
    if (product != null)
    {   System.out.println("\t-- ProductInfo()");
        System.out.println("\t\tManufacture: " + product.getManufactureId().getManufactureId() + "\" +
            manufacture.manufactureInfo(product.getManufactureId().getManufactureId()) +
            " Description: " + product.getDescription() +
            " Price: " + product.getPurchaseCost() +
            " Availability: " + product.getAvailable().toString();
    } else
    {   System.out.println("\t-- ProductInfo(): NullException");
        return null;
    }
}
```
package interceptors;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class PerformanceInterceptor {
    @AroundInvoke
    public Object measureDuration(InvocationContext ctx) throws Exception {
        String nam = ctx.getMethod().getName();
        System.out.println("--> PerformanceInterceptor.measureDuration() of " + nam);
        long start = System.currentTimeMillis();
        Object result = ctx.proceed();
        long elapsed = System.currentTimeMillis() - start;
        System.out.println("<-- PerformanceInterceptor: duration of " + nam + " is " + elapsed + " msec; returns " + result.toString());
        return result;
    }
}

---

Increasing the role of container for binding abstractions

- Beans, POJO’s and Interceptors coexist in the model.
- Container organizes binding of different abstraction using dependency injection
EJB3. Dependency Injection Principle

The principle of Dependency Injection was stated by R. C. Martin in 1996: "High-level modules should not depend upon low-level modules. Both should depend upon abstractions". This means that:

1. Beans, objects and aspects should depend on interfaces.
2. Besides the issue whether we should depend on interface we must deal with problem of instantiation or as the Gang of Four state "you have to instantiate concrete classes somewhere in your system".

EJB3. Dependency Injection Types

Dependency Injection is a mechanism for obtaining references to EJB-related objects relying on the container. There are many ways to instantiate and pass references to concrete classes. Fowler identifies:

- Service Locator
- Constructor Injection
- Method Injection

- **Service Locator** is used in the EJB 2.1 where objects had to be looked up using JNDI (Java Naming and Directory Interface).
- **Constructor Injection** is used in EJB 3.0. Specification mandates that containers bind beans to the JNDI tree by their names at deployment time. There are two sub types: Constructor No Default (CND) and Constructor With Default (CWD).
- **Method Injection** is used in EJB 3.0. There are two sub types: Method No Default (MND) and Method With Default (MWD).
EJB2.1. Service Locator

A service locator is an object that know how to find all the beans. A client asks for implementation of a bean explicitly by a message to a locator.

EJB3.0. Dependency Injection. Constructor Injection

The implementation is not requested by the client explicitly.

class CNDeg {
    B b;
    public CNDeg(B b){
        this.b=b;
    }
}

class CWDeg {
    B b;
    public CWDeg(B b){
        b=new BImp();
    }
    public CWDeg(B b){
        this.b=b;
    }
}
**EJB3.0. Dependency Injection. Method Injection**

```
class MNDeg {
    B b;
    public void setB(B b){
        this.b=b;
    }
}
```

```
class MWDeg {
    B b;
    public MWDeg(B b){
        b=new BImp();
    }
    public void setB(B b){
        this.b=b;
    }
}
```

---

**EJB3.0. Dependency Injection. Context. Advantages**

POJOs exist in the persistence context of a Session (entity manager).
Interceptors exist in the invocation context of a Session.
Other beans exist in the initial context.
These dependencies are injected.

Dependency Injection of different sorts is a useful common feature of frameworks, that simplifies:
- understanding dependencies of a component.
  - Service Locator hides the dependencies
- testing.
  - Implementations can be easily replaces with stubs and mocks.
Abstractions

- Objects
- Aspects
- Components

According to the MDA approach, the definition of a component depends on the level of design.

In order to separate the business functionality from the implementation details, the system is presented as:
- one or more Platform Independent Models (PIM)
- one or more Platform Specific Models (PSM) with sets of interface definitions, each describing how the PIM model is implemented on a different middleware platform.
MDA: PIM and PSM

- The Model Driven Architecture (MDA), an initiative launched in 2001 by the OMG, aims to promote modeling to a central role in the development and management of application systems. In particular, it suggests that “fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code”.
- A specification in PIM – the primary artifact - defines one or more PSMs and sets of interface definitions, each specifying how the base model is implemented on a different middleware platform.
- In the MDA, middleware-specific models and implementations are secondary, rather than primary, artifacts.
- The goal is to generate code from models.
  
  Joe Skabra, Garnter Group, 2005, “In 2010, 50% of IT companies will use code generation”.

PIM level. Separation of concerns

Abstractions: Classes, Aspects.

If based on the business logic and common sense a set of classes does not share state with other classes, this set of classes may be called a component at the PIM level.

Example: Producer and Consumer do not share state. The Producer Classes can be combined into a component: Materials, Stock, Product and Product Request classes may present a Producer. The Consumer classes can be combined into another component. The components may define interface and communicate via message exchange.
PSM level. Separation of concerns

Abstractions:
Classes, Aspects, Components of chosen technologies

Classes, Aspects and Components of the PIM level can be grouped and split at the PSM level into components offered by technologies.

Examples:
- Java Beans can implement components identified at the PIM level
- If a technology is domain oriented, like EJB, it repeats the same patterns of grouping and splitting of the classes, aspects and components from the PIM models, mapping them into prefabricated components at the PSM level. An EJB PIM level Producer is a combination of Entities and Session Beans

UML Profiles for Component technologies

- The Profile mechanism provides a lightweight extension mechanism to the UML standard.
- Profiles are specific packages.
- Stereotypes are metaclasses used to indicate the component types, interface types and association types.
- Tagged values are standard meta attributes.
- A technology profile is consistent with a component technology allowing (partial) code generation from models.
- UML Profiles define platform specific components.
- PIM level components may do not correspond to platform specific components.
Next lecture we are going to

Verify our conclusions about the tendencies in CBD using other component models:
.Net and Web Services

Homework

Assignment 2 EJB3.0 application and its UML specification (when you implement an interceptor, try to present it at the specification diagrams)

Read articles and answer questions:
- Object Management Group. Model Driven Architecture (reader)
- Jan Hannemann and Gregor Kiczales. “Design Patterns Implementation in Java and AspectJ”.
- “Developing Service Oriented Architectures”