Spring Introduction

Overview of the Spring framework

Outline

- Spring overview
- Dependency injection
- Aspect Oriented Programming
- Spring Container
- Spring landscape

Before Spring

- JavaBeans specification (1996.)
  - software component model for Java
  - primarily used as model for building UI widgets

- Enterprise JavaBeans (1998.)
  - server-side components
  - support for enterprise services:
    - transaction support, security, distributed computing

EJB Example - HelloWorldBean

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
public class HelloWorldBean implements SessionBean {
  public void ejbActivate() { }
  public void ejbPassivate() { }
  public void ejbRemove() { }
  public void setSessionContext(SessionContext ctx)
{
  }
  public void ejbCreate() {
  }

  public String sayHello() {
    return "Hello World";
  }
}
```

Spring goal

Spring simplifies Java development.

Spring Example - HelloWorldBean

```
public class HelloWorldBean{
  public String sayHello() {
    return "Hello World";
  }
}
```

Spring goal

- Lightweight POJO-based development
- Declarative programming model
- Based on two programming techniques
  - Dependency Injection
  - Aspect Oriented Programming

Key strategies

- Lightweight and **minimally invasive development** with POJOs
- **Loose coupling** through **dependency injection** and **interface orientation**
- **Declarative programming** model through **aspects** and **common conventions**
- **Boilerplate reduction** through aspects and templates

Dependency injection

- Traditionally
  - each object is responsible to **obtain its own reference** to the objects it collaborates with
  - highly coupled and hard-to-test code
- DI approach
  - objects are given their dependencies on creation time by some **third party that coordinates** each object in the system

JavaProgrammer can only execute Java programming tasks

```
public class JavaProgrammer implements Programmer{
  private JavaProgrammingTask task;
  public JavaProgrammer(){
    task = new JavaProgrammingTask();
  }
  public void executeTask(){
    task.execute();
  }
}
```

How to test JavaProgrammer class?

- we need to check that *execute()* method is invoked on task member each time *executeTask()* method is invoked on JavaProgrammer instance.
- impossible

GoodProgrammer is flexibile enough to execute any programming tasks

```
public class GoodProgrammer implements Programmer{
  private ProgrammingTask task;
  public Programmer(ProgrammingTask task){
    this.task = task;
  }
  public void executeTask(){
    task.execute();
  }
}
```

Task is injected!!!

How to test GoodProgrammer class?

- by mocking Programming task object

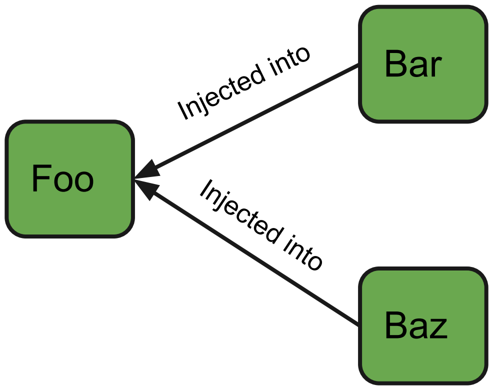
To test GoodProgrammer, we will inject it with a mock ProgrammingTask

```
import static org.mockito.Mockito.*;

public class GoodProgrammerTest {
  public static void main(String[] args){
    ProgrammingTask mockTask = mock(JavaProgrammingTask.class);
    GoodProgrammer programmer = new GoodProgrammer(mockTask);
    programmer.executeTask();
    verify(mockTask, times(1)).execute();
  }
}
```

Dependency injection

- Loose coupling
  - Dependencies defined using **interfaces**
  - Depending object is implementation agnostic
- Easy-to-test
  - injecting mock implementation of dependencies



Dependency injection

- Spring ***application context***
  - loads bean definitions
  - wires them together
- XML based configuration
- Annotation based configuration

Spring application context example

```
<?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="programmer" class="hr.calyx.training.programmer.GoodProgrammer">
          <constructor-arg name="task" ref="task"/>
     </bean>
     <bean id="task" class="hr.calyx.training.programmer.task.JavaProgrammingTask"/>
</beans>
```

DIExample loads the application context containing a programmer.

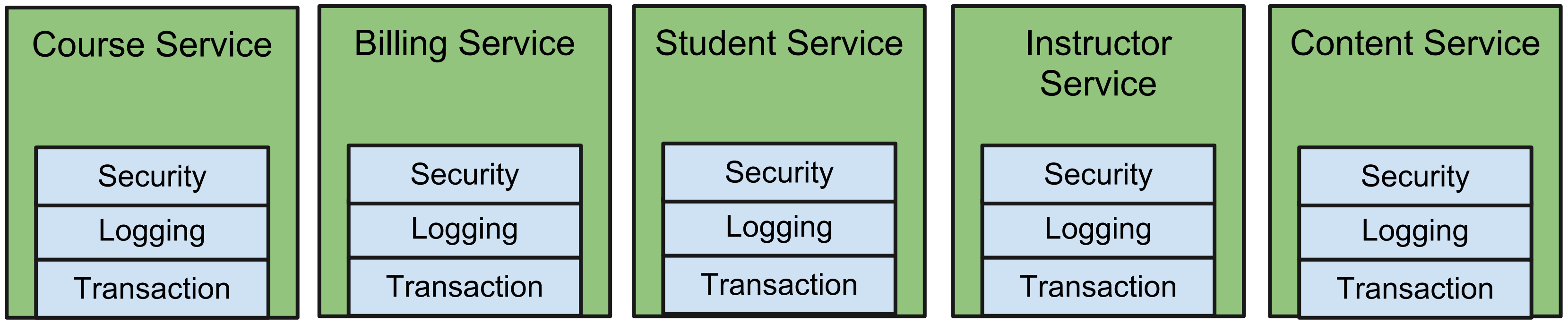
```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class DIExample {
  public static void main(String[] args){
    ApplicationContext context = new ClassPathXmlApplicationContext("programmer.xml");
    Programmer programmer = (Programmer)context.getBean("programmer");
    programmer.executeTask();
  }
}
```

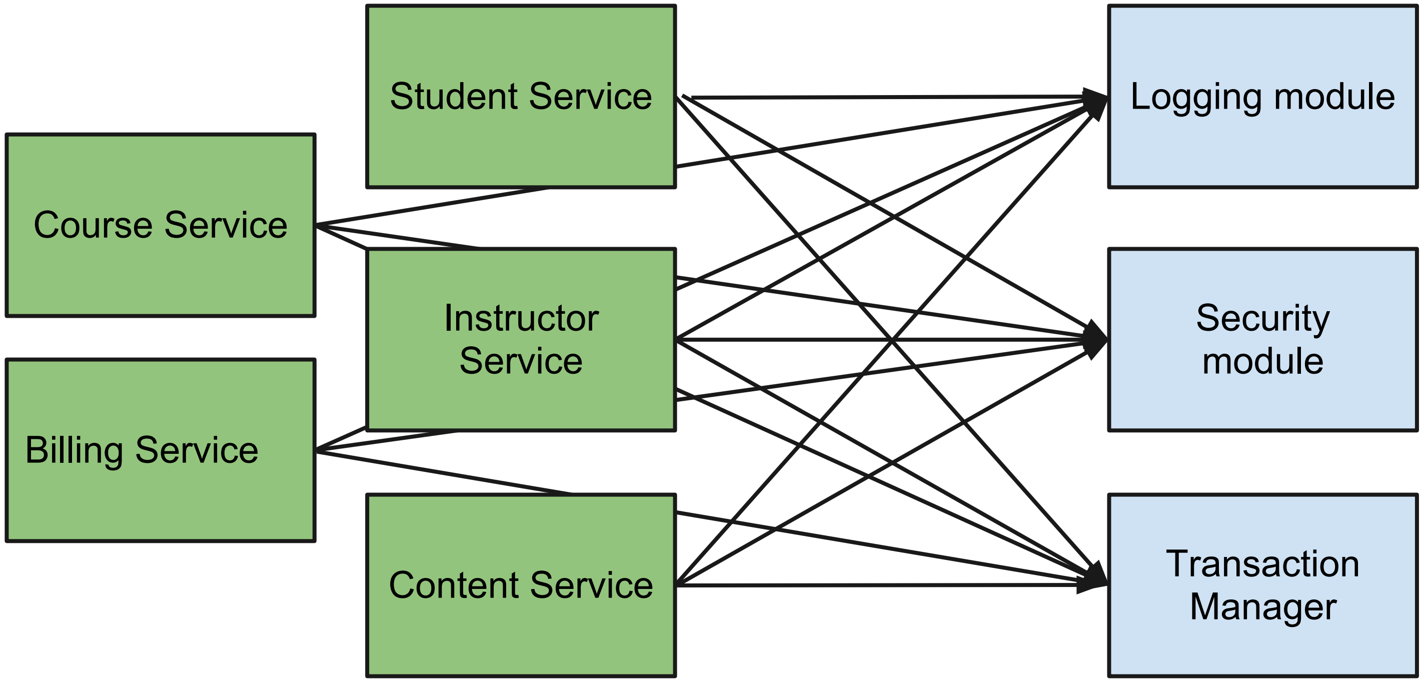
Aspect Oriented Programming

- capture ***cross-cutting concerns*** in reusable components
  - logging, transaction management, security...
- without AOP
  - concerns are duplicated across multiple components
  - components are littered with code that is not aligned with their core functionality
- with AOP
  - modularize common concerns
  - apply to the components that they should affect

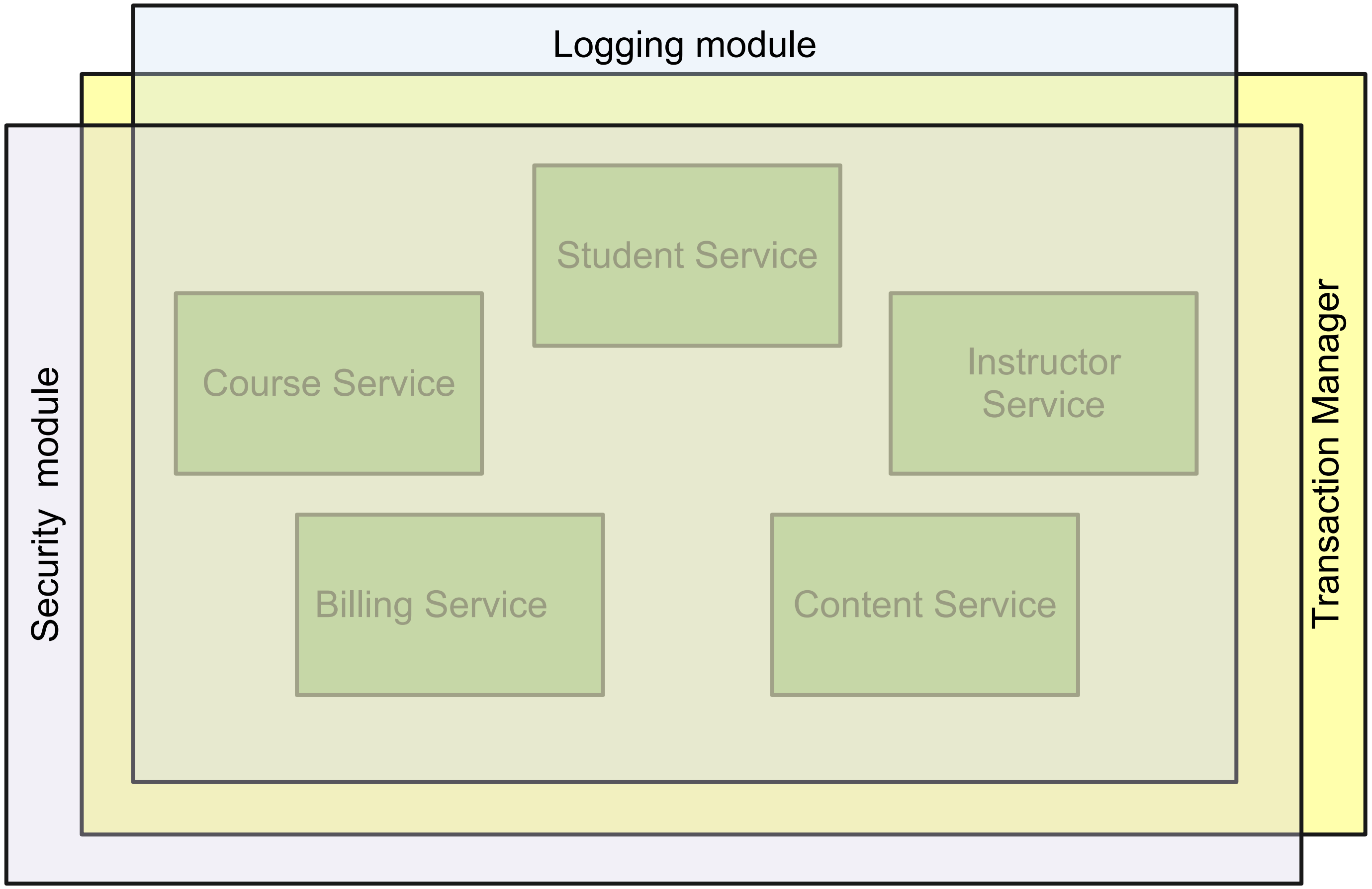
Duplicated cross-cutting concerns



Abstracted cross-cutting concerns



Aspect Oriented Programming



ProjectManager - defines budget and generates invoice for each programming task executed

```
public class ProjectManager implements Manager{
      private ProgrammingTask task;
      public ProjectManager(ProgrammingTask task){
            this.task = task;
      }
      @Override
      public void defineBudget() {
            System.out.println("Defining budget for task...");
      }
      @Override
      public void sendInvoice() {
            System.out.println("Sending invoice to customer...");
      }
}
```

Application context with AOP config file

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

  <bean id="task" class="hr.calyx.training.programmer.task.JavaProgrammingTask" />
  <bean id="programmer" class="hr.calyx.training.programmer.GoodProgrammer">
    <constructor-arg name="task" ref="task" />
    <constructor-arg index="1" value="3" />
  </bean>
  <bean id="manager" class="hr.calyx.training.pm.ProjectManager">
    <constructor-arg name="task" ref="task" />
  </bean>
  <aop:config>
    <aop:aspect ref="manager">
      <aop:pointcut expression="execution(* *.executeTask(..))" id="programerTask" />
      <aop:before pointcut-ref="programerTask" method="defineBudget" />
      <aop:after pointcut-ref="programerTask" method="sendInvoice" />
    </aop:aspect>
  </aop:config>
</beans>
```

Spring templates

- eliminate boilerplate code
  - JDBC, JMS, REST...
-
- Spring's approach
  - boilerplate code is encapsulated in templates
  - programmer can focus on application logic

JDBC API - an example of boilerplate code

```
public Employee getEmployeeById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement("select id, firstname, lastname, salary from employee where id=?");
    stmt.setLong(1, id);
    rs = stmt.executeQuery();
    Employee employee = null;
    if (rs.next()) {
      employee = new Employee();
      employee.setId(rs.getLong("id"));
      employee.setFirstName(rs.getString("firstname"));
      employee.setLastName(rs.getString("lastname"));
      employee.setSalary(rs.getBigDecimal("salary"));
    }
    return employee;
  } catch (SQLException e) {
  } finally {
    if(rs != null) {
      try {
        rs.close();
      } catch(SQLException e) {}
    }
    if(stmt != null) {
      try {
        stmt.close();
      } catch(SQLException e) {}
    }
    if(conn != null) {
      try {
        conn.close();
      } catch(SQLException e) {}
    }
```

Spring JdbcTemplate - focus on database operation

```
public Employee getEmployeeById(long id) {
  return jdbcTemplate.queryForObject(
    "select id, firstname, lastname, salary from employee where id=?",
    new RowMapper<Employee>() {
      public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
        Employee employee = new Employee();
        employee.setId(rs.getLong("id"));
        employee.setFirstName(rs.getString("firstname"));
        employee.setLastName(rs.getString("lastname"));
        employee.setSalary(rs.getBigDecimal("salary"));
        return employee;
      }
    },
    id);
```

Spring container

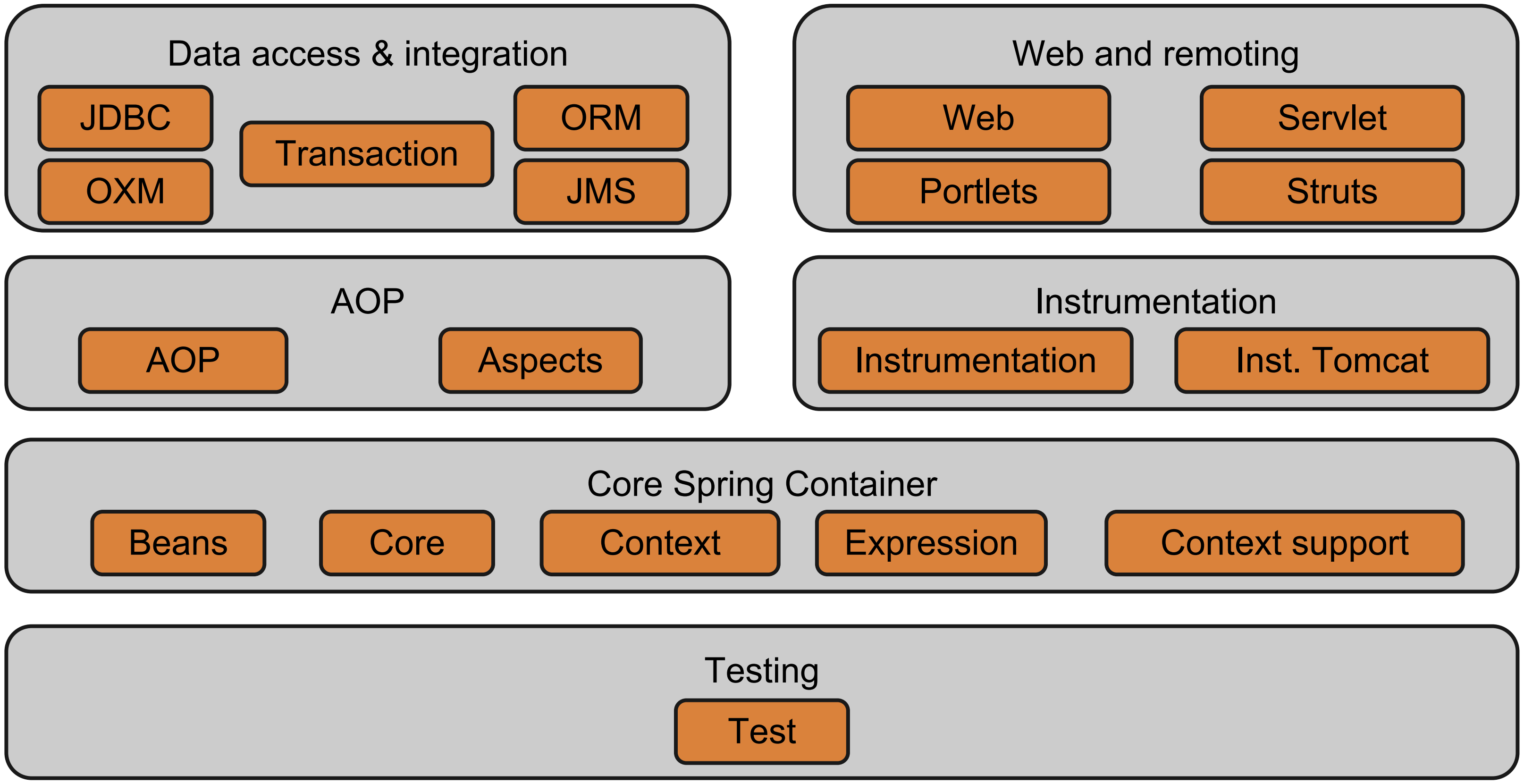
- all application objects live in the container
- manage application components:
  - creating objects
  - wiring them together
  - configuring them
  - managing objects' lifecycle
- two types of implementation:
  - ~~Bean factories~~
  - Application context

Application context

- ClassPathXmlApplicationContext
  - XML file located in the classpath
- FileSystemXmlApplicationContext
  - XML file located in the file system
- XmlWebApplicationContext
  - XML file contained within web application

Spring landscape

- Spring modules



Spring portfolio

- Spring Web Flow
- Spring Web Services
- Spring Security
- Spring Integration
- Spring Batch
- Spring Social
- Spring Mobile
- ...