南京大学
NANJING UNIVERSITY

《软件工程与计算II》

Ch12 详细设计

主要内容

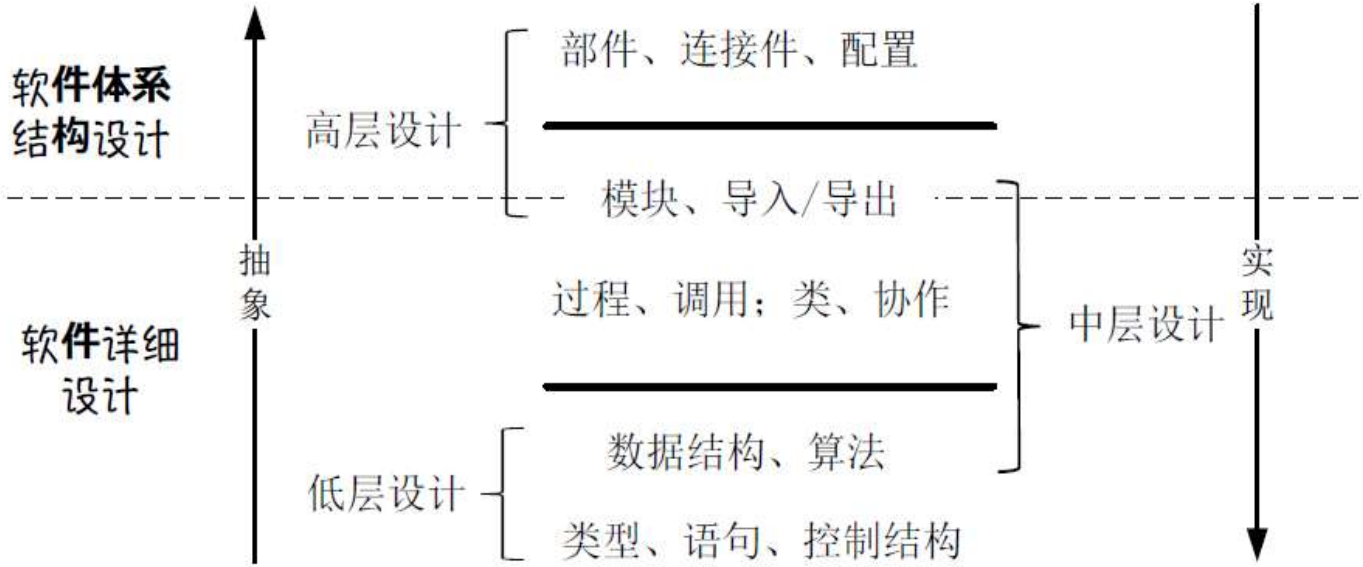
- 详细设计基础
- 面向对象详细设计
- 为类间协作开发集成测试用例
- 详细设计文档描述和评审

主要内容

- 详细设计基础
  - 什么是详细设计
  - 详细设计的出发点
  - 详细设计的上下文
- 面向对象详细设计
- 为类间协作开发集成测试用例
- 结构化详细设计
- 详细设计文档描述和评审

What is Detail Design?

- Mid-level design to a specific module
- Low-level design to objects/classes of the specific module



What is Detail Design?

- Software Architecture defined the specification of a module
- Detail Design implements the module with detail design mechanism
  - Mid-Level: (sub-moduled) ->OO -> Specification of classes
  - Low-Level: DS. + ALG. ->Implementation of the class
- Detail Design requires a designer to consider the aesthetic, functional, and many other aspects of the module
  - Quality attributes in Detail Design
    - Modification , Maintenance, performance …

详细设计的输入

1. 需求规格说明

功能性需求：

要在两座山之间建一座桥

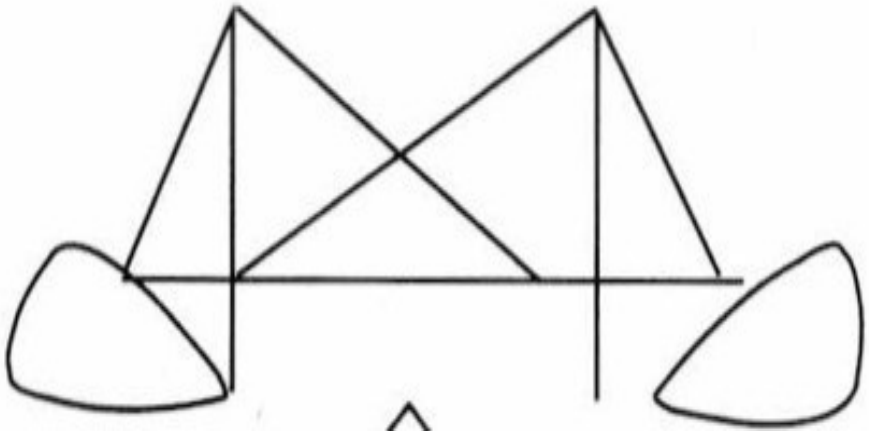
非功能性需求：

使得汽车可以以 100 千米每小时的速度在 5 分钟之内从一座山到达另一座山

2. 体系结构设计

风格：斜拉桥风格（桥墩、桥身和悬索）

原型：斜拉桥模型

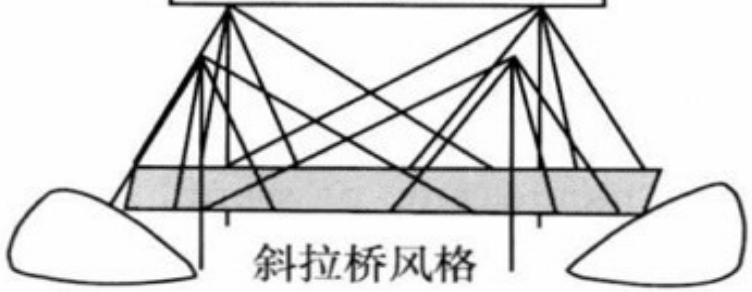


从需求、体系结构设计到详细设计

1. 需求规格说明
功能性需求：
要在两座山之间建一座桥
非功能性需求：
使得汽车可以以 100 千米每小时的速度在 5 分钟之内从一座山到达另一座山

2. 体系结构设计
风格：斜拉桥风格（桥墩、桥身和悬索）
原型：斜拉桥模型

3. 详细设计
悬索：拉索的根数
路面：路面材质



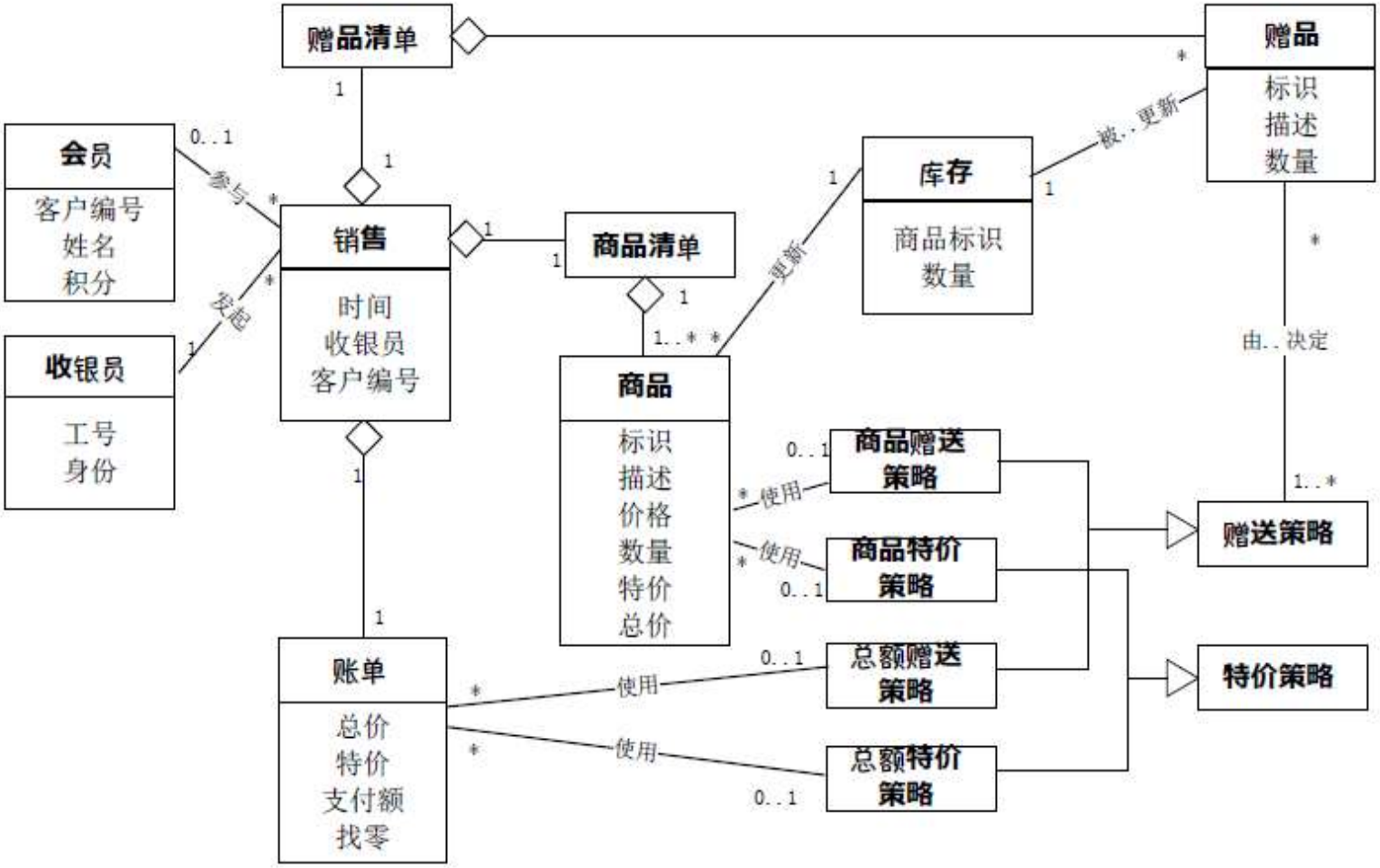
斜拉桥风格

Where from Detail Design Starts ?

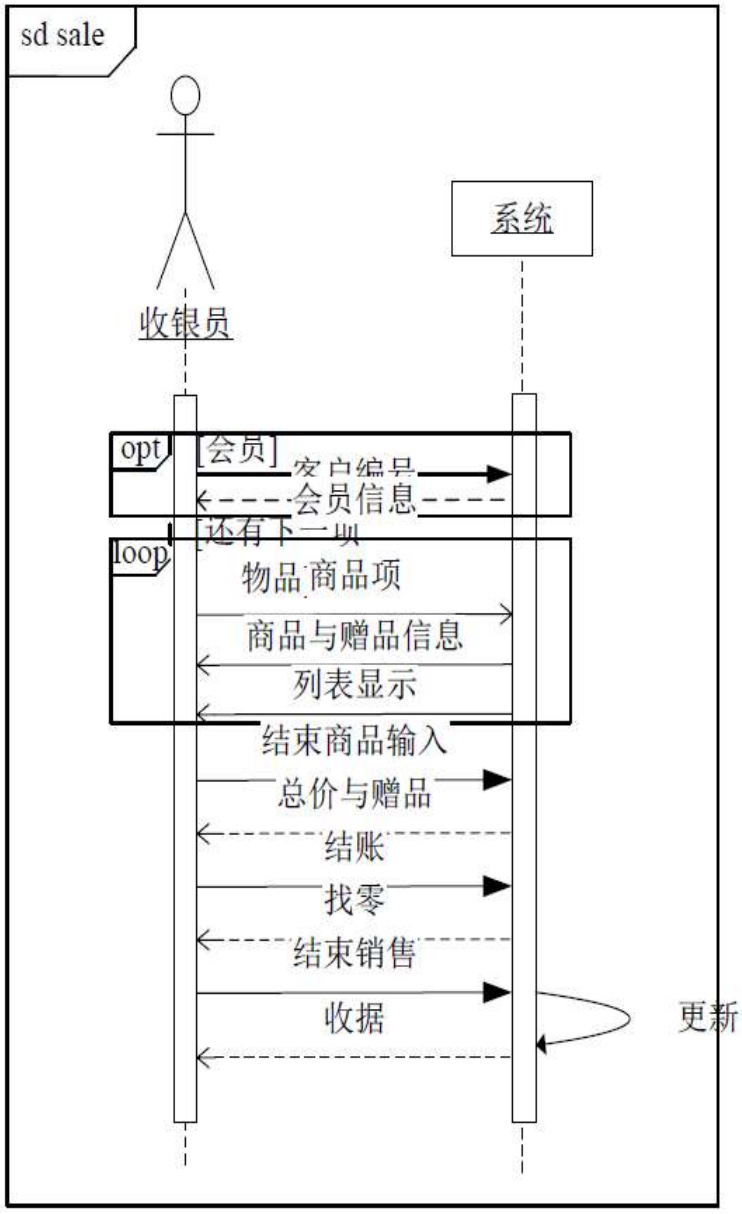
Context

- Specification of modules
  - Export/import interfaces
- Responsibility assignment :
  - Some responsibilities are from RE(SRS)
    - Typically use cases, domain models, sequence diagrams, state diagrams
  - Some others are from implementation decisions

需求：分析类图


赠品清单
赠品
标识
描述
数量
会员
客户编号
姓名
积分
参与
销售
时间
收银员
客户编号
支起
收银员
工号
身份
商品清单
库存
商品标识
数量
被..更新
更新
商品
标识
描述
价格
数量
特价
总价
由..决定
商品赠送策略
使用
商品特价策略
使用
赠送策略
账单
总价
特价
支付额
找零
总额赠送策略
使用
总额特价策略
使用
特价策略

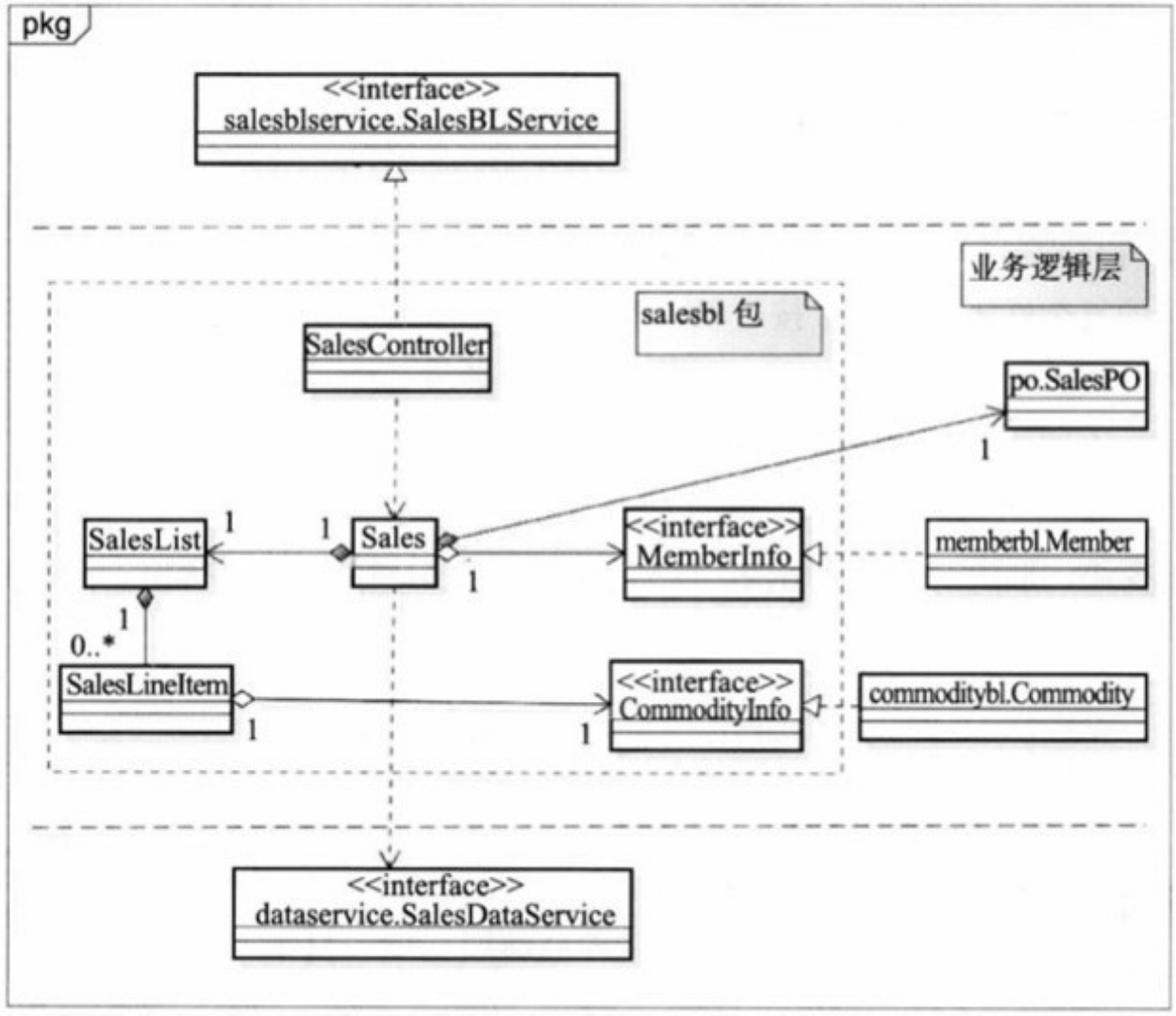
需求：系统顺序图


sd sale
系统
收银员
opt
[会员]
会员信息
[还有下一项]
loop
物品:商品项
商品与赠品信息
列表显示
结束商品输入
总价与赠品
结账
找零
结束销售
收据
更新

软件体系结构：构件之间的接口

```
//被 Presentation 层调用的接口
public interface SalesBLService {
    public CommodityVO getCommodityByID(int id);
    public MemberVO getMember();
    public  CommodityPromotionVO  getCommodityPromotionByID(int
        commodityID);
    public boolean addMember(int id);
    public boolean addCommodity(int id, int quantity);
    public int getTotal(int mode);
    public int getChange(int payment);
    public void endSales();
}
//调用 DataService 层的接口
public interface SalesDataService extends Remote {
    public void init()throws RemoteException;
    public void finish()throws RemoteException;
    public void insert(SalesPO po)throws RemoteException;
    public void delete(SalesPO po)throws RemoteException;
    public void update(SalesPO po)throws RemoteException;
    public SalesPO find(int id)throws RemoteException;
     public ArrayList<PO> finds(String field, int value)throws
RemoteException;
}
```

详细设计的输出


pkg
<<interface>>
salesblservice.SalesBLService
业务逻辑层
salesbl 包
SalesController
po.SalesPO
1
SalesList
1
1
Sales
<<interface>>
MemberInfo
memberbl.Member
1
1
0..*
SalesLineItem
1
<<interface>>
CommodityInfo
commoditybl.Commodity
1
<<interface>>
dataservice.SalesDataService

主要内容

- 面向对象详细设计
  - 面向对象设计思想
    - 职责
    - 协作
  - 面向对象详细设计的过程
    - 设计模型建立
      - 通过职责建立静态设计模型
      - 通过协作建立动态设计模型
    - 设计模式重构

Responsibilities

A responsibility is an obligation to perform a task (an operational responsibility) or to maintain some data (a data responsibility).

- Operational responsibilities are usually fulfilled by operations.
- Data responsibilities are usually fulfilled by attributes.
- Class collaborations may be involved.

Responsibility-Driven Decomposition

- Responsibilities may be stated at different levels of abstraction.
- Responsibilities can be decomposed.
- High-level responsibilities can be assigned to high-level components.
- Responsibility decomposition can be the basis for decomposing components.
- Responsibilities reflect both operational and data obligations, so responsibility-driven decomposition can be different from functional decomposition.

Responsibility Heuristics

- Assigning responsibilities well helps achieve high cohesion and low coupling.
- Make sure module responsibilities do not overlap.
- Place operations and data in a module only if they help to fullfill the module's responsibilities.

Delegation

Delegation is a tactic wherein one module (the delegator) entrusts another module (the delegate) with a responsibility.

What's Collaboration?

"The objects within a program must collaborate; otherwise, the program would consist of only one big object that does everything."

-- Rebecca Wirfs-Brock, et. al.,《Designing Object-Oriented Software》, Prentice Hall, 1990

What's Collaboration?

"Equally important [as inheritance] is the invention of societies of objects that responsibly collaborate with one another. ... These societies form what I call the mechanisms of a system, and thus represent strategic architectural decisions because they transcend individual classes."

-- [The C++ Journal,Vol. 2, NO. 1 1992, "Interview with Grady Booch"]

What's Collaboration?

- An application can be broken down into a set of many different behaviors.
- Each such behavior is implemented by a distinct collaboration between the objects of the application.
- Every collaboration, no matter how small or large, always implements a behavior of the application

What's Collaboration?

- Imagine an object-oriented application as a network of objects connected by relationships.
- Collaborations are the patterns of messages that play through that network in pursuit of a particular behavior
- The collaboration is distributed across the network of objects, and so does not exist in any one place

The needs of collaboration design

- It’s the application behaviors, after all, that we are trying to achieve.
- If the collaborations which implement them are not properly designed, then the application will be inaccurate or brittle

主要内容

- 面向对象详细设计
  - 面向对象设计思想
    - 职责
    - 协作
  - 面向对象详细设计的过程
    - 设计模型建立
      - 通过职责建立静态设计模型
      - 通过协作建立动态设计模型
    - 设计模式重构

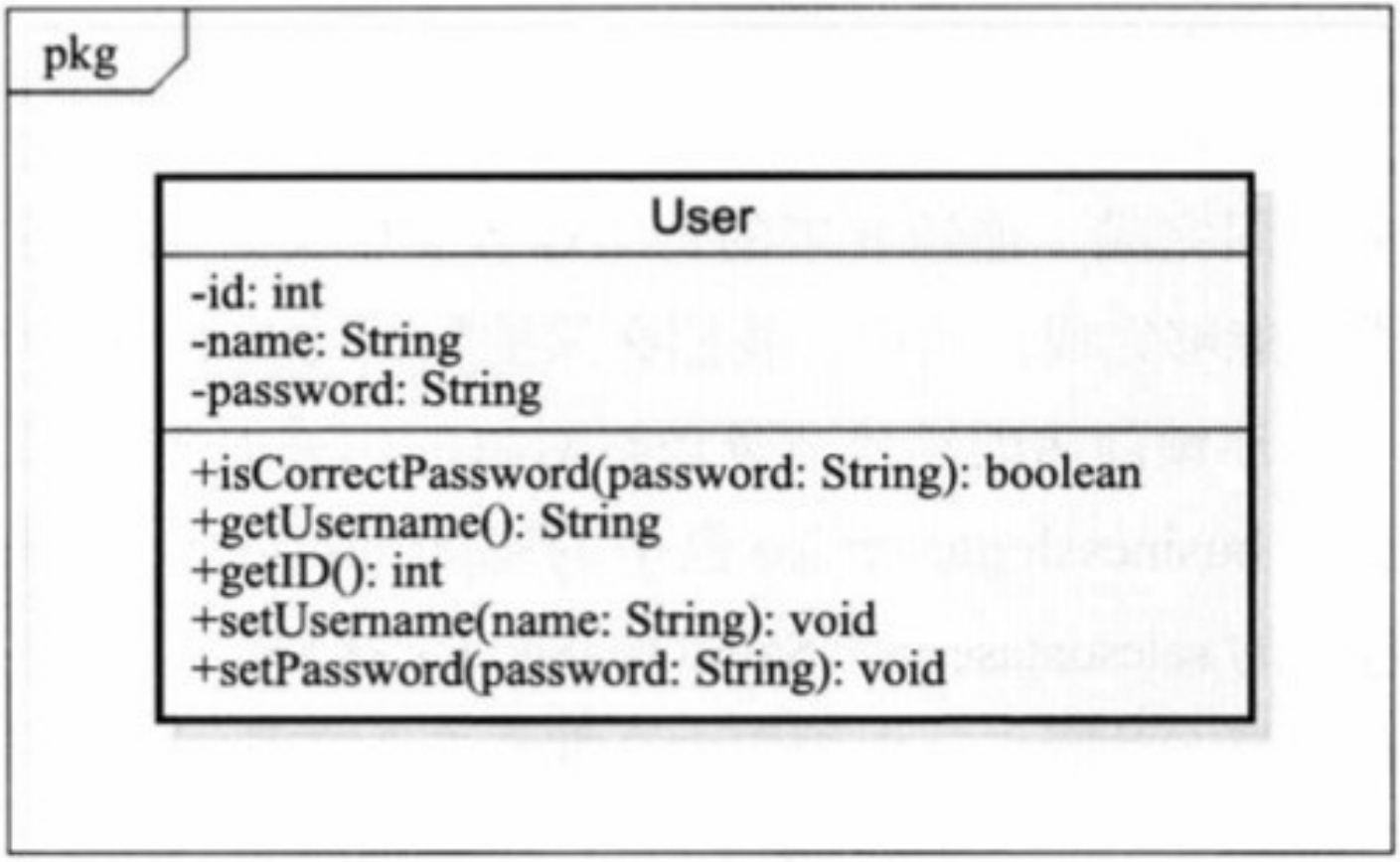
面向对象设计的过程

- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

抽象对象的职责

- 类表达了对对象族的本质特征的抽象
- 构建的蓝图
- 职责
  - 数据职责
  - 行为职责

单一类图


pkg
User
-id: int
-name: String
-password: String
+isCorrectPassword(password: String): boolean
+getUsername(): String
+getID(): int
+setUsername(name: String): void
+setPassword(password: String): void

面向对象设计的过程

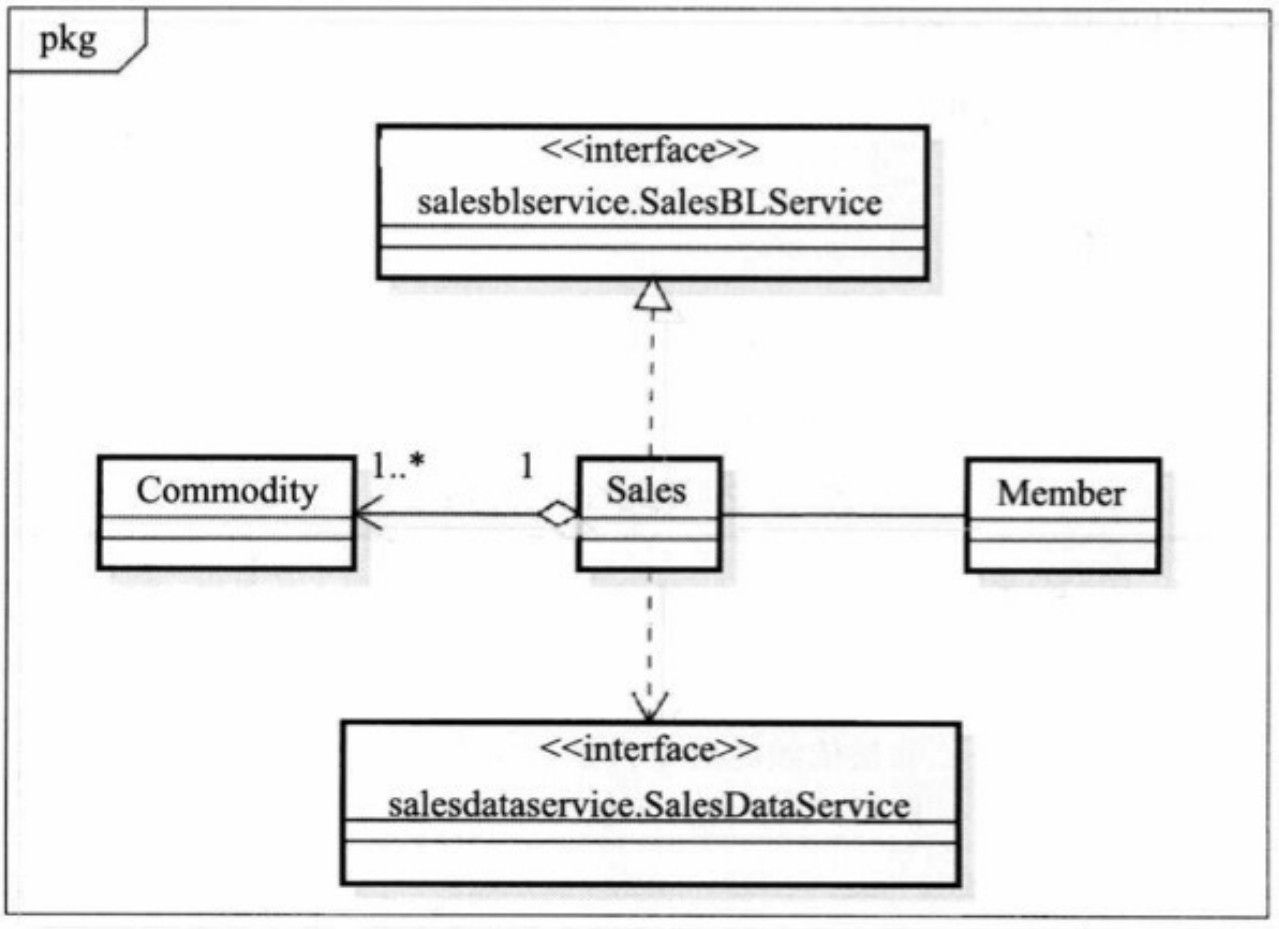
- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

类之间的关系

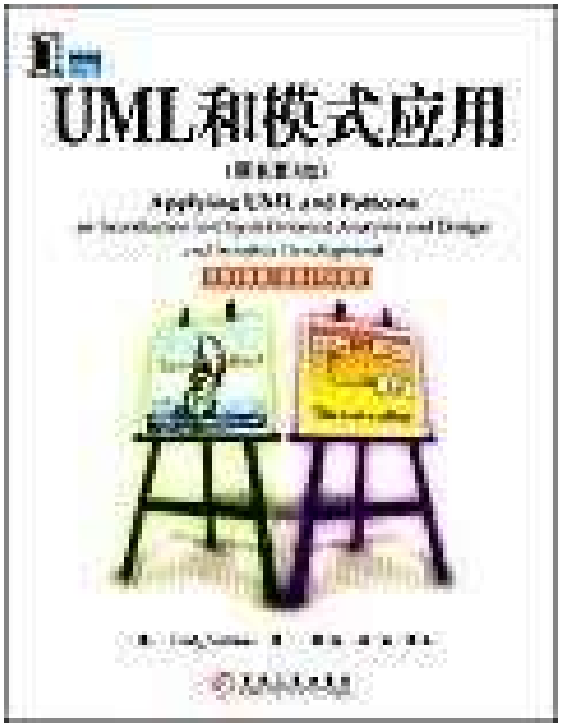
关系类型	关　系	关系短语	解　释	多重性	UML 表示法
General	依赖	A use a B	被依赖的对象只是被作为一种工具使用，其引用并不被另一个对象持有	无	

关系类型	关　系	关系短语	解　释	多重性	UML 表示法
Object Level	普通关联	A has a B	某个对象会长期持有另一个对象的引用。关联的两个对象彼此间没有任何强制性的约束	A：0..* B：0..*	
	聚合	A owns B	它暗含着一种集合所属关系。被聚合的对象还可以再被别的对象关联，所以被聚合对象是可以共享的	A：0..1 B：0..* 集合可以为空	
	组合	B is a part of A	它既要求包含对象对被包含对象的拥有，又要求包含对象与被包含对象的生命期相同。被包含对象还可以再被别的对象关联，所以被包含对象是可以共享的。然而绝不存在两个包含对象对同一个被包含对象的共享	A：0..1 B：1..1 整体存在，部分一定存在	
Class Level	继承	B is A	继承是一种非常强的关系。子类会将父类所有的接口和实现都继承回来。但是，也可以覆盖父类的实现	无	
	实现	B implements A	类实现接口，必须实现接口中的所有方法	无	

类图


pkg
<<interface>>
salesblservice.SalesBLService
Commodity
1..*
1
Sales
Member
<<interface>>
salesdataservice.SalesDataService

UML和模式应用



GRASP Patterns

- General Responsibility Assignment Software Patterns
- Not ‘design patterns’, rather fundamental principles of object design
- Focus on one of the most important aspects of object design: assigning responsibilities to classes

GRASP Patterns

- Low Coupling
- High Cohesion
- Information Expert
- Creator
- Controller

Rule of thumb

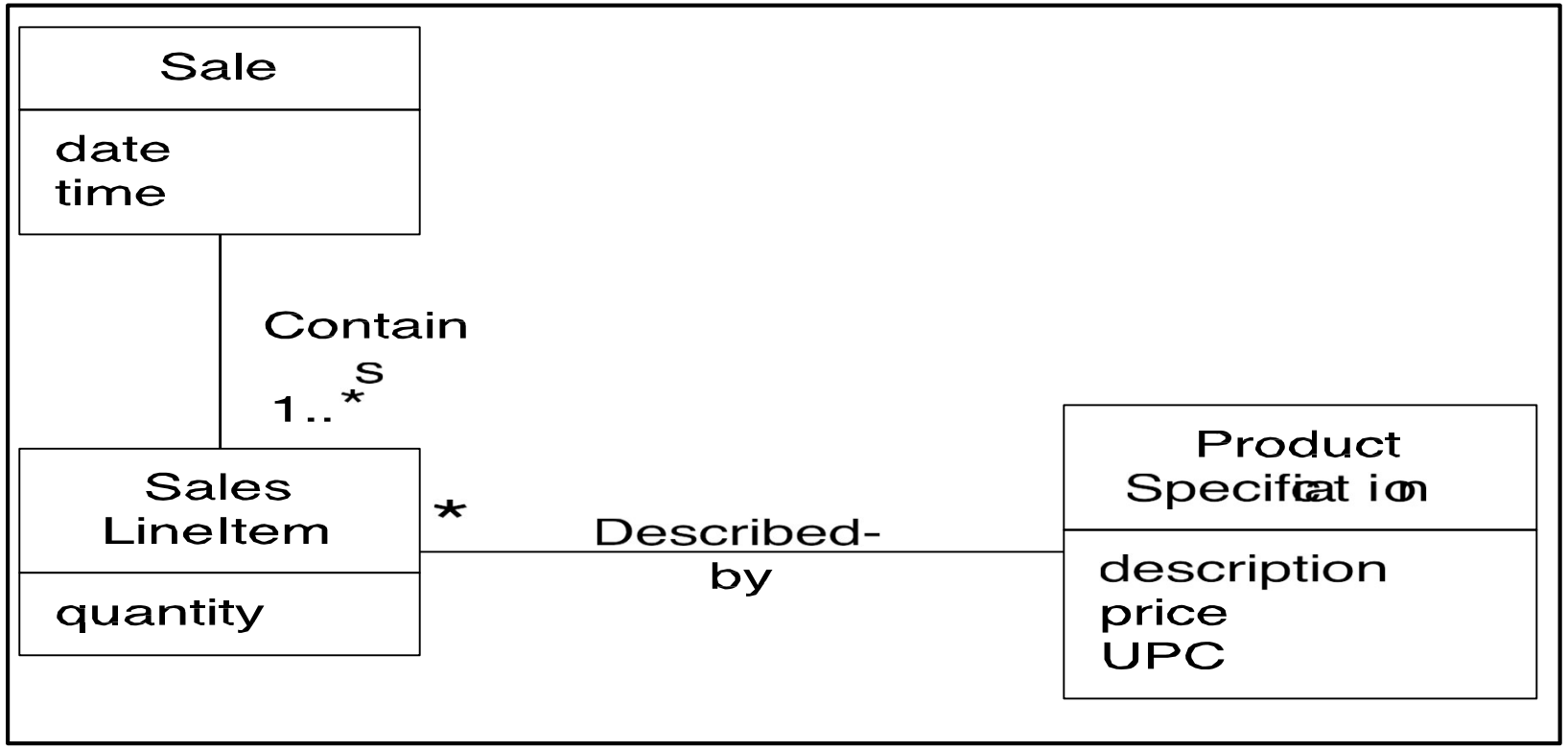
- When there are alternative design choices take a closer look at the cohesion and coupling implications of the alternatives and possibly a the future evolution pressures on the alternatives.
- Choose an alternative with good cohesion, coupling and stability.

Information Expert

- Problem:
  - What is the most basic principle by which responsibilities are assigned in object-oriented design?
- Solution:
  - Assign a responsibility to the class that has the information necessary to fulfill the responsibility.

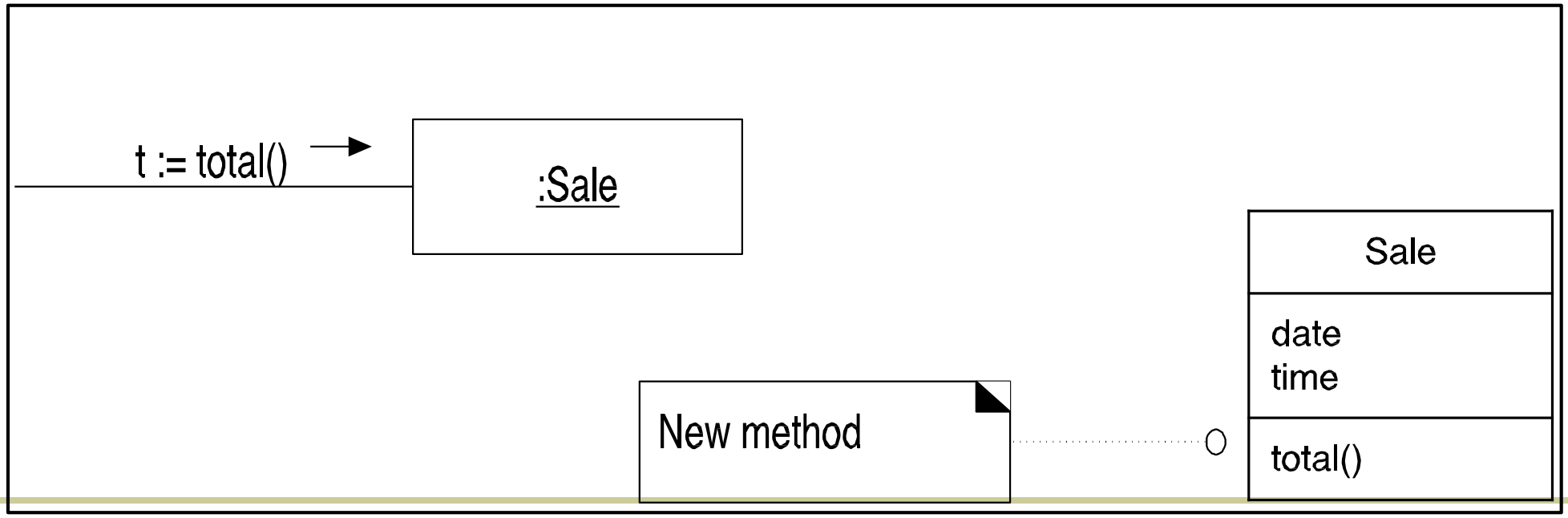
Information Expert : Example

Who is responsible for knowing the grand total of a sale in a typical Point of Sale application?



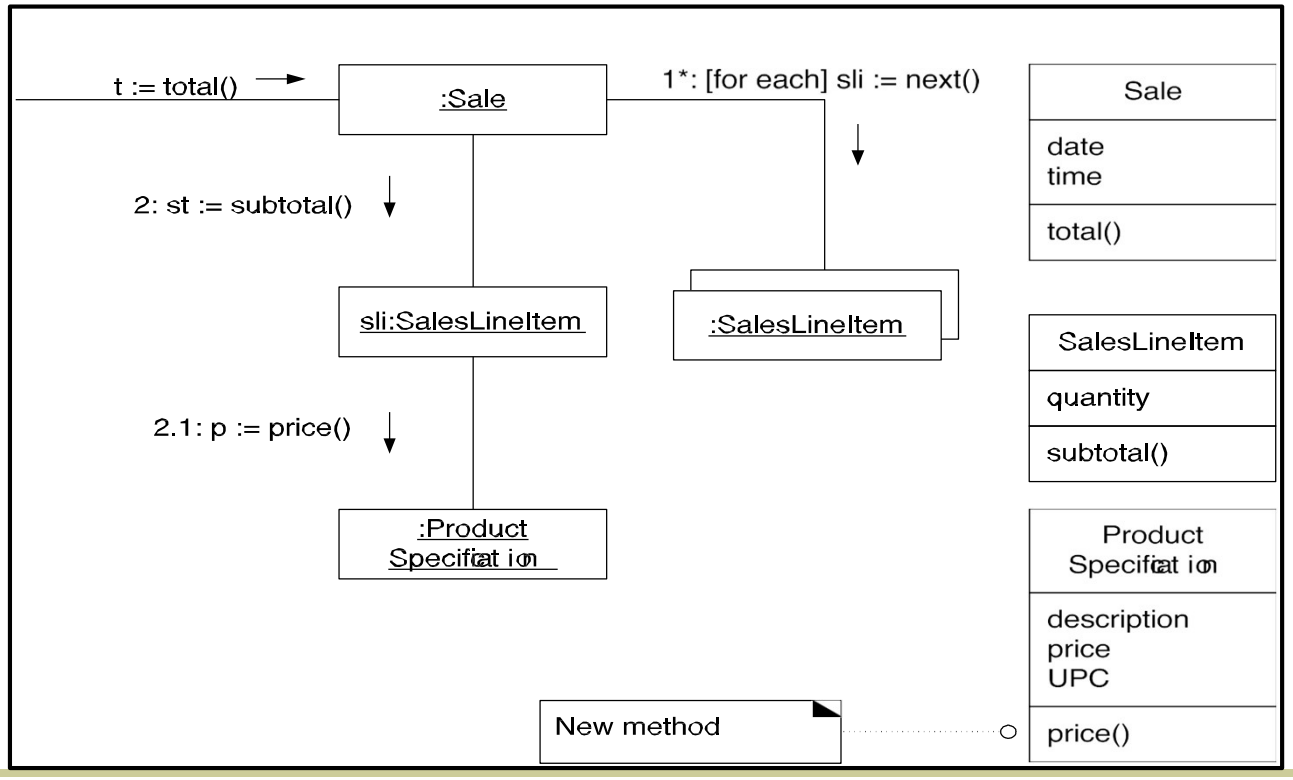
Information Expert : Example

- Need all SalesLineItem instances and their subtotals.
- Only Sale knows this, so Sale is the information expert. Hence:



Information Expert : Example

- But subtotals are needed for each line item(multiply quantity by price).
- By Expert, SalesLineItem is expert, knows quantity and has association with ProductSpecification which knows price.



Information Expert : Example

Hence responsibilities assign to the 3 classes.

Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

Information Expert

- Maintain encapsulation of information
- Promotes low coupling
- Promotes highly cohesive classes

Case Study: 智能热水器

- 智能控制水温
  - 周末水温高
  - 夜晚水温低
  - 生病等特殊情况水温高
  - 度假水温低

概念模型

- Class:
  - WaterHeaterController
    - Mode
    - lowTemp
    - highTemp
    - weekendDays
  - Clock
- Interface:
  - ThemostatDevice

详细设计

WaterHeaterController 和 Clock 怎么交互?

- 轮询
- 通知

怎么知道当前时间是改升温还是降温?

- Controller自己保存特殊时间并计算（比较当前时间和特殊时间）
  - Bad：多个职责。
- 由SpecialTime类保存特殊时间；Controller调用getSpecialTime( ) 得到特殊时间,再计算
  - Bad：数据职责与行为职责的分离
- 由SpecialTime类保存特殊时间,并提供isSpecialTime( );Controller调用方法
  - Good：单一职责

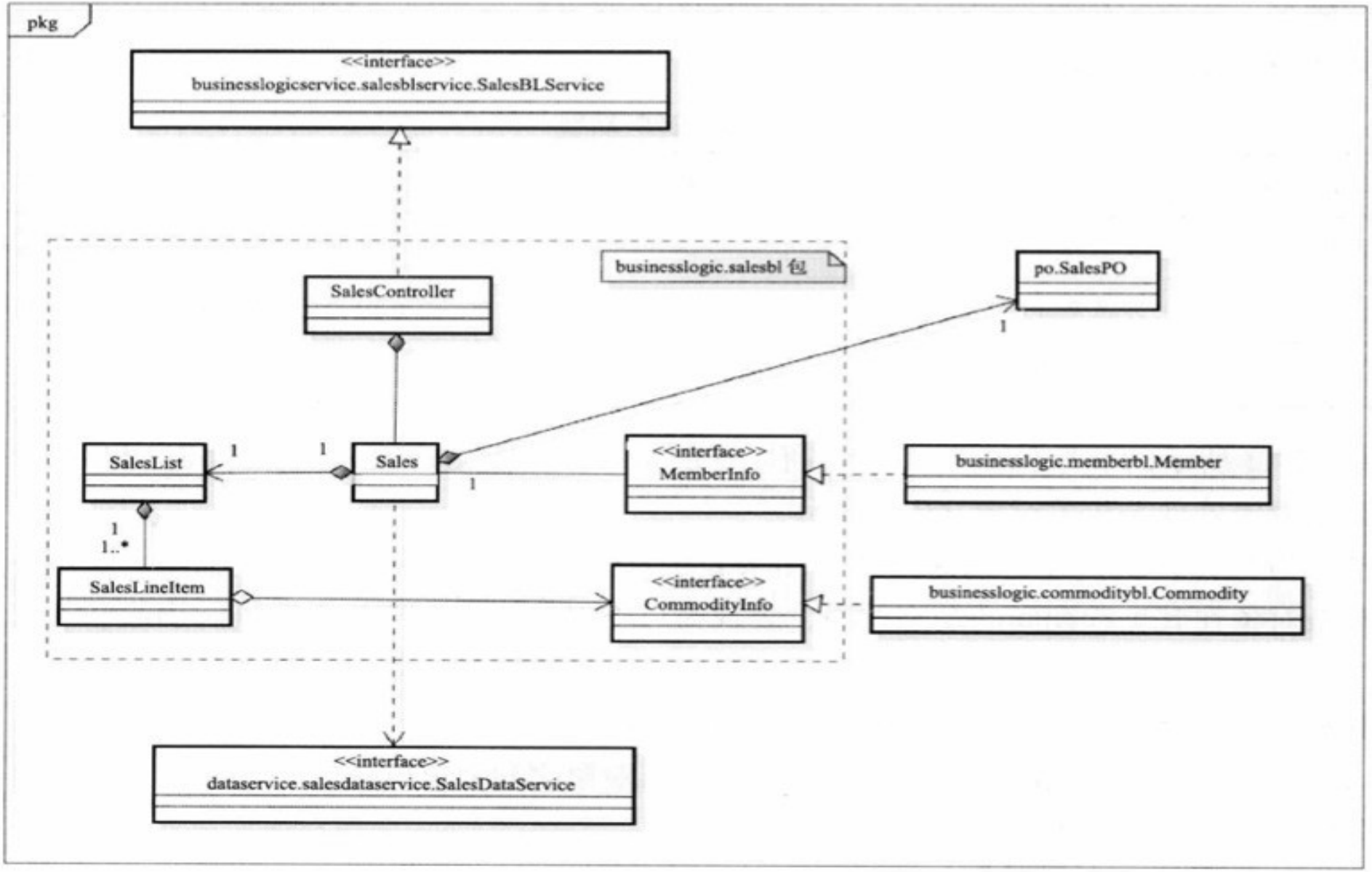
面向对象设计的过程

- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

辅助类

- 接口类
- 记录类(数据类)启动类
- 控制器类
- 实现数据类型的类容器类

添加辅助类后的设计模型



面向对象设计的过程

- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

抽象对象之间的协作

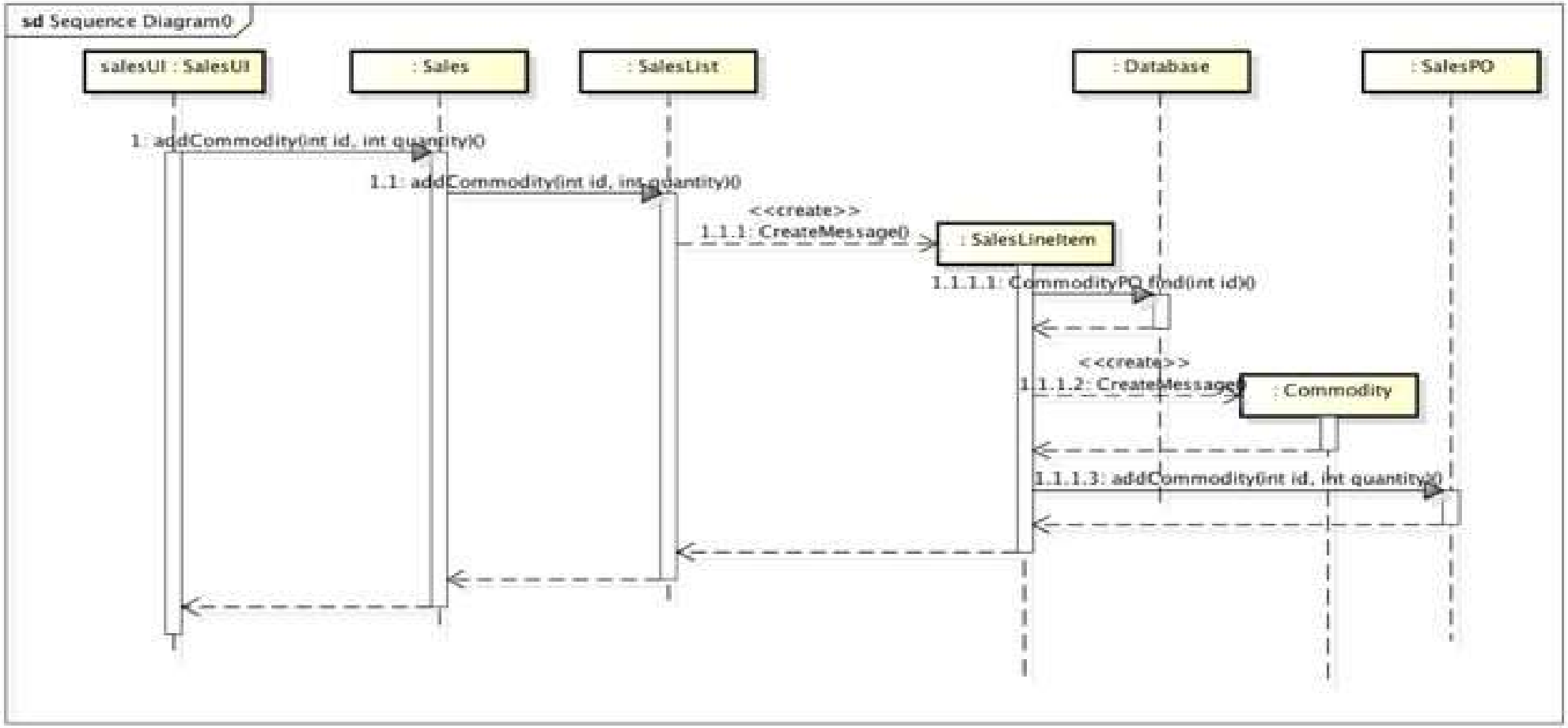
1. 从小到大,将对象的小职责聚合形成大职责;

2. 从大到小,将大职责分配给各个小对象。

这两种方法,一般是同时运用的,共同来完成对协作的抽象。

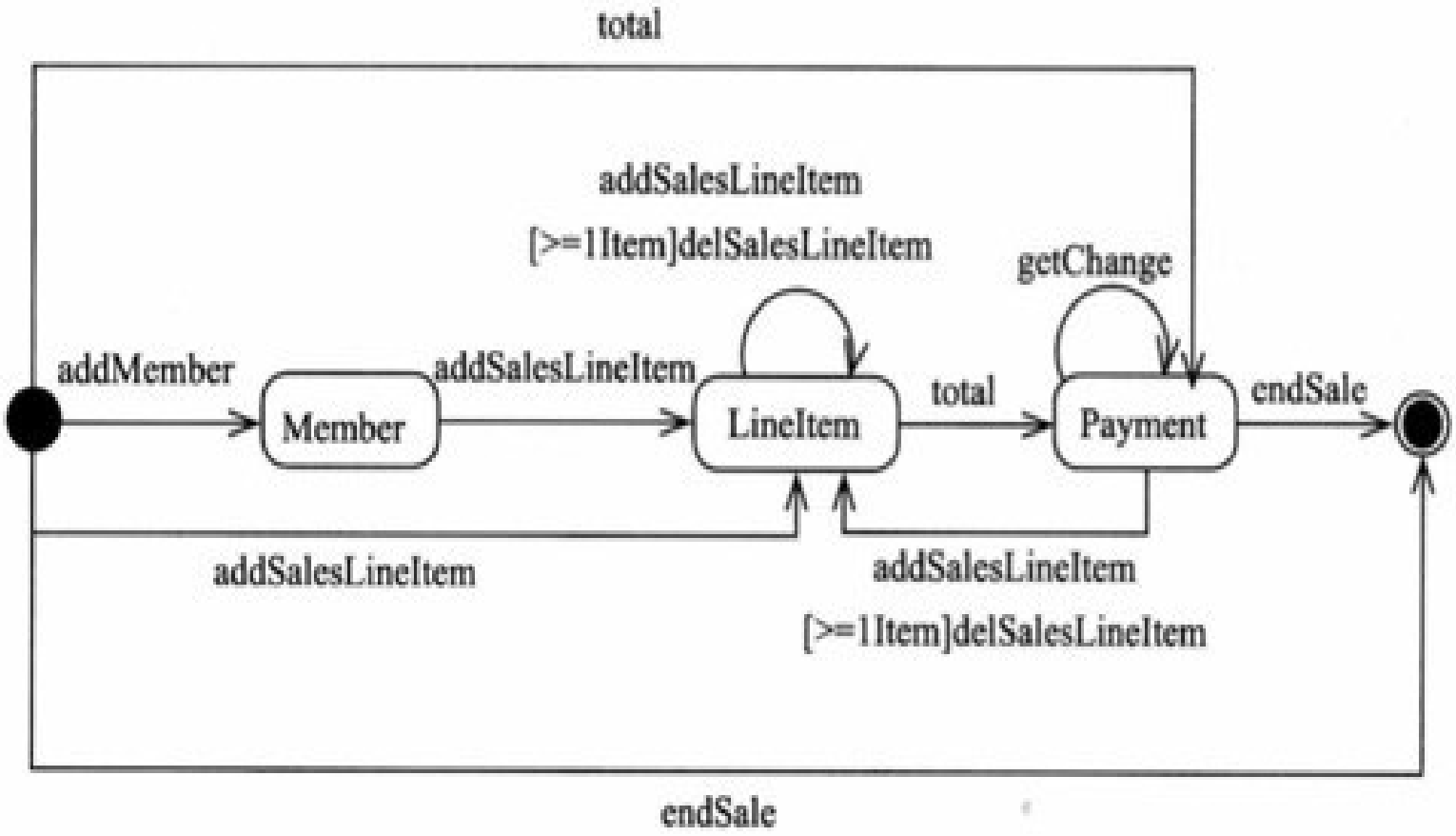
顺序图

可以用顺序图表示对象之间的协作。顺序图是交互图的一种,它表达了对象之间如何通过消息的传递来完成比较大的职责。



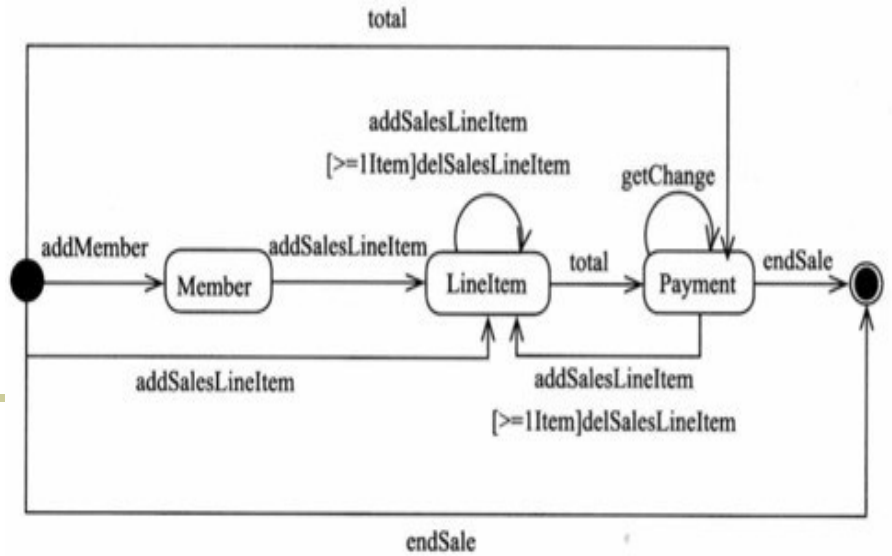
状态图

除了顺序图,我们还可以通过状态图来表达软件的动态模型。UML状态图(State Diagram)



状态图

- 主要用于描述一个复杂对象在其生存期间的动态行为
- 表现为一个对象所经历的状态序列, 引起状态转移的事件(Event),以及因状态转移而伴随的动作(Action)。
- 一般可以用状态机对一个对象的生命周期建模,UML 状态图用于显示状态机(State Machine Diagram),重点在于描述 UML 状态图的控制流。
- 而协作是:用复杂对象的状态图中的 Event 体现出对象之 间消息的传递;用 Action 体现消息引发的对象状态的改变(行为)。



面向对象设计的过程

- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

对象创建者

表 12-4　对象创建者

优　先	场　景	创建地点	创建时机	备　注
高 ↕ 低	唯一属于某个整体的密不可分的一部分（组合关系）	整体对象的属性定义和构造方法	整体对象的创建	例如，销售的业务逻辑对象由销售页面对象创建
	被某一对象记录和管理（单向被关联）	关联对象的方法	业务方法的执行中对象的生命周期起始点	例如，连接池管理对象需要负责创建连接池对象
	创建所需的数据被某个对象所持有	持有数据的对象的业务方法	业务方法的执行中	也可以考虑在此持有数据对象不了解创建时机时，由别的对象创建，由它来初始化
	被某个整体包含（聚合关系）	整体对象的业务方法（非构造方法）	业务方法的执行中	如果某个对象有多个关联，优先选择聚合关联的整体对象。如果某个对象有多个聚合关联的整体对象，则考查整体对象的高内聚和低耦合来决定由谁创建
	其他			通过高内聚和低耦合来决定由谁创建

Creator

- Problem:

  Who is responsible for creating a new instance of some class?

- Solution:

  Determine which class should create instances of a class based on the relationship between potential creator classes and the class to be instantiated.

Creator

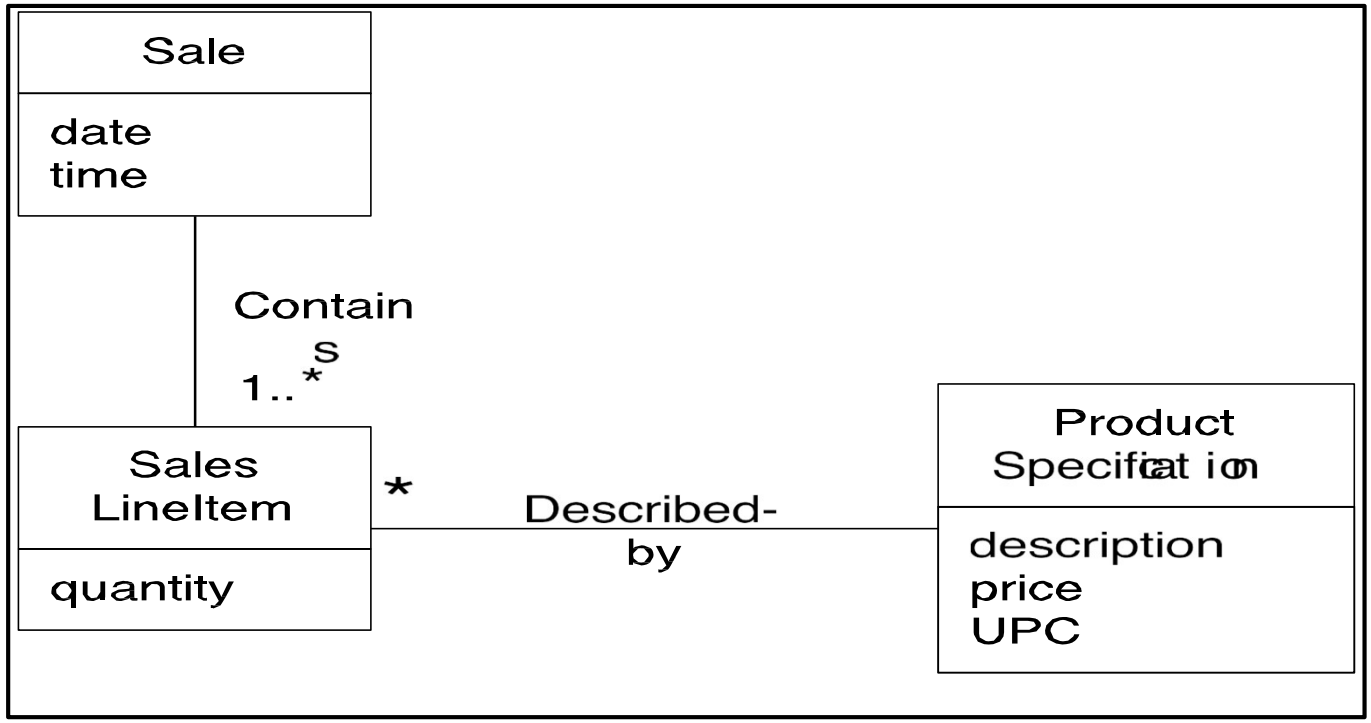
Q:Who has responsibility to create an object?

A: By creator, assign class B responsibility of creating instance of class A if

- B aggregates A objects
- B contains A objects
- B records instances of A objects
- B closely uses A objects
- B has the initializing data for creating A objects
- where there is a choice, prefer B aggregates or contains A objects

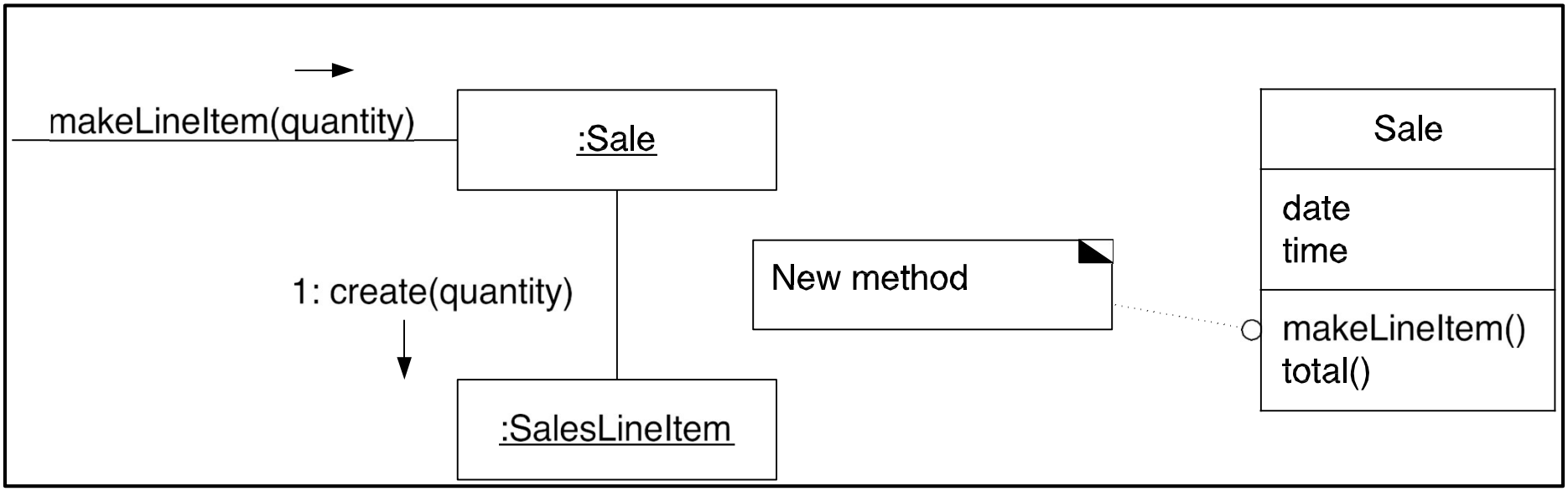
Creator : Example

- Who is responsible for creating SalesLineItem objects?
- Look for a class that aggregates or contains SalesLineItem objects.



Creator : Example

- Creator pattern suggests Sale.
- Collaboration diagram is

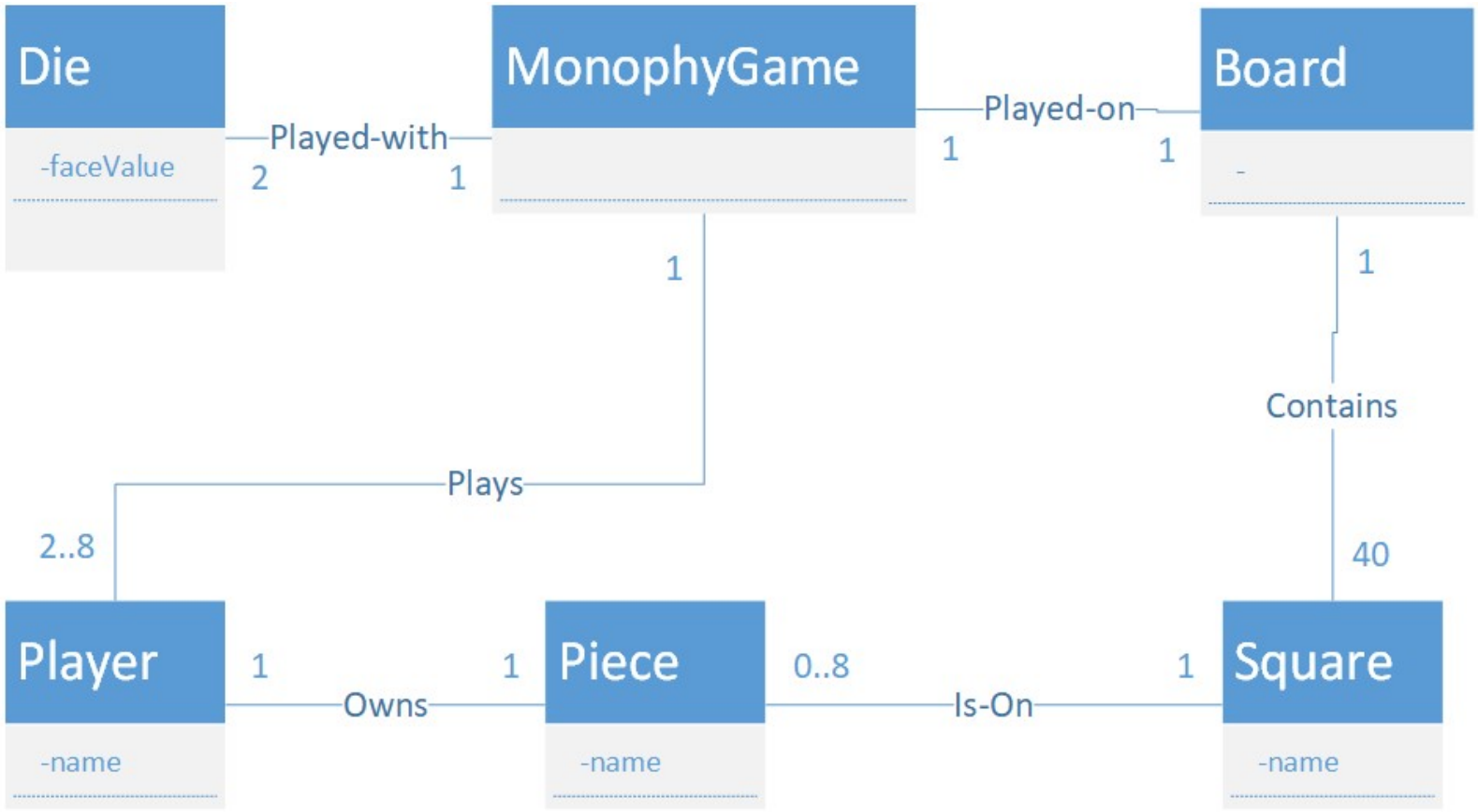


Creator

- Promotes low coupling by making instances of a class responsible for creating objects they need to reference

- By creating the objects themselves, they avoid being dependent on another class to create the object for them

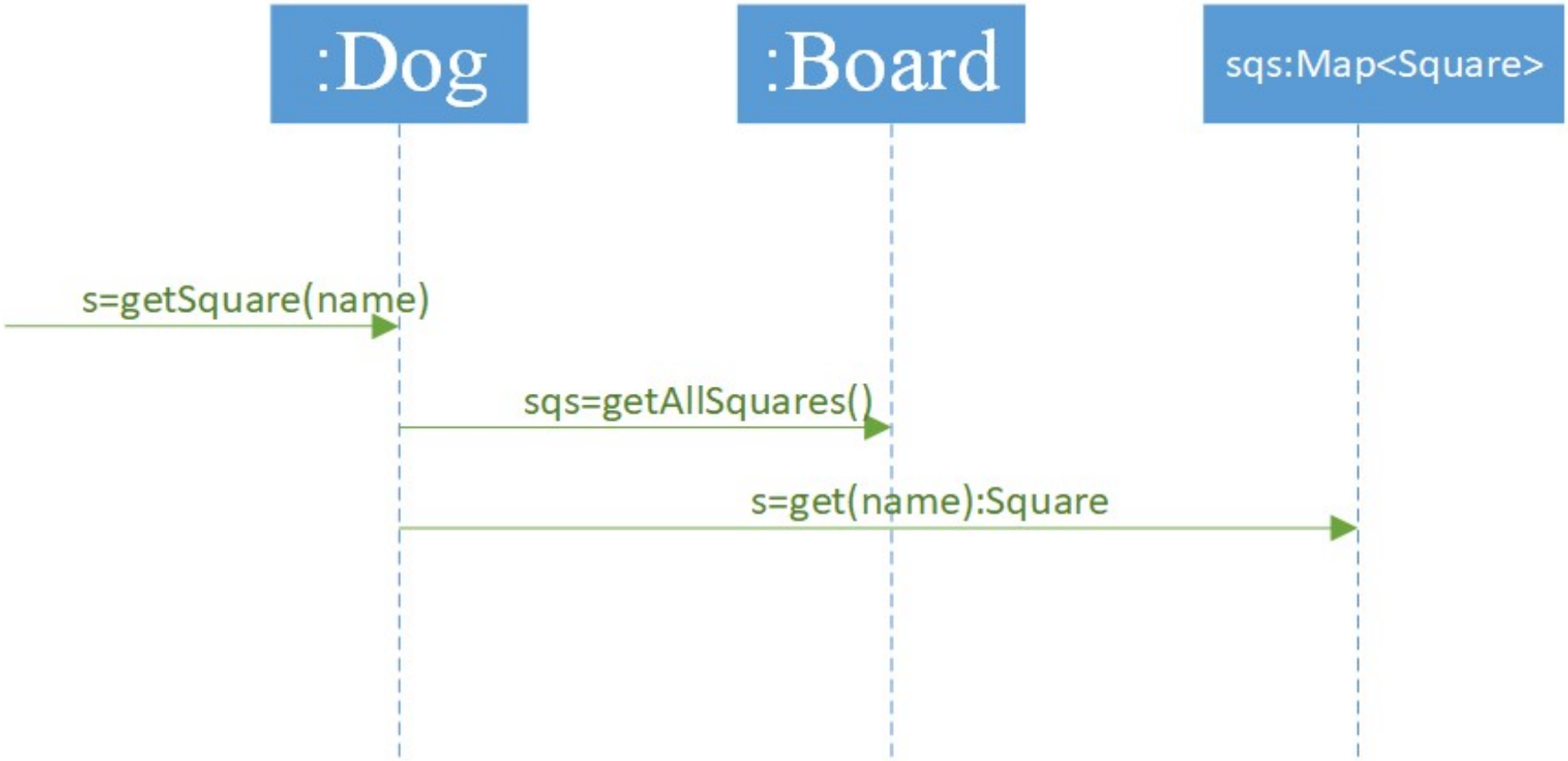
Who creates the Squares/Piece/ Player?



面向对象设计的过程

- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

下面设计正确吗?


:Dog
:Board
sqs:Map<Square>
s=getSquare(name)
sqs=getAllSquares()
s=get(name):Square

Controller

- Problem:

  How to assign responsibility for handling a system event?

- Solution:

  If a program receive events from external sources other than its graphical interface, add an event class to decouple the event source(s) from the objects that actually handle the events.

Controller pattern

Assign the responsibility for handling a system event message to a class representing one of these choices:

- The business or overall organization (a façade controller).
- The overall "system" (a façade controller).
- An animate thing in the domain that would perform the work (a role controller).
- An artificial class (Pure Fabrication) representing the use (a use case controller).

Controller : Example

- System events in Buy Items use case
  - enterItem()
  - endSale()
  - makePayment()
- Who has the responsibility for enterItem()?

Controller :Example

The choice of which one to use will be influenced by other factors such as cohesion and coupling.

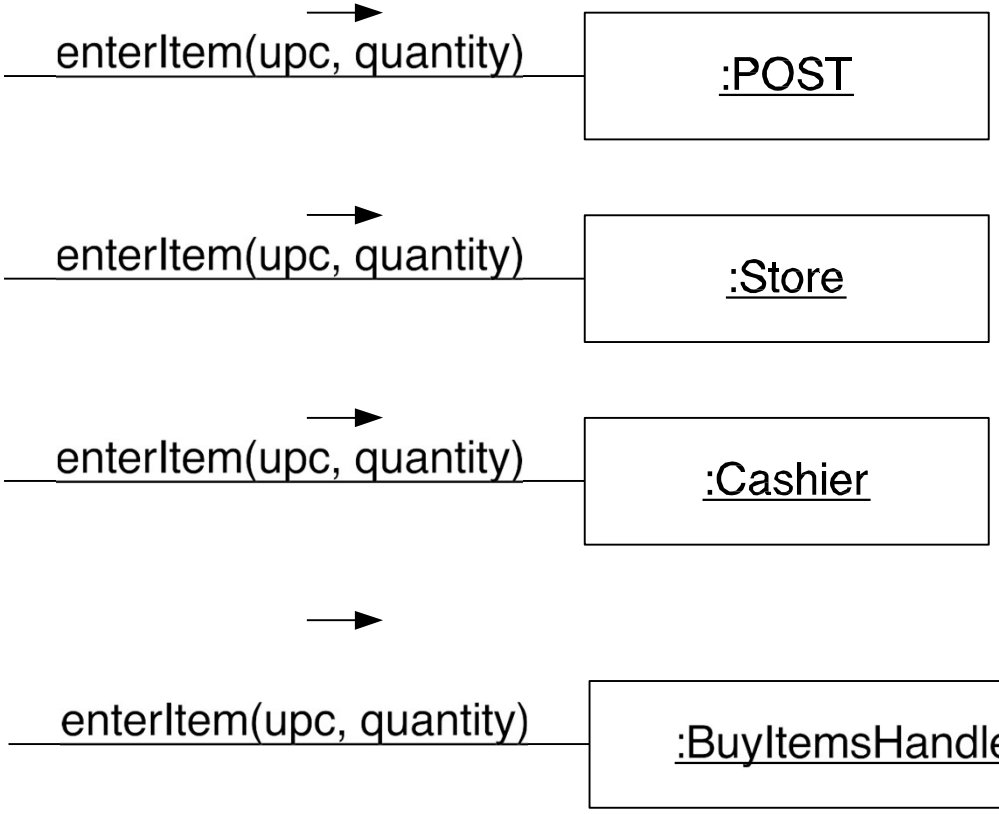
By controller, we have 4 choices:

the overall system Post

the overall business Store

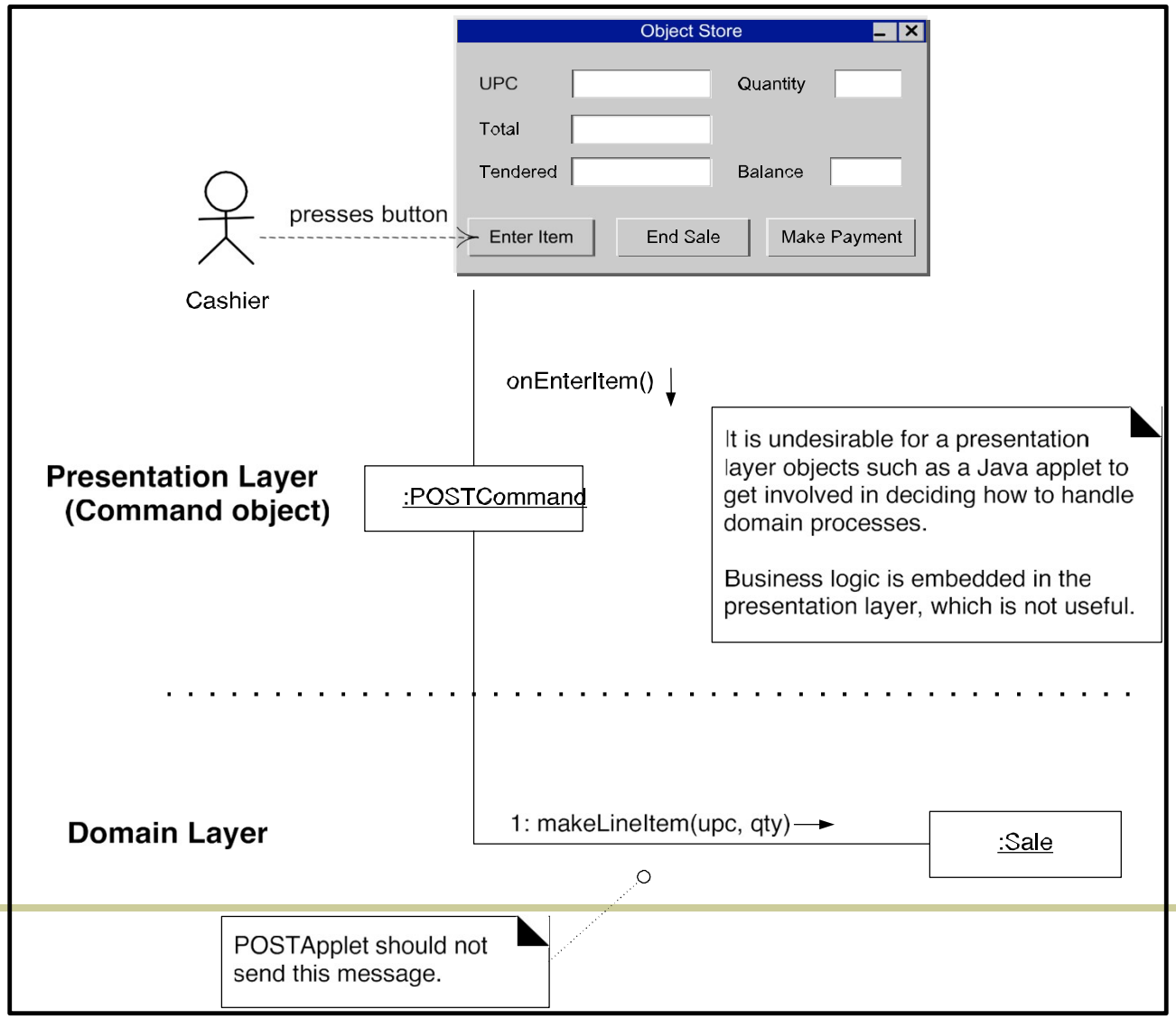
someone in the real world who is active in the task Cashier

an artificial handler of all system events of a use case BuyItemsHandler



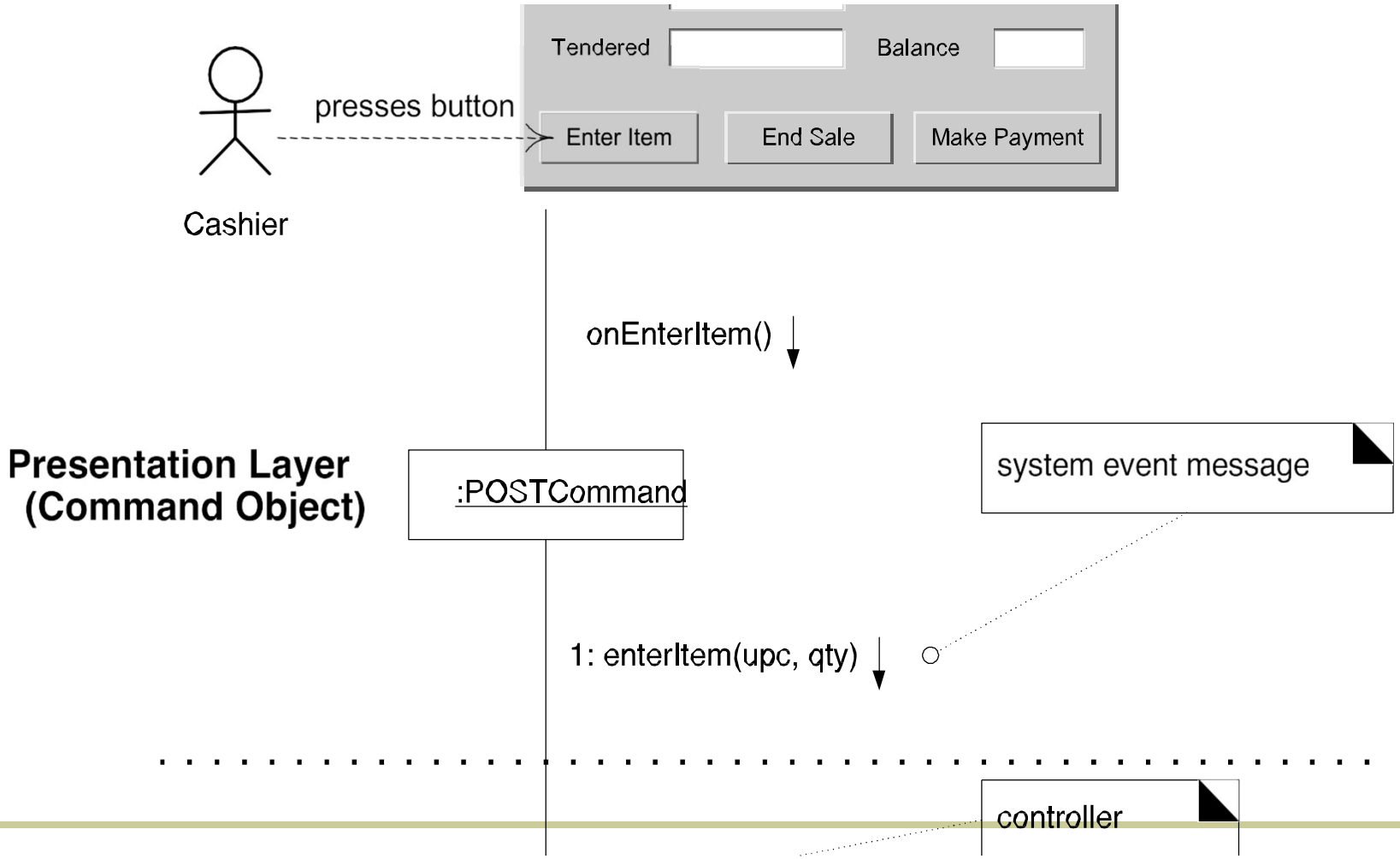
Bad design

Presentation layer coupled to problem domain



Good design

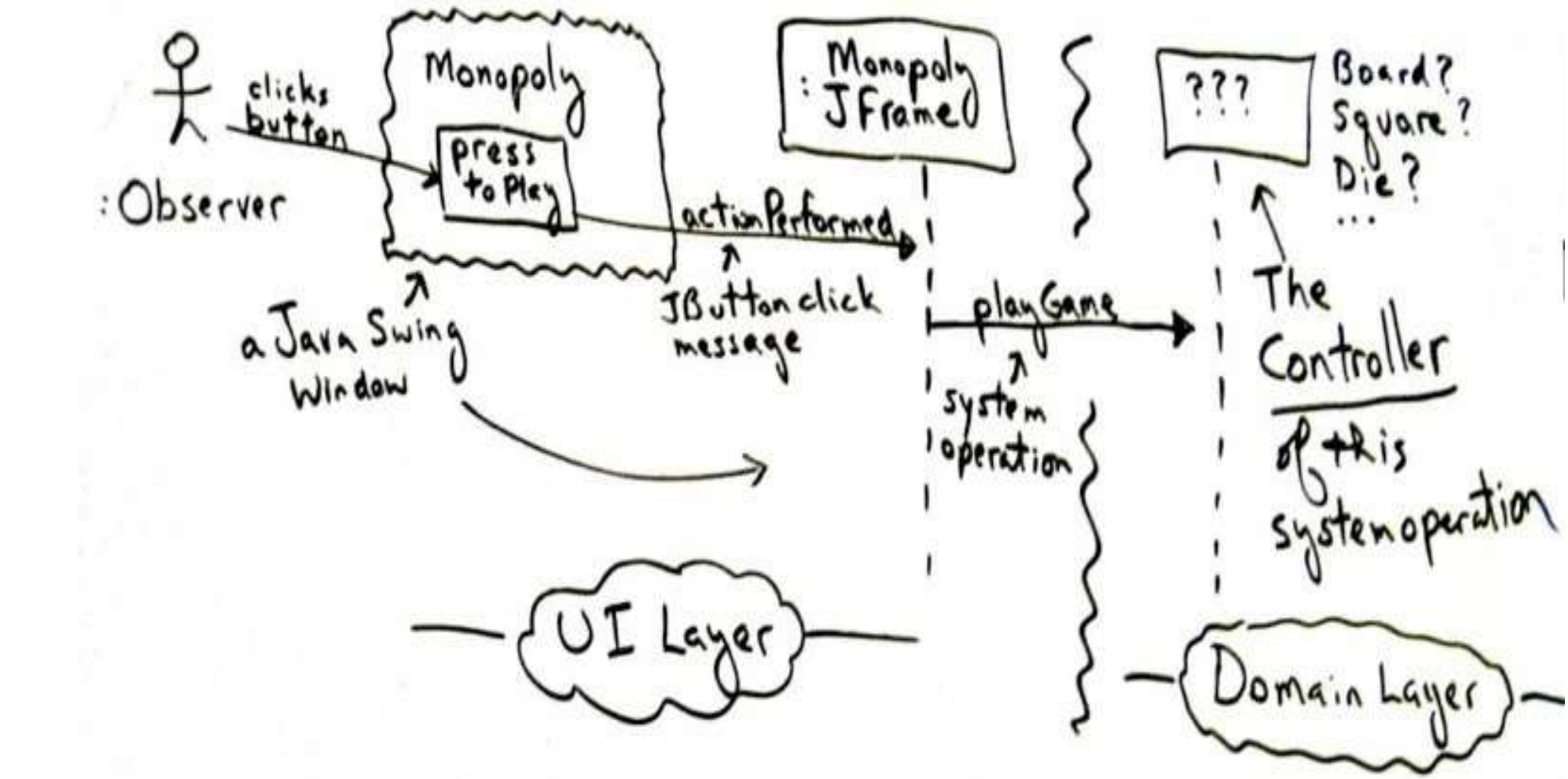
Presentation layer decoupled from problem domain



Controller

- Using a controller object keeps external event sources and internal event handlers independent of each other' type and behavior
- The controller objects can become highly coupled and incohensive with more responsibilities

Who is the controller of playGame operation?



Collaboration of a system behavior

The way that the logic of a system behavior is distributed among objects(components) network.

- Dispersed—Logics of a system behavior is spread widely through the objects network
- Centralized—One extra controller record all logics of a system behavior

Centralized Collaboration Design

Controllers

- Objects that make decisions and direct the actions of others are controllers.
- They always collaborate with others for two reasons:
  - to gather the information in order to make decisions
  - and to call on others to act.
- Their focus typically is on decision making and not on performing subsequent actions.
  - Their ultimate responsibility is often passed to others that have more specific responsibilities for part of a larger task that the controller manages

Collaboration (Control) Styles

A control style is a way that all system behavior is distributed among objects(components) network.

- Dispersed—All system behavior is spread widely through the objects network
- Centralized—A few controller record logics of all system behaviors
- Delegated—Decision making is distributed through the object networks with a few controllers making the main decisions

控制风格

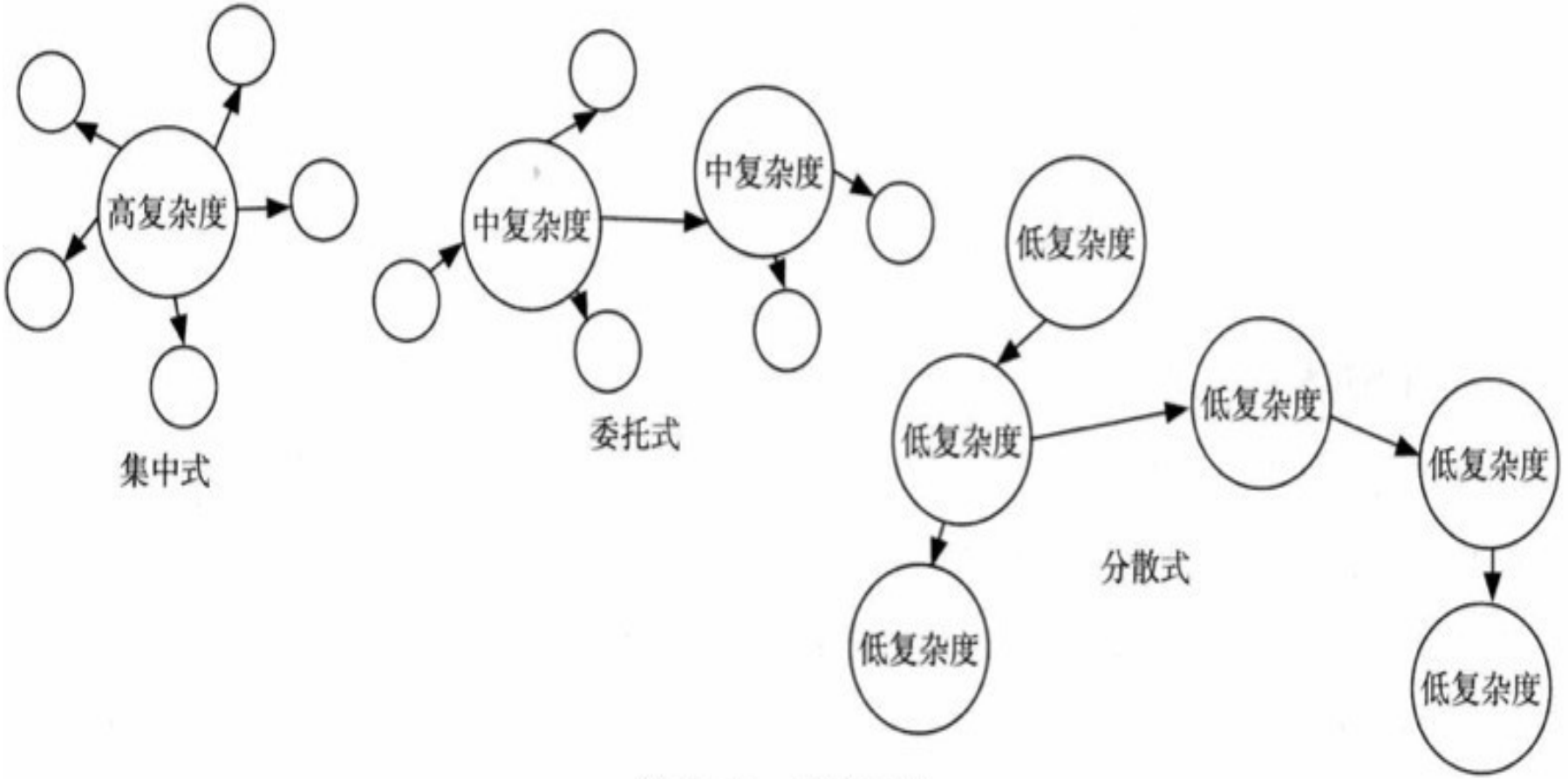
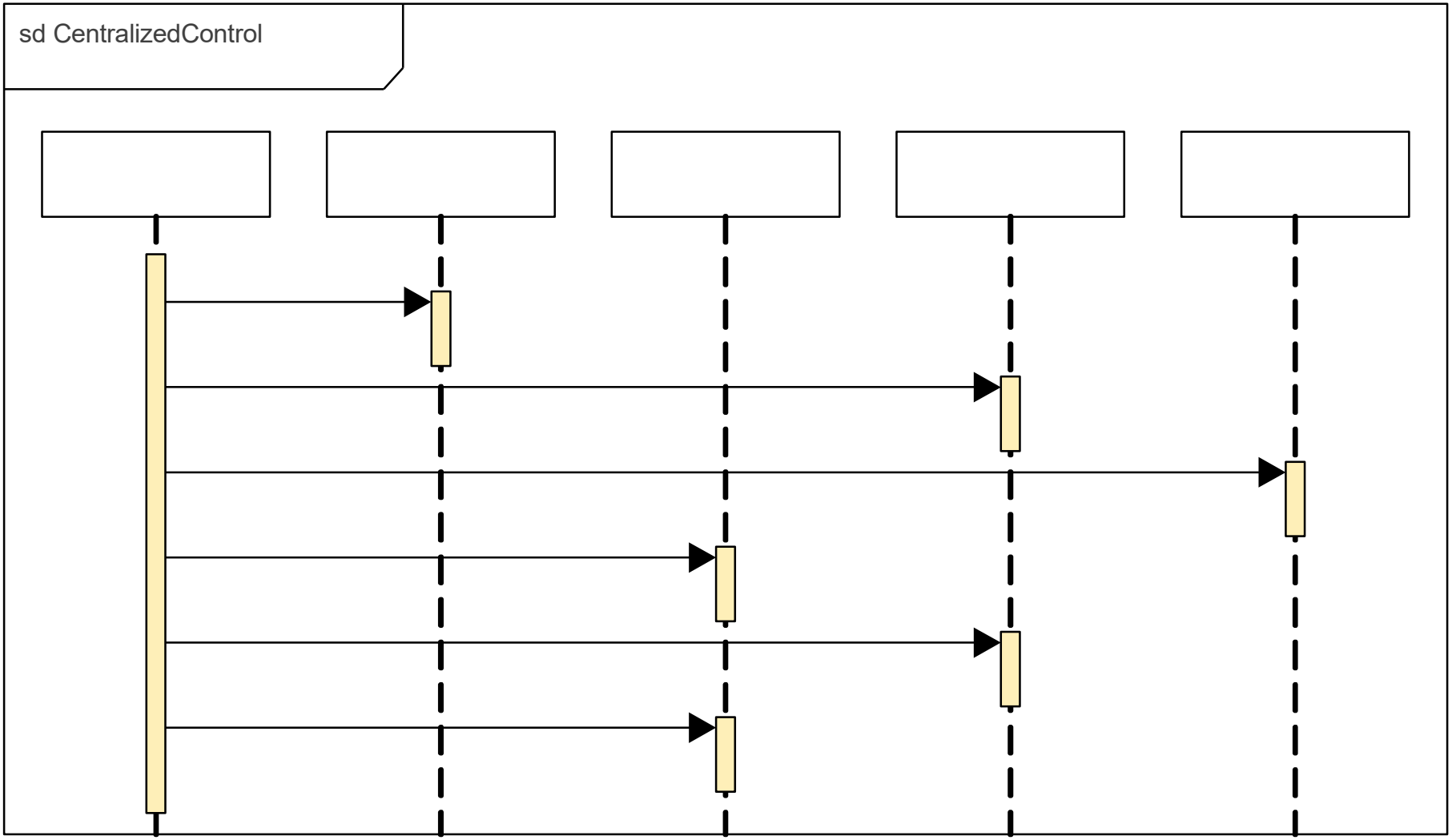


图 12-13　控制风格

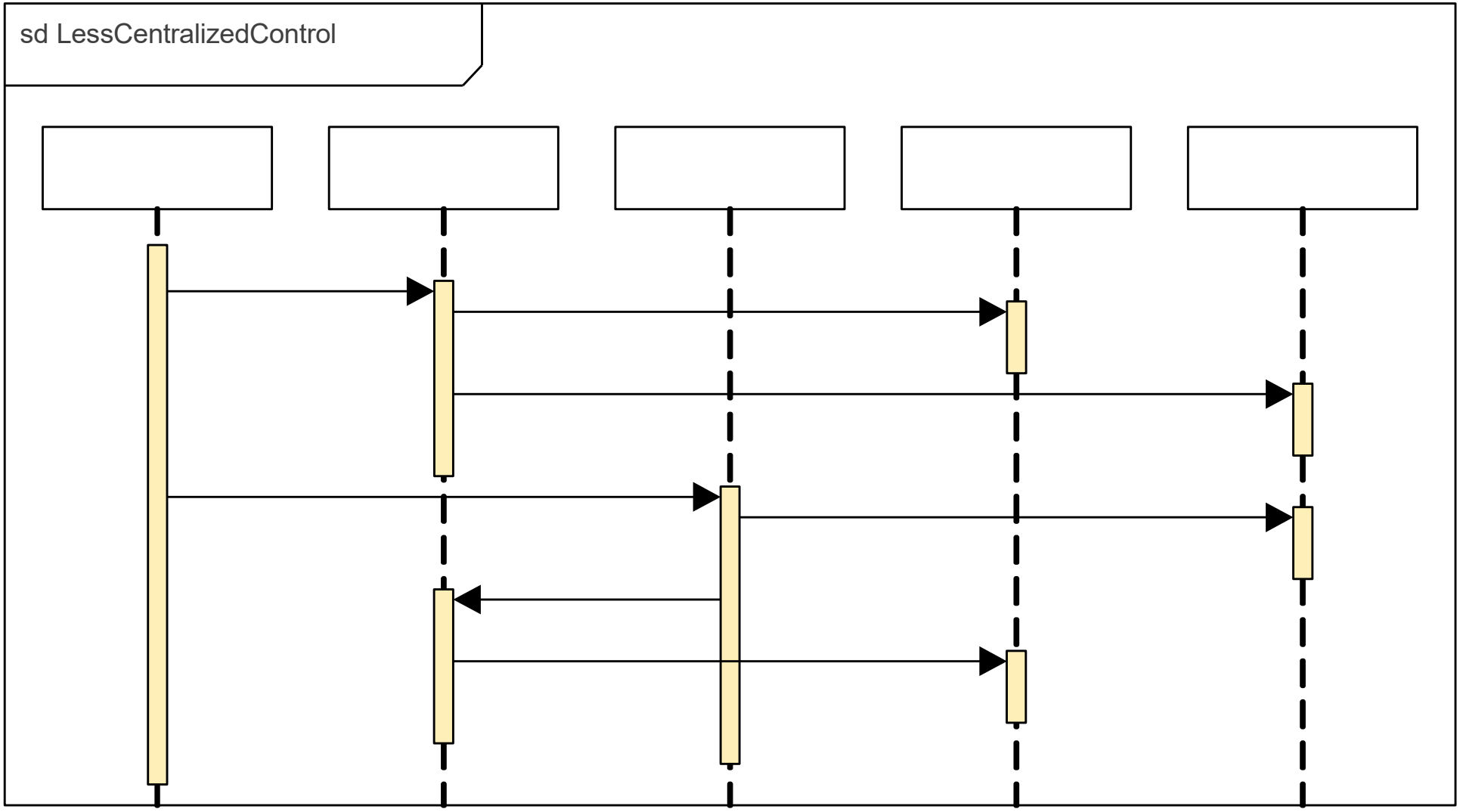
Centralized Control

- Easy to find where decisions are made
- Easy to see how decisions are made and to alter the decision-making process
- Controllers may become bloated—large, complex, and hard to understand, maintain, test, etc.
- Controller may treat other components as data repositories
- Increases coupling
- Destroys information hiding

Centralized Control Form

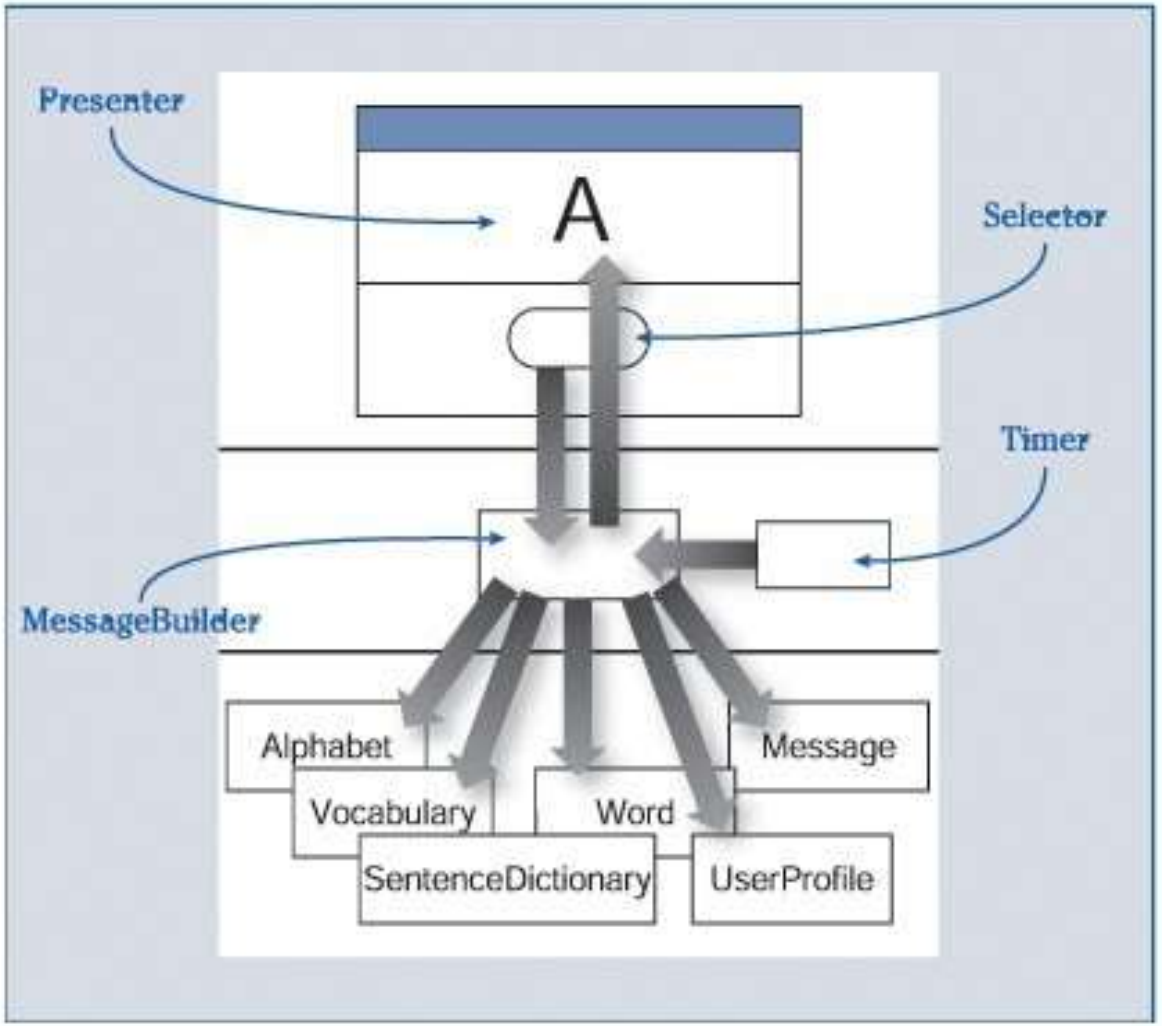


Less-Centralized Control Form

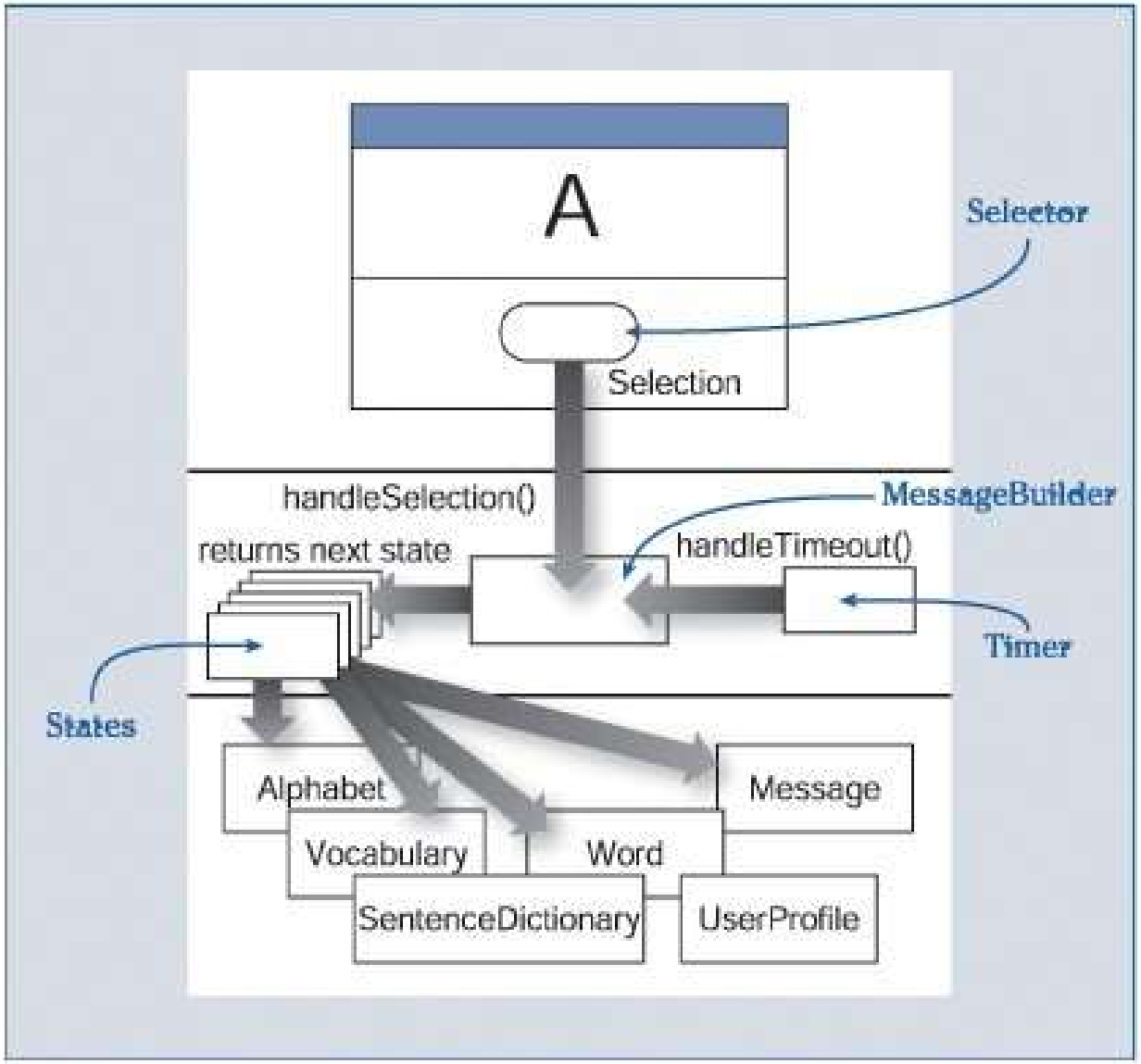


Centralized Control

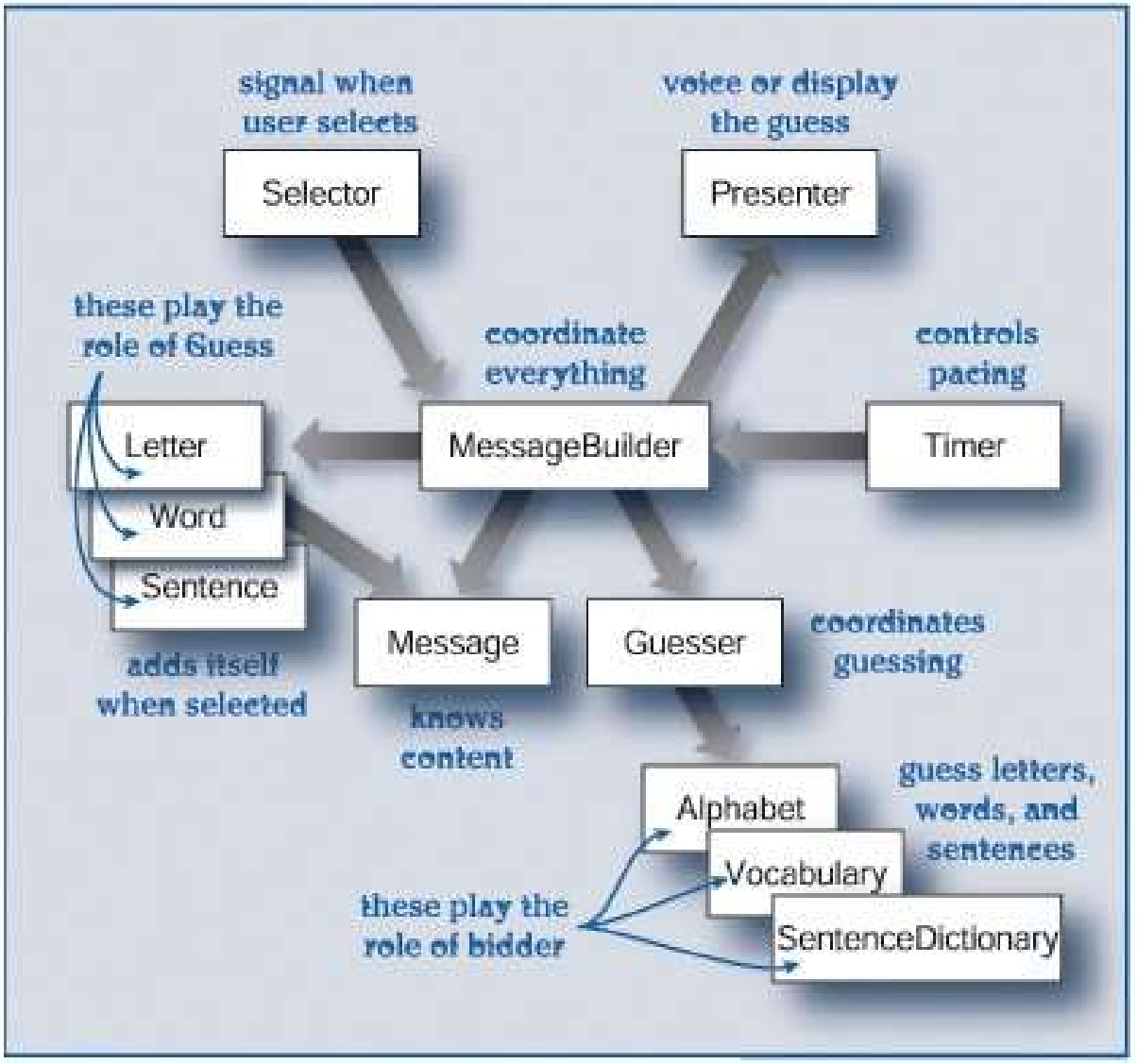
Figure 6-9. The MessageBuilder listens for events and delegates work to others.



Less-Centralized Control



Delegate Control



Control Heuristics 1

- Avoid interaction designs where most messages originate from a single component.
- Keep components small.
- Make sure operational responsibilities are not all assigned to just a few components.
- Make sure operational responsibilities are consistent with data responsibilities.

Dispersed Control Style

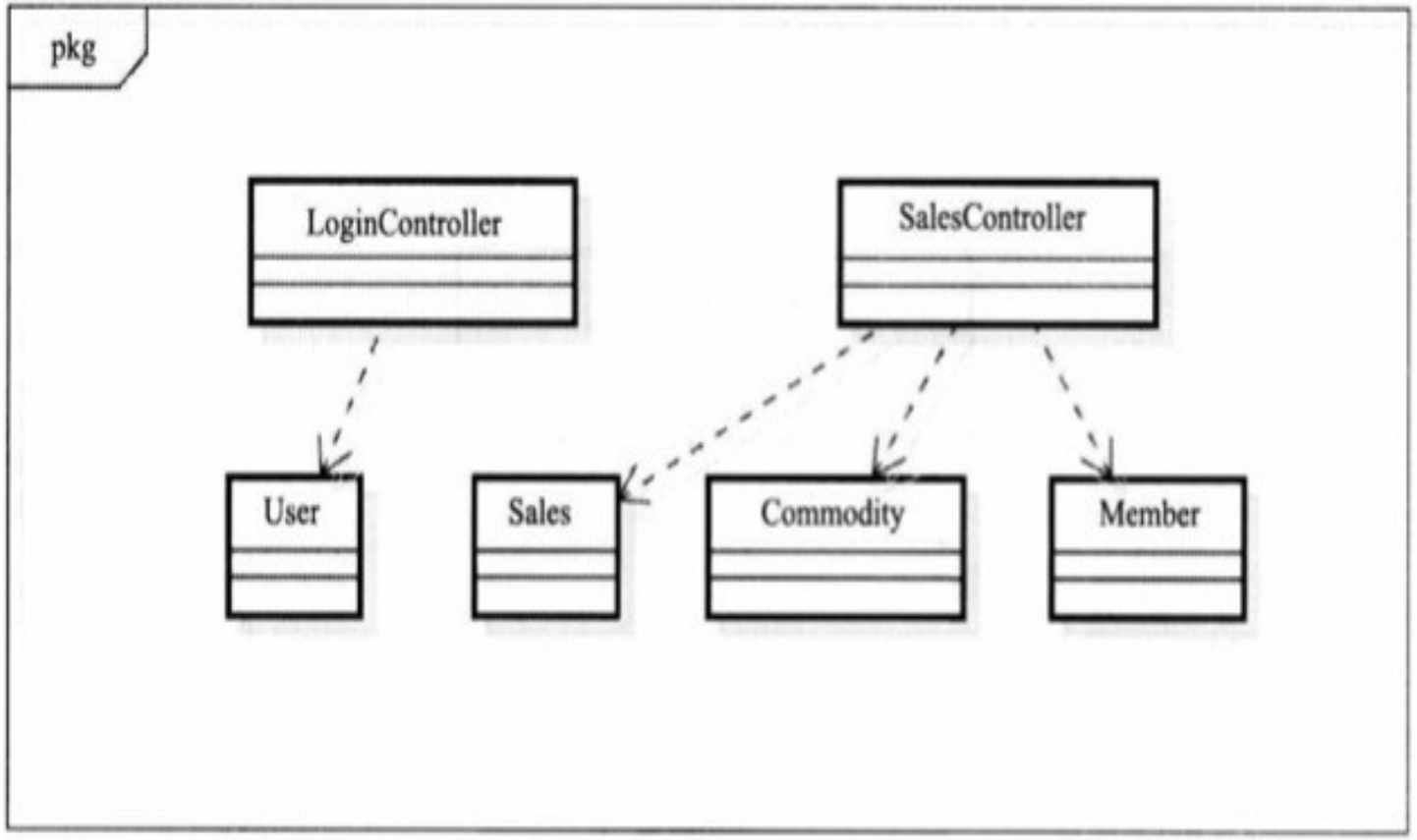
- Characterized by having many components holding little data and having few responsibilities.
- It is hard to understand the flow of control.
- Components are unable to do much on their own, increasing coupling.
- It is hard to hide information.
- Cohesion is usually poor.
- Few modularity principles can be satisfied.

Control Heuristics 2

Avoid interactions that require each component to send many messages.

案例

基于委托式控制风格的业务逻辑层的设计



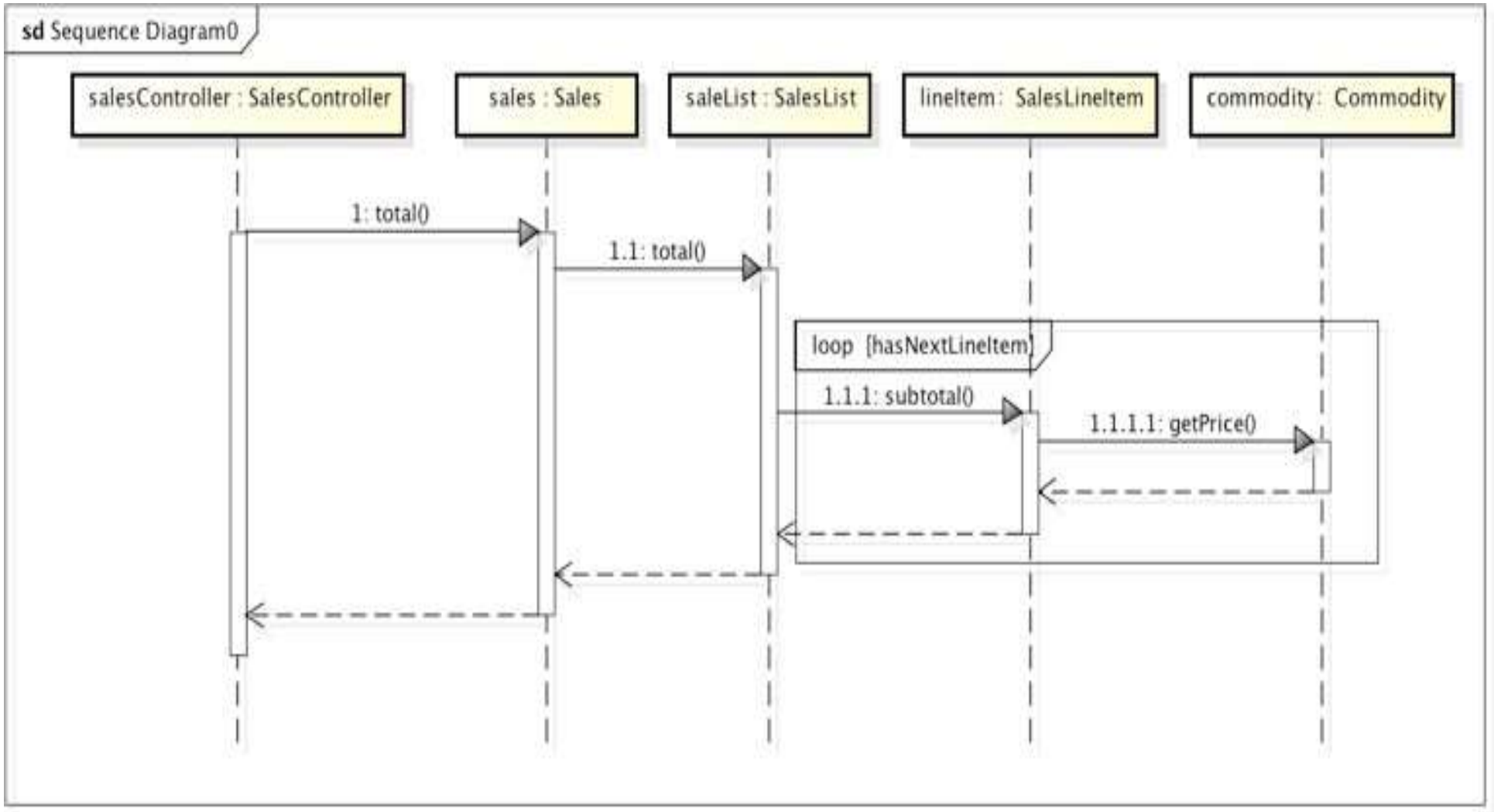
大纲

- 详细设计基础
- 面向对象详细设计
- 为类间协作开发集成测试用例
- 详细设计文档描述和评审

详细设计的集成测试

- 类间协作的集成测试
  - 重点针对复杂逻辑（交互比较多）
  - 自顶向下或者自底向上的集成
- Mock Object
  - 不是stub
- 测试用例

协作集成测试


sd Sequence Diagram0
salesController : SalesController
sales : Sales
saleList : SalesList
lineItem : SalesLineItem
commodity : Commodity
1: total()
1.1: total()
loop [hasNextLineItem]
1.1.1: subtotal()
1.1.1.1: getPrice()

Mock Object

```
public class MockCommodity extends Commodity{
        double price;
        public MockCommodity ( double p){
                price=p;
        }
        public getPrice ( ){
                return price;
        }
}
```

集成测试代码

```
public class TotalIntegratoTnTester {
        @Test
        public void testTotal () {
            MockCommdity commodity1 = new MockCommdity (50);
            MockCommdity commodity2 = new MockCommdity (40);
            SalesLineItem salesLineItem1 =
                                    new SalesLineItem (commodity1, 2);
            SalesLineItem salesLineItem2
                                    = new SalesLineItem (commodity2, 3);

            Sales sale=new Sales();
            sale.addSalesLineItem(salesLineItem1) ;
            sale.addSalesLineItem(salesLineItem2) ;

            assertEquals (220, sale.total () );
         }
}
```

主要内容

- 详细设计基础
- 面向对象详细设计
- 为类间协作开发集成测试用例
- 结构化详细设计
- 详细设计文档描述和评审

1引言

1. 编制目的

表明文档的读者，以及文档主题。

2. 词汇表

文档中用到的缩写、专业词汇等。

3. 参考资料

相关参考文献。

2中层设计

2.1 xxx模块的静态结构和动态行为

2.1.1 xxx模块局部模块的职责

通过逻辑视角、结构视角、依赖视角描述其相应的职责。

2.1.2 xxx模块局部模块的接口规范

各子层的供接口和需接口的规范。

2.1.3 xxx模块的行为

用例执行时，对象之间的消息传递和状态的转移。通常用顺序图、通讯图、状态表示。

2.1.4 xxx模块的实现注解

具体实现时注意点。比如构造方法、枚举、常量、静态方法的说明

2.1.5 业务逻辑模块的设计原理

详细设计验证

- 评审
- 度量
  - 模块化度量
- 测试
  - 协作测试

详细设计的评审

一、基本

1）设计方案自身是否一致?

2）设计制品的详细程度是否合适?

3）设计是否包含了各个视角?

4）多个视角之间是否一致?

二、设计考量

5）设计是否采用了标准技术，而不是晦涩难懂的技术?

6）设计是否强调简洁性重于灵活性?

7）设计是否尽可能简单?

8）设计是否精干? 每个部分都是必需的?

9）如果维护时需求发生变更，需要修改的地方是否支持修改? 是否支持未来的扩展?

10）设计是否支持重用?

11）设计是否具有低复杂性?

12）设计是否是可理解的? 是否没有超越普通人的智力范围?

三、过程考量

13）设计是否覆盖了所有的需求?

14）设计中设计的功能对应需求的哪些部分?

15）是否足够遵循软件体系结构设计的决策?

16）设计的详细程度对后继开发人员是否足够?