Distributed Systems
Component Models and Platforms

Tenshi Hara
tenshi.hara@tu-dresden.de
https://www.rn.inf.tu-dresden.de/hara
Outline

- Motivation for Component Approach
- Software Components - Definition
- Component Platforms
  - Enterprise JavaBeans
  - Microsoft .NET
  - OSGi
Limitations of Object-oriented Approaches

- application and distribution logic mixed in object code
  → limited portability; e.g., transaction specific code
    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
- standardised basic services for distributed systems provided, but manufacturer specific extensions not covered by standards
  → Dependencies on platforms, frameworks, ...
  → Limited reusability & portability
Motivation for Component Approach

Desirable concepts

- Separation of application logic and distribution specific functions → application programmer only develops application logic
- Configuration of distribution specifics for platform specific deployment
- Complete abstraction from
  - Distribution details
  - Basic services
  - Runtime platform
→ Extensive reuse of application logic possible
Solution to before-mentioned limitations and desires

Component-based software development; key ideas:
- simple application development
- Composition of standardised modules
- easy changeability and extension of modules
- cope with high rate of change of today’s systems
- Reusability of modules

What are components?
- Design view: abstract components as basis for software design
- Implementation view: complete, commercially available software building blocks, placed in standardised component frameworks
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**
    - Binding for interface provider and interface user
    - Fulfilment of underlying conditions; e.g., service quality
  - **Realisation Contracts**
    - Describe specification and implementation of components
    - Agreement that the component specification must be respected by every implementation
Software Component – Definition Key Ideas

- Explicit Context Dependencies
  - underlying component model
    (definition of interfaces and composition of components)
  - Information about component platform and implementation
    (e.g., deployment information)

- Relation to Component Standard
  - Platform provides runtime and service environment (i.e., container)
  - Control of component lifecycle (installation, activation, execution)
  - a software artefact (i.e., collection of interfaces & classes) can not
    become a component until linked to a component platform
‘EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology (Java EE).’

- Server-side component model for business applications
- Implementation of application logic on middle tier
- Definition of three component entity types
  - Session Bean
  - Message Driven Bean
  - Entity Bean (JPA)

Used in combination to implement application logic.
Component Platforms – EJB Application Example

Client

Java
RMI

Web
Service

Session Bean
Shopping Basket

Remote Interface

Session Bean
Purchase Order Creation

Remote Interface

Message-Driven Bean
Purchase Order Processing

JMS Provider

EJB Container

Application Server
Sales

Entity
Ordering

Entity
Product Catalog

Application Server
Delivery

Database Server

JMS Provider

Entity
Dispatch List

JMS

Entity
Catalog

Ordering

Dispatch
List

EJB Container

Application
Server
Sales

Distributed Systems – Component Models and Platforms
- Client entry point to server side application logic
- Management of user sessions

- Type 1: Stateless
  - no information about state of session
  - can be deleted and then re-created later
  - e.g., product search component
    → result depends only on current search term

- Type 2: Stateful
  - Information about current session
  - Conversational State
  - e.g., remember contents of shopping basket
    → result of method calls depends on internal state of component
EJB – Message-Driven Beans

- based on Java Message Service technology
- Message-orientated communication over message channels
- asynchronous messages exchange → timely decoupled communication
- Messages can be sent by various components (not just Beans)
- stateless
- Realisation of business processes
**Entity** (Java Persistence API); e.g., Product Catalogue
→ maps 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 DB
- 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
- *Application-managed Entity Manager* – EntityManager instances created and destroyed by the application
EJB - Summary

Enterprise JavaBeans

Session Beans
- Stateless Session Beans
  - Stateless Service; e.g., search engine
- Stateful Session Beans
  - Processing with internal state; e.g., shopping

Message-driven Beans

Entities
- Container-Managed
  - Durable data, automatic persistence mechanism
- Application-Managed
  - Durable data, own persistence solution of the application

Java Persistence API

Multicast-Communication (1:n)
Bean Implementation – EJB Specification 3.0

- Specification of implementation aspects
  - Classes & interfaces of a component, relationships during runtime
  - Methods provided by beans published in form of interfaces
  - Interface specification and class implementation with normal Java notations

- Java annotation – Identification of interface types and bean types
  - Code generation for annotation-based bean management (e.g., @Remote)

- Interface
  - Remote Interface (e.g., through RMI)
  - Local Interface – normal Java method calls → faster than RMI (no call mediation via Stubs)
Bean Implementation – Example Session Bean

```java
import javax.ejb.Remote;
@Remote
public interface ShoppingBasketRemote {
    public boolean addProduct(ProductDescription descr, int quantity);
    public boolean removeProduct(ProductDescription descr);
    public int initiateOrder();
    public int corporateOrder(CorporateID id);
}
```

```java
import javax.ejb.Stateful;
@Stateful
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){...}
    ...
}
```
- Distribution Specifics and Deployment
  - **Deployment descriptor**: character of beans, interfaces, class implementations
  - **Assembly descriptor**: transaction management, state management, security properties
  - very simple, but no specific optimisation possible (only inside the code)

- 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>Stateless</session-type>
    <transaction-type>Container</transaction-type>
   </session>
  </enterprise-beans>
  <assembly-descriptor> ... // See next slide <assembly-descriptor>
  </ejb-jar>
  ```
Distribution Specifics and Deployment

```
<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>
```
Spring Framework

- 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

![Diagram](image-url)

**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
Main Concept - Dependency Injection

- OrderProcessing tightly coupled to payment method
  - code is hard to reuse and test
  - payment method can’t 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();
    }
}
```
- Objects are given their dependencies at creation time by some third party
- Application logic code generally has no dependencies to the spring framework itself

```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();
    }
}
```
  <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> Create CreditCardPayment

Injecting a Dependency with Spring
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
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 – Open Service Gateway Initiative

specification of an open, standardised, component-based service platform

Initial Application Domains
- Home and building automation
- Telematic platform

Increasing number of application areas
(e.g., mobile, automotive, desktop or server-side applications)

OSGi Platforms:
- Open Source: Equinox (Eclipse’s OSGi console), Apache Felix, Knopflerfish
- Commercial: Embedded Server (Prosyst), Service Management Framework (IBM)
- 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
    - Provision of basic services
    - Lightweight service registry
    - Support of remote service (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 → RESOLVED

### Diagram

- **Framework**
  - `org.osgi.framework`
  - `org.osgi.service.http`

- **Bundle A**
  - Export: `org.osgi.service.log`
  - `com.ibm.service.log`
  - `com.ibm.j9`
  - Import: `org.osgi.service.http`
    - `javax.servlet.http`

- **Bundle B**
  - Export: `ericsson.osgi`
    - `javax.servlet`
    - `javax.servlet.http`
  - Import: `org.osgi.service.log`
    - `org.osgi.service.http`

### Code Snippet

```java
public class BundleA {
    public static void main(String[] args) {
        // Initialize Bundle A
    }
}
```
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
	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.
    }
}
Bundles can provide different services

many services already given by OSGi-Framework; e.g.,
- Log Service (message logging)
- HTTP Service
- Service Tracker
- XML Parser

different possibilities to use services
- direct creation of service-instance
- obtain service from service registry via a service reference
- Implementation of a service tracker to handle the dynamics of a bundle/service lifecycle
OSGi Service Registry

1. **create instance**
   - ProductSearchBundle
     (Bundle providing the service ProductSearch)

2. **register Service**
   - Activator
     - ServiceRegistration
       (getReference, setProperties, unregister)

3. **Activator**
   - ProductServiceImpl
     - getProperty()

4. **Activator of bundle using a service**
   - ServiceReference
   - getServiceReference
   - getService(reference)

---

**Service Registry**

<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>ProductServiceImpl</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 (by bundle) via service tracker

```java
public class Activator implements BundleActivator {
    public SearchServiceTracker tracker;
    ...
    public void start(BundleContext bc) throws Exception {
        tracker=new SearchServiceTracker(bc);
        tracker.open();
    }
    ...
}
```
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) {
        ...
    }
}
remote access to services possible since version 4.2.

- entities (service provider or consumer) determine, if they can be accessed remotely

- **Distribution Provider**
  - management of communication between service provider and consumer, support of different communication protocols
  - use of discovery to find service provider
  - creates protocol specific endpoint for remote service
  - Transparency → bundles don’t know if service is local or remote

- **Discovery**
  - publishing of services
  - adding of information: service description, communication data (URL, protocol, etc.)
Distributed OSGi - interaction

- Provider Bundle
  - Booking Service
  - Distribution Provider
    - Booking Endpoint
      - announces
  - Discovery Service
  - register Service
  - listens for registrations of remote services

- Consumer Bundle
  - Booking Client
  - Booking Proxy
    - calls
  - Discovery Client
    - Communication protocol, e.g. JMS, WS*
    - searches
  - Service Registry
    - requests service
    - listens for (unsuccessful) service requests

- OSGi Container
- Discovery Protocol, e.g. UDDI

Register Service
Listens for registrations of remote services
Calls
Announces
Discovery Service

Schill, Springer, Hara
Distributed Systems – Component Models and Platforms
38
OSGi - Evaluation

- **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
Summary

- different component platforms available to support component based development
- various types of middleware available to address different tasks of a distributed system
- characteristics of middleware can vary between particular products
- appropriateness of middleware depends on project requirements
- special challenges involved in debugging distributed systems
- interlocking of language concepts, architecture principles and tools important
References

- OSGi Alliance Homepage: http://www.osgi.org/
- OSGi Platform implementation Equinox: http://www.eclipse.org/equinox/