软件体系结构设计与构建

刘钦

预习测试

- 1. 体系结构设计的过程步骤
- 2. 列举一个非功能性需求对体系结构设计的影响。画出考虑前后的设计图。

预习测试

- 1. 体系结构设计的过程步骤。
- 2. 画出学生选课业务逻辑chooseCoursebl的关键类图。

主要内容

- 体系结构设计
- 体系结构构建
- 体系结构文档化
- 体系结构验证

体系结构设计

体系结构设计过程（简化版）

1. 分析关键需求和项目约束；

2. 通过选择体系结构风格；

逻辑设计

3. 进行软件体系结构逻辑（抽象）设计；

4. 依赖逻辑设计进行软件体系结构（实现）设计；

5. 完善体系结构设计；

物理设计

6. 添加构件接口；

7. 迭代过程3-7

体系结构设计过程

1. 分析关键需求和项目约束；

2. 通过选择体系结构风格；

3. 进行软件体系结构逻辑（抽象）设计；

4. 依赖逻辑设计进行软件体系结构（实现）设计；

5. 完善体系结构设计；

6. 添加构件接口；

7. 迭代过程3-7

体系结构需求

- 功能需求
- 非功能性需求
  - 质量
  - 性能
  - 约束
  - 接口
- 项目约束
  - 开发团队
  - 市场大小
  - 项目预算
  - 项目进度
  - 项目风险
  - 开发环境
  - 开发技术

实践案例

- 1. 需求
  - a) 概要功能需求:10 个功能
  - b) 非功能性需求
    - 安全需求:Security1~3
    - 约束:IC2
- 2. 项目约束
  - a) 开发技术:Java
  - b) 时间较为紧张
  - c) 开发人员:不熟悉 Web 技术

体系结构设计过程

1. 分析关键需求和项目约束；
2. 通过选择体系结构风格；
3. 进行软件体系结构逻辑（抽象）设计；
4. 依赖逻辑设计进行软件体系结构（实现）设计；
5. 完善体系结构设计；
6. 添加构件接口；
7. 迭代过程3-7

实践案例

- 分层风格
  - 协议不变情况下易于修改
  - 能够促进并行开发

体系结构设计过程

1. 分析关键需求和项目约束；
2. 通过选择体系结构风格；
3. 进行软件体系结构逻辑（抽象）设计；
    1. 依据概要功能需求与体系结构风格建立初始设计
    2. 使用非功能性需求与项目约束评价和改进初始设计
4. 依赖逻辑设计进行软件体系结构（实现）设计；
5. 完善体系结构设计；
6. 添加构件接口；
7. 迭代过程3-7

将需求分配到子系统和模块

- 考虑功能的相同性
  - 不同任务，但是相同功能
- 考虑可复用性
  - 结构、数据、行为的可复用性

实践案例

- 销售与退货
- 产品调整、入库、出库与库存分析
- 会员发展与礼品赠送
- 销售策略

- 基本功能
  - 销售
  - 库存
  - 会员
  - 销售策略
  - 用户

功能	对应逻辑包
销售	SalesUI, Sales, SalesData
库存	CommodityUI, Commodity, CommodityData
会员	MemberUI, Member, MemberData
销售策略	PromotionUI, Promotion, PromotionData
调整用户	UserUI, User, UserData

销售是否只用到Sale的数据

功能	对应逻辑包
销售	SalesUI, Sales, SalesData; Commodity, CommodityData; Member, MemberData; Promotion, PromotionData; User, UserData;
库存	CommodityUI, Commodity, CommodityData Sales, SalesData
会员	MemberUI, Member, MemberData Sales, SalesData Commodity, CommodityData
销售策略	PromotionUI, Promotion, PromotionData
调整用户	UserUI, User, UserData

连锁超市管理系统的改进概要功能设计

pkg
presentation
UserUI
SalesUI
MemberUI
CommodityUI
PromotionUI
businesslogic
User
Sales
Member
Commodity
Promotion
data
Database
powered by astah*

初步设计方案 一

pkg
presentation
userui
salesui
memberui
commodityui
promotionui
businesslogic
salesbl
commoditybl
userbl
memberbl
promotionbl
data
userdata
salesdata
memberdata
commoditydata
promotiondata

初步设计方案二

体系结构设计过程

1. 分析关键需求和项目约束；
2. 通过选择体系结构风格；
3. 进行软件体系结构逻辑（抽象）设计；
    1. 依据概要功能需求与体系结构风格建立初始设计
    2. 使用非功能性需求与项目约束评价和改进初始设计
4. 依赖逻辑设计进行软件体系结构（实现）设计；
5. 完善体系结构设计；
6. 添加构件接口；
7. 迭代过程3-7

初步设计的分析

- (1) 能够满足项目约束:
  - 分层风格能促进并行开发,从而缩短开发时间;
  - 分层风格可以使用 Java 技术,而不使用 Web 技术。
- (2) 无法满足安全需求(Security1~3)和网络分布约束(IC2),所以需要改进:
  - 为使其满足安全需求,可以增加用户登录与验证功能,可以建立专门的三个模块 (Presentation、Logic、Data),也可以将该功能并入用户管理功能,即为 userui, user, userdata 三个模块增加新的职责。
  - 为满足网络分布约束,需要将模块分布到客户端和服务器组成的网络上。可以将 Prensentation 层模块部署在客户端,将 Logic 层和 Data 层的模块部署在服务器端。也可将 Prensentation 层和 Logic 层模块部署在客户端,将 Data 层的模块部署在服务器端。一旦相邻 两层被部署到网络两端,那么它们之间的交互就无法通过程序调用来完成,可以考虑将简单 的程序调用转化为远程方法调用 RMI。

连锁超市管理系统最终的软件体系结构逻辑设计方案

体系结构设计过程

1. 分析关键需求和项目约束；

2. 通过选择体系结构风格；

3. 进行软件体系结构逻辑（抽象）设计；

4. 依赖逻辑设计进行软件体系结构（实现）设计；

5. 完善体系结构设计；

6. 添加构件接口；

7. 迭代过程3-7

Package Design Principles

- Reuse-Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principles
- Stable Dependencies Principles
- Stable Abstractions Principles

Cohesion
Reuse &&
Change

Coupling
Compile &&
Link

Common Closure Principle (CCP)
共同封闭原则

- Classes which change together belong together
- Minimize the impact of change for the programmer.
- When a change is needed, it is good for the programmer
- if the change affects as few packages as possible, because of compile and link time and revalidation

CCP in Project?

接口
数据
传递
对
象?

CCP Summary

- Group classes with similar closure together
  - package closed for anticipated changes
- Confines changes to a few packages
- Reduces package release frequency
- Reduces work for the programmer

Common Reuse Principle (CRP)
共同重用原则

- Classes in packages should be reused together
- Packages should be focused, users should use all classes from a package

CRP Summary

- Group classes according to common reuse
  - avoid unnecessary dependencies for users
- Following the CRP often leads to splitting packages
  - Get more, smaller and more focused packages
- Reduces work for the reuser

Tradeoffs Between CCP and CRP

- CCP and CRP principles are mutually exclusive, i.e. they cannot simultaneously be satisfied.
- The CRP makes life easy for reusers, whereas the CCP makes life easier for maintainers.
- The CCP strives to make packages as large as possible, whereas the CRP tries to make packages very small.
- Early in a project, architects may set up the package structure such that CCP dominates and development and maintenance is aided. Later, as the architecture stabilizes, the architects may refactor the package structure to maximize CRP for the external reusers.

Reuse-Release Equivalency Principle (REP)
重用发布等价原则

- The unit of reuse is the unit of release
- It is about reusing software
- Reusable software is external software, you use it but somebody else maintains it.
- There is no difference between commercial and non-commercial external software for reuse.

REP in Project?

Utility Package?

Tools Package?

REP Summary

- Group components (classes) for reusers
- Single classes are usually not reusable
  - Several collaborating classes make up a package
- Classes in a package should form a reusable and releasable module
  - Module provides coherent functionality
- Reduces work for the reuser

The Acyclic Dependencies Principle (ADP)
无环依赖原则

- The dependency structure for packages must be a Directed Acyclic Graph (DAG)
- Stabilize and release a project in pieces
- Organize package dependencies in a top-down hierarchy

What would be required to release the *Protocol* package?

- The engineers would have build their test suite with *CommError*, *GUI*, *Comm*, *ModemControl*, *Analysis*, and *Database* packages!!!

Dependencies are a DAG

- The classes that *CommError* needed are pulled out of *GUI* and placed in a new package named *MessageManager*. Both *GUI* and *CommError* are made to depend upon this new package.

aDependency Cycles
Breaking a Cycle: 1st Way

- Here we see two packages that are bound by a cycle.
- Class A depends upon class X, and class Y depends upon class B.
- We break the cycle by inverting the dependency between Y and B.
- This is done by adding a new interface, BY, to B. This interface has all the methods that Y needs. Y uses this interface and B implements it.

Dependency Cycles Breaking a Cycle: 2nd Way

ADP in project?

- 层次式风格和主程序/子路径风格通常不会发生

- 面向对象式风格尤其要注意
- C/S的MVC风格可能会发生

- 基于数据流、事件/消息、数据共享进行交互的体系结构风格通常不会发生

Stable Dependencies Principle (SDP)
稳定依赖原则

- Dependencies should point in the direction of stability
- Stability: corresponds to effort required to change a package
- Stable package: hard to change within the project
- Stability can be quantified

- X is stable

- Y is instable

SDP Example

包的稳定性度量

- 一种方法是计算进、出该包的依赖关系的数目。可以使用这些数值来计算该包的位置稳定性（positional stability）。
  - （Ca ）输入耦合度（Afferent Coupling）： 指处于该包的外部并依赖于该包内的类的类的数目。
  - （Ce ）输出耦合度（Efferent Coupling）： 指处于该包的内部并依赖于该包外的类的类的数目。
  - （不稳定性I）
    - I = Ce / (Ca +Ce)

- 该度量的取值范围是[0 ， 1]。I=0表示该包具有最大的稳定性。I=1表示该包具有最大的不稳定性。通过计算和一个包内的类有依赖关系的包外的类的数目，就可以计算出试题Ca 和Ce 。

- 当一个包的I度量值为 1 时，就意味着没有任何其他的包依赖于该包（Ca =0）；而该包却依赖于其他的包（Ce>0）。这是一个包最不稳定的状态：它是不承担责任且有依赖性的。因为没有包依赖于它，所以它就没有不改变理由，而它所依赖的包会给它提供丰富的更改理由。

- 另一方面，当一个包的I度量值为 0 时，就意味着其他包会依赖于该包（Ca >0），但是该包却不依赖于任何其他的包（Ce =0)。它是负有责任且无依赖性的。这种包达到了最大程度的稳定性。它的依赖者使其难以更改，而且没有任何依赖关系会迫使它去改变

SDP in Project?

MVC?

Stable Abstractions Principle (SAP)
稳定抽象原则

- Stable packages should be abstract packages.
- Unstable packages should be concrete packages.
- Stable packages contain high level design.
- Making them abstract opens them for extension but closes them for modifications (OCP).
- Some flexibility is left in the stable hard-to-change packages.

抽象性度量

- 包的抽象性用抽象类的数目和包中所有类的数目进行计算。
- 假如说包中类的总数是Nc, 抽象类的数目是Na ,那么抽象度A = Na/Nc

距离D=|Abstraction+Instability-1|/根号2
规范化的距离D‘=|Abstraction+Instability-1|

到主序列的距离

SAP in Project?

抽象接口包

包设计的过程

- 迭代的过程
  - 先CCP 原则对把可能一同变化的类组织成包进行发布,
  - 随着系统的不断增长,我们开始关注创建可重用的元素,于是开始使用 CRP 和 REP 来指导包的组合。
  - 最后使用 ADP、SDP、SAP 对包图进行度量，去掉不好的依赖。

体系结构设计过程

1. 分析关键需求和项目约束；
2. 通过选择体系结构风格；
3. 进行软件体系结构逻辑（抽象）设计；
4. 依赖逻辑设计进行软件体系结构（实现）设计；
    1. 开发包(构件)设计
    2. 运行时的进程
    3. 物理部署
5. 完善体系结构设计；
6. 添加构件接口；
7. 迭代过程3-7

commoditybl	*commoditybl*	*commoditydata, salesbl*
commoditydata	*commoditydata*	
memberui	*memberui*	*memberbl*
memberbl	*memberbl*	*memberdata, salesbl, commoditybl*
memberdata	*memberdata*	
promotionui	*promotionui*	*promotionbl*
promotionbl	*promotionbl*	*promotiondata*
promotiondata	*promotiondata*	
userui	*userui*	*user*
userbl	*userbl*	*userdata*
userdata	*userdata*	

初始物理包

表 10-4 连锁超市管理系统的最终开发包设计

开发（物理）包	依赖的其他开发包
mainui	userui, salesui, memberui, commodityui, promotionui, vo
salesui	salesblservice,界面类库包, vo
salesblservice	
salesbl	salesblservice, salesdataservice, po, promotionbl, userbl
salesdataservice	Java RMI, po
salesdata	databaseutility, po, salesdataservice
commodityui	commodityblservice，界面类库包
commodityblservice	
commoditybl	commodityblservice, commoditydataservice, po , salesbl
commoditydataservice	Java RMI, po
commoditydata	Java RMI, po, databaseutility
memberui	memberblservice，界面类库包
memberblservice	
memberbl	memberblservice, memberdataservice, po, salesbl, commoditybl
memberdataservice	Java RMI, po
memberdata	Java RMI, po, databaseutility
promotionui	promotionblservice，界面类库包
promotionblservice	
promotionbl	promotionblservice, promotiondataservice, vo
promotiondataservice	Java RMI, po
promotiondata	Java RMI, po, databaseutility
userui	userblservice，界面类库包
userblservice	
userbl	UserInterface, UserDataClient, UserPO
userdataservice	Java RMI, po
userdata	RMI, po, databaseutility
vo	
po	
utilitybl	
界面类库包	
Java RMI	
databaseutility	JDBC

细节考虑

(1) Presentation 层与 Logic 层被置于客户端,Data 层被置于服务器端,那么 Logic 层的开发包依赖于 Data 层的开发包是不可能的。

a) 可以考虑使用 RMI 技术,RMI 技术会将 Data 层开发包分解为置于客户端的 dataservice 接口包和置于服务器的 data 开发包。这样一来,Logic 层开发包依赖于 dataservice 包,dataservice 和 data 层的开发包都依赖于 RMI 类库包。

(2) 所有的 Data 层开发包都需要进行数据持久化(例如读写数据库、读写文件等),所以它们会有一些重复代码,可以将重复代码独立为新的开发包然后所有的 Data 层开发包都依赖于 databaseutility,databaseutility 会依赖于 JDBC 类库包或者 IO 类 库包。

细节考虑

表 10-4 连锁超市管理系统的最终开发包设计

开发（物理）包	依赖的其他开发包
mainui	userui, salesui, memberui, commodityui, promotionui, vo
salesui	salesblservice,界面类库包, vo
salesblservice	
salesbl	salesblservice, salesdataservice, po, promotionbl, userbl
salesdataservice	Java RMI, po
salesdata	databaseutility, po, salesdataservice
commodityui	commodityblservice，界面类库包
commodityblservice	
commoditybl	commodityblservice, commoditydataservice, po , salesbl
commoditydataservice	Java RMI, po
commoditydata	Java RMI, po, databaseutility
memberui	memberblservice，界面类库包
memberblservice	
memberbl	memberblservice, memberdataservice, po, salesbl, commoditybl
memberdataservice	Java RMI, po
memberdata	Java RMI, po, databaseutility
promotionui	promotionblservice，界面类库包
promotionblservice	
promotionbl	promotionblservice, promotiondataservice, vo
promotiondataservice	Java RMI, po
promotiondata	Java RMI, po, databaseutility
userui	userblservice，界面类库包
userblservice	
userbl	UserInterface, UserDataClient, UserPO
userdataservice	Java RMI, po
userdata	RMI, po, databaseutility
vo	
po	
utilitybl	
界面类库包	
Java RMI	
databaseutility	JDBC

- (3) 所有的 Presentation 层开发包都需要使用图形类型建立界面,都要依赖于图形界 面类库包。
- (4) 此外,Presentation 层实现时,由 mainui 包负责整个页面之间的跳转逻辑。其它 各包负责各自页面自身的功能。

表 10-4 连锁超市管理系统的最终开发包设计

开发（物理）包	依赖的其他开发包
mainui	userui, salesui, memberui, commodityui, promotionui, vo
salesui	salesblservice,界面类库包, vo
salesblservice	
salesbl	salesblservice, salesdataservice, po, promotionbl, userbl
salesdataservice	Java RMI, po
salesdata	databaseutility, po, salesdataservice
commodityui	commodityblservice，界面类库包
commodityblservice	
commoditybl	commodityblservice, commoditydataservice, po , salesbl
commoditydataservice	Java RMI, po
commoditydata	Java RMI, po, databaseutility
memberui	memberblservice，界面类库包
memberblservice	
memberbl	memberblservice, memberdataservice, po, salesbl, commoditybl
memberdataservice	Java RMI, po
memberdata	Java RMI, po, databaseutility
promotionui	promotionblservice，界面类库包
promotionblservice	
promotionbl	promotionblservice, promotiondataservice, vo
promotiondataservice	Java RMI, po
promotiondata	Java RMI, po, databaseutility
userui	userblservice，界面类库包
userblservice	
userbl	UserInterface, UserDataClient, UserPO
userdataservice	Java RMI, po
userdata	RMI, po, databaseutility
vo	
po	
utilitybl	
界面类库包	
Java RMI	
databaseutility	JDBC

细节考虑

- 5) 在分层风格的典型设计中,不希望高层直接依赖于低层,而是为低层建立接口包, 实现依赖倒置原则(参见 15.2.3), 所以应该调整为:各 Prensentatin 层开发包(调 用)依赖于 Logic 层接口包 businesslogicaservice 包,Logic 层开发包(实现)依赖 于 Logic 层接口包 businesslogicaservice 包。

- (6) 在分层风格的典型设计中,Presentation 层与 Logic 层之间、Logic 层与 Data 层之 间可能会传递复杂数据对象,那么相邻两层都需要使用数据对象声明,所以需要将 数据对象声明单独独立为开发包(VO 包与 PO 包),或者将数据对象声明放入接口 包(VO 包放入 Logic 接口包,PO 包独立)。

细节考虑

表 10-4 连锁超市管理系统的最终开发包设计

开发（物理）包	依赖的其他开发包
mainui	userui, salesui, memberui, commodityui, promotionui, vo
salesui	salesblservice,界面类库包, vo
salesblservice	
salesbl	salesblservice, salesdataservice, po, promotionbl, userbl
salesdataservice	Java RMI, po
salesdata	databaseutility, po, salesdataservice
commodityui	commodityblservice，界面类库包
commodityblservice	
commoditybl	commodityblservice, commoditydataservice, po , salesbl
commoditydataservice	Java RMI, po
commoditydata	Java RMI, po, databaseutility
memberui	memberblservice，界面类库包
memberblservice	
memberbl	memberblservice, memberdataservice, po, salesbl, commoditybl
memberdataservice	Java RMI, po
memberdata	Java RMI, po, databaseutility
promotionui	promotionblservice，界面类库包
promotionblservice	
promotionbl	promotionblservice, promotiondataservice, vo
promotiondataservice	Java RMI, po
promotiondata	Java RMI, po, databaseutility
userui	userblservice，界面类库包
userblservice	
userbl	UserInterface, UserDataClient, UserPO
userdataservice	Java RMI, po
userdata	RMI, po, databaseutility
vo	
po	
utilitybl	
界面类库包	
Java RMI	
databaseutility	JDBC

(7) 开发包的循环依赖现象需要消除,对此可以使用依赖倒置原则(参见 15.2.3)将 循环依赖变为单向依赖:

a) Sales与Commodity:将部分Commodity类抽象接口commodityInfoService置入Sales 包,这样 Commodity 单向依赖于 Sales(实现接口+调用)。

b) Sales 与 Member:将部分 Member 类抽象接口置入 Sales 包,这样 Member 单向依赖于 Sales(实现接口+调用)。

(8) 在 Logic 层中,一些关于初始化和业务逻辑层上下文的工作被分配到 utilitybl 包中去。

连锁超市管理系统开发包图
(客户端)

连锁超市管理系统开发包图
(服务器端)

进程图

pkg
ClientNode[4..6]
{OS = Window 7}
{JDK = JDK 6.0}
Client
RMIStub
<<artifact>>
client.jar
<<artifact>>
server.jar
Java RMI / LAN
<<artifact>>
IDataRemoteService_Client_Stub.class
ServerNode
{OS = Window 7}
{JDK = JDK 6.0}
Server
powered by Astah

部署图

体系结构设计过程

1. 分析关键需求和项目约束；
2. 通过选择体系结构风格；
3. 进行软件体系结构逻辑（抽象）设计；
4. 依赖逻辑设计进行软件体系结构（实现）设计；
5. 完善体系结构设计；
    1. 完善软件体系结构设计
    2. 细化软件体系结构设计
6. 添加构件接口；
7. 迭代过程3-7

图 10-10 客户端模块视图

图 10-11 服务器端模块视图

表 10-5 客户端各层的职责

层	职责
启动模块	负责初始化网络通信机制，启动用户界面。
用户界面层	基于窗口的连锁超市客户端用户界面。
业务逻辑层	对于用户界面的输入响应和业务处理逻辑。
客户端网络模块	利用 Java RMI 机制查找 RMI 服务

表 10-6 服务器端各层的职责

层	职责
启动模块	负责初始化网络通信机制，启动用户界面。
用户界面层	基于窗口的连锁超市服务器用户界面。
业务逻辑层	对于用户界面的输入响应和业务处理逻辑。
数据层	负责数据的持久化及数据访问接口。

常见完善的方面：
启动、现场初始化、监控、环境建立与维护、清理现场...

完善启动和网络链接

细化salesbl模块

数据定义

- 接口的数据对象
- 关键类的重要数据结构

- Value Object (VO)
- Persistent Object(PO)
  - 用POJO实现

Java Entities

- An entity is an object that represents a persistent business entity such as an account or a customer.
- No logic / business methods
- Entities must persist between the sessions or transactions that use them.
- Entities are stored in files or databases
- Entities are beans
  - Simple or EJB.

POJO（Plain Old Java Objects）= Simple Java Entities
Example: A Person Entity

```
public class Person extends Entity {
   private String first;
   private String last;
   private Address address;
   private Phone phone;
   public Phone getPhone() { return phone; }
   public void setPhone(Phone p) { phone = p; }
   // etc.
}
```

Value Objects

- A value object holds the attributes of one or more entities in public fields.
- Pass value objects, not entities, between layers.
- Implementation of Serializable should be considered
- Value objects can update and create entities.
- Entities can create value objects.

Example: Person VO

```
public class PersonVO extends ValueObject {
   public String first;
   public String last;
   public int addressOid;
   public String street;
   public String apartment;
   public String city;
   public String state ;
   public String zip;
   public int phoneOid;
   public String phone;
   public void update(Person per) { ... }
   public Person makePerson() { ... }
}
```

Address Book Entities

项目实践

- VO :View层与Logic之间的数据传递
  - Customer(VIP)VO、CommodityVO、SaleLineItemVO、PaymentVO、GiftVO…
- PO：Logic与Data层之间的数据传递
  - SalesPO、SaleLineItemPO、CustomerPO、CommodityPO、PaymentPO、GiftPO…

持久化对象 UserPO的细化

- public class UserPO implements Serializable { int id;
  - String name; String password; UserRole role;
  - public UserPO(int i, String n, String p, UserRole r){ id = i;
    - name = n; password = p; role = r;
  - }
  - public String getName(){
    - return name;
  - }
  - public int getID(){
    - return id;
  - }
  - public String getPassword(){
    - return password; }
  - public UserRole getRole(){
    - return role;
  - }
- }

数据类型	数据定义
序列化持久化	序列化数据保持在.ser 文件中。 UserPO 类包含用户的用户名，密码几个属性。 CommodityPO 类是包含商品的编号、价格、数量和名字几个属性。 MemberPO 类包含会员的编号、姓名、生日、性别、电话、积分几个属性。 SalesPO 类是保存销售时的数据的类，包含编号、会员编号、商品列表、总价、折扣、客户支付金额、找零金额等属性。 SalesLineItemPO 是保持销售记录中一行的信息类：商品编号、数量、小计。 CommodityGiftPromotionPO 包含商品 ID，促销起始日，促销结束日，礼品编号，礼品数量。 … …
Txt 持久化	序列化数据保持在.txt 文件中。 以商品数据为例每行分别对应 货号：商品名称：价格：数量。 中间用：隔开。 123:杯子:10:32 456:桌子:20:22
数据库持久化	包含 User 表、Commodity 表、Member 表、Sales 表、SalesLineItem 表、CommodityGiftPromotion 表、CommdoityPricePromotion 表、GiftLineItem 表。

数据持久化的细化

体系结构设计过程

1. 分析关键需求和项目约束；
2. 通过选择体系结构风格；
3. 进行软件体系结构逻辑（抽象）设计；
4. 依赖逻辑设计进行软件体系结构（实现）设计；
5. 完善体系结构设计；
6. 添加构件接口；
7. 迭代过程3-7

构件初步设计

- 根据分配的需求确定模块对外接口
- 初步设计关键类
- 编写接口规范

makenewsale()
VO V2P=findV2P(ID)
VO item=
enterItem(ID, num)
int total()
int change=
makepayment(VO payment)
List<VO> gift=getGift()
Sales
POJO V2P findV2P(ID)
V2P.update(POJO V2P)
POJO item=
findItem(ID)
Product.update
(POJO item)
POJO alg=
getStrategy(ID)
total
Sale.
update(POJO Sale)
Payment.
update(POJO
payment)

确定模块对外接口

提供的服务（供接口）		
LoginController.login	语法	public ResultMessage login(long id, String password);
	前置条件	用户输入有效的用户编号和密码。user 成员变量初始化为 null。
	后置条件	根据 id 查找是否存在相应的用户，根据用户信息验证其密码正确性，如果通过验证记录当前用户。
需要的服务（需接口）		
服务名	服务	
UserDataService.find	根据 ID 进行查找用户信息	

编写接口规范

体系结构构建

体系结构构建

- 包创建
- 重要文件的创建
- 定义构件之间的接口
- 关键需求的实现

Client (~/Documents/workspace/ConveniStoreSystem
src
businesslogic
commoditybl
memberbl
promotionbl
salesbl
userbl
utilitybl
businesslogicservice
dataservice
commoditydataservice
datafactoryservice
memberdataservice
promotiondataservice
salesdataservice
userdataservice
po
presentation
commodityui
mainui
memberui
promotionui
salesui
userui
vo
Client.iml

Server (~/Documents/workspace/ConveniStoreSystem
src
data
commoditydata
databaseutility
datafactory
memberdata
promotiondata
salesdata
userdata
dataservice
commoditydataservice
datafactoryservice
initdataservice
memberdataservice
promotiondataservice
salesdataservice
userdataservice
po
commodity.ser
commodity.txt
commodityGift.txt
member.ser
member.txt
sales.txt
Server.iml
user.ser

包的创建

重要文件的创建

- 可选
  - 数据文件
  - 系统文件
  - 配置文件
  - …

Server (~/Documents/workspace/ConveniStoreSystem
src
data
commoditydata
CommodityDataServiceMySqlImpl
databaseutility
datafactory
DataFactoryMySqlImpl
DataFactorySerializableImpl
DataFactoryTxtFileImpl
memberdata
promotiondata
salesdata
userdata
dataservice
commoditydataservice
CommodityDataService
datafactoryservice
initdataservice
memberdataservice
promotiondataservice
salesdataservice
userdataservice
po
commodity.ser
commodity.txt
commodityGift.txt
member.ser
member.txt
sales.txt
Server.iml
user.ser
user.txt
build-Client.xml
build-Server.xml

文件列表

SalesBLService的定义

- public interface SalesBLService {
  //销售界面得到商品和商品促销的信息
  public CommodityVO getCommodityByID(int id);
  public ArrayList<CommodityPromotionVO> getCommodityPromotionListByID(intcommodityID);

- //销售界面得到会员的信息
  public MemberVO getMember ();

- //销售的步骤
  public ResultMessage addMember(int id);
  public ResultMessage addCommodity(int id, int quantity);
  public double getTotal(int mode);
  public double getChange(double payment);
  public void endSales();

- }

关键需求的实现

- 实现一些关键功能需求。
  - 比如连锁超市系统中销售用例就是 一个最关键的用例。它所覆盖的面最广,如果销售用例可以实现的话,其它用例的实现就比 较有信心了。而且这个需求需要做到端到端的实现。
  - 比如,连锁超市系统中存在客户端和服务器端。那么我们就需要在客户端通过 GUI 界面 发出指令,通过客户端的业务逻辑处理,访问服务器端的数据,对数据进行修改。只有这样 才能表明当前的体系结构可以胜任功能性需求。
- 对原型的非功能性指标进行估算和验证。如果出 现不符合非功能性需求和项目约束的情况,我们还需要重新对体系结构设计进行调整。
  - 比如,连锁超市系统中我们用文件来保存数据,而当数据越来越多,特别是销售记录要 保存 1 年。这时候,客户端跨网络对服务器上文件的查找就比较慢,可能会使得我们不满足 实时性的要求。这时候,可能我们就要调整体系结构中数据存储的方案,比如换成数据库。 或者在客户端的逻辑层和数据层之间,加设数据映射层。在客户端运行初始化时,将数据从 服务器端载入到客户端的内存中去。

体系结构集成与测试

集成的策略

- 当体系结构中原型各个模块的代码都编写完成并经过单元测试之后,需要将所有模块组 合起来形成整个软件原型系统,这就是集成。集成的目的是为了逐步让各个模块合成为一个 系统来工作,从而验证整个系统的功能、性能、可靠性等需求。对于被集成起来的系统一般 主要是通过其暴露出来的接口,伪装一定的参数和输入,进行黑盒测试。
- 根据从模块之间集成的先后顺序,一般有下列几种常见的集成策略:
  - 大爆炸式
  - 增量式
    - 自顶向下式
    - 自底向上式
    - 三明治式
    - 持续集成

自顶向下集成的优点:

- 按深度优先可以首先实现和验证一个完整的功能需求;
- 只需最顶端一个驱动(Driver);
- 利于故障定位。

自顶向下集成的缺点:

- 桩的开发量大;
- 底层验证被推迟,且底层组件测试不充分。

自顶向下

自底向上集成的优点:
是对底层组件行为较早验证;
底层组件开发可以并行;
桩的工作量少;
利于故障定位。
自底向上集成的缺点:
驱动的开发工作量大;
对高层的验证被推迟,设计上的高层错误不能被及时发现。

自底向上

持续集成

- 一种增量集成方法,但它提倡尽早集成和频繁集成。
- 尽早集成是指不需要总是等待一个模块开发完成才把它集成起来,而是在开发之初就利 用 Stub 集成起来。
- 频繁集成是指开发者每次完成一些开发任务之后,就可以用开发结果替换 Stub 中的相 应组件,进行集成与测试。一般来说,每人每天至少集成一次,也可以多次。
- 结合尽早集成和频繁集成的办法,持续集成可以做到:
  - 防止软件开发中出现无法集成与发布的情况。因为软件项目在任何时刻都是可以集成和发布的。
  - 有利于检查和发现集成缺陷。因为最早的版本主要集成了简单的 Stub,比较容易做到没有错误。后续代码逐渐开发完成后,频繁集成又使得即使出现集成问题也能 够尽快发现、尽快解决。
- 持续集成的频率很高,所以手动的集成对软件工程师来说是无法接受的,必须利用版本 控制工具和持续集成工具。如果程序员可以仅仅使用一条命令就完成一次完整的集成,开发 团队才有动力并且能够坚持进行频繁的集成。

依据模块接口建立桩程序Stub

- Stub
  - 为了完成程序的编译和连接而使用的暂时代码
  - 对外模拟和代替承担模块接口的关键类
  - 比真实程序简单的多，使用最为简单的逻辑

<<interface>>
SalesBLService
SalesController
Sales
- member : Member
- salesLineItemMap : HashMap<int,SalesLineItem>
- total : double
- salesPO : SalesPO
+ addMember(memID : int) : void
+ addSalesLineItem(commodityID : int, quantity : int) : void
+ addCommodityQuantity(commodityID : int, quantity : int) : void
+ getTotal() : double
+ getChange() : double
+ endSales() : void

Stub定义

```
public interface SalesBLService _Stub implements SalesBLService {
    String commodityName;
    double commodityPrice;
    double commodityDiscount;
    CommodityPromotion promotion;
    String memberName;
    Int memberPoint;
    double total;
    double change;

    public SalesBLService_Stub(String cn, double cp, double cd, CommodityPromotion p,String mn, int mp, double t, double c){
        commodityName = cn;
        commodityPrice = cp;
        commodityDiscount = cd;
        promotion = p;
        memberName = mn;
        memberPoint = mp;
        total = t;
        change = c;
    }
    //销售界面得到商品和商品促销的信息
    public CommodityVO getCommodityByID(int id){
        return new CommodityVO(name, price, commodityDiscount );
    }
    public ArrayList<CommodityPromotionVO> getCommodityPromotionListByID(int commodityID){
        ArrayList<CommodityPromotionVO> commodityPromotionList = new ArrayList<CommodityPromotionVO>();
        commodityPromotionList.add(new CommodityPromotionVO(promotion));
        return commodityPromotionList;
    }
```

Stub使用

```
public class SalesView{
    ……
    SalesBLService salesBl = new SalesBLService_Stub ("Cup",10.0,2.0, new CommodityPromotion(), "Karen", 500, 57.0, 3.0);
    ……
}
```

Driver定义

```
public class SalesBLService _Driver{
    public void drive(SalesBLService salesBLService){
        ResultMessage    result = salesBLService. addMember(0911024);
        If(result== ResultMessage.Exist) System.out.println("Member exists\n");
        ……
        salesBLService.endSales();
    }
}
```

Driver使用

```
public class Client{
    public static void main(String[] args){
      SalesBLService salesController = new SalesController();
      SalesBLService _Driver driver = new SalesBLService _Driver (salesController);
      driver.drive(salesController);
    }
}
```

集成与构建

- 集成
  - 使用桩程序辅助集成
  - 编译与链接
- 持续集成
  - 逐步编写各个模块内部程序，替换相应的桩程序
  - 真实程序不仅实现业务逻辑，而且会使用其他模块的接口程序（真实程序或者桩程序）
  - 每次替换之后都进行集成与测试

编写驱动程序，在桩程序帮助下进行集成测试

项目实践

- 客户端
  - View层：HCI特殊测试
  - Logical层：
    - 驱动模拟来自View层的请求
    - 桩程序模拟远程Data层
- 服务器端（Data层）
  - 驱动模拟来自客户端的请求
  - 桩程序模拟未完成的Data模块接口

体系结构文档化

体系结构设计文档模版

IEEE1471-2000

1 引言

1.1 编制目的

表明文档的读者，以及文档主题。

1.2 词汇表

文档中用到的缩写、专业词汇等。

1.3 参考资料

相关参考文献。

2 产品概述

产品需求的概述，以及设计的需求分析文档的引用。

3 体系结构模型

3.1 整体架构描述

系统整体的模块的分解：主要是组成视角和逻辑视角。

3.2 xx 模块的分解

模块内部的分解：主要是结构视角和接口视角。一般会用类图、构件图等。

3.2.1 xx 模块的职责

每个类相应的职责说明。

3.2.2 xx 模块的接口规范

模块供接口和需接口的说明。

3.2.3 启动模块的设计原理

设计理由的说明。

……（各个模块的设计描述）

3.7 界面模块的设计

界面跳转的整体设计，界面的风格的描述。

3.8 运行时组件

进程图。

4 模型之间的映射

4.1 调用关系映射

模块之间相互调用的关系。

4.2 数据模型

信息视角。一般会定义相应的数据格式。

5 系统体系结构设计思路

总体的设计的大思路和设计中一些特别的细节。

体系结构评审

评审的角度

- 设计方案正确性、先进性、可行性;
- 系统组成、系统要求及接口协调的合理性;
- 对于功能模块的输入参数、输出参数的定义是否明确;
- 系统性能、可靠性、安全性要求是否合理;
- 文档的描述是否清晰、明确。

体系结构评审的方法

- 对结果的评审
  - Checklist
- 对设计决策的评审

Checklist

- 1. 体系结构设计是否为后续开发提供了一个足够的视角?
- 2. 体是否所有的功能都被分配给了具体模块?
- 3. 是否所有的非功能属性都得到了满足?
- 4. 是否所有的项目约束都得到了满足?
- 5. 系结构设计是否为后继设计提供了简洁的概述、背景信息、限制条件和清晰的 组织结构?
- 6. 体系结构设计是否能应对可能发生的变更?
- 7. 体系结构设计是否关注点在详细设计和用户接口的层次之上?
- 8. 不同的体系结构设计视角的依赖是否一致?
- 9. 系统环境是否定义,包括硬件、软件、和外部系统?