Aspects (AOP)
AspectJ
AOP

- Aspect-oriented programming
- „separation of concerns“
  - concern ~ a part of program code related to a particular functionality

- typically understood as an extension of OOP
- solves the problem that it is not always possible to put a code for a single functionality to a single (or several) classes
  - contrary, code is through the all application
Application modularity

XML parsing in Tomcat

URL manipulating in Tomcat

logging
AspectJ

- http://www.eclipse.org/aspectj/
- a Java extension
  - 1 concept – **joinpoint**
    - a place in a program for adding code
  - several constructs
    - **pointcut**
      - definition of joinpoint(s)
    - **advice**
      - code to be added
    - **inter-type declaration**
      - extending a class declaration
    - **aspect**
      - a “class” that can contain the above mentioned constructs
Pointcut

- call(void Point.setX(int))
- call(void Point.setX(int)) ||
  call(void Point.setY(int))
- call(void FigureElement.setXY(int,int)) ||
  call(void Point.setX(int)) || call(void Point.setY(int)) ||
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point))
- pointcut move():
  call(void FigureElement.setXY(int,int)) ||
  call(void Point.setX(int)) || call(void Point.setY(int)) ||
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));
- call(public * Figure.* (...)
Advice

- before(): move() {
    System.out.println("about to move");
}

- after() returning: move() {
    System.out.println("just successfully moved");
}
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}

Aspect

aspect PointObserving {
    private Vector Point.observers = new Vector();
    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    pointcut changes(Point p): target(p) && call(void Point.set*(int));
    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }
}

static void updateObserver(Point p, Screen s) {
    s.display(p);
}
Aspect

- aspect SimpleTracing {
  pointcut tracedCall():
      call(void FigureElement.draw(GraphicsContext));

  before(): tracedCall() {
      System.out.println("Entering: " + thisJoinPoint);
  }
}

- aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;

  before(): call(void Line.rotate(double)) {
      rotateCount++;
  }

  before(): call(void Point.set*(int))
      && cflow(call(void Line.rotate(double))) {
      setCount++;
  }
}
Aspects can be defined directly in Java via annotations:

```java
@Aspect
public class Foo {

    @Pointcut("call(* *.*(..))")
    void anyCall() {}

    @Before("call(* org.aspectprogrammer..*(*(..)) && this(Foo))")
    public void callFromFoo() {
    }
}
```
JAVA

JEE
Java Enterprise Edition
Overview

JEE

JSE

JME

Java Card API
“Enterprise” applications

- “big enterprise” applications
- required features
  - re-usability
  - loosely coupled
  - transactions
  - declarative interface
  - persistence
  - security
  - distributed applications
  - ...
3-tier architecture

**Presentation tier**
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

**Logic tier**
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

**Data tier**
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

EJB
(first, briefly EJB 2, i.e. old EJB)
Overview

- Enterprise Java Beans
- components
- runs in a server
  - an EJB container
- local and remote access
- the container offers many services
  - persistence
  - security
  - transactions
  - scalability
  - concurrency
• kinds of beans
  – session beans
    • stateless
      – no state kept
    • statefull
      – a state is kept
  – message-driven beans
  – entity beans
Session beans

- implement business logic
  - logic tier
- not persistent
Session beans

• 2 interfaces
  – business interface
    • extends EJBObject
    • declares provided methods
  – home interface
    • extends EJBHome
    • defines “create” methods

• implementation
  – a class extending SessionBean
  – has methods from the business interface
    but does not implement it!
  – must have following methods
    • public void ejbRemove()
    • public void ejbActivate()
    • public void ejbPassivate()
    • public void setSessionContext(SessionContext sc)
    • public void ejbCreate()
Entity beans

- access to persistent data (database)
  - data tier
- persistence
  - container managed
  - bean managed
Entity Beans

- 2 interfaces
  - home
    - create
    - findByPrimaryKey
  - business
    - get/set methods for data
- implementation
  - extends EntityBean
  - does not implement the business interface
  - get/set methods are abstract
Message-driven beans

- implementing the MessageListener interface
  - onMessage()
EJB container

- EJB container “ties” the interface and implementation
  - generates stubs and skeletons

Image source: B. Eckel: Thinking in Enterprise Java
Deployment

- deployment descriptor
- EAR
Java

Spring
Overview

- 2002
- critique of EJB
  - to complex
  - hard to be used
  - hard to be tested
  - RemoteException everywhere
  - …
- Rod Johnson: Expert One-on-One
  J2EE Design and Development
  - critique of EJB +
    proposal of a better architecture
  - Spring foundations
Overview

• Spring
  – http://www.spring.io/
  – based on POJO
    • plain old Java objects
    • but can be integrated with EJB
  – “lightweight” solution
    • the smallest possible dependency of application code on Spring
    • no server necessary
      – suitable for any type of application
  – effort for integration with other frameworks
    • not to “reinvent the wheel”
    • to use proven existing solutions
Spring Framework Runtime

Data Access/Integration
- JDBC
- OXM
- ORM
- JMS
- Transactions

Web
- WebSocket
- Servlet
- Web
- Portlet

AOP

Aspects

Instrumentation

Messaging

Core Container
- Beans
- Core
- Context
- SpEL

Test
Spring core

- the org.springframework.beans package
- an “inversion of control” container
  - Dependency Injection
  - Hollywood Principle: "Don't call me, I'll call you."

- objects are not interconnected in code but in a configuration file
- an object is not responsible for searching its dependencies
- dependencies are declared
  - a container “provides” them – sets them via setters
    - common naming conventions setXxx()
    - or via parameters of constructors
- no special requirements on objects
Spring core

- objects created via a “factory”
  - the interface
    org.springframework.beans.factory.BeanFactory
  - the most used factories
    - DefaultListableBeanFactory
public class nameBean {
    String name;

    public void setName(String a) {
        name = a;
    }

    public String getName() {
        return name;
    }
}

<bean id="bean1" class="nameBean">
    <property name="name">
        <value>Tom</value>
    </property>
</bean>

<bean id="bean" class="beanImpl">
    <property name="conn">
        <ref bean="bean2"/>
    </property>
</bean>

<bean id="bean2" class="bean2impl"/>

- interconnecting objects
Spring and data tier

- anything can be used
  - JDBC
  - ORM
    - Hibernate
    - ...
- can be used separately
  - simplified DB usage
  - unified exceptions
  - ...
JdbcTemplate template = new JdbcTemplate(dataSource);
List names = template.query("SELECT USER.NAME FROM USER",
    new RowMapper() {
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            return rs.getString(1);
        }
    });

int youngUserCount = template.queryForInt("SELECT COUNT(0) FROM USER WHERE USER.AGE < ?", new Object[] { new Integer(25) });

class UserQuery extends MappingSqlQuery {
    public UserQuery(DataSource datasource) {
        super(datasource, "SELECT * FROM PUB_USER_ADDRESS WHERE USER_ID = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    protected Object mapRow(ResultSet rs, int rownum) throws SQLException{
        User user = new User();
        user.setId(rs.getLong("USER_ID"));  user.setForename(rs.getString("FORENAME"));
        return user; }
    public User findUser(long id) { return (User) findObject(id); }

    User user = userQuery.findUser(25);
Spring AOP

- implemented in plain Java
  - can be integrate with AspectJ
- intended for functionality for which aspects are ideal
  - originally for adding JEE services to Spring
    - transactions
    - logging
    - ...
Other Spring parts

- Spring MVC
  - a web MVC framework
  - inspired by the Struts framework
  - does not prescribe what should be used for generating pages
    - JSP
    - template systems (Velocity,...)
    - ...

- EJB
  - instead of POJO, EJBs can be used

- ...

Spring Roo

- framework easy generation of enterprise applications
  - roughly
  creation of an application using a “wizard” in several steps
Overview

- inspired by Springem
- instead of implementing interfaces, annotations are used
- using “dependency injection”
- no need to use descriptors
- …
- entity beans replaced by Java Persistence API
  - “mapping” classes to tables in relational database
  - JPQL query language
    - “SQL over objects”
Session bean – example

@Remote
public interface Converter {
    public BigDecimal dollarToYen(BigDecimal dollars);
}

@Stateless
public class ConverterBean implements converter.ejb.Converter {
    private BigDecimal euroRate = new BigDecimal("0.0070");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
@MessageDriven(mappedName="MDBQueue")
public class MDB implements MessageListener {
    public void onMessage(Message msg) {
        System.out.println("Got message!");
    }
}
@Entity
@Table(name = "phonebook")
public class PhoneBook implements Serializable {
    @Column(name="number") private String number;
    @Column(name="name") private String name;

    public PhoneBook() {}

    public PhoneBook(String name, String number) {
        this.name = name;
        this.number = number;
    }

    @Id public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number; }
}