

# 系统架构师培训

## 一应用架构设计

• 第一章：	企业应用架构基础	3
• 第二章：	表现层设计	30
• 第三章：	业务层设计	55
• 第四章：	数据访问层设计	107
• 第五章：	通用服务设计	137
• 第六章：	企业应用集成（EAI）	182
• 第七章：	面向服务架构（SOA）设计	195
• 第八章：	应用框架的设计与实现	224

# 第一章

## 企业应用架构基础

- 架构师的角色
  - 系统的规模
  - 系统的分布
- 架构满足风险管理的需要
  - 高层规划的目标：
    - 部分失效时系统的强健性
    - 处理请求负载
    - 并发使用的扩展能力

- 架构的功能
  - 技术职责
    - 标识对架构重要的用例
    - 指导架构原型的开发
  - 管理职责
    - 成本管理
      - 技术和风险转移的方法
    - 沟通管理
      - 与项目干系人和团队成员的有效合作的沟通技巧

	架构	设计
抽象级别	高层的、广泛的,很少关注细节	底层的、特定的,关注更多的细节
提交物	系统和子系统规划, 架构原型	组件设计、代码规范
关注点	非功能性需求、风险管理	功能性需求

- 面向对象的分析和设计职责
  - 基于组件设计的关键：
    - 抽象
    - 封装
    - 内聚
    - 耦合



- 系统架构师

- 可视化硬件和服务器的设计与实现
- 有数据库设计、容量规划、服务器集群、负载平衡及容错策略等方面的经验
- 提供支持**RAS**的部署环境
- 通常称为系统架构师或基础平台架构师



- 应用架构师

- 可视化应用软件和组件集成的设计和实现
- 有典型的业务应用、集成应用和 OO方法方面的经验
- 提供实现端到端功能并支持非功能性需求的应用结构

- 架构的关键点
  - 架构过程
  - 实现技术
  - 风险管理
  - 模式使用
  - 原型开发



- 创建满足QoS需求的蓝图
  - 典型的架构文档
    - 愿景文档
    - 需求规范
    - 风险识别和转移计划
    - 应用的域模型
    - 上下文环境描述
    - 项目计划
    - 假设列表

- 评估实现技术
  - 考虑技术决策点
  - 确保团队正确地使用了所选技术



- 识别及控制风险
  - 非功能性需求
    - 业务规则
    - 约束
    - 系统质量
  - 风险评估
  - 成本分析

- 使用适当的模式
  - 设计模式
    - 支持功能性需求
  - 架构模式
    - 支持非功能性需求



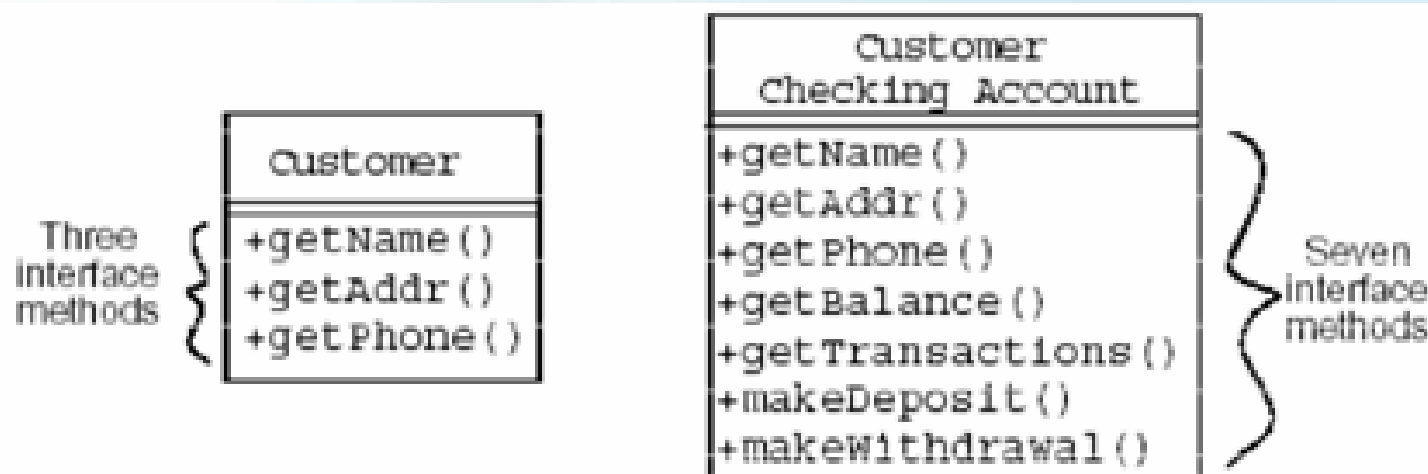
- 开发原型
  - 架构原型描述系统并按照经验确定计划是否得到满足
  - 包括：
    - 域模型
    - 交互图



- 识别关键的风险因素
  - 系统的灵活性
  - 网络通信和布局
  - 事务模型
  - 成本管理



- 面向对象设计的本质
  - 使用抽象定义边界
  - 限制接口的粒度



- 网络通讯的指南
  - 有效使用带宽：
    - 按大块发送数据，需要更少的来回往复
  - 最小化请求的频度：
    - 小心设计UI
    - 小心设计远程 API

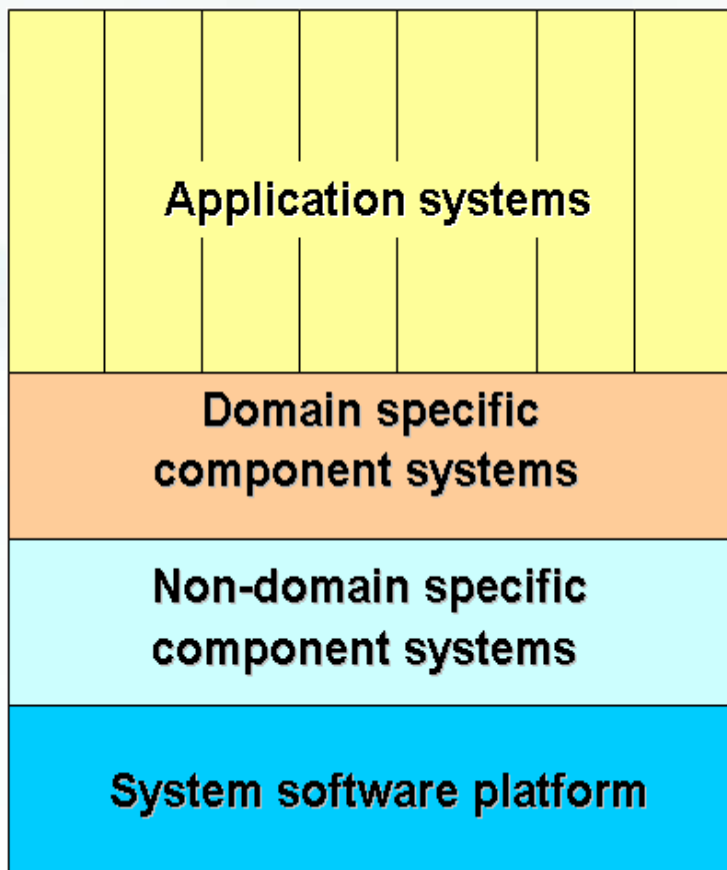
- 有效使用事务控制共享资源
  - 何时事务是必须的
  - 何时事务不是必须的
    - 尽可能避免事务
  - 事务模型的获得
    - 影响性能和吞吐量
    - 死锁情况

- 利用复用控制成本
  - 选择相关的模式



- **Layer模式**

- 确保抽象边界的定义和使用



- 各种特定的应用系统

- 通用组件,如**GUI**创建器、与**DBMS**的接口、操作系统服务、**ORB**, **OLE**组件等

- 操作系统、**DBMS**、**OLE**、基础类库等

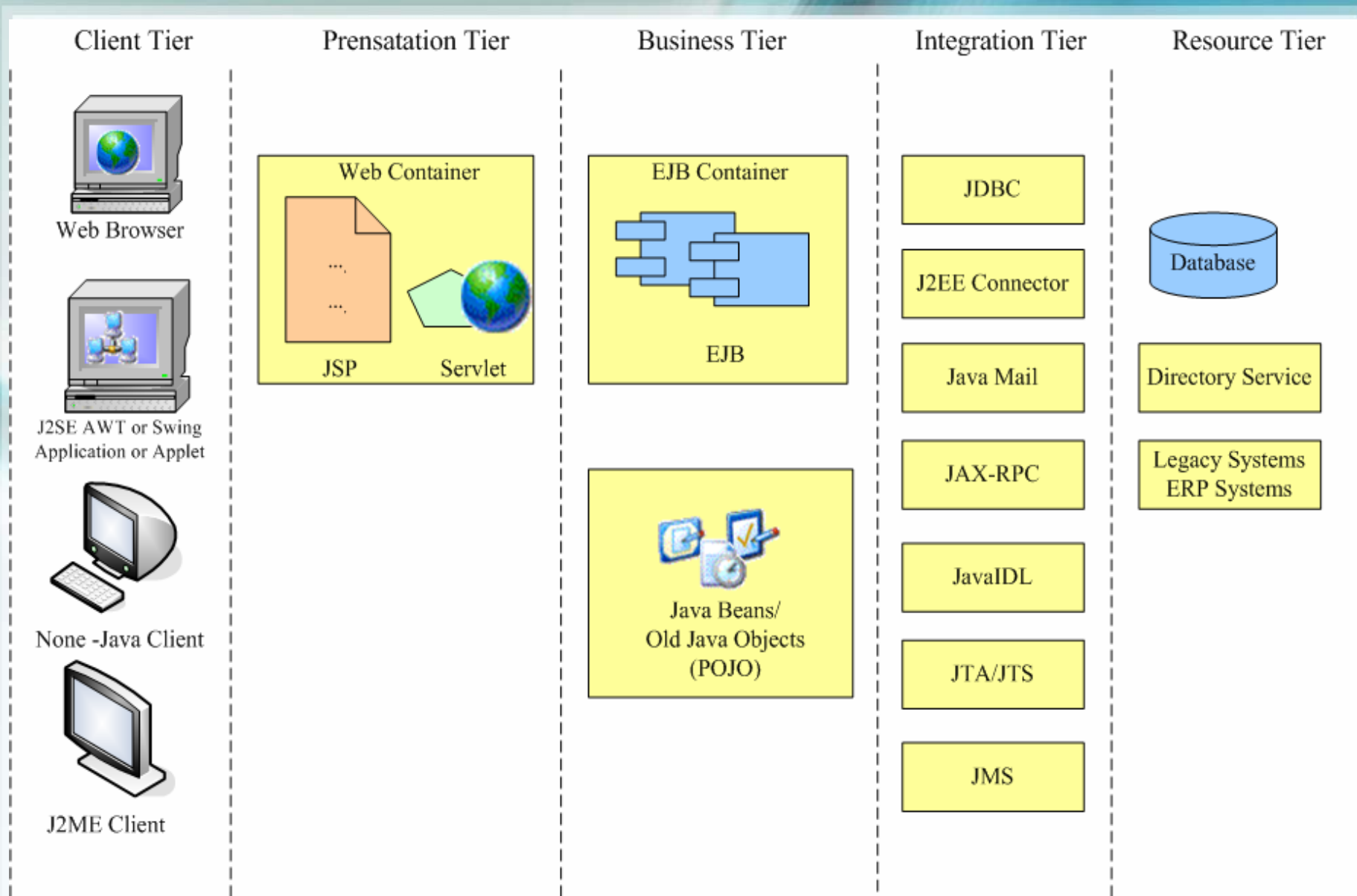
- Layer模式在 J2EE中的应用：
  - 应用程序
    - 提供满足功能需求的具体组件的实现
  - 虚拟平台
    - 提供应用程序组件实现的 API
  - 上层平台
    - 提供应用程序层次组件的基础架构设施
  - 下层平台
    - 提供支持以上层次的操作系统
  - 硬件平台
    - 提供支持以上层次所需的硬件

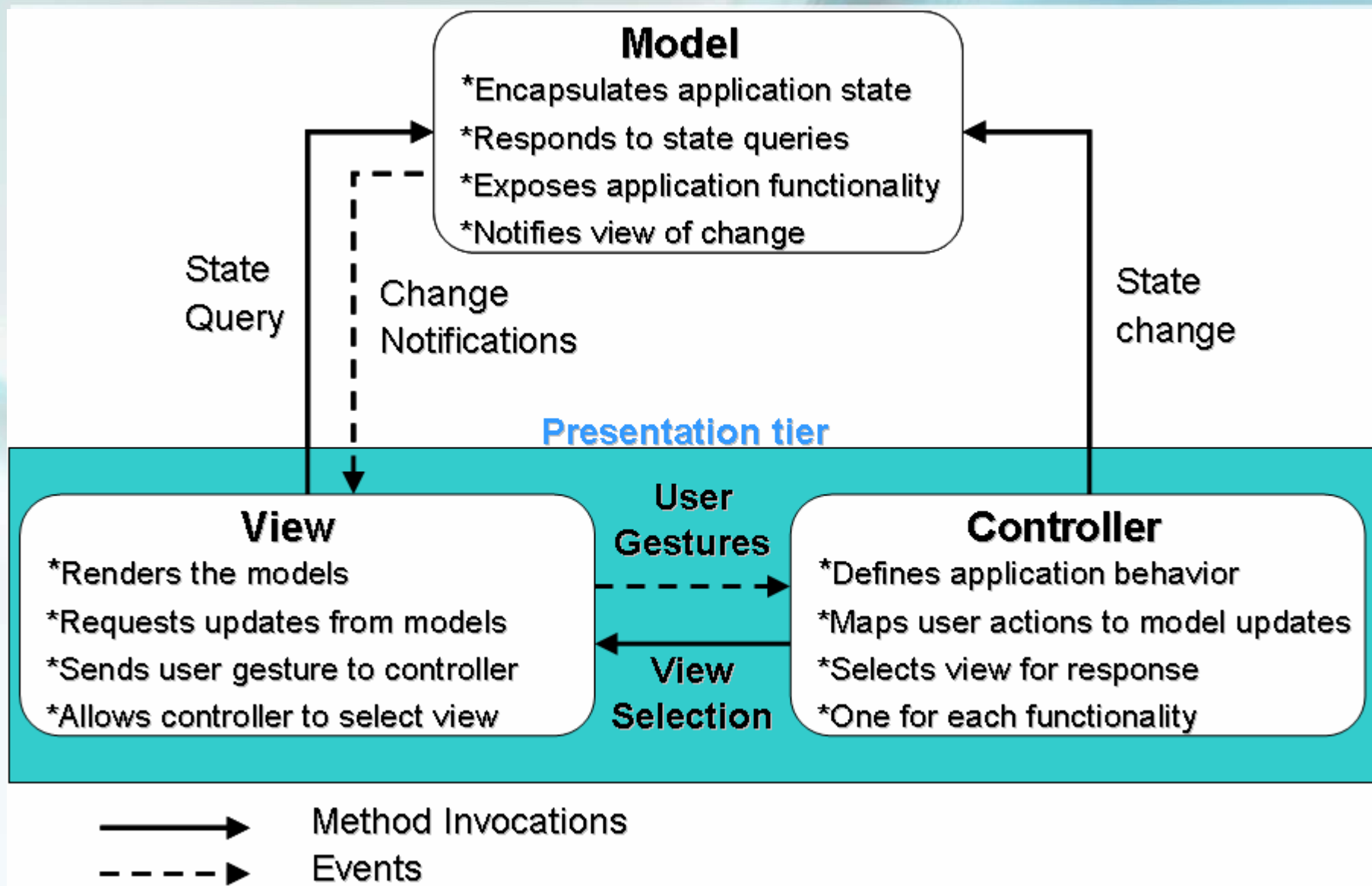


- Tier模式
  - 解決方案
    - 客户/服务器
    - 表现层/业务层/数据层
    - .....



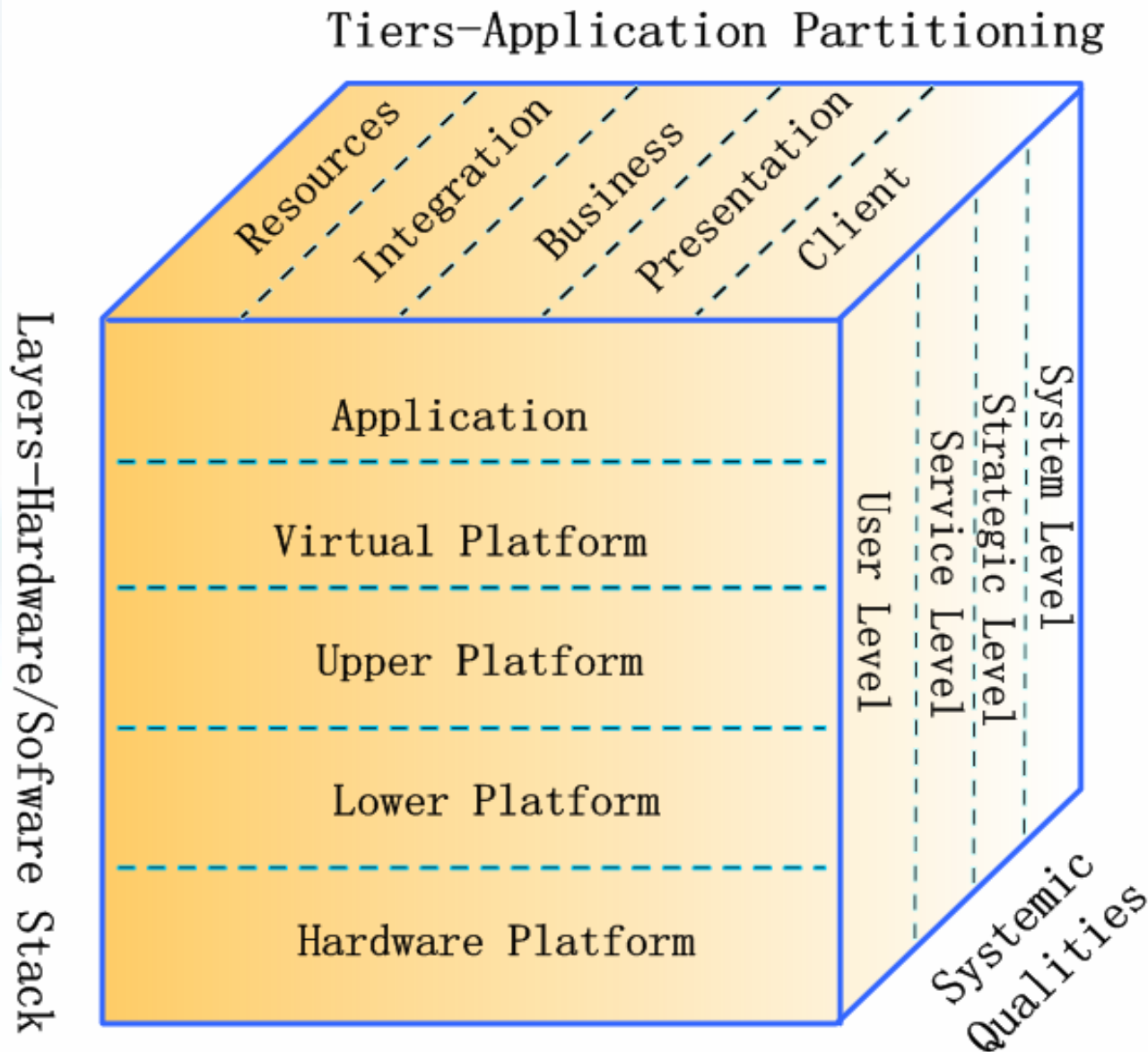
# J2EE中各Tier技术



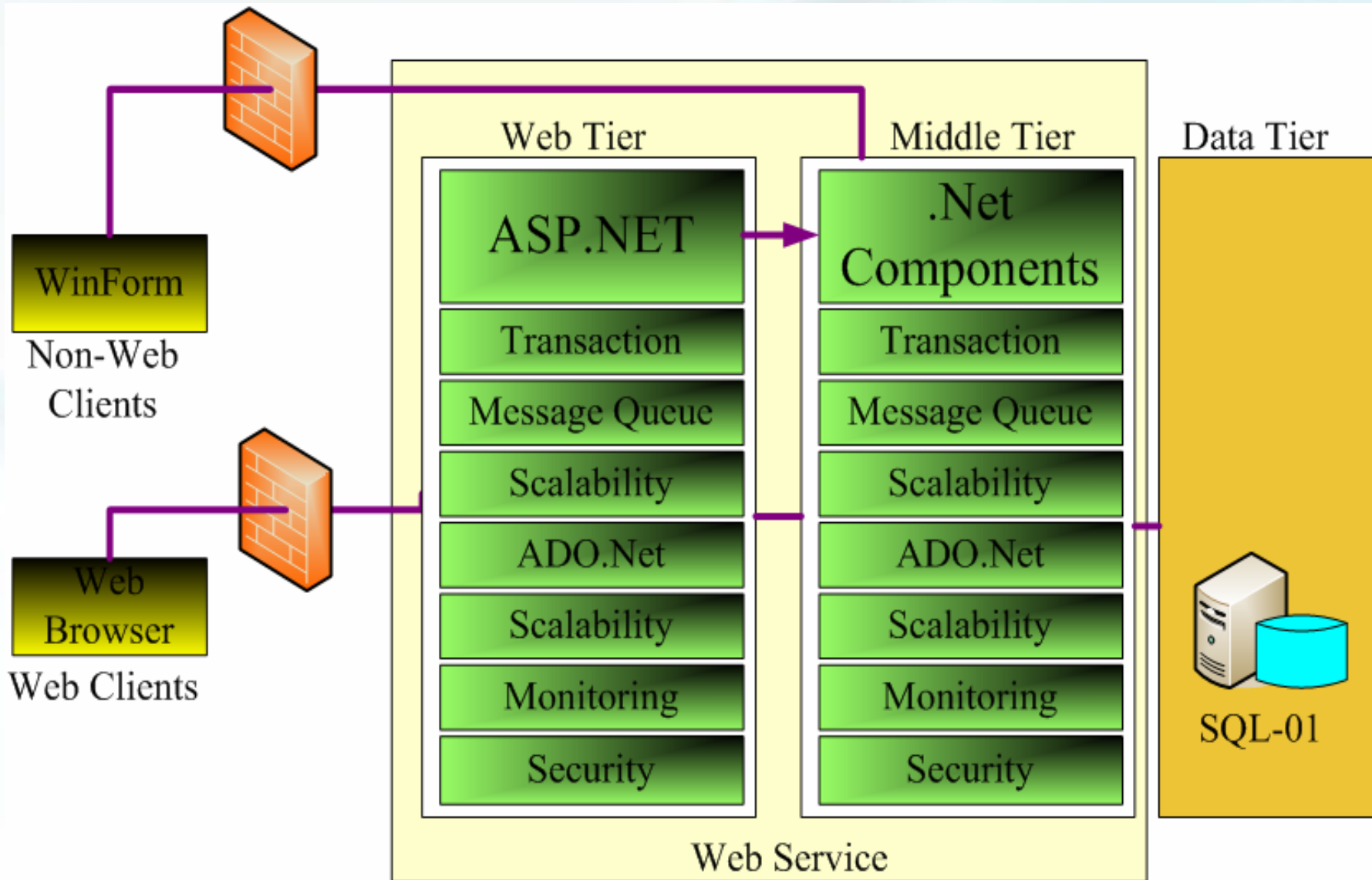


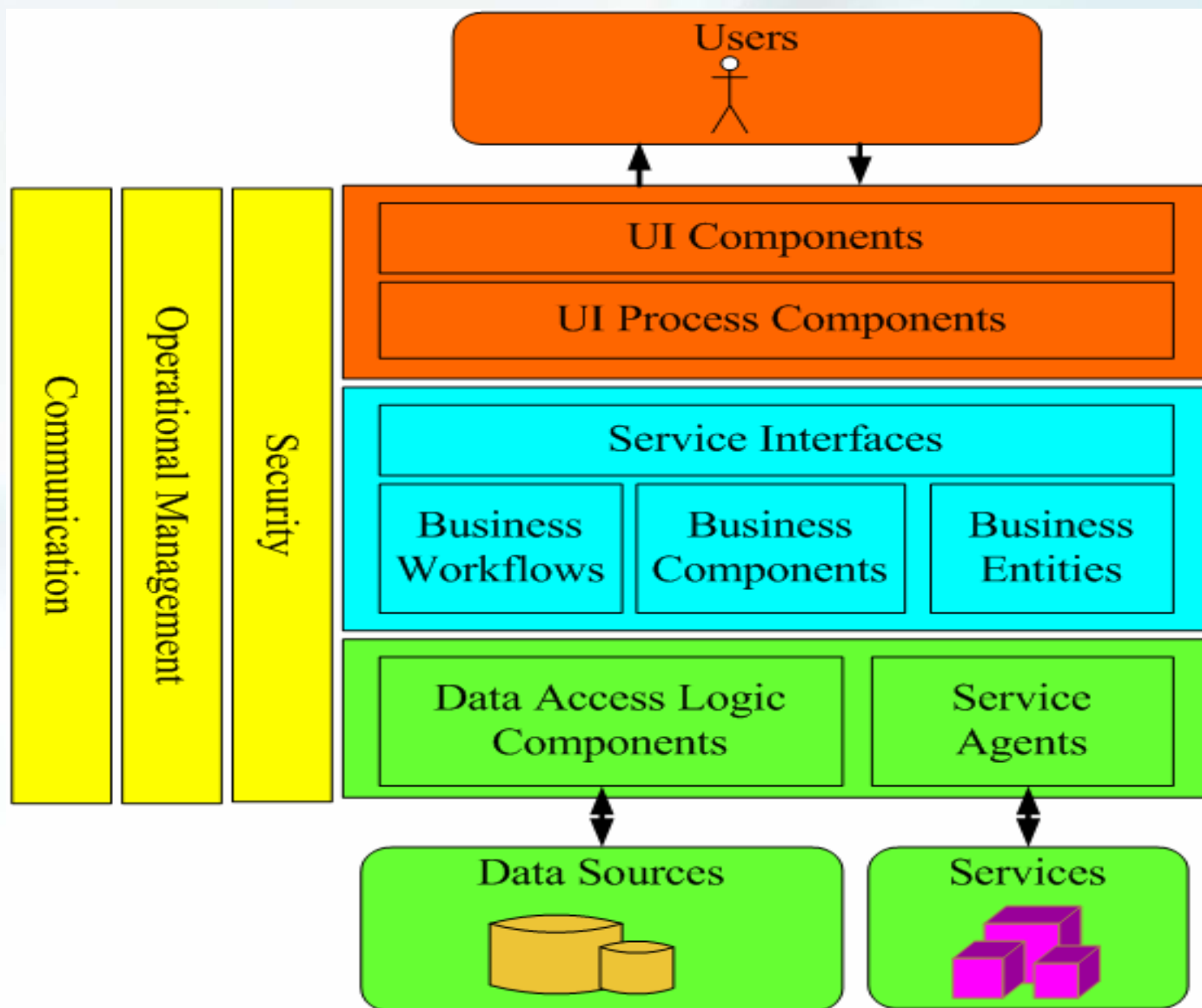
- 使用可靠的框架
  - 通过装配模式构建框架
  - 框架成为装配系统的模式
  - 框架应用到特定的问题域

# SunTone 3-D 架构框架



# .Net架构



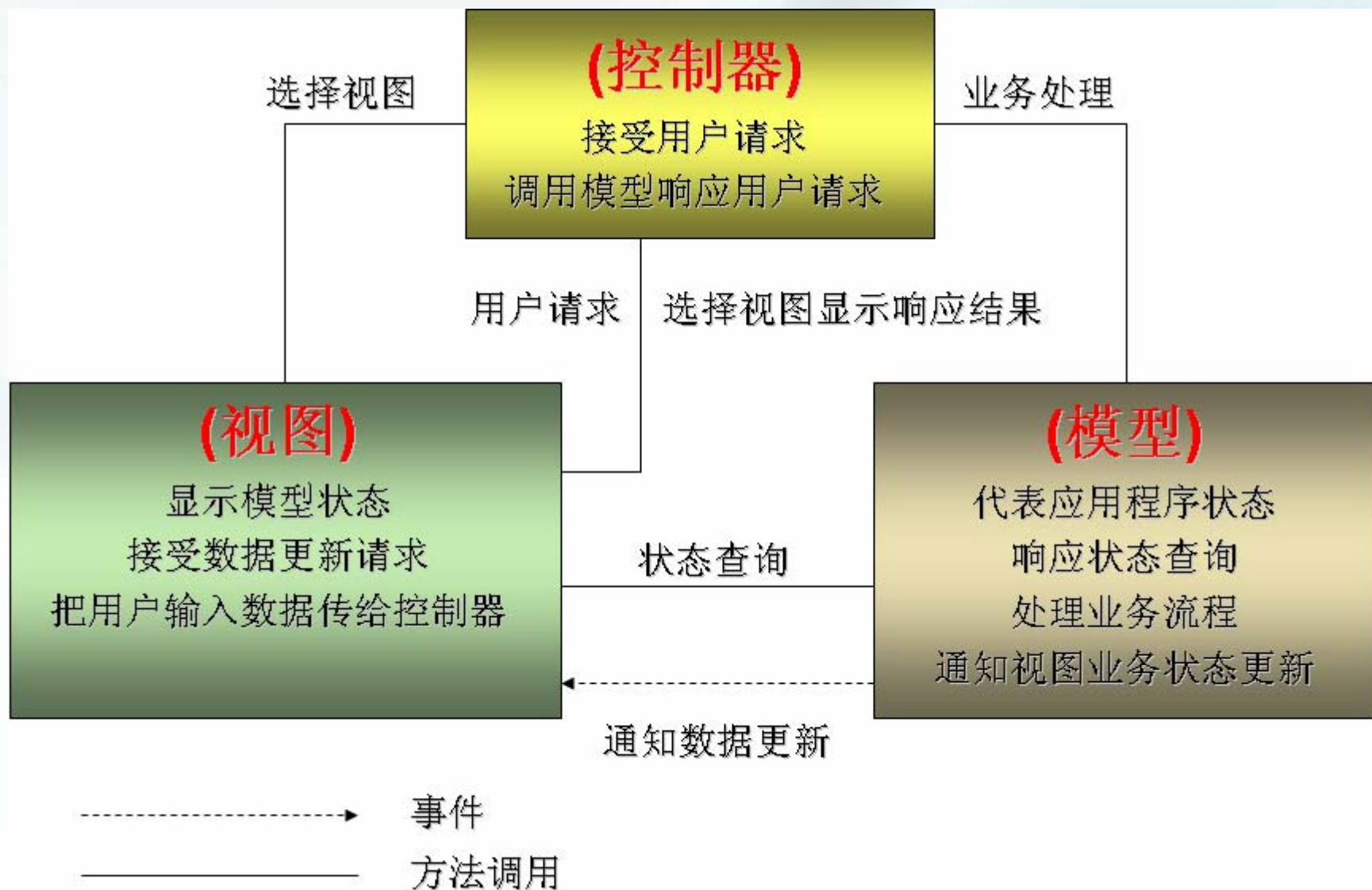




# 第二章

## 表现层设计

- 浏览器
  - 不同版本的浏览器对于HTML/DHTML的支持程度
  - 用户的系统安装了哪些组件
  - 应用是否需要访问用户的本地资源
  - HTML、DHTML、脚本语言 (Java、JavaScript或VBScript)、 CSS等
- 浏览器和 Web Server



- **Web控制器职责**

- 接收用户请求
- 获取请求参数
- 验证
- 根据用户的不同请求，调用对应的模型组件来执行相应的业务逻辑
- 获取业务逻辑执行结果
- 根据当前的状态数据及业务逻辑的处理结果，选择适合的视图组件返回给客户

- 拦截过滤器

- 问题:

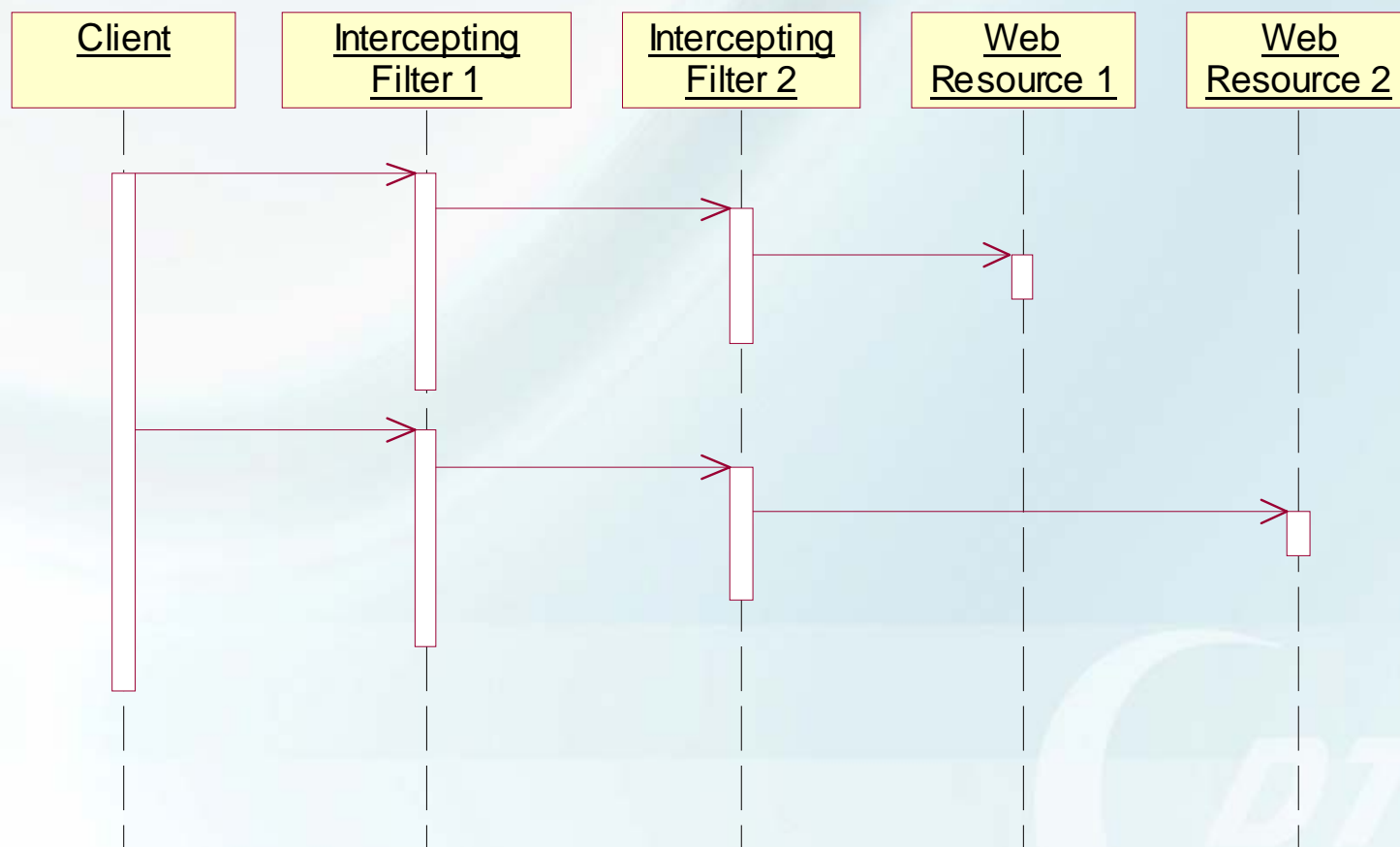
- 表现层的请求可能需要不同的处理
    - 某些请求可能需要预处理，而某些请求可能需要后续处理

## — 解决方案

- 创建可插入的过滤器以标准的方式处理通用服务，而不需要改变核心的请求处理代码
- 过滤器拦截输入的请求和输出的响应，以进行预处理或后续处理



## — 示例





- 前端控制器

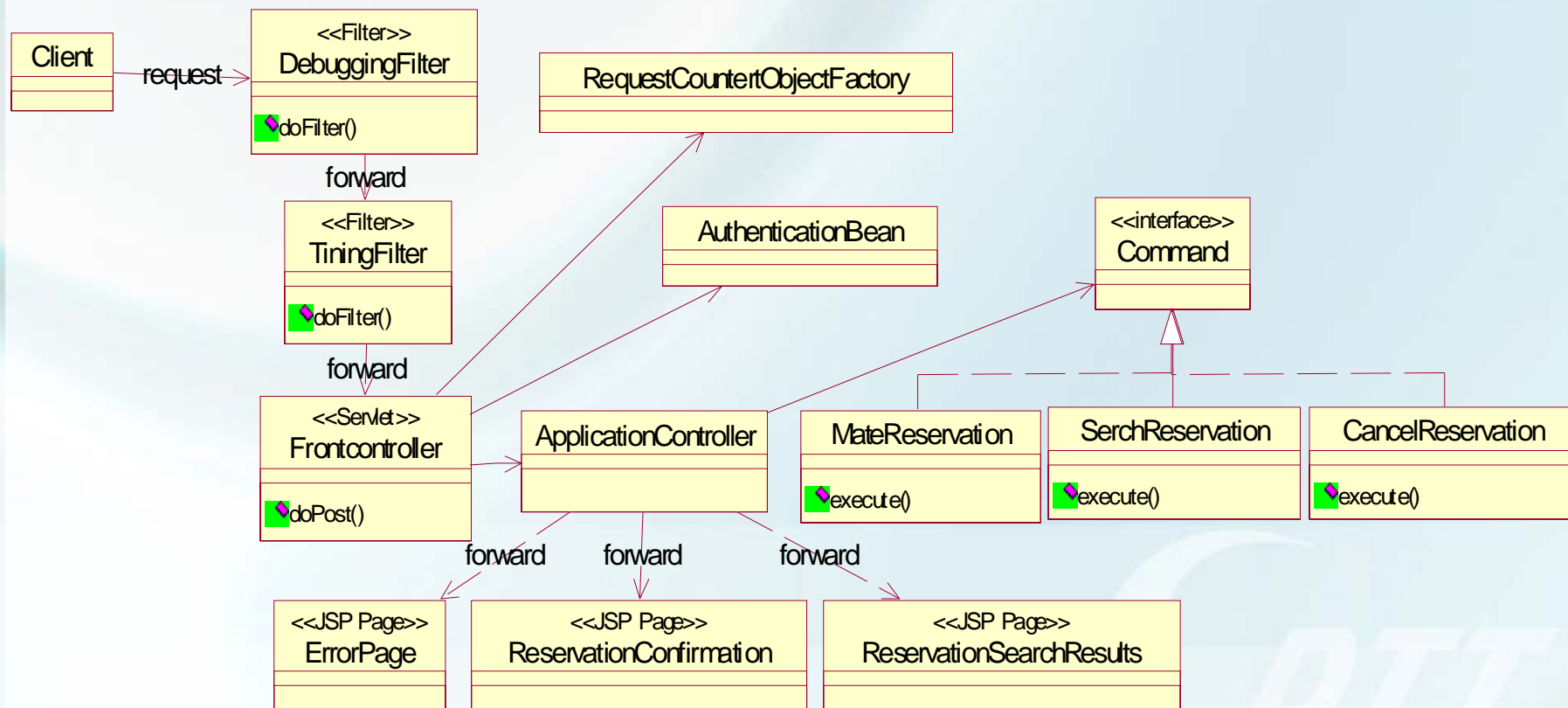
- 问题:

- 系统缺少一个集中处理请求的机制,会导致对每个请求都要完成的活动被随意地放在多个组件中
    - 通用的系统服务（如安全和审计）不应当在每个视图组中都重复

## — 解决方案

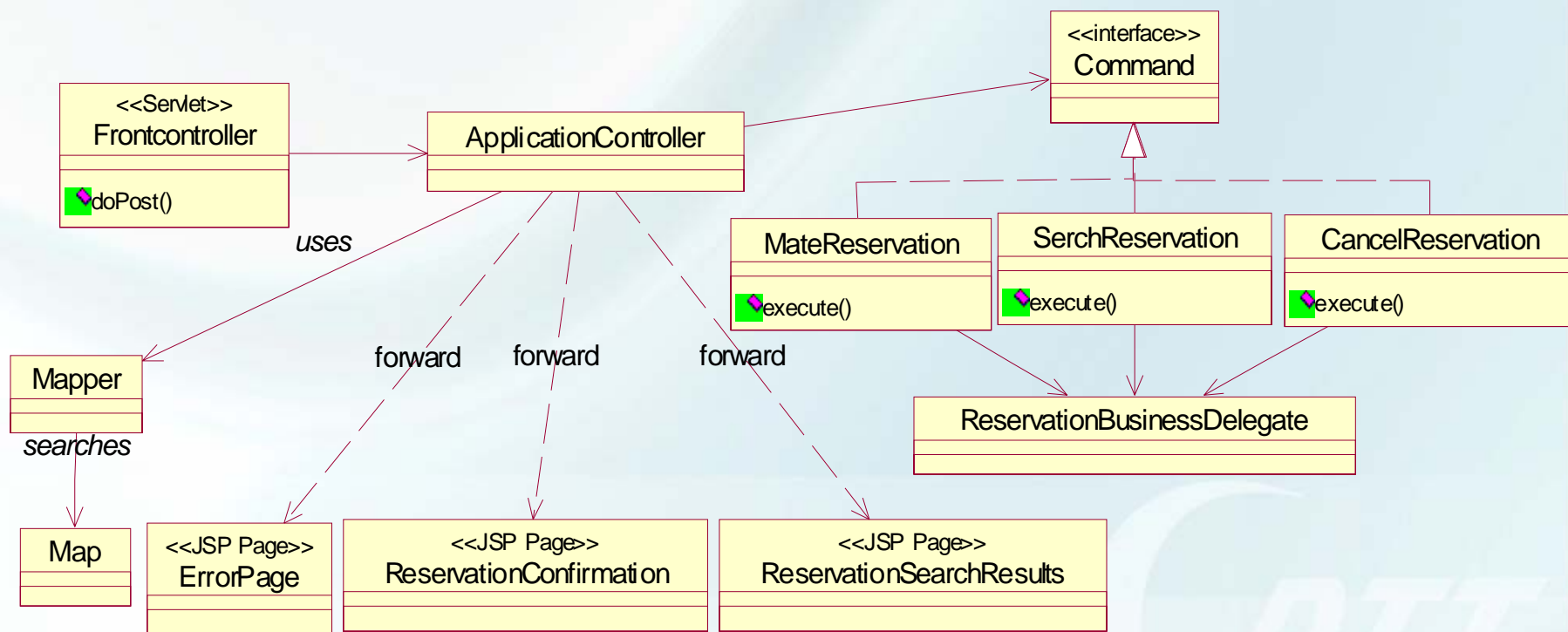
- 提供一个集中处理请求的点
  - 调用安全服务，如认证和授权
  - 代理业务处理
  - 管理相应的视图选择
  - 处理错误
  - 管理内容的创建策略

## - 示例



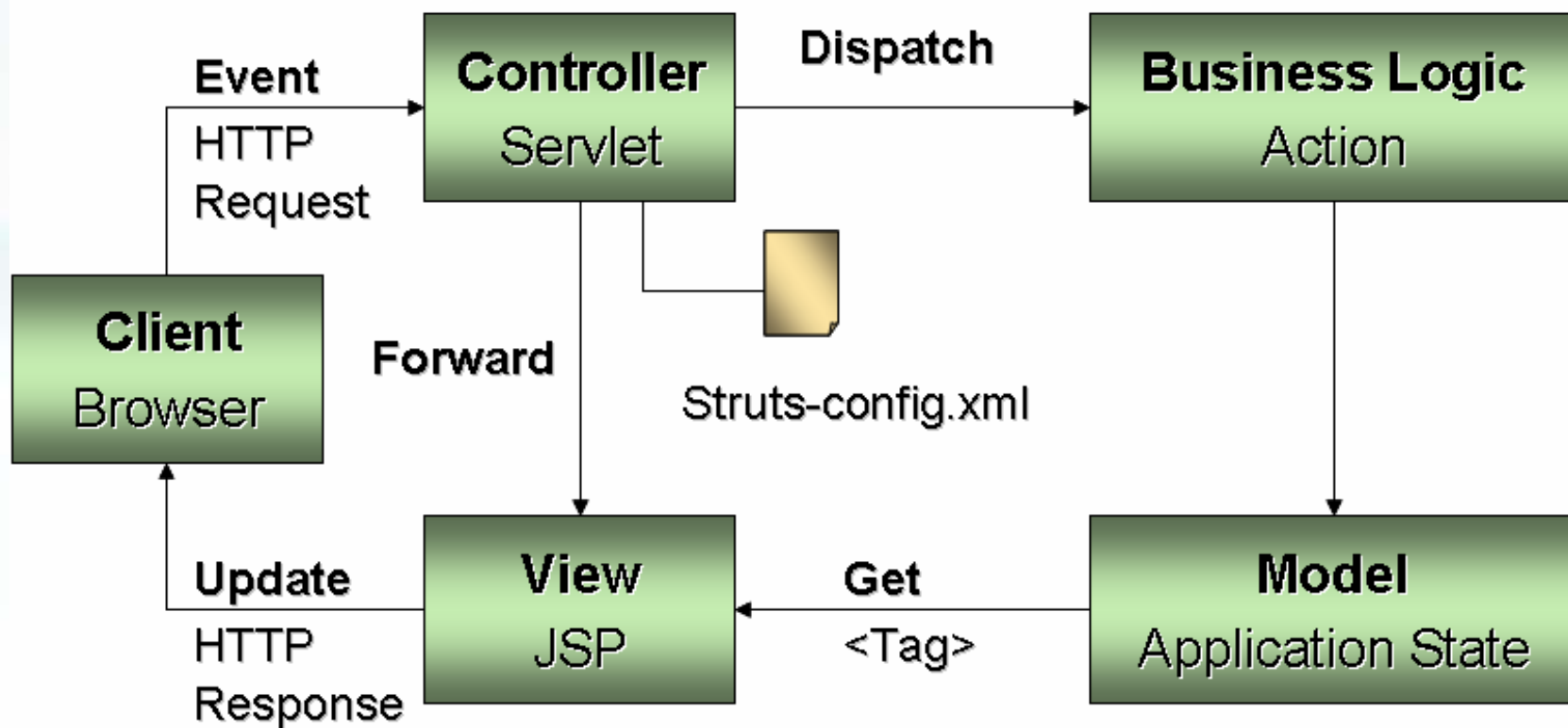
- 应用程序控制器：
  - 动作管理
    - 控制器决定要调用哪个动作
    - 该动作接下来会调用业务处理过程
  - 视图管理
    - 控制器决定将请求转发到哪个视图

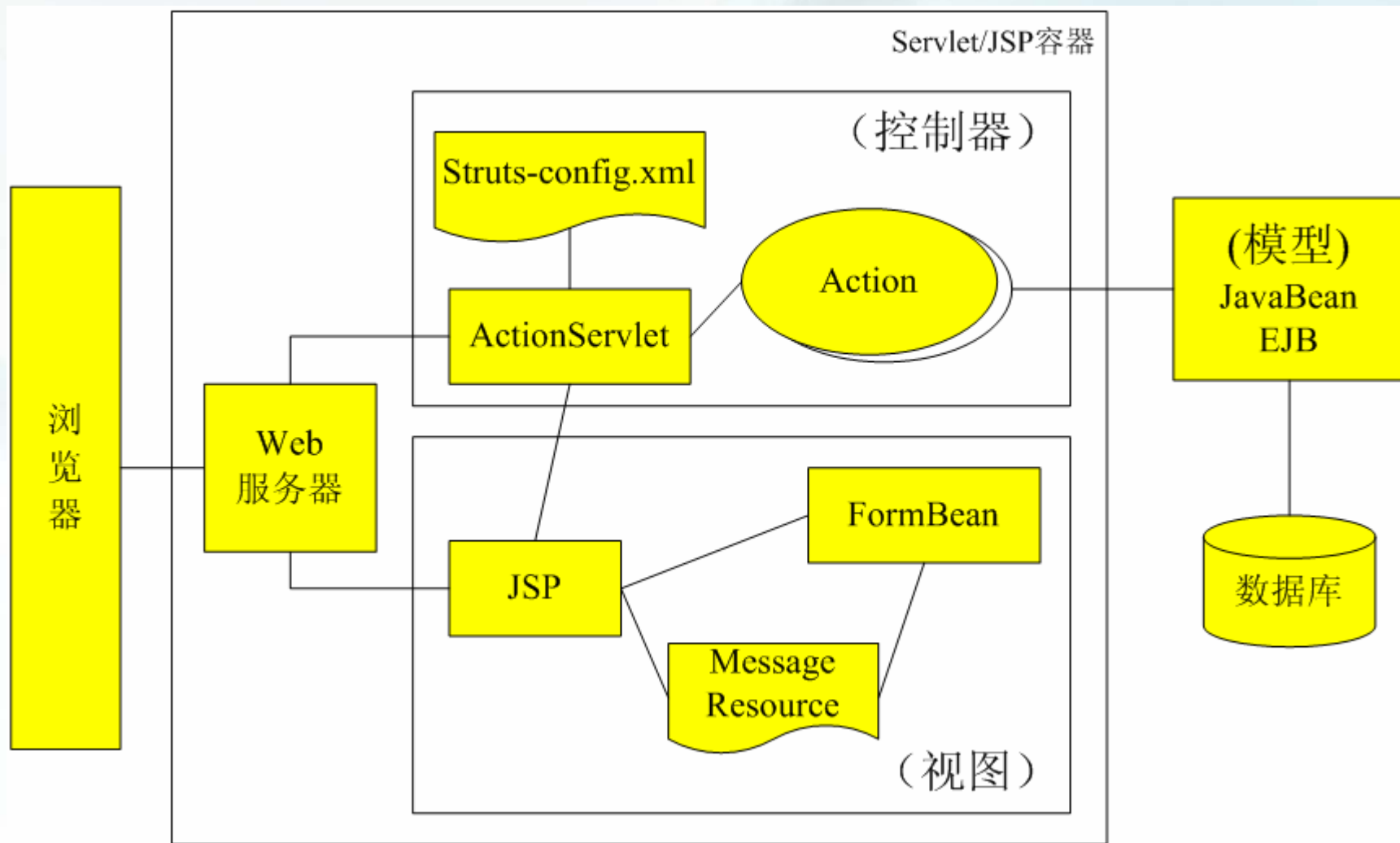
# - 示例一:



## — 示例二:

- Struts





- Struts 控制器

- Struts 框架重要的组件

- ActionServlet
    - RequestProcessor
    - ActionMapping
    - ActionForward

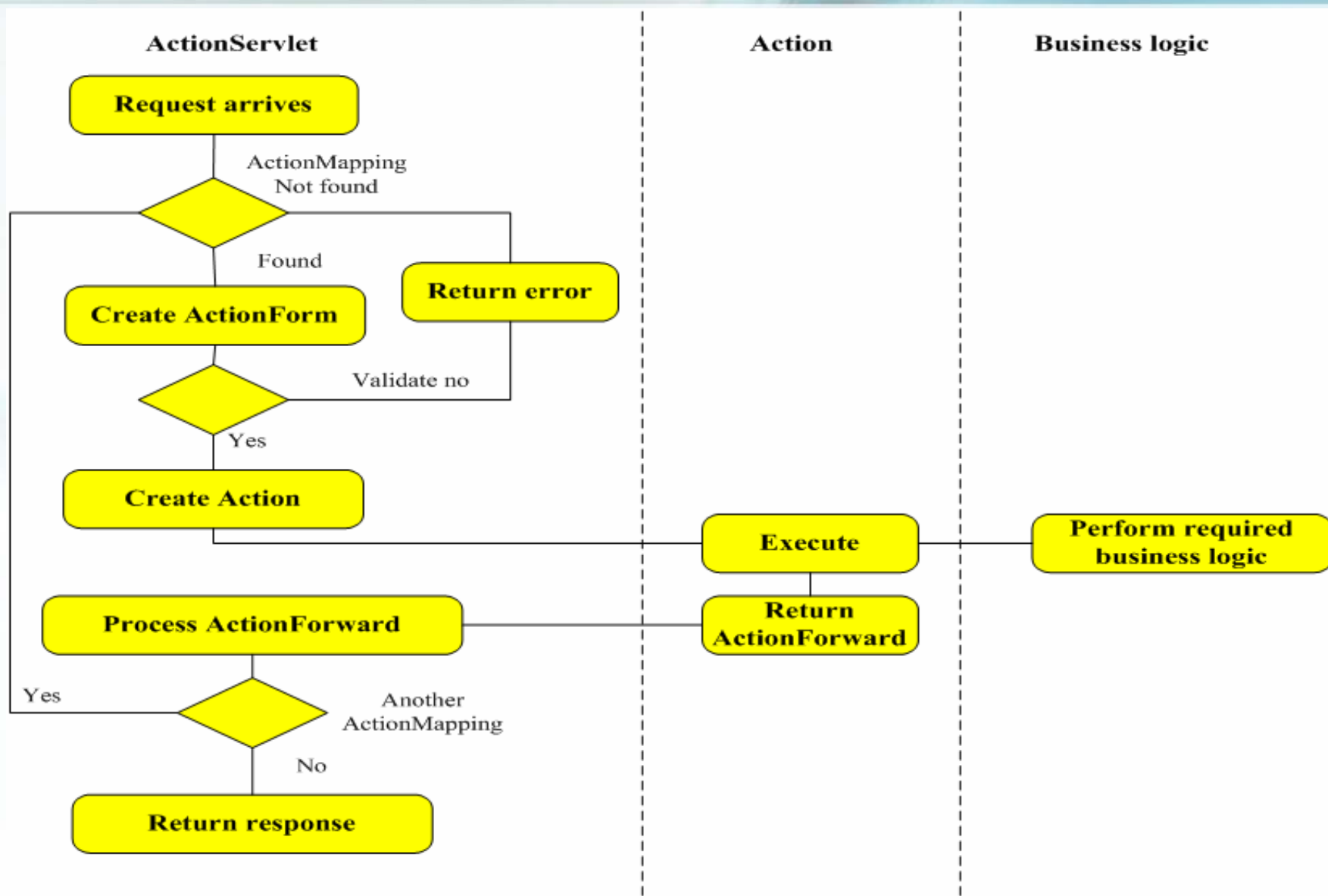
- Struts-config.xml

- 配置信息

- Action类

- 负责调用模型的方法，更新模型的状态，并帮助控制器应用程序的流程





## • Struts ActionForm

```
public class UserActionForm
```

```
...
```

```
    private String password;
```

```
    private String userName;
```

```
    public String getPassword() {
```

```
        return password;
```

```
    }
```

```
    public void setPassword(String password) {
```

```
        this.password = password;
```

```
    }
```

```
    ...public ActionErrorsvalidate(ActionMapping actionMapping,
```

```
        HttpServletRequest httpRequest) {
```

```
        return null;
```

```
    }public void reset(ActionMapping actionMapping,
```

```
        HttpServletRequest servletRequest) {
```

```
    }
```

```
...
```

## • Struts Action

```
public class LoginAction extends Action {  
    public ActionForward execute(ActionMapping actionMapping,  
                                ActionForm actionForm,  
                                HttpServletRequest servletRequest,  
                                HttpServletResponse servletResponse) {  
        UserActionForm userActionForm = (UserActionForm) actionForm;  
        String username = userActionForm.getUserName();  
        String password = userActionForm.getPassword();  
        UserBean user = new UserBean();  
        boolean flag = user.verifyUser(username, password);  
        if (flag) {  
            return actionMapping.findForward("success");  
        } else {return actionMapping.findForward("error");  
        }  
    }  
}
```

## ● 模型组件

```
public class UserBean {  
    public UserBean() {  
    }public boolean verifyUser(String userName, String password) {  
    if (userName.equals("guest") &&  
        password.equals("guest")) {  
        return true;  
    }else {return false;  
    }  
    }  
}
```

# • Struts-config.xml

```
<struts -config>
  <form-beans>
    <form-bean name="userActionForm" type="userstruts.UserActionForm" />
  </form-beans>
  <action-mappings>
    <action name="userActionForm" path="/loginAction" scope="session"
      type="userstruts.LoginAction">
      <forward name="success" path="/success.jsp " />
      <forward name="error" path="/error.jsp" />
    </action>
  </action-mappings>
</struts-config>
```

- 页面
  - 页面的Cache依赖于Http1.1协议
    - 需要客户端支持
  - Cache保存的位置
    - 客户端、服务器、代理服务器
  - Cache的有效条件
    - 过期时间
  - 页面的Cache的版本
    - 可以根据 Request中的请求参数或Http Header的不同保存不同版本的Cache
  - 页面Cache也可以通过程序实现

## ● 状态保存

- 需要保存的状态是客户端、还是服务器端
- 如果在服务器端，状态全局的，还是局部的
- 状态是否有效期
- 关闭浏览器以后，状态是否还需要保存
- 状态数据有没有可能在网络上传输，传输过程中是否需要加密、是否防篡改

- 用户界面(UI)组件
  - 类型：
    - Console、Web、Plug-in
  - 搭建 UI的框架
    - 自定义标签
    - AJAX
    - .....



## — 动态页面生成

- XML
- 模板技术

```
<screen name="main">  
    <parameter key="banner" value="/banner.jsp" />  
    <parameter key="sidebar" value="/sidebar.jsp" />  
    <parameter key="body" value="/main.jsp" />  
    <parameter key="footer" value="/footer.jsp" />  
</screen>
```

- UI流程（UIP）组件：
  - 作用：
    - 隔离了UI与业务逻辑层
    - 对流程中的 UI进行了管理
    - 提供了状态保存和传递的机制
  - 定制导航流程
    - 在配置文件中定制的流程

# 第三章 业务层设计

- 实现业务规则及执行业务工作的组件
  - 实现业务功能，是对特定业务逻辑和内部业务流程的封装
  - 负责发起事务，是根事务发起者，支持事务与补偿交易
  - 通过封装已存在的业务能够获得更高等级的操作和业务逻辑

- 业务组件与事务

- 为了保证业务处理的完整性，业务组件必须提供事务的支持

- 是事务的发起者，必须参与事务的投票
    - 能发起或参与异构系统的分布式事务，设置组件事务属性
    - 为业务处理提供补偿交易处理

- 实现事务方式
  - 数据库事务与应用事务
  - 手工事务与自动事务



## ● 设计要点

- 对于大型的系统，在保证性能的前提下，保证组件结构的可扩展性
- 尽量保持组件之间的松耦合，允许并行、渐进及独立的开发与测试
- 尽可能采用基于消息的通讯
- 确定透过服务接口所暴露的处理流程是能处理多次重复信息的情况。
- 仔细选择事务边界，设置适当的事务隔离度
- 选择和保持用一致的数据格式作为输入和返回参数

- 常用模式

- 管道模式：

- 以顺序方式执行动作与查询
    - 定义完成业务功能所需一系列步骤
    - 所有的步骤按照顺序执行
    - 每一步包含读写数据操作以便确认管道状态
    - 可以调用或不调用外部服务，步骤可以包含调用异步服务

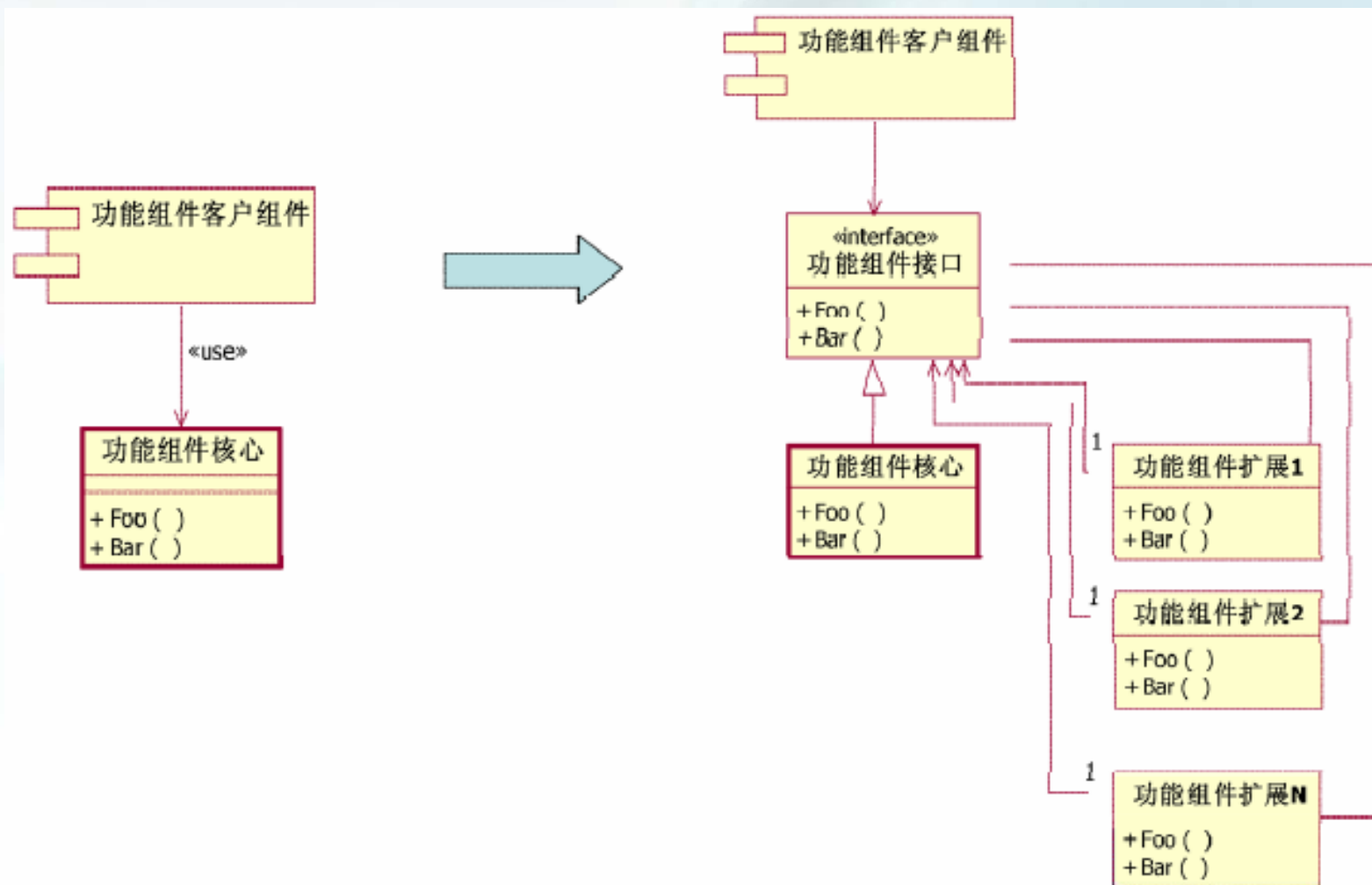


## ● 事件模式

- 事件在特定的业务条件发生，可以编写相应的代码来响应这些事件
- 如果有许多活动发生，但所有活动都收到相同的启动数据且无法与彼此通讯，可以采用该模式
- 可以是并行或顺序执行
- 执行顺序则不一定是固定的顺序

- 可扩展的组件结构
  - 扩展不影响已有客户代码的正常执行
  - 扩展是在运行期动态完成，而不需要在编译期确定
  - 扩展可以以较小的颗粒度进行
  - 不应引入过多的开发量，同时保证代码的可管理性和可维护性

# ● 基于接口的可扩展性设计



- 业务流程（ **Business Workflow** ）
  - 具有各种不同功能的活动相连的一组有相互关系的任务
  - 有起点和终点，而且都是可重复的，由多个业务过程（ **Business Process** ）组成
  - 业务过程包含多个业务步骤，且具有一定顺序
  - 定义及协调长期执行业务过程，支持长事务

- 业务流程特点

- 提供迅速实现业务规则和适应商业目标改变的能力
- 提供衡量这些改变的影响的能力
- 将每一步业务操作、资源管理，以及流程独立的分离
- 以前后一致的方式定义、改变和实现业务流程

- 业务流程种类

- 基于人的业务流程

- 每个人都得面对他或她必须完成、批准或执行的电子文档

- 基于规则的自动化流程

- 规则引擎指定一步一步的操作，基于产生的规则自动执行业务应用，这种工作流现在也在朝支持基于人的工作流的方向发展

- 业务流程的实现

- 流程引擎

- 实现业务流程同时管理活动的启用和终止或商业功能

- 资源管理器

- 资源管理器使实现商业功能或活动所必须的资源具有可用性

- 调度程序

- 审计管理器

- 安全管理器

- 业务流程
  - 管理包含多个步骤以及长期执行交易的处理过程
  - 业务过程包含长事务
  - 提供调用业务过程的接口，可让应用程序与其它服务进行交流或协作
- 只使用业务组件实现业务处理流程的情况
  - 业务功能可以实现为单一交易
  - 需要将功能和逻辑封装，多个业务过程重复加以使用
  - 需要对数据和逻辑流程进行精密的控制



<b>XLANG</b>	Microsoft 的 XLANG：用于 BizTalk 的业务模型语言，该语言是可运行 EAI 的 .NET 组件。BizTalk 编制（BizTalk Orchestration）是工作流引擎，BizTalk 编制设计器（BizTalk Orchestration Designer）是基于 XLANG 的可视化业务流程模型工具。
<b>BPEL4WS</b>	用于 Web 服务的业务流程执行语言（Business Process Execution Language for Web Services）是用于 Web 服务编制、工作流和组合的 WSFL 和 XLANG 的协作合并。该语言还尚未被提交到 IT 标准组织。
<b>Wf-XML</b>	工作流管理联盟（Workflow Management Coalition, WfMC）中的 Wf-XML 和工作流参考模型（Workflow Reference Model）：Wf-XML 是一种基于 XML 的工作流互操作性信息的编码。工作流参考模型是一种底层工作流系统体系结构的描述。目前 Wf-XML 没有与 SOAP 和 WSDL 绑定。
<b>WSFL</b>	IBM Web 服务流语言（IBM Web Services Flow Language）：指定了 Web 服务组合的两种类型 1）一个被认为是流模型（flowModel）的可执行业务流程，和 2）一个被认为是统一模型（globalModel）的业务合作。与 SOAP、UDDI 和 WSDL 兼容。
<b>ebXMLBPSS</b>	电子商务过渡工作组（eBusiness Transition Working Group）继承了业务流程规范方案（Business Process Specification Schema（BPSS））的 ebXML 层中的工作流对话和编制，ebXML 定义了许多基于 XML 的电子商务的协议和层。
<b>WSCI</b>	Sun/BEA/Intalio/SAP 联盟的 Web 服务编排接口（Web Services Choreography Interface）“是基于 XML 的接口描述语言，该语言描述了参与和其他服务的编排交互作用的 Web 服务所交换的信息流。”
<b>WSCL</b>	W3C 的 Web 服务对话语言（W3C's Web Services Conversation Language）：Hewlett-Packard 向 W3C 的提交，该提交允许定义 Web 服务的抽象接口（也就是，Web 服务支持的企业级对话或公共流程），以及交换的 XML 文档及其文档的排序。
<b>PIPs</b>	RosettaNet 的伙伴接口流程（Partner Interface Process）：定义了贸易伙伴与指定的系统到系统（system-to-system）的基于 XML 的对话之间的业务流程。许多 PIP 被用来定义各种伙伴情况。
<b>JDF</b>	CIP4 的工作定义格式（Job Definition Format）是一种即将使用的用于图形艺术（Graphics Arts）工业的工作流工业标准，该标准用于简化不同应用程序和系统之间的信息交换。

- 服务接口是为服务提供的进入点：
  - 是一个软件实体，将其实现为处理映射和转换服务的外观（**Facade**）组件
  - 与服务进行通讯，并强制执行通讯的处理流程及原则
  - 服务接口暴露方法，这些方法可被个别调用或以特定的顺序被调用

- 服务接口作用

- 提供业务处理的调用点
- 实现了缓冲，映射，以及简单的格式和架构转换
- 不实现业务逻辑
  - 通过服务接口将应用程序业务逻辑与通信协议、数据转换和SLA的履行分开
- 进行消息安全控制
- 区隔内部系统实现
  - 为对内部实现进行变更时，不需要变更服务接口
  - 需要验证传进的消息

- 服务接口的设计:

- 将服务接口视为应用程序的边界
- 同一功能发布多种服务接口, 不同的服务接口可以执行不同的**SLA**, 以及不同事务处理能力
- 实现外观组件, 帮助隔离业务的变化
- 事务性传输或非事务性传输
- 尽可能提高与其它平台和服务的交互操作性

- 服务接口的实现
  - 交换信息，负责实现通信时的所有细节：
    - 网络协议
    - 数据格式
    - 安全性
    - 服务级别协议

- 服务接口设计模式

- 外观(Façade)模式

- 目标:

- 子系统提供单一接口给用户端

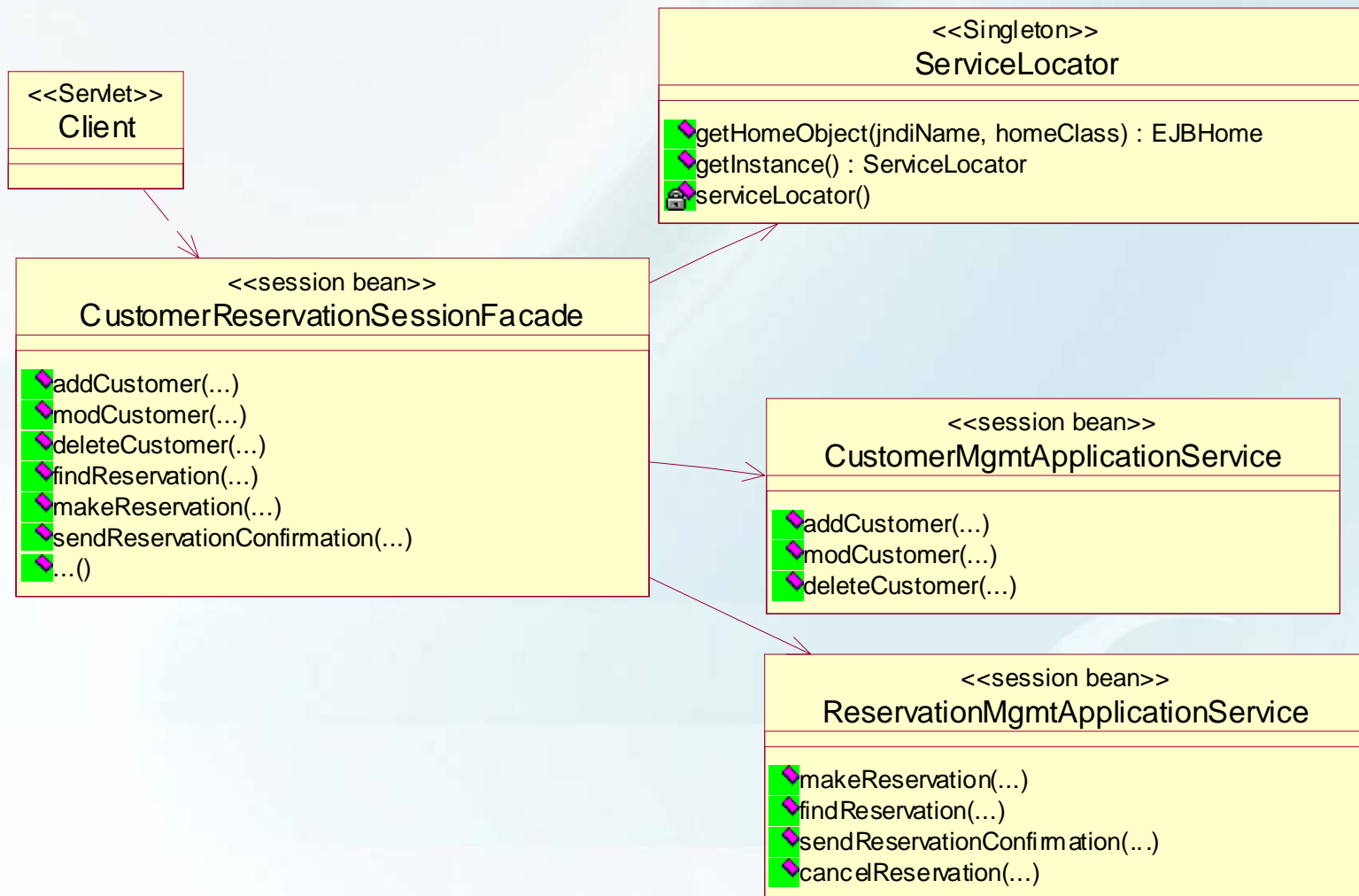
- 问题 :

- 子系统内的类分别 提供部分功能，用户端必须调用每个类， 导致两层间联系复杂， 不易维护,违反封装原则

- 效果:

- 简化设计，易于维护

## — 示例





- 实现服务接口的方式
  - XML Web 服务
    - 使用SOAP 和 HTTP
  - 消息队列方式
    - 一般采用系统服务中的队列组件



- 服务接口的事务管理
  - 在设计事务边界时,保证操作可以在发生错误时重试
    - 保证建立事务的各种资源
    - 标记事务发起组件的属性为“require”
    - 保证所有的子组件事务属性设置为 "require"或 "support "属性
  - 利用消息的事务消息机制
  - 如果消息机制不支持事务，必须在服务接口的代码中发起根事务

- 服务接口的优缺点

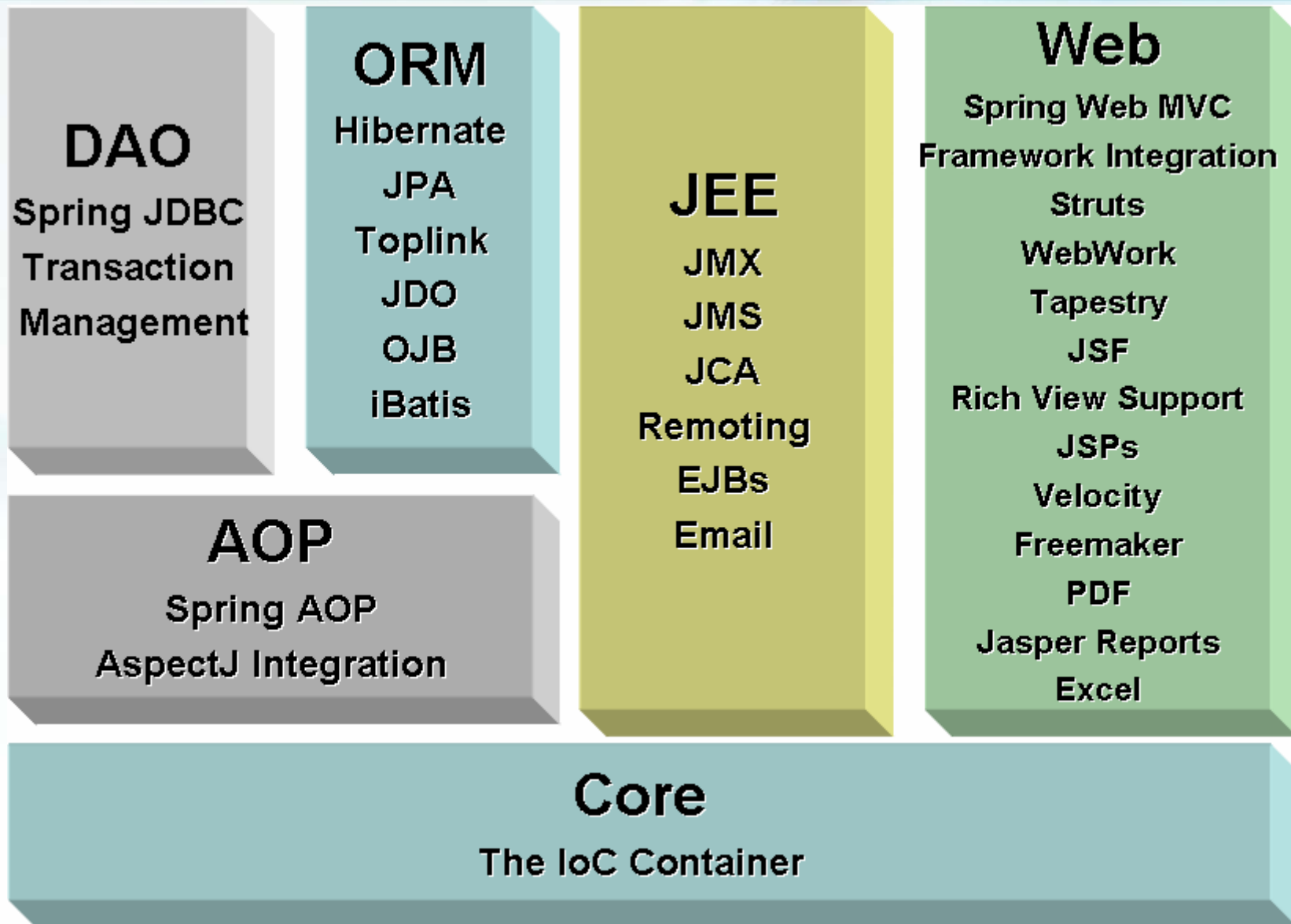
- 优点

- 服务接口机制与应用逻辑分隔
    - 部署灵活性

- 缺点

- 接口粒度设计
    - 增加在更改服务所需的工作量
    - 增加了复杂性和性能开销

- 用于构造应用程序的轻量级框架
  - IoC ( Inversion of Control ) 容器、非侵入性的框架
  - AOP ( Aspect-oriented programming )
  - 提供对持久层、事务的支持
  - 提供 MVC Web 框架的实现
  - 对常用的企业服务API提供一致的模型封装
  - 对于现存的各种框架 ( Struts、 JSF、 Hibernate等提供了整合方案



- 依赖注入（ **Dependency Injection, DI** ）  
的基本原则
  - 应用对象不应该负责查找资源或者其它依赖的协作对象
  - 配置对象的工作应该由 **IoC** 容器完成
  - “查找资源”的逻辑应该从应用代码中抽取出来,交给容器负责

- 依赖注入类型

- 类型1

- 服务需要实现专门的接口，通过接口，由对象提供这些服务，可以从对象查询依赖性

- 类型2

- 通过JavaBean 的属性（例如 **setter** 方法）设置依赖性

- 类型 3

- 依赖性以构造方法的形式提供，不以 JavaBean 属性的形式公开

## – Setter注入

```
<bean id="exampleBean" class="examples.ExampleBean">
```

```
    <!-- setter injection using the nested <ref/> element -->
```

```
    <property name="beanOne">
```

```
        <ref bean="anotherExampleBean"/>
```

```
    </property>
```

```
    <!-- setter injection using the neater 'ref' attribute -->
```

```
    <property name="beanTwo" ref="yetAnotherBean"/>
```

```
    <property name="integerProperty" value="1"/>
```

```
</bean>
```

```
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
```

```
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

- 控制反转（IoC）容器

- 是指应用代码的运行框架

- 提供服务：

- 生命周期管理
    - 查找服务
    - 配置服务
    - 依赖管理

- 容器特点：

- 可接插性
    - 一致性
    - 一站式
    - 提供企业级服务



## — 实例化容器

```
Resource resource = new FileSystemResource("MyBean.xml");  
BeanFactory factory = new XMLBeanFactory(resource);  
MyBean mybean = (MyBean) factory.getBean("mybean");
```

## — Bean例:

```
package example.spring;
```

```
    public class HelloWorld {  
        private String hello;
```

```
        public String getHello() {  
            return hello;  
        }  
        public void setHello(String hello) {  
            this.hello = hello;  
        }  
    }
```

```
        public void show(){  
            System.out.println("---message---"+getHello());  
        }  
    }
```

## — 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="myBean" class="example.spring.HelloWorld">
        <property name="hello">
            <value>Hello Spring!</value>
        </property>
    </bean>
</beans>
```

## — 使用Bean:

```
package example.spring;
```

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

```
import org.springframework.context.support.FileSystemXmlApplicationContext;
```

```
public class TestHelloWorld {
```

```
    public static void main(String[] args) {
```

```
        ApplicationContext ctx = new
```

```
        FileSystemXmlApplicationContext("src/myspring.xml");
```

```
        HelloWorld hw = (HelloWorld) ctx.getBean("myBean");
```

```
        hw.show ();
```

```
    }
```

```
}
```

- 面向方面的编程（AOP）
  - 是一种编程技术
  - 允许程序员对横切关注点或横切典型的职责分界线的行为（例如日志和事务管理）进行模块化
  - 将那些影响多个类的行为封装到可重用的模块中

## — 创建自己的Interceptor

```
public class MyInterceptor implements MethodInterceptor {  
    private final Log logger = LogFactory.getLog(getClass());  
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {  
        logger.info("Beginning method (1): " +  
            methodInvocation.getMethod().getDeclaringClass() + "." +  
            methodInvocation.getMethod().getName() + "()");  
        long startTime = System.currentTimeMillis();  
        try{  
            Object result = methodInvocation.proceed();  
            return result;  
        }finally{  
            logger.info("Ending method (1): " +  
                methodInvocation.getMethod().getDeclaringClass() + "." +  
                methodInvocation.getMethod().getName() + "()");  
            logger.info("Method invocation time (1): " +  
                (System.currentTimeMillis() - startTime) + " ms.");  
        }  
    }  
}
```

## — 编写业务对象及其接口

```
public interface BusinessInterface {  
    public void hello();  
}  
  
public class BusinessInterfaceImpl implements BusinessInterface{  
    public void hello() {  
        System.out.println("hello Spring AOP.");  
    }  
}
```



## — 使用Interceptor

```
<bean id="businessTarget"
  class="com.spring.testaop.BusinessInterfaceImpl"/>
<bean id="myInterceptor" class="com.spring.testaop.MyInterceptor"/>

<bean id="businessBean"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces ">
    <value>com.spring.testaop.BusinessInterface</value>
  </property>
  <property name="interceptorNames">
    <list>
      <value>myInterceptor</value>
      <value>businessTarget</value>
    </list>
  </property>
</bean>
```



## — 测试类

```
ClassPathResource resource = new  
    ClassPathResource("com/spring/testaop/aop_bean.xml");  
XmlBeanFactory beanFactory = new XmlBeanFactory(resource);  
  
BusinessInterface businessBean = (BusinessInterface)  
    beanFactory.getBean("businessBean");  
businessBean.hello();
```

- 业务实体
  - 提供了业务数据的封装
  - 将显示数据与实际的存储隔离，保证数据的独立性，提高可重用性



- 业务组件与业务实体的关系
  - 管理业务实体的生命周期
  - 根据业务需求选取业务实体集合
  - 管理业务实体间的关系
  - 并非所有的情况都必须有业务实体，业务组件可以直接调用数据访问层

- 业务实体的形式

- XML:

- 可用于执行查询直接传回，或由DataSet/ResultSet做数据转换

- DataSet/ResultSet :

- 数据访问结果

- 自定义业务实体:

- 符合面向对象概念，程序逻辑简单，数据转换的额外开销最多

- 利用XML来表现业务实体，注意：
  - 确定XML文档包含一个业务实体还是多个业务实体的集合
  - 用命名空间唯一确认表现业务实体的XML文档
  - 为属性和元素选择合适的名称，表现业务实体数据

## — DTD定义例

< ? xml version="1.0" encoding="GB2312"? >

< !ELEMENT 联系人列表 (联系人)\* >

< !ELEMENT 联系人 (姓名, ID, 公司, EMAIL, 电话, 地址) >

< !ELEMENT 地址 ( 街道, 城市, 省份 ) >

< !ELEMENT 姓名 (#PCDATA) >

< !ELEMENT ID(#PCDATA) >

< !ELEMENT 公司 (#PCDATA) >

< !ELEMENT EMAIL(#PCDATA) >

< !ELEMENT 电话 (#PCDATA) >

< !ELEMENT 街道 (#PCDATA) >

< !ELEMENT 城市 (#PCDATA) >

< !ELEMENT 省份 (#PCDATA) >

## – XML Schema定义例:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="loan">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="initialBalance" type="balance" />
        <xs:element name="currentBalance" type="balance" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="balance">
    <xs:restriction base="xs:decimal">
      <xs:totalDigits value="10" />
      <xs:fractionDigits value="2" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

- 优点：
  - 国际标准支持
  - 灵活性
  - 互操作性
- 缺点：
  - 性能
  - 保留类型精度
  - 验证 XML
  - 显示 XML，不能在用户界面自动显示XML数据
  - 解析XML
  - XML排序
  - 没有私有信息



- 自定义业务实体
  - 实体组件包含数据的快照
    - 其数据只能保证与发起读取事务时上下文环境一致
  - 组件储存数据值，并透过其属性加以显露
  - 业务实体不直接存取数据库
  - 业务实体不初始化任何事务交易
  - 考虑是否将业务实体设计为序列化



## — 设计建议

- 自定义业务实体实现要简单
- 从封装通用方法的基类继承
- 业务实体实现一组公用的接口
- 利用内部数据集或XML文件保存复杂数据
- 将验证规则分离为元数据
- 设计保证实体之间的关系
- 通过数据存取逻辑组件操作数据库数据



## — 实现原则

- 为每个实体定义唯一标识
- 在类或其它结构之间选择
- 显示业务实体的状态
- 在用户自定义业务实体中表现子集合以及层次数据
  - 集合类，例如ArrayList
  - 数据集，例如DataSet



## —其它考虑:

- 支持用户界面组件进行数据绑定
- 为内部数据变化提供事件
- 对业务实体组件进行序列化



## — 优点

- 自行控制序列化实现，有利于性能调优
- 隔离内部格式和应用程序所使用的数据结构描述
- 代码可读性
- 封装
  - 给数据增加自己的操作
- 对复杂系统建模
  - 用自定义实体类定义一个良好的接口，将复杂的问题隐藏在类中
- 本地验证
- 私有成员
- 隐藏不想暴露给调用者的信息



## — 缺点

- 业务实体集合
  - 一个业务实体对象表现单个的业务实体，而不是业务实体的集合
- 序列化
  - 在业务实体中必须实现自己的序列化机制
- 业务实体对象中表现复杂的数据关系和层次数据
- 数据查询和排序
- 部署
- 对系统服务客户端的支持
- 可扩展性的问题

# 第四章

## 数据访问层设计

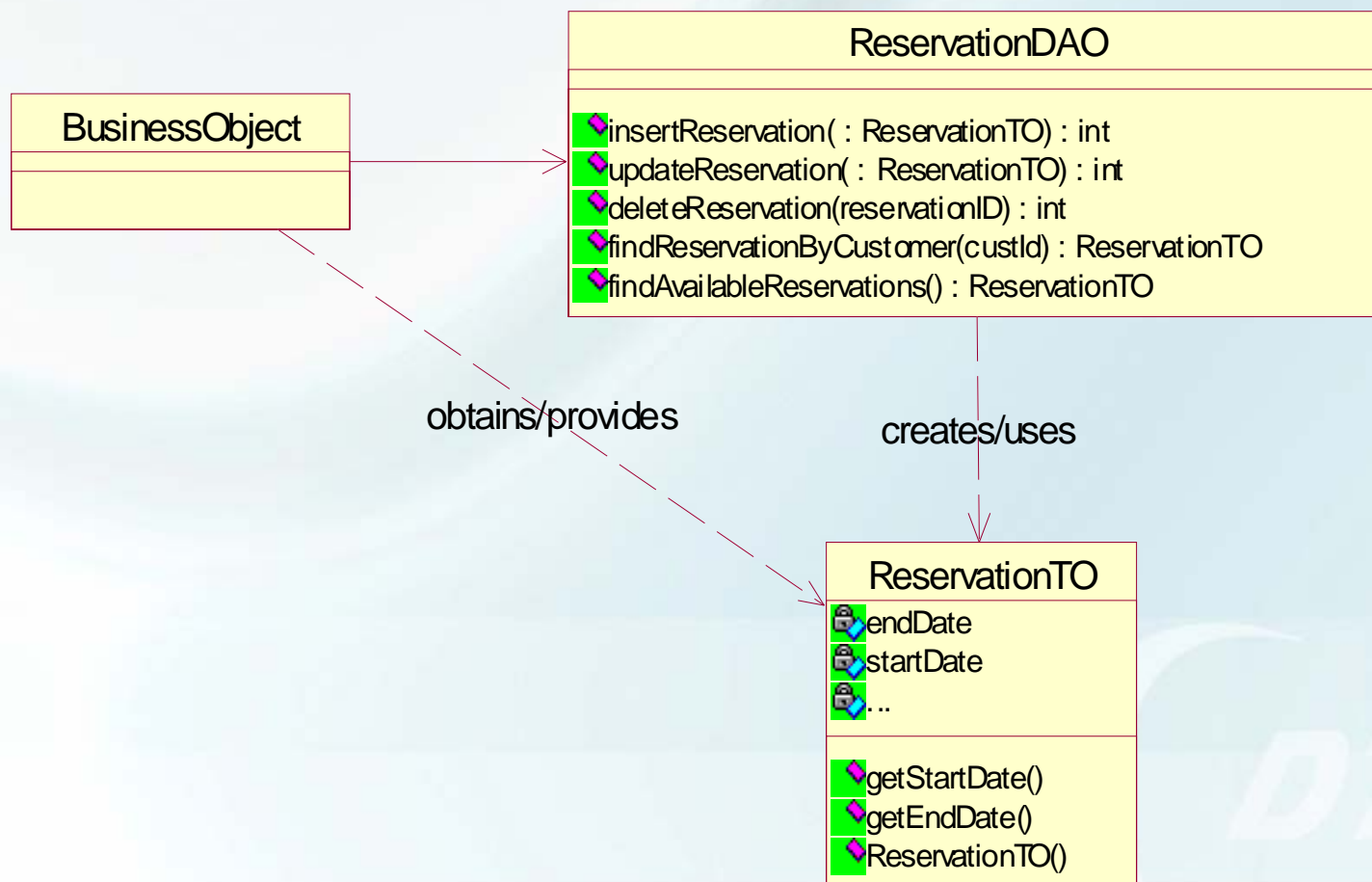
- 数据源
  - 可以是关系数据库、文件系统等
- 数据访问层
  - 隔离业务逻辑层和数据存储
  - 好处
    - 增加代码重用性
    - 尽可能消除业务逻辑层对数据源的依赖
      - 可以通过配置文件进行改变
    - 隐藏了数据操作的细节，可以实现各种优化



- 数据访问对象
  - 使用DAO抽象并封装对数据源的所有访问
  - DAO管理数据源的连接以获取和存储数据
  - 使用工厂管理DAO



## — 示例



- DAO设计考虑

- 提供无状态的方法来实现对数据源的操作
- 实现涉及到三方面问题：
  - 无状态
  - 如何划分组件
  - 参数如何定义



- DAO组件的接口定义和划分
  - 先定义好DAO组件的接口
    - 可以通过不同工厂类来建立不同DAO实现类
  - DAO组件的划分
    - 可以考虑按照业务实体进行划分
    - 某一组件最好只访问一种数据源



## — DAO组件与事务

- 保证事务完整性

- 如何传递事务上下文?
- 事务是否跨越不同的数据源(分布式事务)?



## — DAO组件的输入输出参数

- 输入参数
  - 一般读操作所要传递的参数
    - » 查询条件、分页信息
  - 一般写操作需要转递的参数
    - » 更新数据
- 返回值
  - 普通类型
  - 结果集
  - 自定义的数据结构



- **DAO组件和存储过程**
  - 利用存储过程
    - 有助于优化和提高性能
    - 有助于将某些二维表的结果集转换为另外一种二维表
  - 不适合使用存储过程的操作
    - 查询/修改的条件或参数是变化的
    - 存储过程尽量避免实现业务逻辑
    - 有些操作用**SQL**语句很难写

- 设计通用DAO

- 为数据访问层提供通用的数据访问接口
- 减少数据访问操作的代码
- 简化执行SQL语句和调用存储过程的代码
- 进行数据库连接的管理
- 包括数据库名称、认证信息等
- 在不同的数据源之间，可以提供统一的接口



- 考虑：
  - 数据架构规划与数据库设计
  - 数据库设计与类的设计
  - 数据库设计与XML设计
  - 数据库性能规划
  - 封装数据库设计



- 带CRUD操作的自定义业务实体
  - 优点：
    - 封装
      - 自定义业务实体封装了下游数据存取逻辑组件的操作
    - 接口调用
      - 调用者只需要通过一个接口与业务实体数据打交道
    - 私有成员
      - 可以隐藏不希望暴露给调用者的信息
  - 缺点：
    - 处理业务实体类的集合
    - 增加开发时间

- 数据传输对象（DTO）
  - 为业务实体对象与实际的关系数据存储提供松散耦合关系
  - 一个数据传输对象是一个没有业务逻辑但可序列化的数据实体



## — 数据传输对象的设计

- 数据传输对象在实现上可以是一个只有构造器和简单类型字段成员（或属性）的类或结构体
- 内部唯一标识与外部唯一标识
- 数据实体成员
- 数据版本信息
  - 用于支持并发检测



- 对象关系映射（ORM）
  - 作用
    - 通过结构化的映射使程序更强健
    - 减少错误代码
    - 始终对性能进行优化
    - 数据源独立性
  - 目标：
    - 利用关系型数据库的优点
    - 保持面向对象的语言对象方式
    - 做更少的工作，减少DBA压力

## ●对象—关系数据库的匹配

对象	关系数据库
类的属性（基本类型）	表的列
类	表
1:n/n:1	外键
n:m	关联表
继承	...

- ORM 实现:

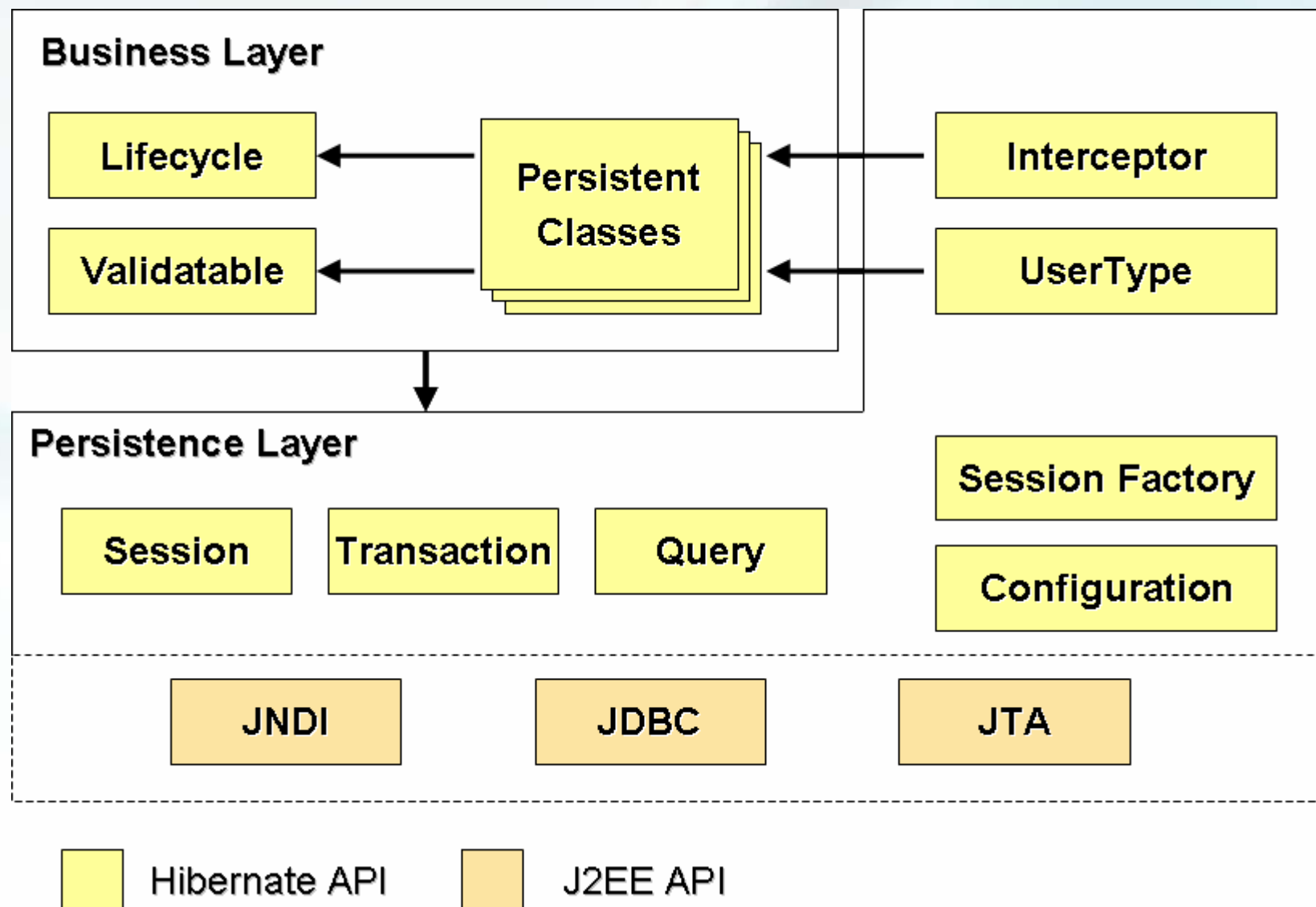
- 透明的持久化:
  - 用面向对象的语言直接操作关系型数据的能力
- 自动进行数据同步问题的检查
- 灵活的获取与缓存
  - 实现Lazy Loading, Outer Join Fetching, Runtime SQL Generation功能
- 继承的映射策略
- 开发工具

- 具体实现方式

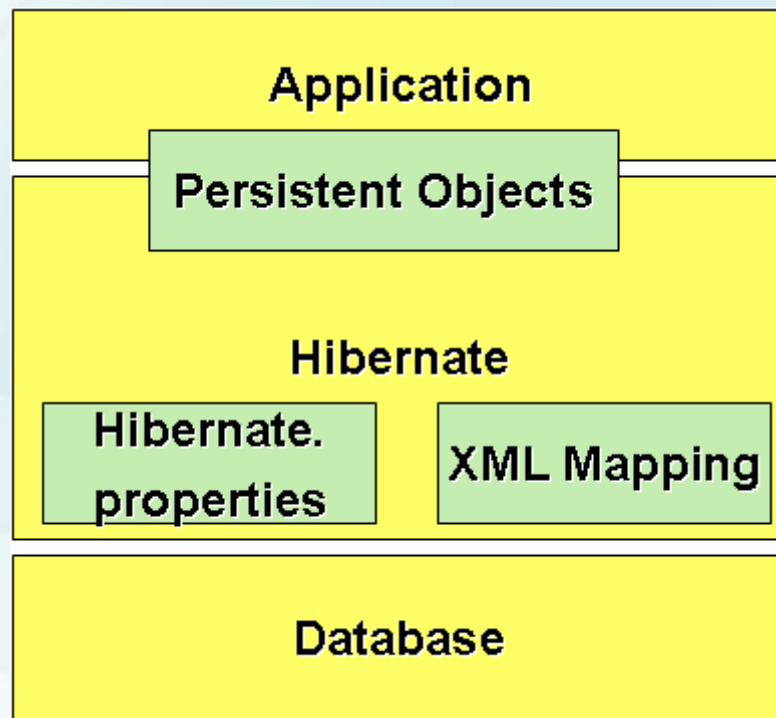
- 利用反射机制，在运行时自动产生SQL语句，执行ORM操作
- 通过ORM工具，生成代码，通过生成的代码来实现ORM







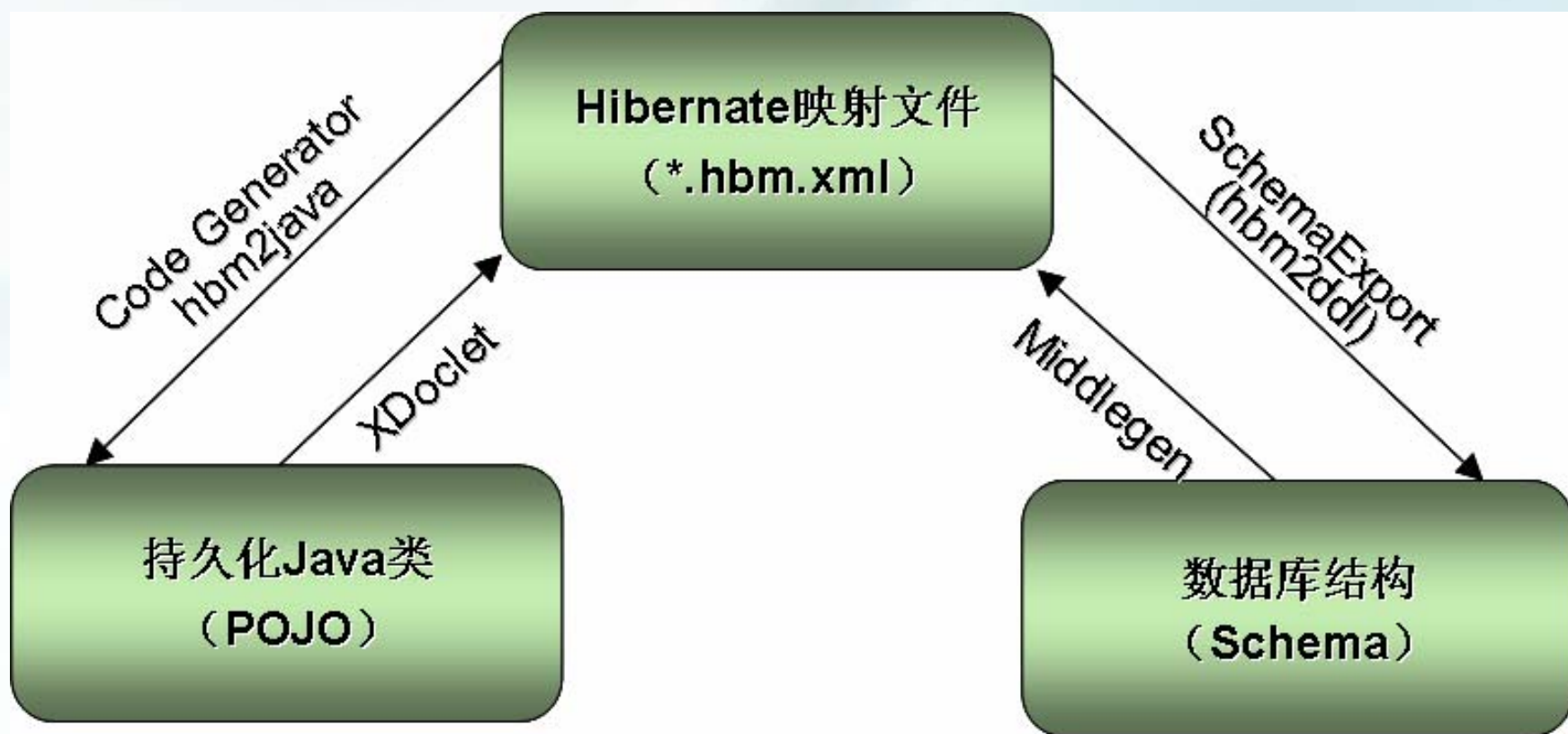
- **Hibernate开发**
  - 持久化类设计
  - 持久化类和关系数据库的映射
  - 应用开发



## — 持久化类 — **User.java**

```
public class User {  
    private Long id;  
    private String name;  
    private Date birthday;  
    private String email;  
    public User(){  
    }  
    public User(String name, Date birthday, String email){  
        .....?Get/Set  
    }  
}
```

## 持久化类和关系数据库的映射



## — 映射文件—User.hbm.xml

```
<hibernate-mapping>
<class name=" com.test.hibernate.User " table="TBL_USER">
  <id name="id" column="ID">
    <generator class="native"/>
  </id>
  <property name="name" column="NAME"/>
  <property name="birthday" column="BIRTHDAY"/>
  <property name="email" column="EMAIL"/>
</class>
</hibernate-mapping>
```

## — 映射一对多关联

- User-Address的映射

```
<class name="com.test.hibernate.User" table="T_USER">  
  <id name="id" column="userId">  
    <generator class="native"/></id>  
    <set name="addresses" lazy="true" cascade="all">  
      <key column="addressId"/>  
      <one-to-many class="com.test.hibernate.Address"/>  
    </set>  
  </class>
```

```
<class name="com.test.hibernate.Address" table="T_ADDRESS">  
  <id name="id" column="addressId"> <generator class="native"/></id>  
</class>
```

- Spring事务管理
  - 编程式事务处理
    - 在一个catch代码块中对任何异常时行回滚处理
    - 或，通过一个IoC模板类和一个回调实现
  - 声明式事务处理
    - 最适合用AOP，回事务管理有一个很明确的横切概念





## ● 程式事务处理

- 声明数据源
- 声明一个事务管理类

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory ">
    <ref bean="sessionFactory " />
  </property>
</bean>
```

- 在代码中加入事务处理代码

```
TransactionDefinition td = new TransactionDefinition();
TransactionStatus ts = transactionManager.getTransaction(td);
try{//do sth
    transactionManager.commit(ts);
}catch(Exception e){
    transactionManager.rollback(ts);
}
```



## • 声明式事务处理

- 使用 TransactionInterceptor 拦截器和常用的代理类 TransactionProxyFactoryBean
- 定义数据源，事务管理类
- 定义事务拦截器

```
< bean id = "transactionInterceptor"  
  class="org.springframework.transaction.interceptor.TransactionInterceptor">  
  < property name="transactionManager">  
    < ref bean="transactionManager"/>  
  < /property>  
  < property name="transactionAttributeSource">  
    <value> com.test.UserManager.*=PROPAGATION_REQUIRED </value>  
  < /property>  
< /bean>
```

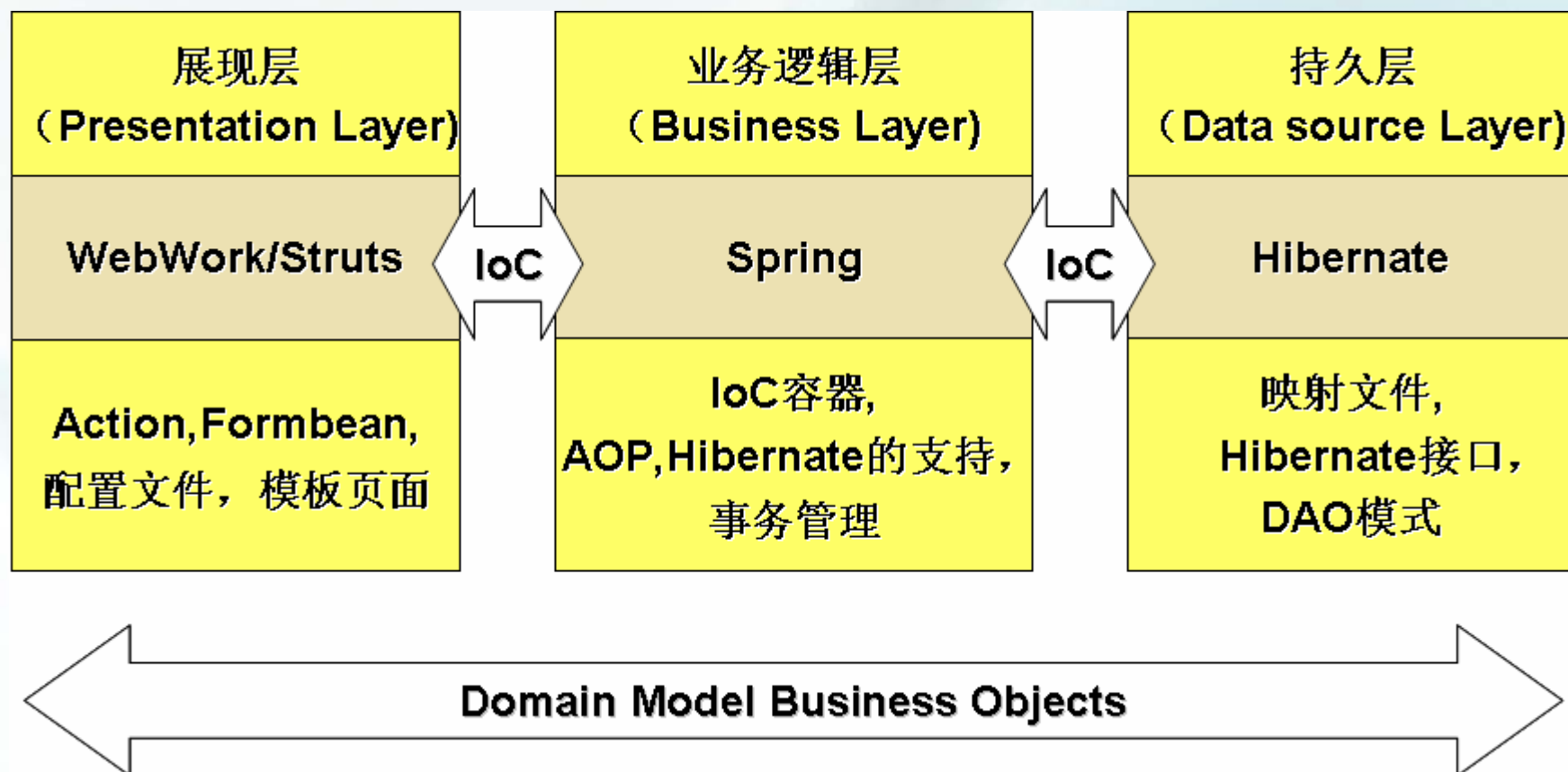
- 为组件声明一个代理类: ProxyFactoryBean

```
<bean id="userManager"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
    <value>com.test.UserManager</value>
  </property>
  <property name="interceptorNames">
    <list>
      <idref local="transactionInterceptor"/>
    </list>
  </property>
</bean>
```

# ● 使用TransactionProxyFactoryBean:

```
< bean id="userManager"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  < property name="transactionManager">
    < ref bean="transactionManager"/>
  < /property>
  < property name="target">
    < ref local="userManagerTarget"/>
  < /property>
  < property name="transactionAttributes">
    < props>
      < prop key="insert*"> PROPAGATION_REQUIRED
    < /prop>
      < prop key="update*"> PROPAGATION_REQUIRED
    < /prop>
      < prop key="*"> PROPAGATION_REQUIRED,readOnly
    < /prop>
    < /props>
  < /property>
< /bean>
```

# 基于B/S的典型三层架构



# 第五章

## 通用服务设计

- 设计流程
  - 分析潜在的威胁
  - 评估威胁并设定优先级
  - 针对威胁进行分类
  - 根据威胁选择安全技术
  - 设计对应的安全服务



- 针对系统安全性的威胁
  - 标识欺骗 (Spoofing identity)
  - 篡改数据 (Tampering with data)
  - 可否认性 (Reputability)
  - 信息泄露 (Information disclosure)
  - 拒绝服务 (Denial of service)
  - 特权升级 (Elevation of privilege)



- 设计基本原则
  - 使用成熟的安全系统
  - 以小人之心度输入数据
  - 外部系统是不安全的
  - 最小授权
  - 减少外部接口
  - 缺省使用安全模式
  - 安全不是似是而非
  - 从 **STRIDE** 思考
  - 在入口处检查
  - 从管理上保护好你的系统



- 系统安全的五个角度
  - 认证
  - 授权
  - 安全通讯
  - 属性管理
  - 审计



- 认证

- 考虑:

- 什么是认证?
    - 认证操作应该在哪里实现?
    - 认证的设计目标是什么?
    - 如何进行认证设计?



## — 常用传递标识流的方法

- 传递用户信息，重新认证
- 传递认证凭证
- 单一认证解决方案
  - 异构认证环境
  - 支持系统间交换
- 运行在同一上下文中



## — 认证机制

- 成熟的认证系统
  - 活动目录
  - LDAP, ...
- 自定义认证系统
  - 自定义标识对象
  - 应用中不要保存密码，保存凭证
  - 审计所有的失败认证
  - 提供访问认证凭证的方法



## — 表现层认证

- 用户界面组件
- 不需要在界面流程组件中认证



## — HTTP Form认证方法配置例

```
<user-data-constraint>
```

```
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
```

```
</user-data-constraint>
```

```
<login-config>
```

```
  <auth-method>FORM</auth-method>
```

```
  <form-login-page>/estore/login.jsp</form-login-page>
```

```
  <form-error-page>/estore/error.html</form-error-page>
```

```
</login-config>
```



## — 业务层认证

- 业务组件的认证
  - 业务组件必须要认证
  - 可以通过服务接口认证
- 业务实体的认证
  - 访问内部业务实体无需认证
  - 访问外部业务实体必须认证
    - » 通过辅助的认证组件实现认证过程



## — 数据访问层认证

- 服务帐号
  - 使用情况:
    - » 对帐号缺少控制
    - » 应用系统和存储系统采用不同的认证机制
- 扮演（Impersonating the caller）
  - 使用情况:
    - » 数据存储需要根据用户信息进行授权
    - » 审计每个用户的活动
  - 扮演方式:
    - » 平台的扮演服务
    - » 单一认证服务



- 授权：
  - 两个方面：
    - 用户的权限
    - 代码的执行权限
  - 基本方法
    - 系统授权
    - 代码访问权限机制
    - 应用本身的授权管理



## — 用户访问权限-访问控制列表（ACLs）

- 分级：
  - 高级覆盖低级
  - 同级类Deny优先

## — 基于角色的授权

- 减少代码修改
- 减少维护工作量
- 角色可交叉



## • Web授权例

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>CheckOut</web-resource-name>
    <url-pattern>CheckOut</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>customer</role-name>
  </auth-constraint>
</security-constraint>
```



## — 自定义应用系统的授权

- Principal对象
- 经过认证的授权数据访问
- 授权信息中不要包含凭证和认证信息
- 使用缓存技术提高性能
- 提供离线功能
- 可扩展的架构
- 同时使用代码访问权限



- 安全通讯
  - 针对信道的安全：
    - SSL
    - IPSec
    - 自定义加密信道
    - VPN
  - 针对数据的安全：
    - 数字签名
    - 加密信息
      - 全部
      - 敏感信息



- 属性管理
  - 属性管理的数据包括：
    - 用户的系统配置
    - 用户数据
  - 相关技术：
    - 缓冲技术
    - 加密和Hashing
  - 存储方式：
    - 活动目录
    - 数据库
    - 本地文件

- 审计

- 可以看作是"安全的日志"
  - 用户操作
  - 业务活动
- 考虑的安全重点
  - 反抵赖和防篡改
  - 安全的存储
- 常用技术：
  - 数字签名
  - 基于平台的认证
  - 代码访问权限
  - 可利用Principal对象实现接口

- 通讯策略
  - 三要素：
    - 同步性、格式和协议
  - 模式：
    - 无连接：基于消息
    - 有连接：分布式组件
  - 考虑因素：
    - 扩展性
    - 可用性
    - 可管理性



- 通讯选择

- 应用间通讯:

- 建议采用消息方式:

- XML Web Service

- 消息队列

- 应用内通讯:

- 使用基于消息机制的层间通讯

- 例外: 事务/身份流

- 建议:

- 尽量使用Queue和Cache技术

- 评估封装异步操作实现同步操作

- 异步消息通讯机制
  - 优点
    - 扩展性和可用性
    - 更短的响应时间
  - 更容易实现负载均衡和容错
  - 可实现透明的服务定位
  - 更符合现实的业务模式
  - 更容易定义**SLA**
  - 传输协议的无关性



## — 缺点

- 无状态
- 无消息匹配机制
- 消息延迟
- 没有事务处理
- 可能存在重复性消息
- 无消息序列



## — 异步处理需要考虑的问题

- 状态通知和轮询
- 处理超时情况
- 创建和执行补偿操作



## — 基于消息的应用

- 核心应用
- 大型分布式系统
- 对外的服务应用
- 需要离线操作的应用
  - 可能需要处理同步冲突



## — 消息队列

- 优点和特点
  - 基于Internet的存储转发机制
  - 确保顺序和可靠性的通讯机制
  - 可以利用群集技术实现高可用性
- 设计要点
  - 发送和接收方式
  - 消息格式



- 同步技术考虑
  - 端点
  - 协议
  - 格式



- 同步 vs. 异步
  - 需要事务流或者安全认证信息？
  - 开放给外部系统？
    - 需要封装业务功能？
  - 需要认证调用者？





- 通讯格式、架构和协议的决定因素：
  - 发送和接收的可控性
  - 对性能和安全的要求
  - 与企业外应用的交互要求



- 缓存（Cache）
  - 所解决的问题
    - Performance
    - Scalability
    - Availability
  - 缓存是必须在设计时考虑的问题
    - 理解系统提供的缓存技术
    - 考虑面对的环境



- 数据缓存的必要性：
  - 减少跨进程的通信
  - 减少对数据库的访问
  - 减少磁盘操作



- 状态

- 某个特定时刻，系统的数据的状况
- 状态的生命周期
  - 永久状态：
    - 在一个应用里永久有效的数据
  - 进程状态：
    - 状态仅在进程范围内有效
  - 会话状态：
    - 在一个特定用户的会话期间有效
  - 消息状态：
    - 在一个消息的处理期间有效

## — 状态的范围

- 物理范围：状态可被访问的物理的位置
  - 组织
  - 应用服务器集群
  - 机器
  - 应用域
- 逻辑范围
  - Application
  - Business process
  - Role
  - User



- 状态的失效程度（**staleness**）
  - 缓存的状态和原始数据之间的差别程度
  - 两方面影响：
    - 变化的可能性
    - 变化的相关性
  - 系统对不同状态的失效程度具有不同的容忍度
  - 状态转变过程
    - 决定缓存的数据形式

- 缓存技术

- 远程单实例缓存（Remoting Singleton Caching）

- 提供跨越应用域和机器的缓存服务
    - Singleton Object保证单实例
    - 需要控制租约的超期



- 内存映射文件（Memory-Mapped Files）
  - 用于跨越应用域的缓存
  - 良好的性能
  - 开发复杂，互操作
- 数据库缓存
  - 适合于直接缓存表数据
  - 不利于缓存对象
  - 良好的伸缩性





- 通过静态变量缓存
  - 在一个应用域以内
  - 最好的性能
  - 精心设计的锁策略



- 分布式系统中的缓存
  - 服务器端Session状态
  - 客户端Cache和状态



- 分布式应用中的缓存

- 表现层的缓存

- 缓存 Web 页面、部分页面，以及需要填充大量数据的Form
    - 其它缓存：
      - 用户身份
      - 业务数据
      - 配置数据
      - 元数据



## — 业务服务层的缓存

- 服务接口中
  - 缓存与事务无关的服务的状态数据，以及配置数据
- 业务流程中
  - 可以缓存一个工作流的多个步骤之间的状态
  - 注意：这些状态通常被保存在数据库或文件系统中
- 业务组件中
  - 缓存处理过程中的不同状态
- 业务实体中
  - 按照原始格式缓存
  - 按照使用格式缓存
    - » 使用业务实体工厂为特定的应用特定的业务实体

## — 数据服务层的缓冲

- 数据访问组件中
  - 缓存非事务的查询返回的原始数据
- 数据访问辅助类中
  - 配置信息
  - 存储过程的参数



- 物理部署的考虑

- 选择一种**Cache**技术，支持灵活的物理存储位置
  - DB Server, File, XML file...
- 尽量使用一种类型的**Cache**技术为分布式系统的不同层次提供**Cache**服务
- 大部分**Cache**机制支持name/value方式
  - 注意命名规则 ("UI\_", "BI\_", "SRV\_")



- 管理缓存的内容

- 载入数据到缓存

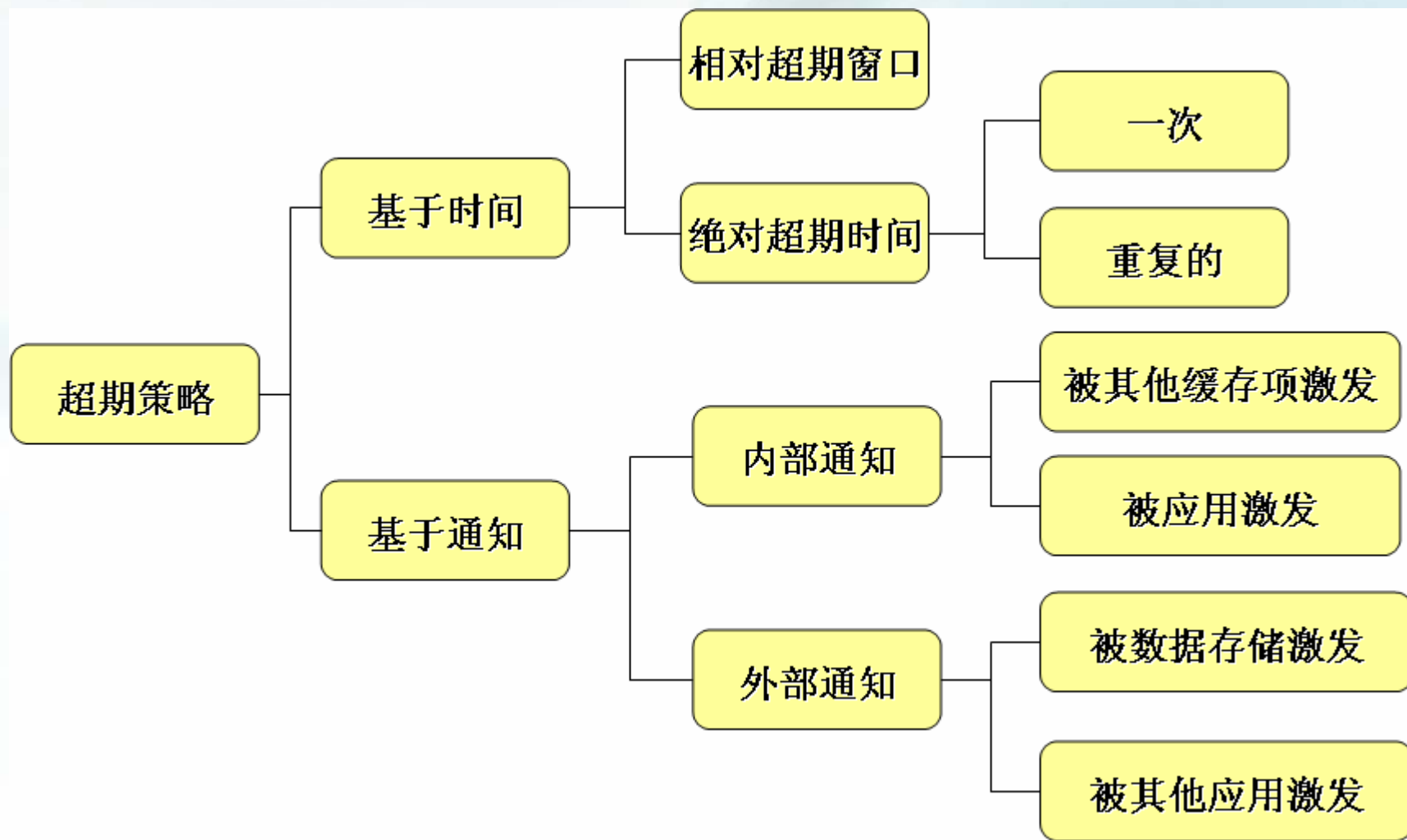
- 主动载入数据

- 访问缓存时无需检测是否存在
      - 如果缓存是只读的，可以简化锁的设计
      - 不适用于太大的数据

- 被动/延迟载入数据

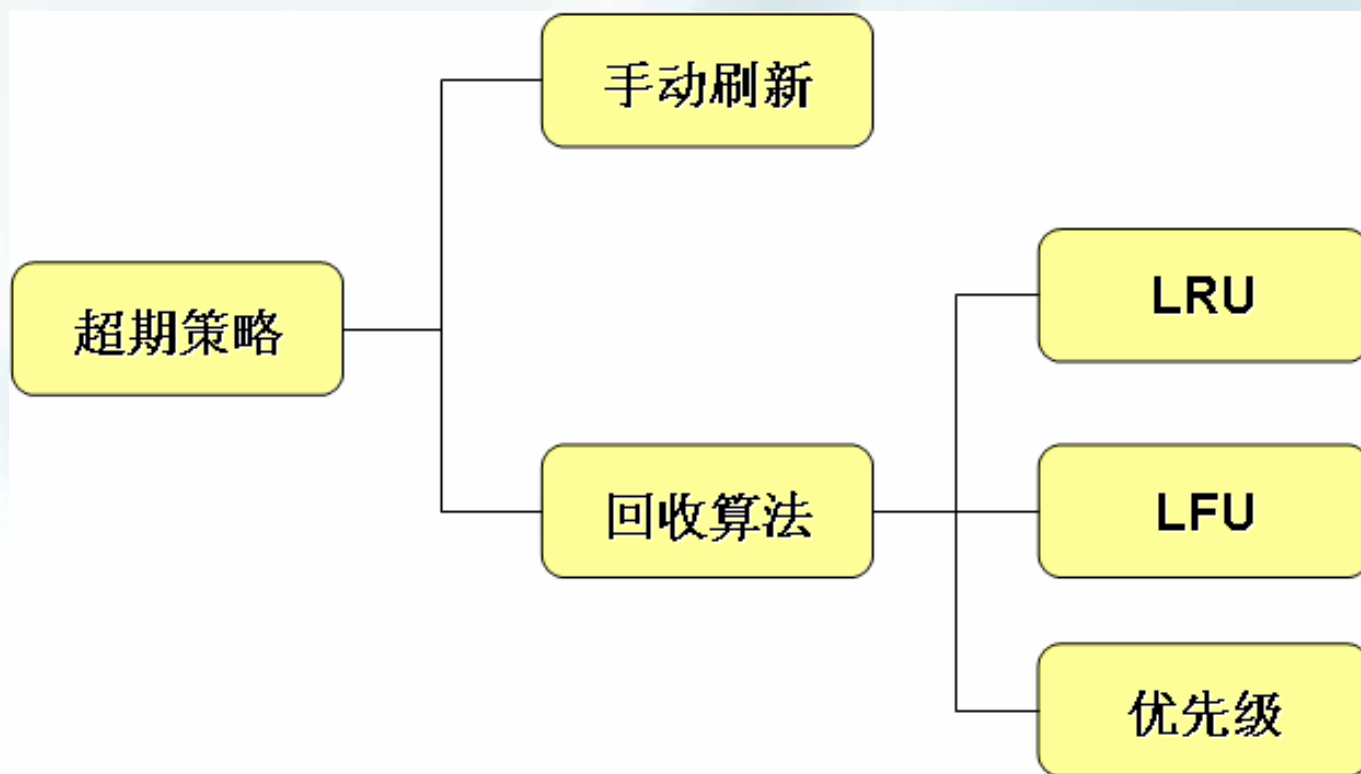
- 节省系统资源，降低初始化的时间
      - 不能假设数据已经在缓存中，每次访问需要检测
      - 需要良好的锁设计

## 缓存超期策略





## ● 回收缓存



# 第六章

## 企业应用集成

- 企业的现状

- 企业内部存在多个不同应用系统
- 应用系统间需要共享数据，共享服务功能，共享处理流程

- EAI实施

- 企业的信息建设是一个漫长的过程
- 面临不同的平台、技术、产品和应用需求
- 信息技术本身迅速发展
- 企业结构、应用需求不断发展变化

- 类型

- 数据集成

- 直接访问其他系统的数据
    - 共享、交换数据

- 应用接口集成

- 调用其他系统对外开发的服务接口
    - 共享、交换数据
    - 共享现有应用逻辑

- 界面集成

- 一般Web应用可以使用

- 模式参考模型



- EAI业务模式

- 满足业务需求，提供以下功能：

- 集成业务处理步骤
    - 集成异构应用
    - 集成异构数据源

- 满足业务需求，达成如下目标：

- 快速的应用集成
    - 组件和服务的重用性
    - 集成服务灵活，易于访问

- 概念模型

- 提供一个能够快捷、灵活地集成业务应用和业务数据的EAI方案
  - 兼顾到安全和可管理性要求，能够支持业务处理步骤集成
- 信息技术的影响
  - 应用系统集成
    - 首要需求是应用集成
    - 通过接口集成应用
    - 应尽可能减少对应用的影响
  - 可管理的集成
    - 具体集成方案不断发展
    - 接口不断增加，风险不断增加
    - 要求有效的管理
  - 数据交换
    - 直接和数据源交换或者通过接口访问

- 逻辑服务层方案
  - EAI服务
  - 传输服务
  - 安全服务
  - 管理服务
- 物理服务
  - 使用EAI技术支持EAI逻辑模式中定义的各种逻辑服务





- 实现层
  - 使用EAI技术来实现常见EAI方案
  - 三种类型集成方案
    - 处理流程集成
    - 应用集成
    - 数据集成



- 实施方法

- 理解企业业务流程
- 认识当前企业应用之间面临的问题
- 建立业务数据模型
- 整理业务流程
- 整理现有应用服务接口
- 整理业务事件
- 规划信息数据流
- 计划业务规模和性能需求
- 制定运维方案



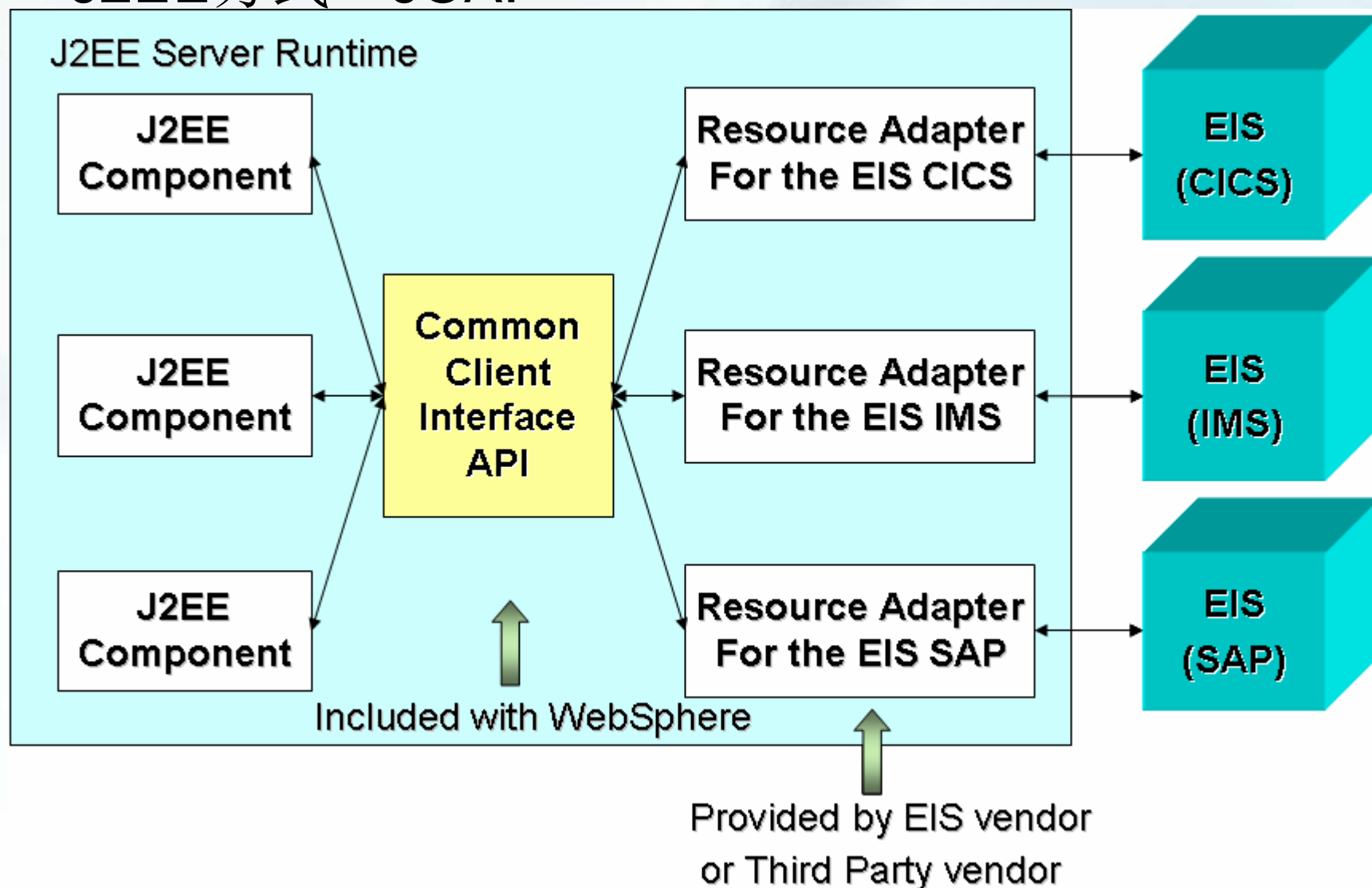
- 应用集成考虑要点
  - 同步与异步
  - 确保数据完整性
  - 管理事务
  - 流程自动化



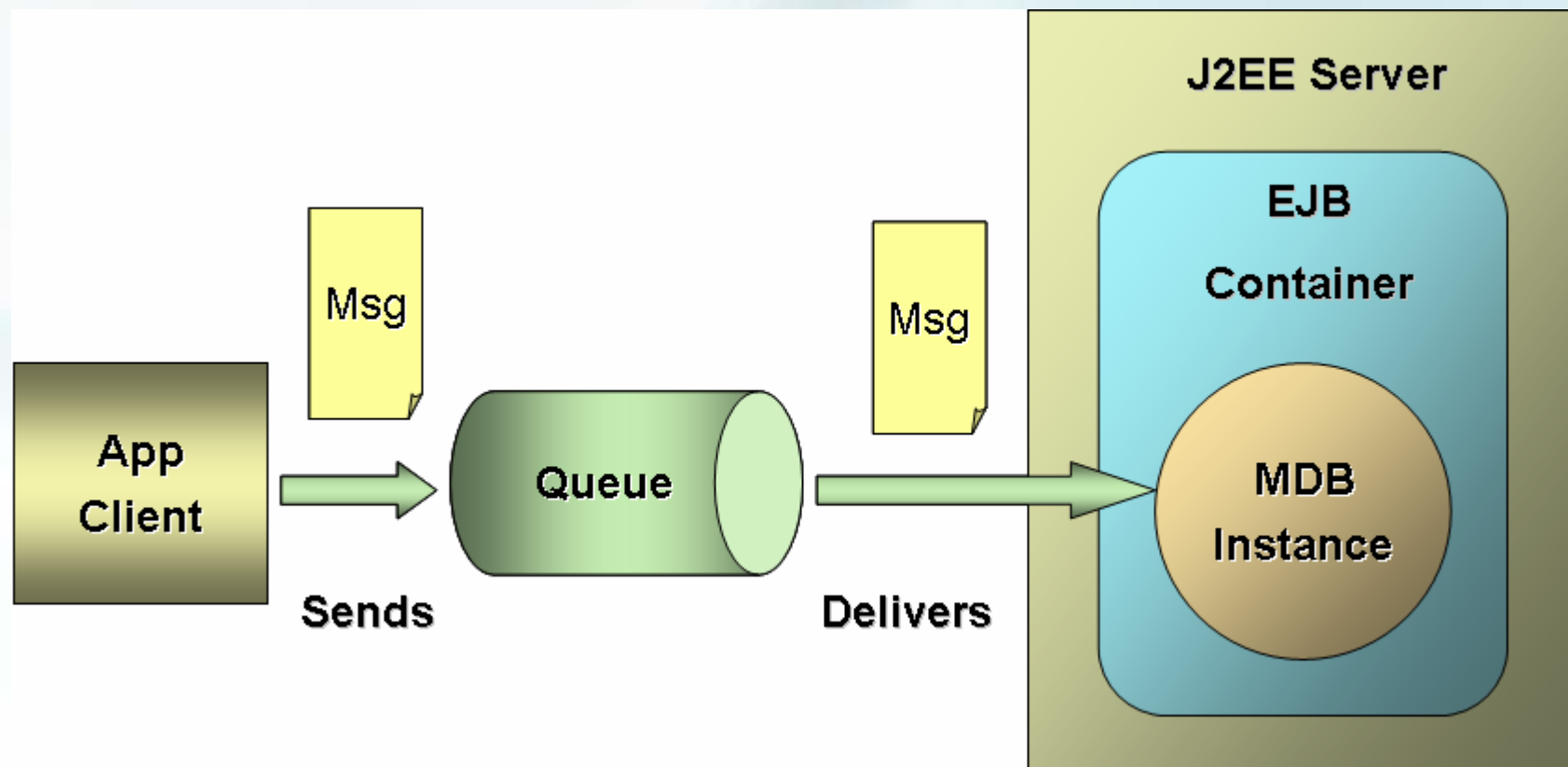
- 典型的集成方法
  - 使用客户/服务器结构
    - 同步适配器
    - 异步适配器
  - 使用消息代理
  - 使用应用服务器



## - J2EE方式-JCA:



## — J2EE方式—JMS与EJB



# 第七章

## 面向服务架构(SOA)设计

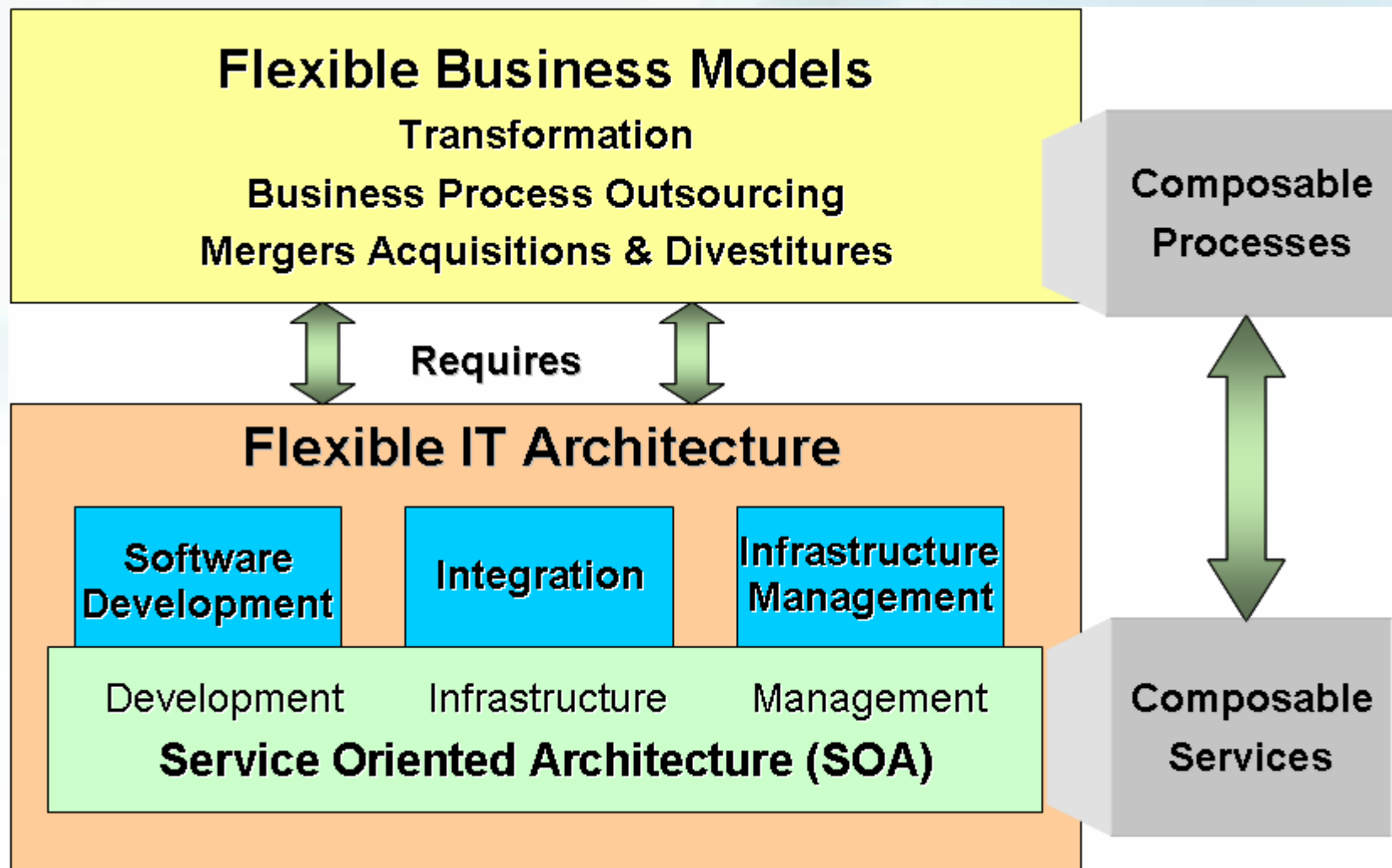
## ● SOA

- 一种体系结构风格
- 将应用程序的不同功能单元（服务）通过服务间定义良好的接口和契约联系起来
- 接口是采用中立的方式进行定义的
- 构建在各种系统中的服务可以以统一和通用的方式进行交互





- 实现可变的IT架构

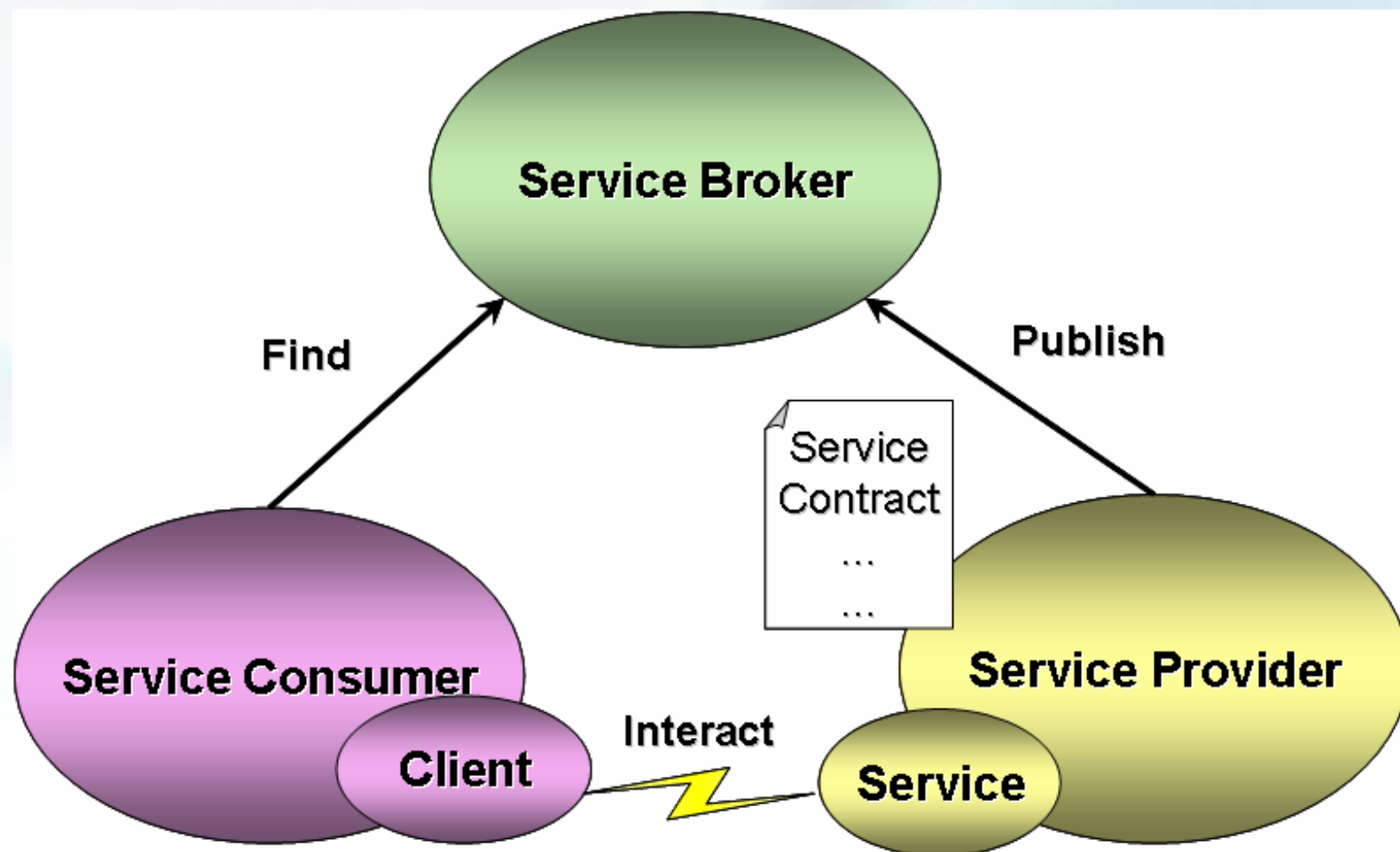


- SOA方法论
  - 服务
  - 服务库
  - 服务总线
  - 服务交互模式
  - SOA的过程、原则和工具



- 服务（Service）
  - 定义良好的，自包含的，不依赖于上下文和其它服务的一组功能
- SOA特征
  - 本质上是一组服务的集合
  - 自包含和模块化
  - 互操作性
  - 松散耦合
  - 位置透明
  - 可组合性
  - 明确定义的接口

## • SOA组成部分

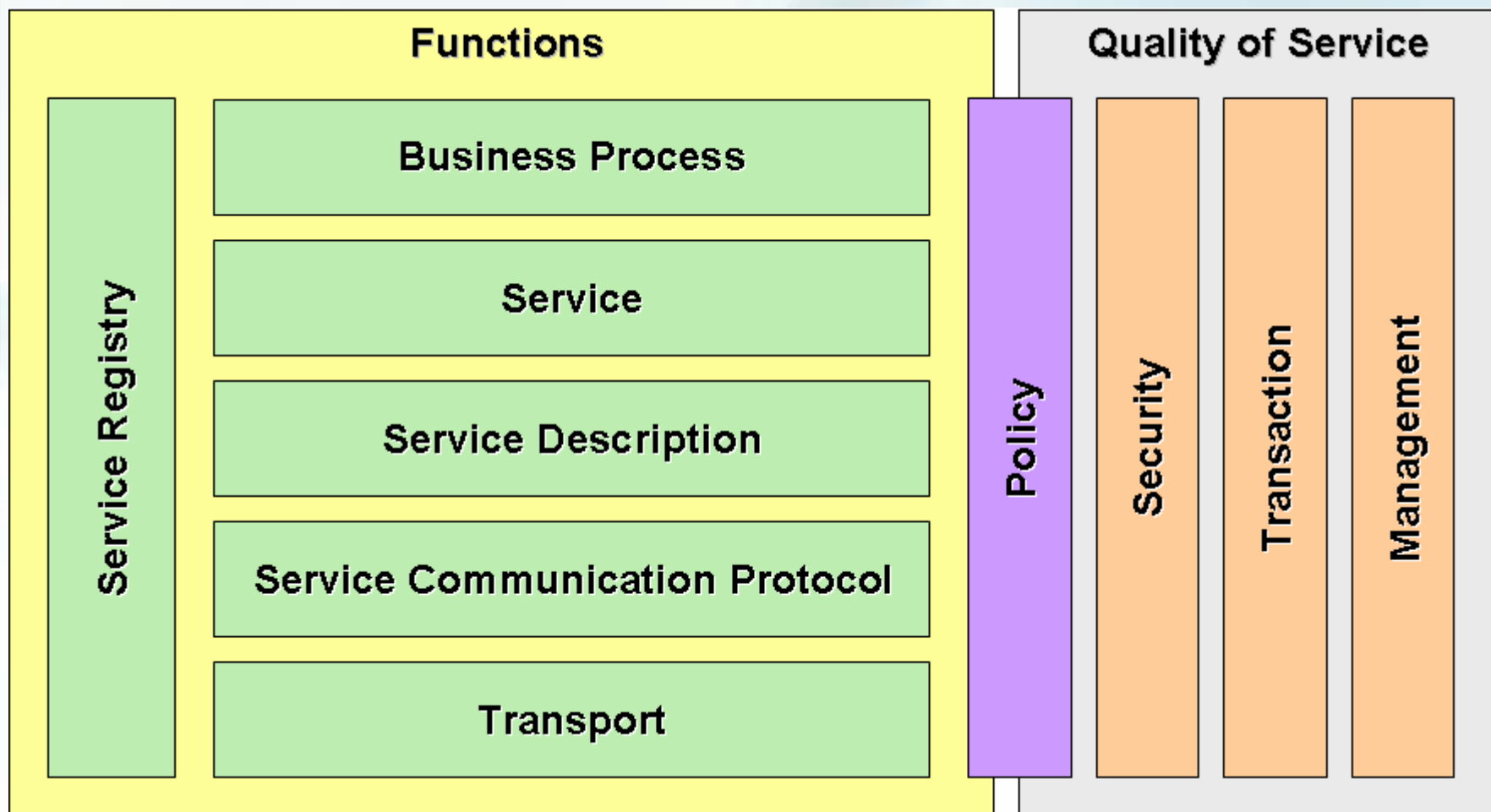


- 服务的编排

- 业务流程按照一组有序的活动进行
- 可以编排服务，以实现业务流程
  - 即，业务流程中的每一步都实现为一个服务
- 业务流程自己也是一个服务
- 使用**BPEL** 描述服务编排
- 服务编排（**Choreography**）提供：
  - 处理应用和使用者的组合
  - 事务及其补偿
  - 操纵流程数据

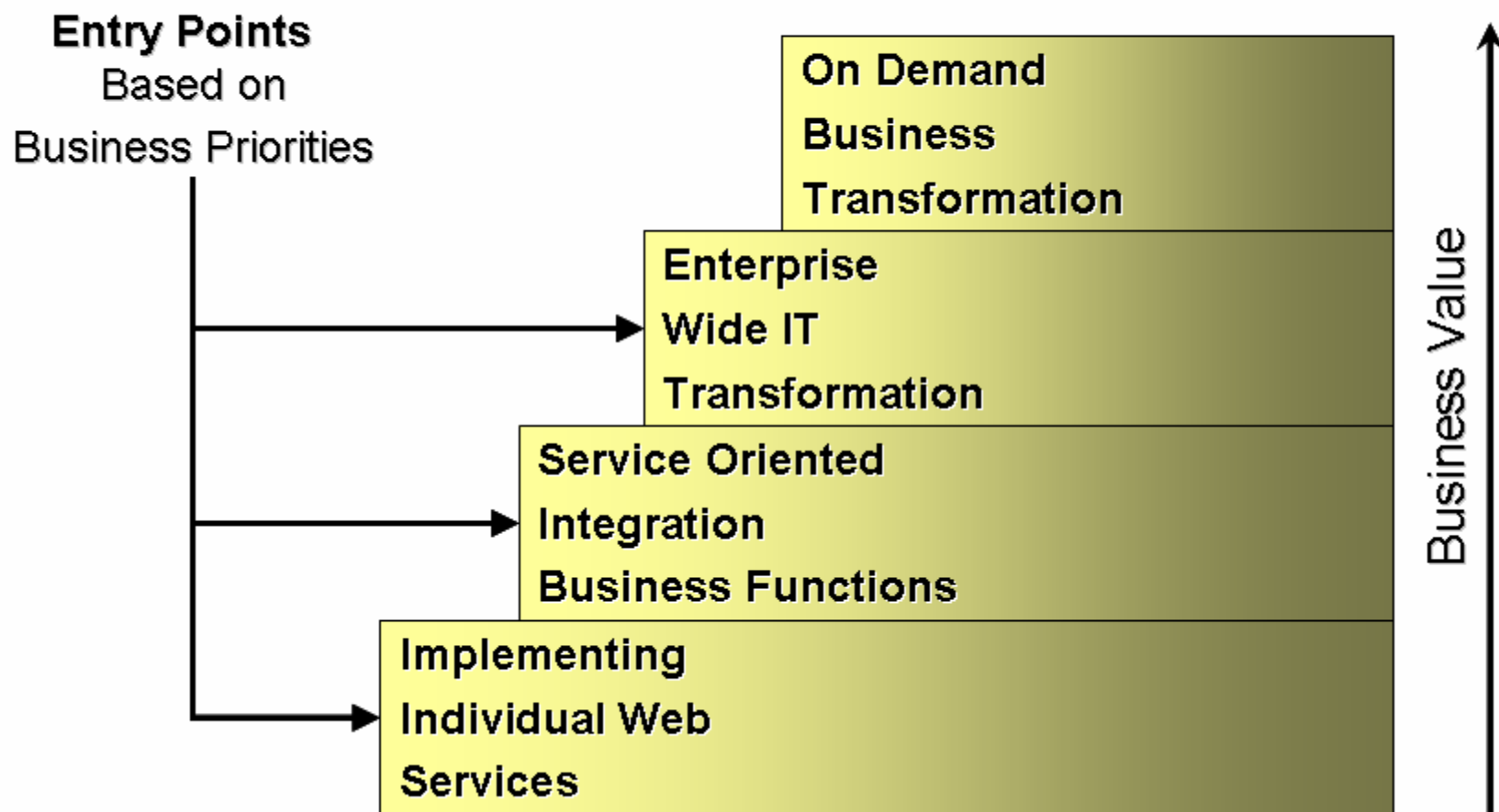
- 服务规范：
  - 结构规范
    - 定义可以调用的操作和由这些操作销毁或创造出的消息
  - 行为规范
    - 表示服务客户和所指定服务之间的预定义的有意义的协议或会话
  - 策略规范
    - 表示服务的策略主张和约束，如安全性、可管理性等

# SOA协议栈



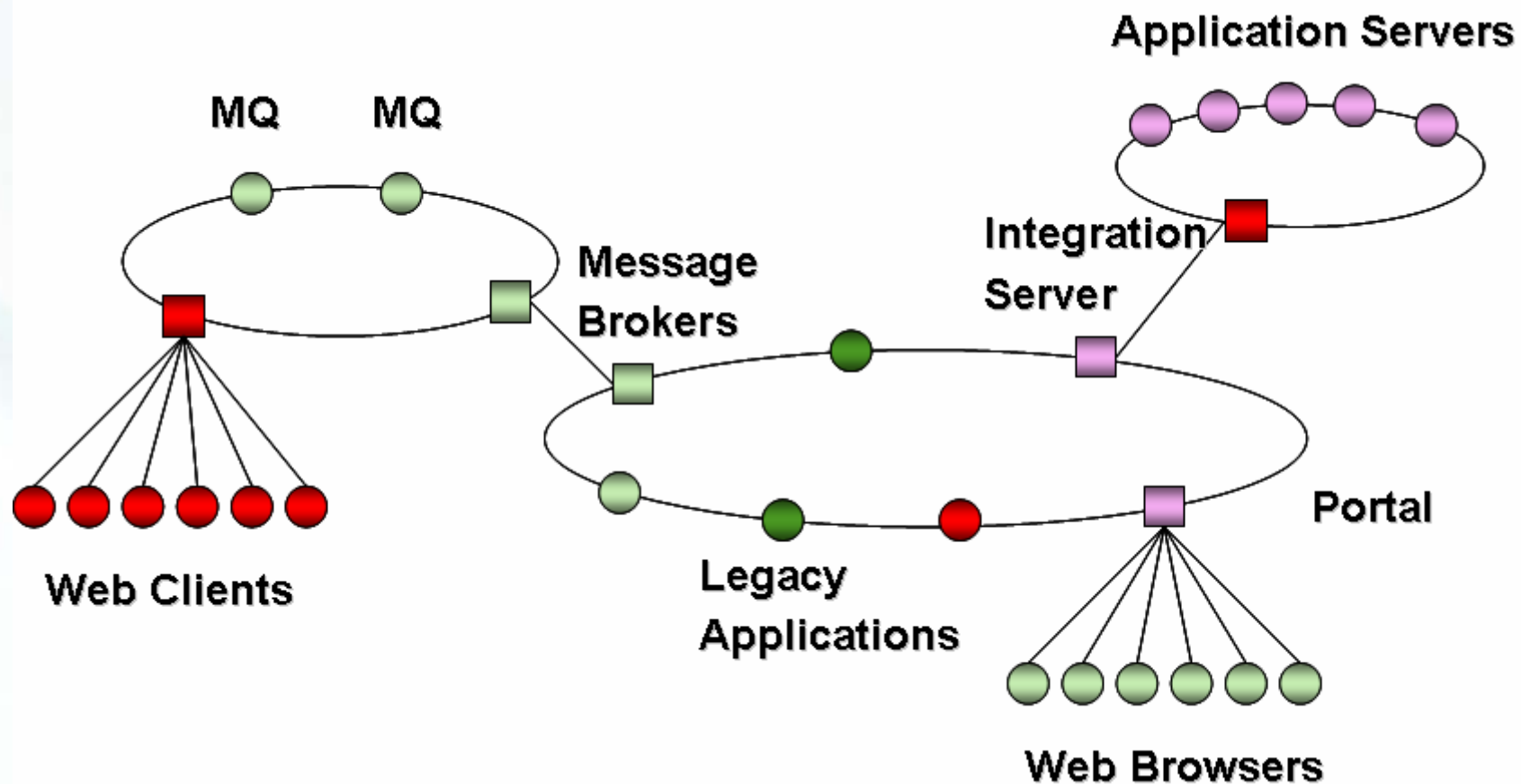
- 面向服务的集成

## SOA Adoption Levels





- 企业服务总线 (ESB)



## ● ESB特点

- 是一种逻辑架构
  - 提供内部连接服务
  - 服务按照每个事务所需要的QoS彼此交互
- 连接并集成企业IT业务
  - 在不同的地点，使用不同的传输方式，可以跨越组织
- 协调服务的请求与响应
  - 执行转换和路由
  - 使连接类型透明
- 允许使用多种协议
  - 如：SOAP/HTTP可以转换成SOAP/JMS，或反之

- ESB的服务

- 中介服务

- 路由、转换

- 事件服务

- 发表/订阅

- 传输服务

- 同步/异步
    - 持久化/非持久化
    - 松耦合/紧耦合

- 基于标准的

- HTTP/HTTPS（可选,WS-Reliable Messaging）
    - JMS, JAX-RPC, SOAP
    - WS-Security, WS-Policy, WS-Addressing

- 开发面向服务的应用
  - Everything is a service
- 理解并定义服务
  - 良好的服务分解是一个挑战
  - 保持粗粒度
  - 某些服务实现已存在，其它的需要创建
- 确定服务接口及交互模式
  - 数据交互采用何种格式？
  - 对交互的服务质量有何要求？
  - 哪些地方需要协调或适合进行协调？
- 根据业务流程定义服务的编排
  - 复合服务是否反映出业务流程？
  - 什么是业务流程

- OO的扩展遇到了挑战
  - 随着时间的推移，接口继承的复杂度在累积
  - 随着系统间距离的延伸，调用成本在上升，类型系统的不同步
  - 扩展组件的功能成本高，不可确定未来需求，不可堆叠的扩展方式
  - 重用与标准化，重用是OO的第一原则，难以维持和维护复杂的重用标准和机制



- OO vs. SOA
  - OO仍然适用于服务的开发
    - 明显的性能优势
    - 成熟的设计与开发方法
  - SOA适用于系统的互联
    - 互操作性的要求强于性能的要求



- SOAD及其组成部分
  - OOAD、BPM和EA



## ● BPM

- 在功能工作单元之上提供了端到端视图，但通常都没有触及架构和实现领域
- 在用于Web 服务的业务流程执行语言（BPEL）之前，BPM 表示法缺少操作语义
- 许多流程建模与开发活动彼此分离





- **OOAD**

- **OO** 支持应用程序分析、设计和开发的完整生命周期
  - **OO**设计的目标是能够进行快速而有效的设计、开发以及执行灵活且可扩展的应用程序
    - 从 **OO** 的角度看，每件事情都是对象
- **SOAD** 应该尽可能多地利用**OO** 分析技术
  - 将**OO** 分析成功地应用于**SOA** 项目
    - 必须一次分析多个系统，用例模型必须继续扮演重要的角色
    - **SOAD** 主要是流程，而不是用户驱动的。**SOAD** 需要**BPM** 和用例建模活动之间的强链接

- 与 SO 有关的 OO 设计的主要问题

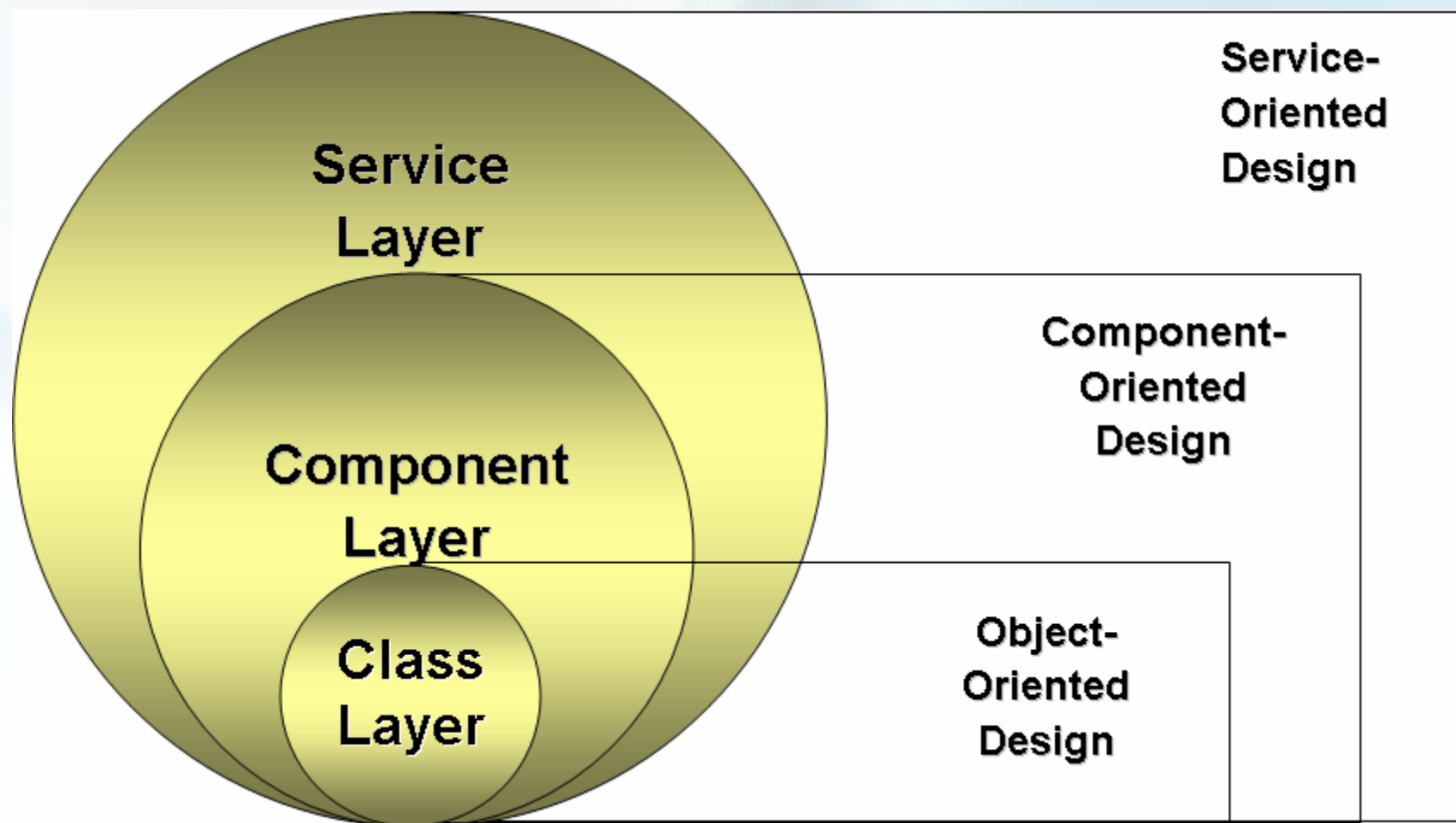
- OOAD

- 粒度级别集中在类级
    - 对于业务服务建模来说，这样的抽象级别过低
    - 诸如继承这样的强关联产生了相关方之间一定程度的紧耦合（因而具有依赖性）

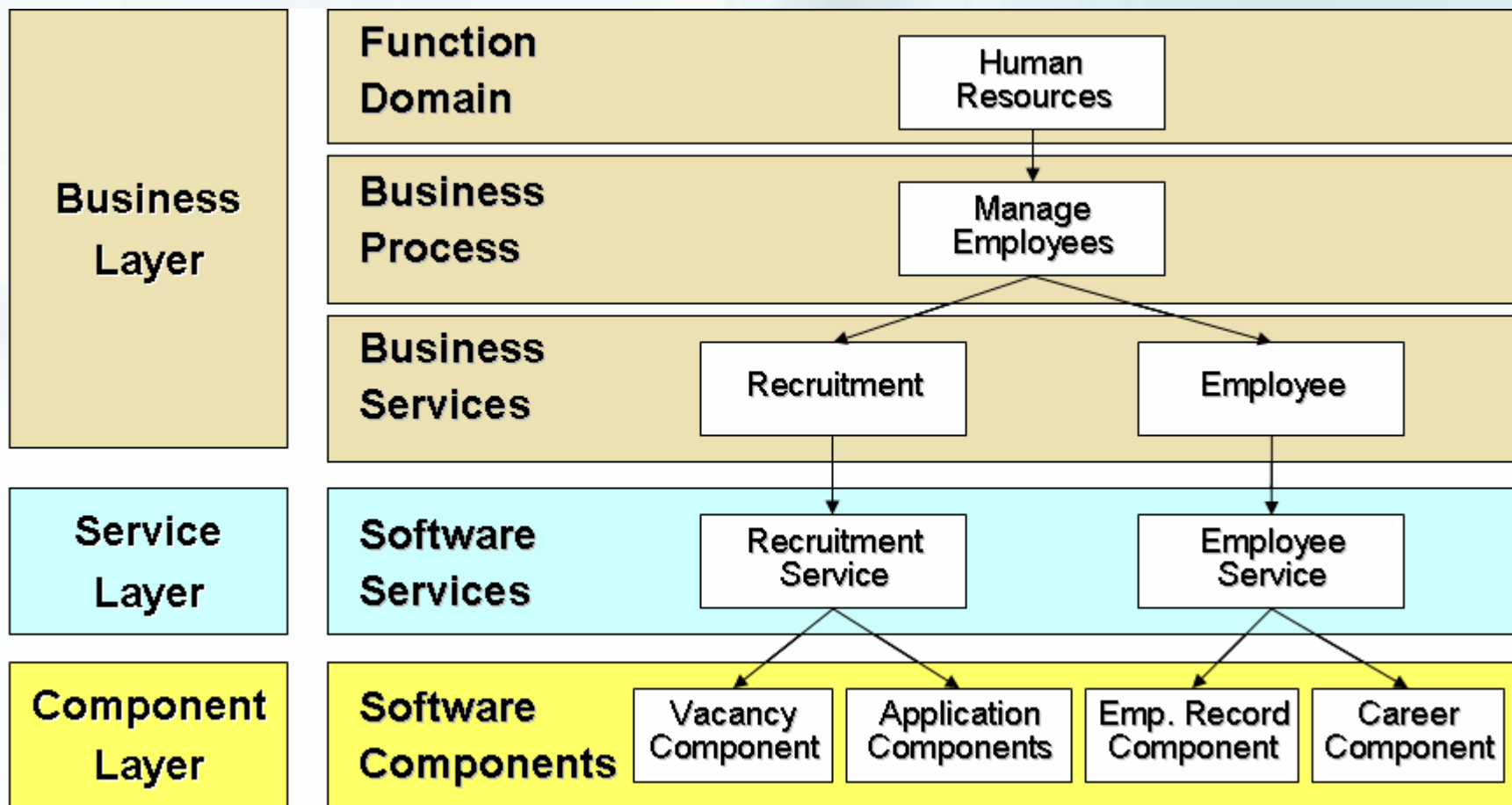
- SO

- 试图通过松耦合来促进灵活性和敏捷性
    - 在SOA 中还没有服务实例的跨平台继承支持和表示法来避免需要处理服务生命周期维护管理问题（如远程垃圾收集）

## • 设计层次



## • SOAD服务定义层次



## ● SOA服务建模方法

- 自顶向下的业务建模是SOA 建模提供起点
- 将创建SOA 解决方案所涉及集成的现有遗留系统分解成服务、操作、业务流程和业务规则
- 将现有的应用程序和厂商软件包分解成表示相关操作组的独立服务集（自底向上方法）
- 从应用程序中将业务流程和规则抽象为由业务编排模型管理的单独BPM

- **Web服务是一种接口**
  - 描述了一组操作
  - 可以使用标准的XML消息传递技术
  - 经由网络访问这些操作
- **Web服务的应用成为松散耦合的、面向组件的、跨技术的实现**
- **Web服务优点**
  - 互操作性
  - 可伸缩性
  - 高可靠性
  - 可管理性



- 不同视角看Web服务

- 业务视角

- Web服务是一业务过程或是业务过程中的步骤

- 技术视角

- Web服务就是关于集成的技术
    - Web服务只是一个或多个相关操作的集合，这些操作可以由网络访问，并可以用服务描述来描述



- Web 服务中的QoS性能
  - 可用性
  - 可靠性
  - 完整性
  - 安全性
  - 可访问性

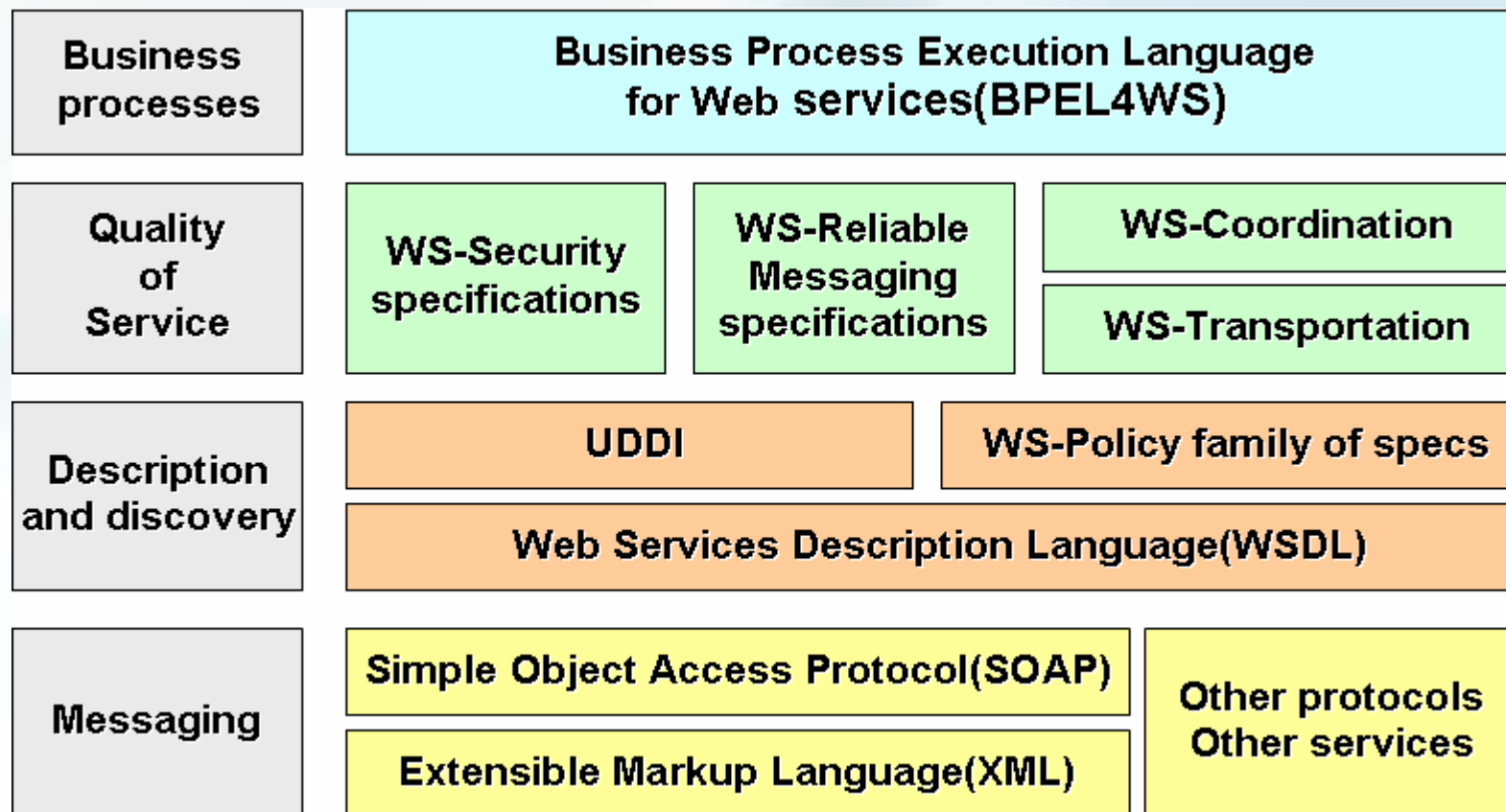




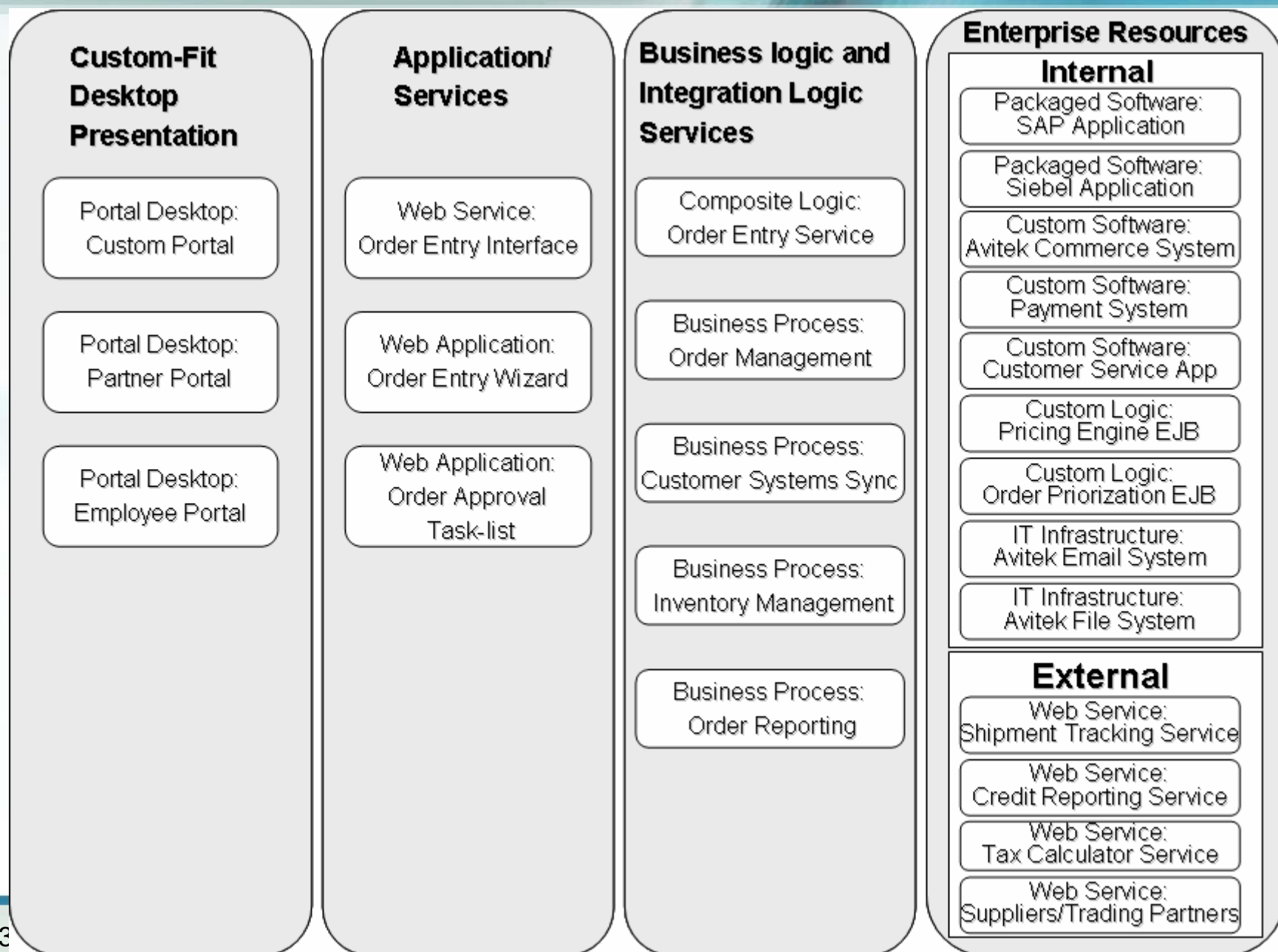
- **Web服务核心技术**
  - 服务描述—WSDL
  - 服务发布、发现和绑定—UDDI
  - 服务调用和执行—SOAP
  - 服务实现—各种实现技术



# • Web Service Stack for SOA



# • WS应用示例:



# 第八章

## 应用框架的设计与实现

- 应用框架 (Application Framework)
  - 提供了结构和模板，开发者以此为基线 (Baseline) 来构建其应用系统
  - 用于描述有助于软件应用开发的一组可重用的设计和代码
  - "结构"一词反映了任何框架的本质
    - 结构帮助我们将应用中不断变化的细枝部分，组织成易于理解的少数几个部分



- 应用框架的发展
  - 用户界面
  - 通用应用开发
  - 业务领域



- 应用框架的作用
  - 模块化 (Modularity)
  - 可重用性 (Reusability)
  - 可扩展性 (Extensibility)
  - 简单性 (Simplicity)
  - 可维护性 (Maintainability)

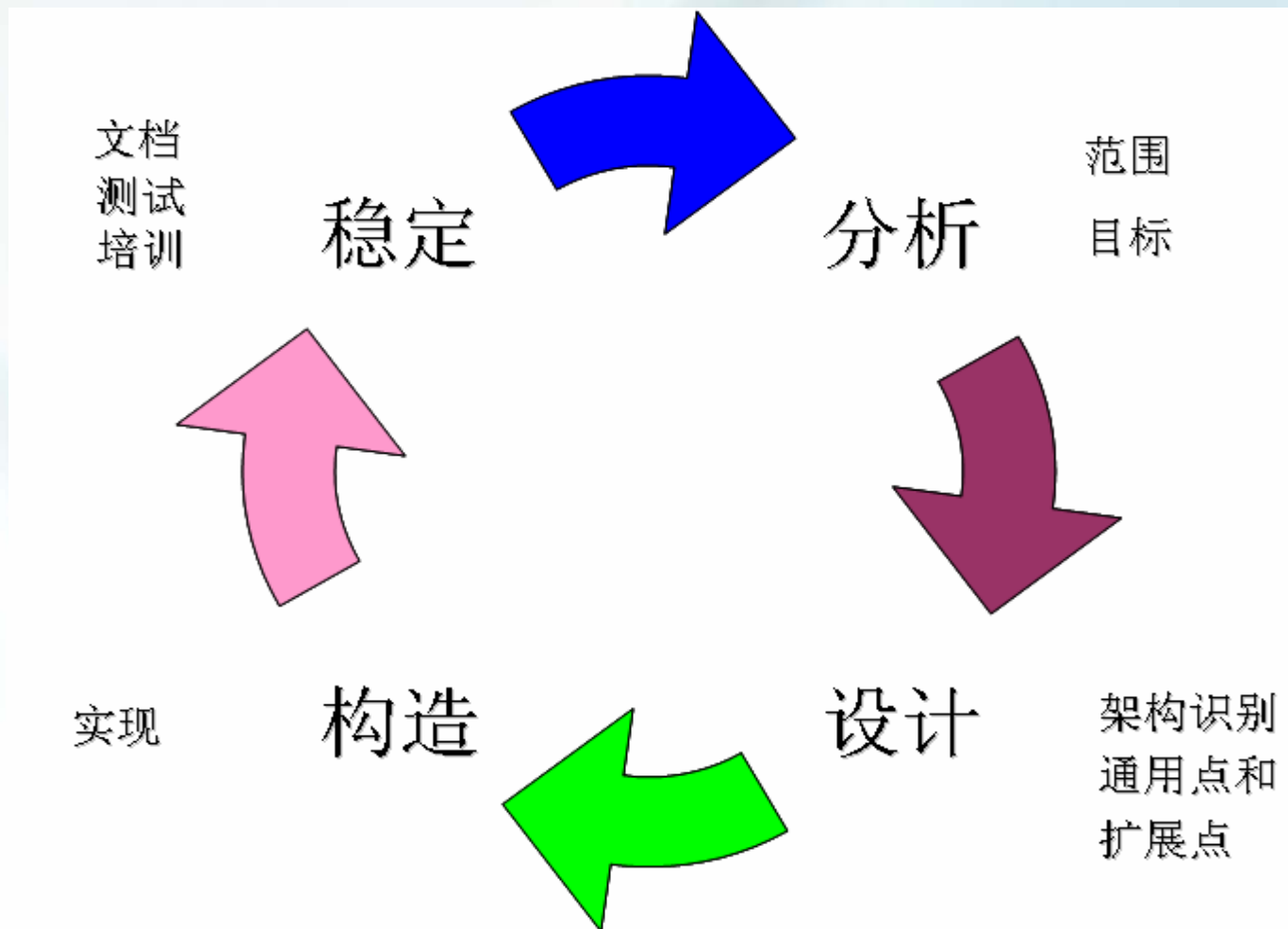


- 应用框架的分层
  - 业务应用层 (Business Application)
  - 应用框架层 (Application Framework)
  - 基础框架层 (Foundation Framework)
  - 操作系统层 (Operating System)





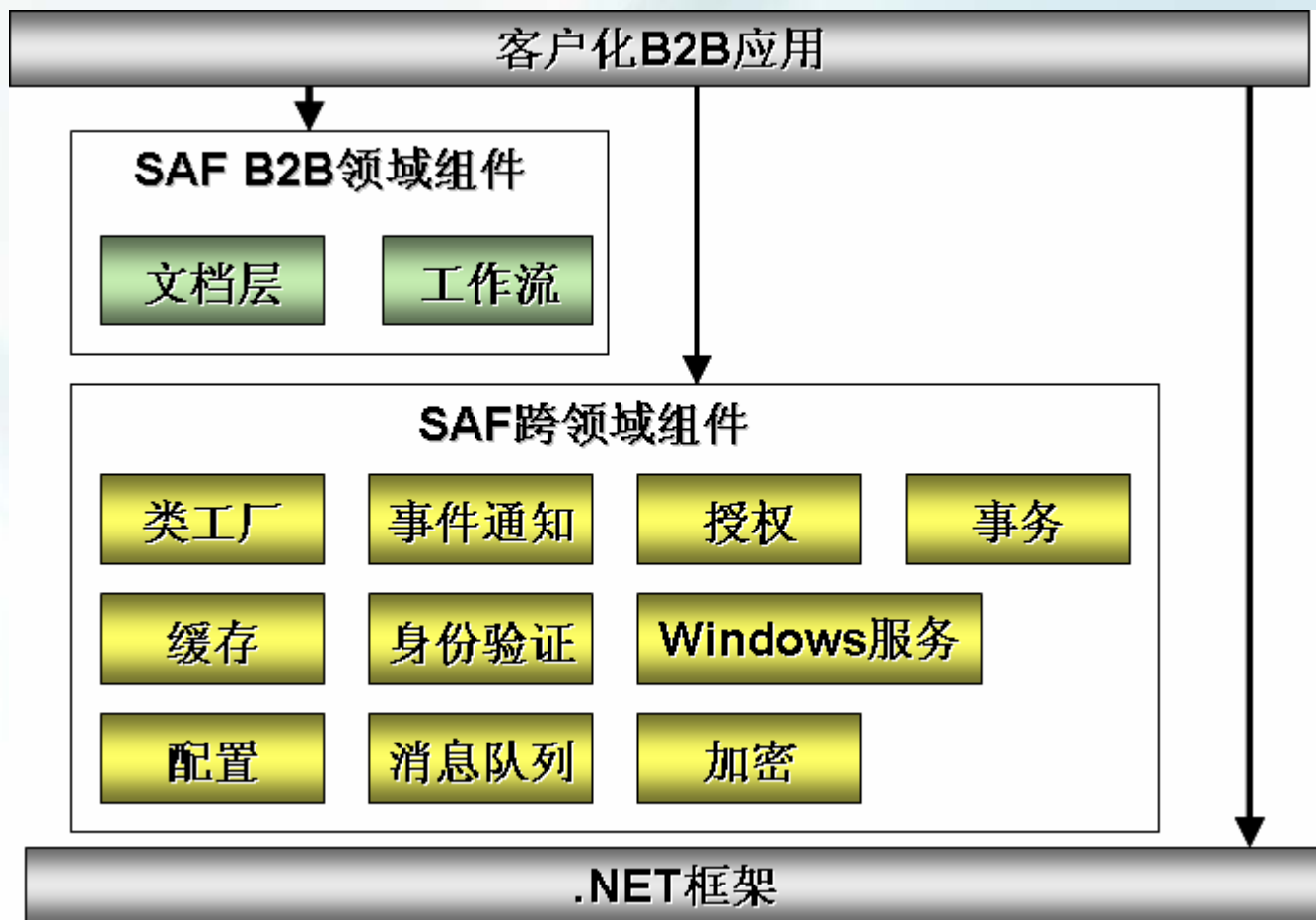
- 框架开发过程包括的四个主要阶段：
  - 分析、设计、实现和稳定



- 开发框架的技术
  - 通用点 (Common spots)
  - 扩展点 (Hot spots): 占位符号/扩展点
  - 设计模式 (Design pattern)
- 开发框架的方法
  - 黑盒框架 (Black-box framework)
  - 白盒框架 (White-box framework)
  - 灰盒框架 (Gray-box framework)



- SAF（Simplified Application Framework）
  - B2B例：



## — SAF的基本组件

- 类工厂服务 (Class Factory Service)
- 缓存服务 (Caching Service)
- 配置服务 (Configuration Service)
- 事件通知服务 (Event Notification Service)
- 消息队列服务 (Message Queue Service)
- 授权服务 (Authorization Service)
- 身份验证服务 (Authentication Service)
- 加密服务 (Cryptography Service)
- 事务服务 (Transaction Service)
- Windows服务"服务 (Window Service Service)

- **SAF B2B领域组件**
  - 文档层服务 (Document Layer Service)
  - workflow服务 (Workflow Service)



- 类工厂服务

- 动机和目标:

- 减少"指定具体类"的要求
    - 为开发者隐藏对象创建的复杂性
    - 无论是否是远程对象, 使对象的创建不受位置、版本等限制

- 由类工厂根据实际设置来决定类的位置、版本等

- 实现

- 反射技术
    - 抽象工厂模式



- 缓存服务

- 动机和目标:

- 提供对象存取机制
    - 为应用开发者提供简单易用的接口
    - 使开发具有更好的强健性和可伸缩性

- 实现

- 静态变量与哈希表(Hashtable)
    - 采用XML
    - 策略模式
    - 单例模式

- 配置服务

- 动机和目标:

- 为应用配置提供统一的解决方案
    - 可以为应用的不同部分定义多个配置
    - 也可以定义一个控制其它散布配置的配置类来进行集中式管理
      - 减少应用中的重复配置数据

- 实现

- 将 XML用于应用配置





## ● 事件通知服务

### — 动机和目标：

- 为开发者在应用中实现异步过程提供简单的方法
- 解耦事件客户和事件服务器，让它们能够独立地改变
- 使属于不同进程域或系统的应用可以通过事件通知相互通信

### — 实现：

- 使用应用服务器的消息服务
- 通过事件订阅和分发模型（或称事件通知模型）
- 观察者模式
- 中介者模式

- 消息队列服务

- 动机和目标

- 处理各种消息队列实现方案所支持的消息队列功能

- 实现

- 使用应用服务器的消息服务
    - 桥接模式



- 授权服务

- 动机和目标

- 在不修改和重新编译代码的情况下，就能重新设置业务组件访问权限
    - 需要开发自定义安全特性进行安全权限检查

- 实现

- 使用应用服务器的安全服务，可以直接通过配置文件改变业务组件的权限



- 身份验证服务

- 动机和目标

- 为应用提供单点登录服务

- 即使应用中各部分识别用户的方式不尽相同，由该组件协调，以保证用户只需验证一次就获得所有部分的"信任"



- 加密服务

- 动机和目标

- 提供一组类简化数据的加密和解密，使开发者可以保证敏感信息的安全



- 事务服务

- 动机和目标

- 支持开发者将多个方法组织到一个事务中
    - 支持用户为事务组件指定不同的事务特征，而无需建立重复的事务组件

- 实现

- 事务服务的设计
    - 事务锁和隔离级别



- “Windows服务”服务

- 动机和目标:

- 开发Windows服务应用，需要Windows服务应用创建一个特殊的"服务"
    - 和常规的控制台应用或Windows应用不同，为安装Windows服务应用，需要开发一个特殊的安装项目



- 文档层服务

- 动机和目标

- 将不同的文档处理装配成一条文档处理链
    - 可以通过修改配置来调整链中的文档处理，以达到改变文档处理过程的目的

- 实现

- 装饰模式





- workflow 服务

- 动机和目标

- 用于多业务组件处理文档
    - 使用该组件提供的服务，开发者能够抽取出协调逻辑，放入和业务组件分离的专用组件中

- 实现

- 访问者模式



# 讨论与交流



DTT