Distributed Systems

Lecture 7: Component Models

Prof. Dr. Alexander Schill
http://www.rn.inf.tu-dresden.de
Motivation for Component Approach

Software Components - Definition

Component Platforms

- EJB (Enterprise JavaBeans)
- Spring Framework as simplified EJB alternative
- Microsoft .NET
- OSGi (formerly Open Services Gateway Initiative)
Motivation and Goals

- Application and distribution logic mixed in object-oriented code → limited portability and reusability
  - e.g. Transaction specific code
    ```java
    Warehouse.con.commit();
    ...
    catch (SQLException se) { Warehouse.con.rollback(); }
    ```

- Reuse based on libraries, frameworks and abstract programming interfaces
- Reuse through inheritance (e.g. in frameworks) creates dependencies, not known or evident to the programmer
- Standardized basic services for distributed systems provided, but manufacturer specific extensions not covered by standards

→ Goal: **Separation between application logic** (implemented by application programmer) and **distributed systems functionality** (to be provided by component platform in a more reusable way)
“A component is a **unit** of composition with **contractually specified interfaces** and **explicit context dependency** only. A component can be deployed independently and is subject to composition by third parties.”

[C. Szyperski]

- **Unit** – Specified, implemented and deployed only as a whole
- **Contractual interfaces**
  - Use Contracts - interface provider and interface user
    - Fulfilment of underlying conditions: e.g. service quality
  - Realisation Contracts - component specification and implementations
    - Agreement that the component specification must be respected by every implementation
- **Explicit context dependency**
  - No hidden dependencies on internal super-classes, special libraries
  - Clear definition of component model and component platform
  - Behaviour can be defined and modified by universal component attributes
Component Platforms – EJB: Application Example

EJB focuses on:
- Server side
- Transactions
- Persistence
- Security

Prof. Dr. Alexander Schill
EJB – Session Beans and Message Driven Beans

- **Session Bean**
  - Client entry point to server side application logic for user sessions
  - Type 1: Stateless
    - No information about state of session, can be deleted and then re-created later
    - E.g. Product search component (i.e. depending only on current search term)
  - Type 2: Stateful
    - Information about current session with conversational state guiding method execution
    - E.g. Contents of shopping basket

- **Message Driven Bean**
  - Based on Java Messaging Service technology, stateless
  - Asynchronous message exchange ➔ timely decoupled communication
  - E.g. used for realisation of business processes like purchase order processing
import javax.ejb.Remote;

@Remote // Annotation enabling remote access via RMI
public interface ShoppingBasketRemote {
    public boolean addProduct(ProductDescription descr, int quantity);
    public boolean removeProduct(ProductDescription descr);
    public int initiateOrder();
    public int corporateOrder(CorporateID id);
    ...
}

import javax.ejb.Stateful;

@Stateful // Annotation declaring stateful type of session bean
public class ShoppingBasketBean implements ShoppingBasketRemote {
    //Constructor
    public ShoppingBasketBean() {...}
    public boolean addProduct(ProductDescription descr, int quantity){...}
    public boolean removeProduct(ProductDescription descr){...}
    public int initiateOrder(){...}
    public int corporateOrder(CorporateID id){...}
    ...
}
EJB - Entities

- **Entity** (Java Persistence API) – e.g. Product Catalogue
  - Mapping of objects to (relational) database storage
    - POJO (Plain Old Java Object) represents one table, instances represent rows
    - Java Annotations/xml-file for mapping between objects and database
    - Java Persistence Query Language: SQL-like language for entities → portable to different databases

- **Entity Manager** manages entities & persistence
  - No remote interface → Callable via local session beans or message driven beans
  - Associated with a Persistence Context = data store
  - **Container-Managed Entity Manager** – all components in the same transaction use the same persistence context (for example, automatic database mapping)
  - **Application-Managed Entity Manager** – EntityManager instances created and destroyed by the application (for example, by individual database access mechanisms using JDBC or via file system calls)
• **Deployment descriptor**: character of beans, interfaces, class implementations
• **Assembly descriptor**: transaction management, state management, security properties
• Very simple, but no specific optimization possible (only inside the code)
• Further simplified in recent EJB versions, but general principle shown below:

Example Deployment Descriptor:

```xml
<ejb-jar>
  <display-name>ShoppingBasket</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>ShoppingBasketBean</ejb-name>
      <remote>tud.onlinetrader.orderprocessing.ShoppingBasketRemote</remote>
      <ejb-class>tud.onlinetrader.orderprocessing.ShoppingBasketBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor> ... // See next slide <assembly-descriptor>
</ejb-jar>
```
Deployment of EJB

```xml
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name> ShoppingBasketBean </ejb-name>
      <method-name> initiateOrder </method-name>
    </method>
    <trans-attribute> RequiresNew </trans-attribute>
  </container-transaction> ...

  <method-permission>
    <role-name> CorporateClient </role-name>
    <method>
      <ejb-name> ShoppingBasketBean </ejb-name>
      <method-name> corporateOrder </method-name>
    </method>
  </method-permission>
</assembly-descriptor>
```
Enterprise JavaBeans

Session Beans

- Stateless Session Beans
  - Stateless Service, for instance, Search engine

- Stateful Session Beans
  - Processing with internal state, for instance, shopping

Message-driven Beans

- Multicast-Communication (1:n)

Java Persistence API

Entities

- Container-Managed
  - Durable data, automatic persistence mechanism.

- Application-Managed
  - Durable data, own persistence solution of the appl.
- Open source application framework for Java
- Lightweight alternative for EJB
  - JavaBeans as components
  - Any POJO (plain old Java object) can be a component
  - Applicable for Java SE and Java EE applications
- Spring container responsible for creation, wiring, configuration and lifecycle management

- Data Access
  - Spring JDBC
  - Transaction management
- ORM
  - Hibernate
  - JPA
  - TopLink
  - JDO
  - ...
- JEE
  - JMS
  - JCA
  - EJBs
  - JMX
  - ...
- Web
  - Spring Web MVC
  - Struts
  - WebWork
  - JSF
  - JSPs
  - ...
- AOP
  - Spring AOP, AspectJ Integration
- Core
  - Spring Container

Distributed Systems – Lecture 7: Component Models and Platforms

Prof. Dr. Alexander Schill
OrderProcessing tightly coupled to payment method
  - code is hard to reuse and test
  - payment method can not be exchanged easily

Coupling is necessary but should be carefully managed

```java
public class OrderProcessing {
    private CreditcardPayment payment;

    public OrderProcessing() {
        this.payment = new CreditcardPayment();
    }

    public void doPayment() {
        payment.pay();
    }
}
```
dependency injection - principle

- Objects are given their dependencies at creation time by some third party
- Application logic code generally has no dependencies to the spring framework itself
- Principle can also be applied to aspects like transactions, security, persistence

```java
public class OrderProcessing {
// Interface to be implemented by any payment method
private Payment payment;

// Dependency injection by constructor argument
public OrderProcessing(Payment payment) {
    this.payment = payment;
}
public void doPayment() {
    payment.pay();
}
```
Injecting a Dependency with Spring

```xml
<?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.xsd">

    <bean id="order" class="de.tud.springexample.OrderProcessing">
        <constructor-arg ref="payment" />
    </bean>

    <bean id="payment" class="de.tud.springexample.CreditCardPayment">
        <constructor-arg value="#{T(System).out}" />
    </bean>

</beans>
```
Component Platforms – Microsoft .Net

.NET Applications (Assemblies)

.NET Development Tools

.NET 3.0

WCF (Windows Communication Foundation)

WPF (Presentation Foundation)

WCS (Card Space)

WWF (Workflow Foundation)

Class Libraries (Base classes, ASP.NET, ADO.NET, WinForms)

Common Language Runtime (CLR)

Operating System
Component Platforms – Microsoft .Net

- Service based platform (similar to Java EE)

- .NET Framework (core)
  - Runtime environment (CLR) for basic functions (garbage collection, memory management, etc.)
  - Class libraries, APIs, services

- .NET Components (= Assemblies)

- .NET Services
  - WCF: MS Message Queue, MS Transaction Server, Web Services, etc.
  - WPF: GUI, media, etc.
  - WWF/WCF: workflows, etc.

- MSIL (Microsoft Intermediate Language)
  - Source Code $\rightarrow$ MSIL $\rightarrow$ Machine Code
  - Interoperability, independence from specific programming language
OSGi – formerly Open Service Gateway Initiative
Specification of an open, standardized, component-based service platform by the OSGi Alliance (large industry consortium)

Initial Application Domains:
• Home and building automation, Telematics platform

Increasing number of application areas
• Mobile, automotive, desktop or server-side applications

OSGi Platforms:
• Open Source: Equinox (Eclipse's OSGi console), Apache Felix, Knopflerfish
• Commercial: mbedded Server (Prosyst), Service Management Framework (IBM)
Component Platforms - OSGi

- Local component platform based on Java
  - Bundles as components
  - Multiple bundles running on one Java VM
  - Sharing of components between applications

- Additional Features on top of Java
  - Bundle Management
    - Lifecycle-management at runtime
    - Dynamic deployment and binding
    - Remote management architecture
  - Service Model
    - Provisioning of basic services
    - Lightweight service registry
    - Support of remote services (Distributed OSGi)
  - Dependency management
    - Explicit import/export of interfaces of bundles
    - Versioning of bundles and interfaces
Components in OSGi are called Bundles
- interface definition
- java classes and further resources
- bundle description - manifest file

Unit of deployment is bundle
Unit of composition is service

Bundle compiled into JAR
- Class files
- Resources
- Manifest
  - describes contents of JAR
  - deployment information, e.g. import and export of java packages for shared use
  - Description of dependencies
- Bundle Interface Definition

```java
package tud.onlinetrader.Productsearch;

public interface ProductSearch {
    public Enumeration getProductsByCategory(String category);
    public Enumeration getProductsByDescription(String description);
    ...
}
```

- Bundle Interface Implementation

```java
package tud.onlinetrader.Productsearch.impl;

public class ProductSearchImpl implements ProductSearch{
    public Enumeration getProductsByCategory(String category){......}
    public Enumeration getProductsByDescription(String description){......}
    ...
}
```
The dependencies of a bundle
- **Package dependencies**
- Service dependencies
- Runtime environment dependencies

Dependencies defined through manifest file: Import-/Export-Package

Framework resolves package dependencies statically

- Bundle State: INSTALLED \(\rightarrow\) RESOLVED
OSGi – Bundle Lifecycle

Install
- Evaluation of manifest file
- Checking availability of bundle classes and resources
- Persistent saving

Resolve
- Resolution of static dependencies to external bundles and java packages

Uninstall
- Must take place explicitly through the OSGi platform
- Removal of all persistently saved parts of the bundle
Bundle – Activator Implementation

```java
package tud.onlinetrader.Productsearch.impl;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
...
public class Activator implements BundleActivator {
    public static BundleContext bc = null;

    public void start(BundleContext bc) throws Exception {
        Activator.bc = bc;
        //Code to be executed, when bundle starts, e.g. starting own services / services of other bundles
    }
    public void stop(BundleContext bc) throws Exception {
        Activator.bc = null;
        //Code to be executed, when bundle stops.
    }
}
```
OSGi Service Registry

1. create instance
   - ProductSearchBundle (Bundle providing the service ProductSearch)
   - ServiceRegistration (getReference, setProperties, unregister)

2. register Service
   - Activator

3. Activator
   - Activator of bundle using a service
     - ServiceReference

4. getServiceReference
   - getService(reference)

- Service Registry Table:
<table>
<thead>
<tr>
<th>service class name</th>
<th>service instance</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductSearch.class.getName()</td>
<td>ProductSearchImpl</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
1. Register Service (bundle, that provides service):

   ```java
   public class Activator implements BundleActivator {
       public void start(BundleContext bc) throws Exception {
           ...
           bc.registerService(ProductSearch.class.getName(), new ProductSearchImpl(), properties); // Properties of the service ...
       }
   }
   public void stop(BundleContext bc) throws Exception {
       //automatic deregistration
   }
   ```

2a). Service use (bundle, that uses service ) via ServiceReference

   ```java
   ServiceReference servRef = bc.getServiceReference(ProductSearch.class.getName());
   if (servRef != null){
       ProductSearch ps = (ProductSearch) bc.getService(servRef);
       if(ps!=null){ ps.getProductsByCategory(“home”); ...
   }
   bc.ungetService(servRef);
   ```
2b. Dynamic service use (bundle, that uses service) via service tracker
public class Activator implements BundleActivator {
    public SearchServiceTracker tracker;
    ...
    public void start(BundleContext bc) throws Exception {
        tracker=new SearchServiceTracker(bc);
        tracker.open(); }
}

3. Service Tracker Implementation (bundle, that uses service)
public class SearchServiceTracker extends ServiceTracker{
    // If tracked service is registered, bind to service
    public Object addingService(ServiceReference reference) {
        ProductSearch ps =(ProductSearch)bc.getService(reference); ...
    }
    // rebind if tracked service is modified
    public void modifiedService(ServiceReference reference, Object service){...}
    // unbind if tracked service is deregistered
    public void removedService(ServiceReference reference, Object service){...}
- Uniform local and remote access to services enabled, transparent for bundles
- Various communication protocols supported by distribution provider
- Discovery service for binding via service description and protocol details
Advantages

- Open standard, widely adopted/supported
  - E.g. Equinox, Apache Felix
- Component Model + Framework
- Explicit statement of dependencies (Imports/Exports)
- Building of modular systems
- Dynamic service- and lifecycle-management

Disadvantages

- Breaking down everything to modules may be painful
- Open source libraries may not be compatible with OSGi
- Enterprise features (transactions, etc.) not completely supported


OSGi Alliance Homepage: http://www.osgi.org/

OSGi Platform implementation Equinox: http://www.eclipse.org/equinox/