System Analysis and Design

L17. Designing Objects with Responsibilities

Topics

- Modeling for Object Design
- Responsibility-Driven Design
- GRASP Patterns
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion

Understanding OOD

- After
  - identifying your requirements and
  - creating a domain model,
- Add methods to the appropriate classes
- Define the messaging between the objects to fulfill the requirements.

- Such vague advice doesn't help us, because deep principles and issues are involved.
- The Critical Tool for software development is Not UML but **a mind well educated in design principles**

About Object Design

- What has been done?
  Prior activities and artifacts
- How do things relate?
  Influence of prior artifacts (e.g., use cases) on OO design
- How much design modeling to do, and how?
- What is the output?

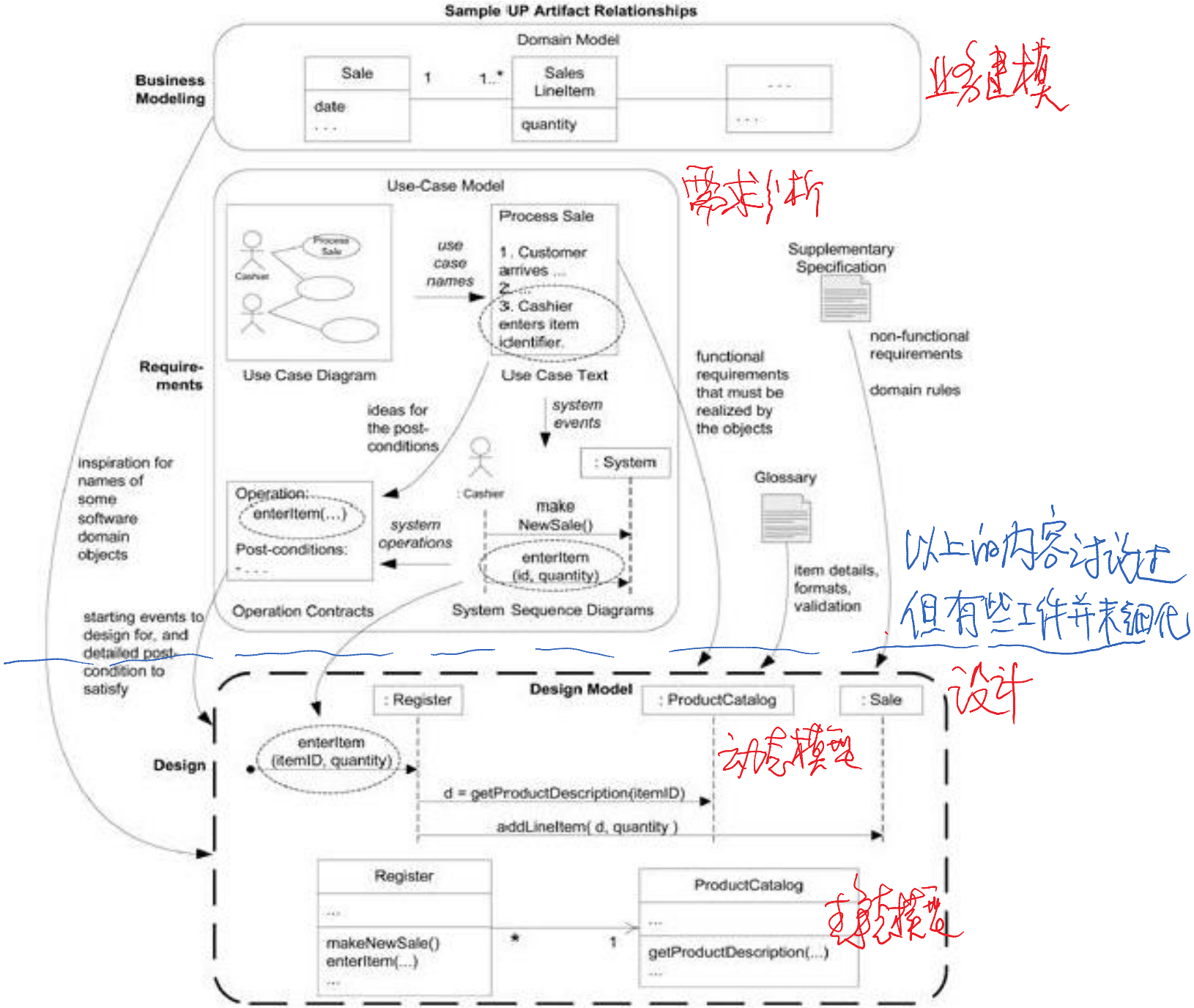
- Especially, you should understand how the analysis artifacts relate to object design. 理解分析工件与对象设计的关系

到此为止已完成的工作：

Process Comes to the Point

The first ***two-day requirements workshop*** is finished.	The chief architect and business agree to implement and test some ***scenarios of Process Sale in the first three-week*** timeboxed iteration.
Three of the twenty use cases those that are the most architecturally significant and of high business value have been analyzed in detail, including, of course, the *Process Sale* use case. (The UP recommends, as typical with iterative methods, analyzing only ***10%-20% of the requirements*** in detail before starting to program.)	***Other artifacts*** have been started: Supplementary Specification, Glossary, and Domain Model.
Programming experiments have resolved the show-stopper technical questions, such as whether a Java Swing UI will work on a touch screen.	The chief architect has drawn some ideas for the ***large-scale logical architecture***, using UML package diagrams. This is part of the UP Design Model.

Artifact influence on OO design



Artifact Inputs to Object Design

The **use case text** defines the visible behavior that the software objects must ultimately support objects are designed to "realize“ (implement) the use cases. In the UP, this OO design is called, not surprisingly, the **use case realization**.	**The Supplementary Specification** defines the non-functional goals, such as internalization, our objects must satisfy.
The **system sequence diagrams** identify the system operation messages, which are the starting messages on our interaction diagrams of collaborating objects.	The **Glossary** clarifies details of parameters or data coming in from the UI layer, data being passed to the database, and detailed item-specific logic or validation requirements, such as the legal formats and validation for product UPCs (universal product codes).
The **operation contracts** may complement the use case text to clarify what the software objects must achieve in a system operation. The post-conditions define detailed achievements.	The **Domain Model** suggests some names and attributes of software domain objects in the domain layer of the software architecture.

What Are Activities of Object Design?

- Start **coding**, with test-first development
- Start some **UML modeling** for object design
- Or start with **another modeling** technique such as CRC cards

Modeling for Object Design

- We draw both interaction diagrams (dynamic modeling) and complementary class diagrams (static modeling)
- Most importantly, apply various OO design principles, such as **GRASP** and **GoF design patterns**.
- The overall approach to doing the OO design modeling will be based on the *metaphor* of **responsibility-driven design** (RDD)
  - thinking about how to assign responsibilities to collaborating objects.

During a Modeling Day

- Working in small groups for 2-6 hours either at the walls or with tools
- Doing different kinds of modeling for the difficult, creative parts of the design (UML drawings, prototyping tools, sketches, and so forth)
- Doing some code generation with a UML tool

- On last Tuesday of the iteration, stops modeling and begin to do programming.
- Models are to understand and communicate, not to document

对象设计的输出工件.

Outputs of Object Deign

What's been created during the modeling day?

- (Specifically for object design) UML interaction, class, and package diagrams for the difficult parts of the design that we wished to explore before coding
- UI sketches and prototypes
- Database models
- Report sketches and prototypes

Responsibilities

- UML defines a **responsibility** as "a contract or obligation of a classifier"
- Responsibilities are of two types: *doing* and *knowing*.
- **Doing** responsibilities
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- **Knowing** responsibilities
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

Responsibility-Driven Design

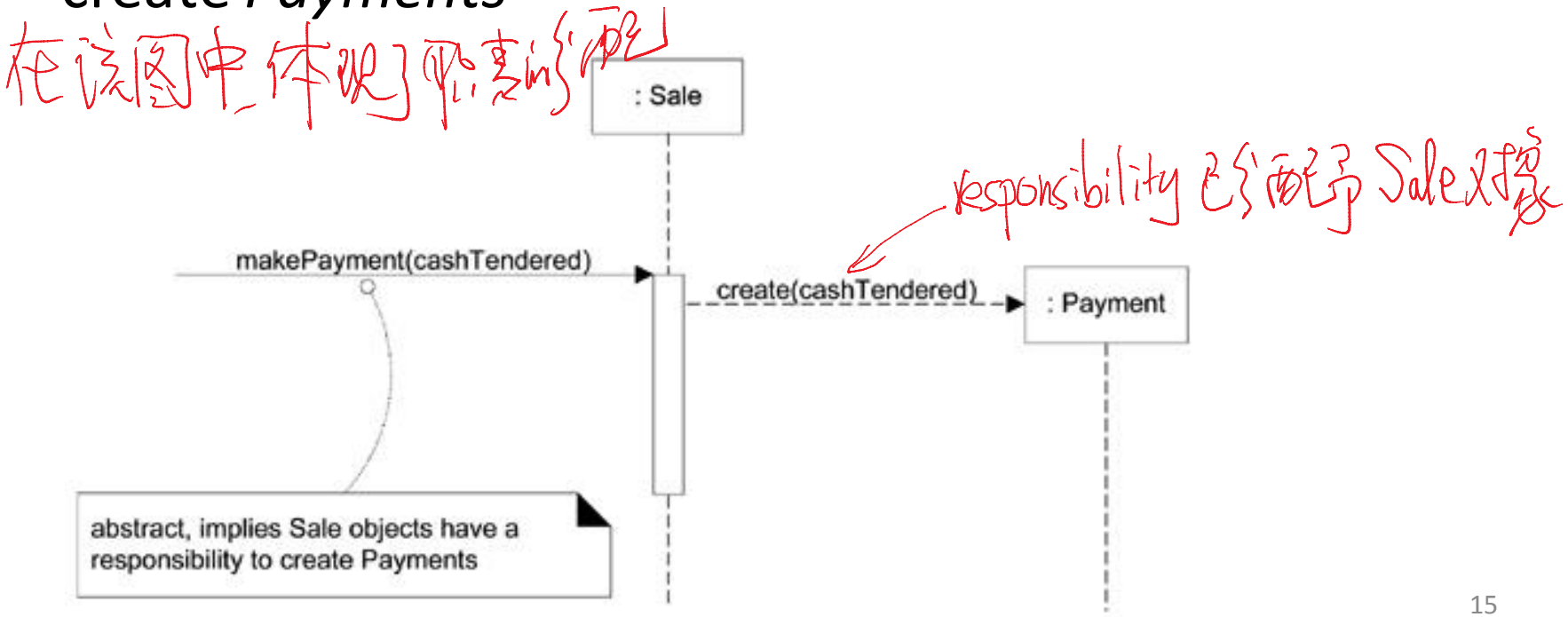
- Responsibilities are assigned to classes of objects during object design 我们的任务是给类分配职责.
- **Knowing** responsibilities are often inspired by domain model, and this Lows representational gap 知识职责一般由领域模型所指示.
- The translation of responsibilities into classes and methods is influenced by the *granularity* of the responsibility. 职责粒度影响编码
- A responsibility is not the same thing as a method, but methods fulfill responsibilities. 方法可以实现职责
- RDD also includes the idea of **collaboration** 设计时要考虑对象间的协作关系
- RDD is a general metaphor for thinking about OO software design RDD是OOD的总思想.
- RDD leads to viewing an OO design as a *community of collaborating responsible objects*. 把任职责的对象协作工作构成一个群体.

GRASP

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**attern
- GRASP names and describes some basic principles to assign responsibilities to support RDD 是基本的原则(RDD)
- This approach to understanding and using design principles is based on *patterns of assigning responsibilities*. 基于职责分配的一组原则(模式)
- GRASP is a learning aid for OO design with responsibilities 是学习OOD的工具

Apply GRASP Principles While Drawing and Coding

- You can apply the GRASP principles while drawing UML interaction diagrams, and also while coding
- *Sale* objects have been given a responsibility to create *Payments*



Design Patterns 设计模式

- General principles and idiomatic solutions for creating software 是创建软件的一般性原则和成熟的方案
- It is usually codified in a structured format describing the problem and solution 以结构性格式描述
- In OO design, a **pattern** is a named description of a problem and solution that can be applied to new contexts 可应用于新的环境
- GRASP defines nine basic OO design principles or basic building blocks in design. GRASP定义了九个这样的原则

GRASP Patterns

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

GRASP原则1：

Creator Pattern

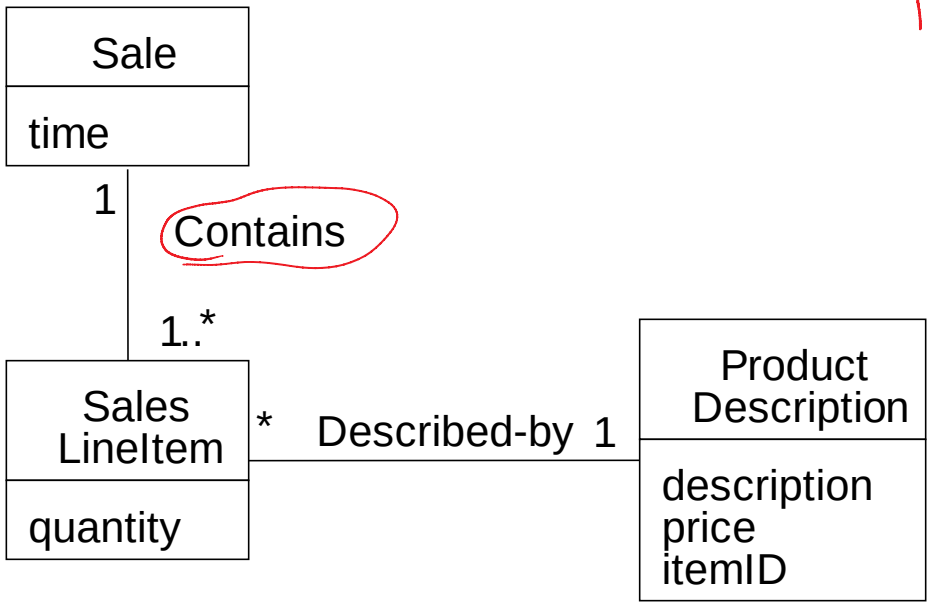
如何分配创建实例"职责"？

- **Problem**: Who should be responsible for creating a new instance of some class?
- **Solution**: assign class B the responsibility to create an instance of A if one or more are true:
  - B contains or aggregates A
  - B records A
  - B closely uses A
  - B has the initializing data for A

若其一或多个条件成立，可以赋予B创建A实例职责。

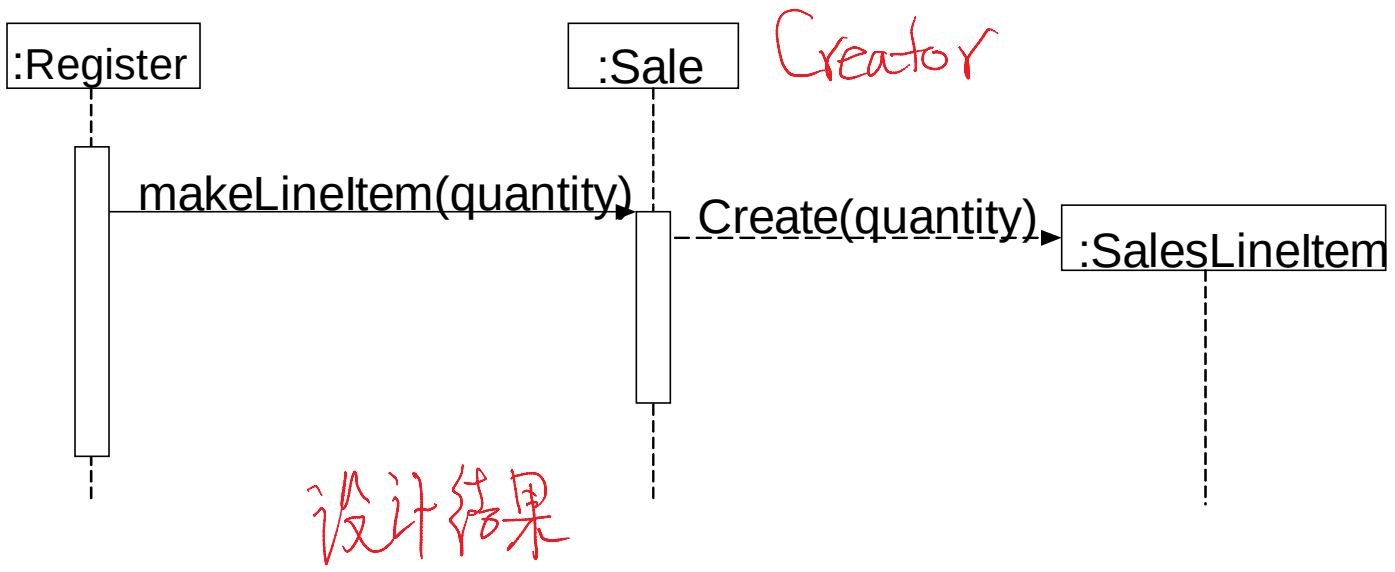
NextGen POS Example

- Who should be responsible for creating a SalesLineItem instance?
  - By Creator, look for a class that aggregates, contains SalesLineItem instances.



NextGen POS Example

- A Sale contains (aggregates) many SalesLineItem objects
  - Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.
  - This leads to the design of object interactions shown
  - This assignment of responsibilities requires that a *makeLineItem* method be defined in Sale



不适用的情况

Creator - Contraindications

- When not to use:
  - When creating requires complexity, such as using existing instances of a class, conditionally creating an instance from one class or another based upon some external property, etc. 复杂性时不适用.
  - In these cases, delegate creation to a helper class called a *Concrete Factory* or *Abstract Factory*

此时用其它模式 (具体工厂或抽象工厂)

(我们后面会讨论这种情况)

Benefits and Related Patterns

- Benefits
  - **Low coupling** is supported, which implies lower maintenance dependencies and higher opportunities for reuse.
    - Coupling is not increased because the created class is visible to the creator class, due to the existing associations.
- Related Patterns or Principles
  - Low Coupling
  - Concrete Factory and Abstract Factory

GRASP原则2：

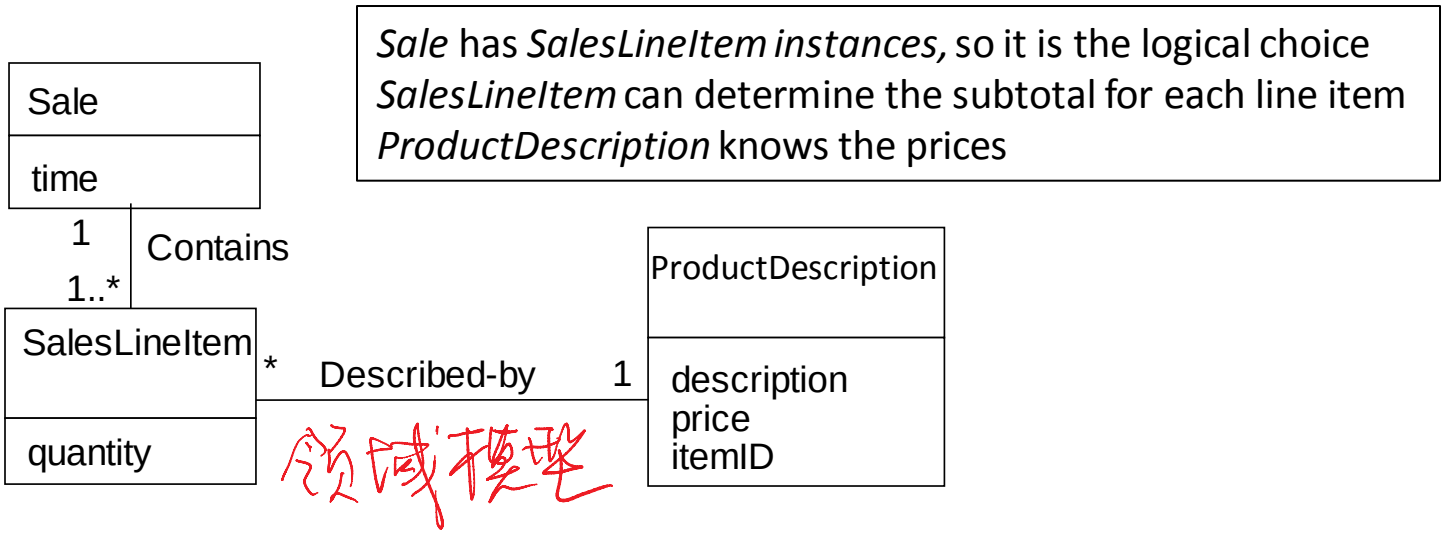
Information Expert

- Problem:
  What is a basic principle by which to assign responsibilities to objects? 职责分配的原则是什么？
- Solution:
  Assign a responsibility to the class that has the information needed to fulfill it.
  如果类具有实现职责所需的信息，则将职责分配给它。

NextGEN POS Example

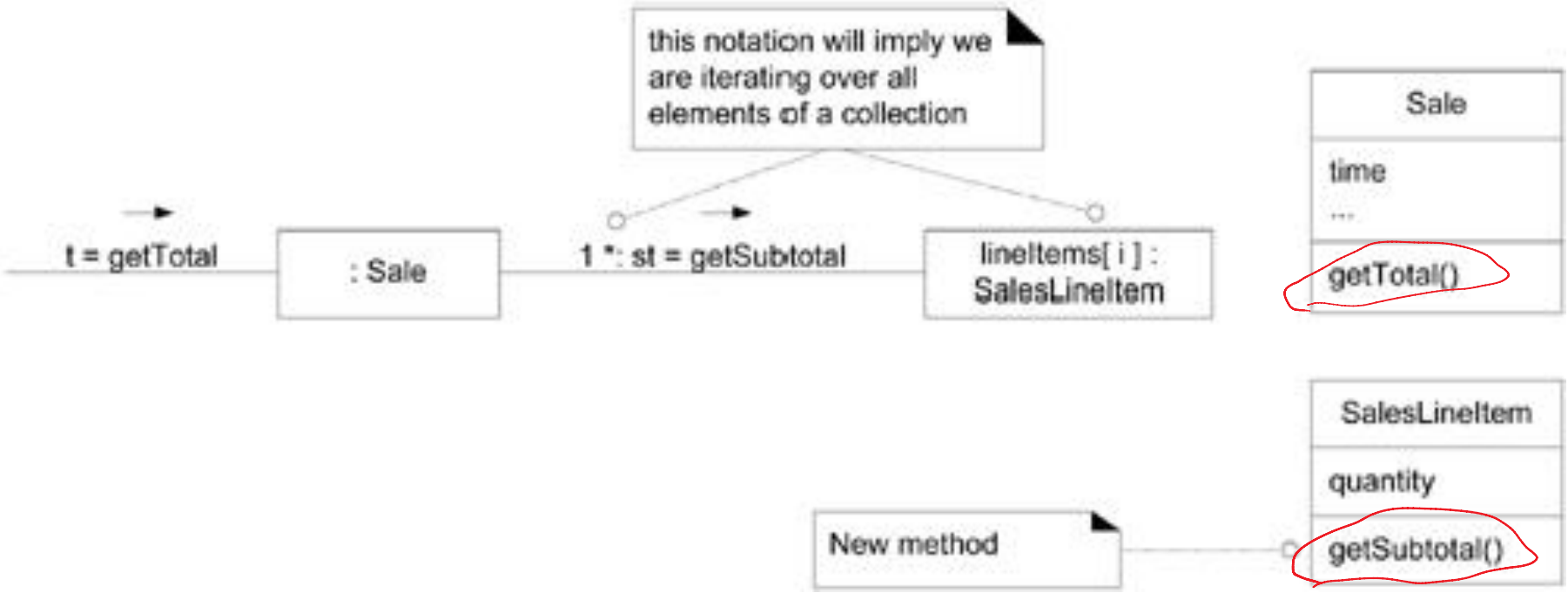
Some class needs to know the grand total of all the SalesLineItem instances of a sale.

- Who should be responsible for knowing that?
- Information Expert of the Domain Model : look for class that has the information needed to determine the total - Sale.
  - give the responsibility of knowing its total - method getTotal.
- This approach supports **low representational gap** between software design of objects and the concepts of real domain.



Responsibility Assignment

Sale has *SalesLineItem instances,* so it is the logical choice
SalesLineItem can determine the subtotal for each line item
ProductDescription knows the prices



Information Expert

- This is the class that has the information necessary to fulfill some responsibility.
- If there are relevant classes in the design model, look there first.
- If not, look at the domain model

不适用的地方

Expert -- Contraindications

- When there are problems with coupling or cohesion 存在耦合与内聚问题时，不适用
- For example, who should be responsible for saving a *Sale* in a database?
- Sale has all the sale data, so you could logically think it would store that in the database.
- However, that makes Sale dependent upon database (third layer) logic. 导致Sale与数据库本层密耦合！

GRASP 原則3:

Low Coupling

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements
- An element with Low Coupling is not dependent upon too many other elements. But how many is too many?
- Problems with high coupling: 強耦合導致的問題.
  - Forced local changes because of changes in related classes
  - Harder to understand in isolation
  - Harder to reuse because it requires other classes

Low Coupling 原则

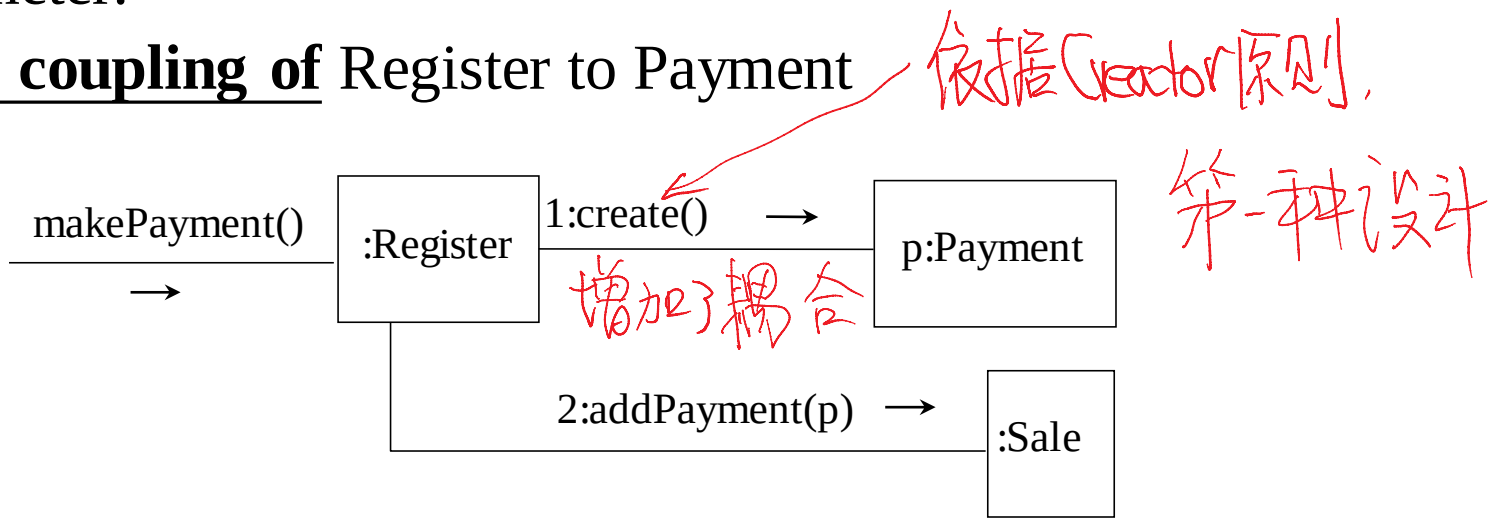
- Problems: 如何减少变化产生的影响？
  - How to reduce the impact of change?
- Solution:
  - Assign responsibility so coupling remains low
  - This reduces the impact of change 分配职责时保证耦合低，这样降低了变化的影响

We use Low Coupling to *evaluate* existing designs or to evaluate the choice between new alternatives, all other things being equal, we should prefer a design whose coupling is lower than the alternatives. 在设计时选择低耦合的选项.

NextGen Example

Design 1

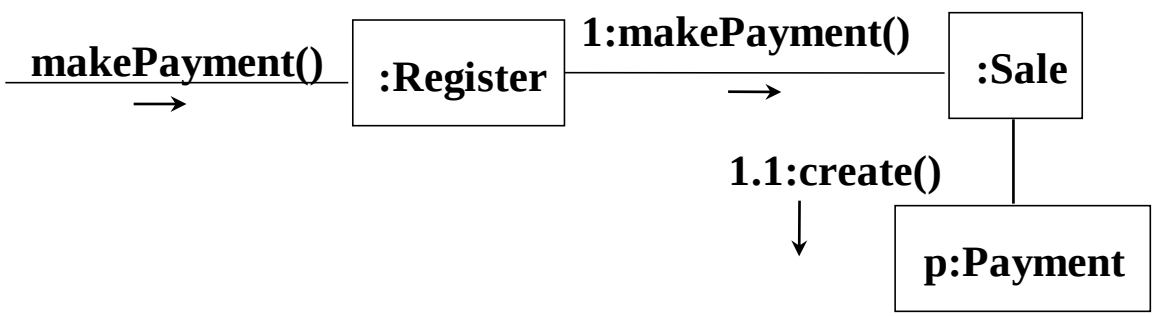
- Since a Register "records" a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment.
- The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.
- **Adds coupling of** Register to Payment



NextGen example

Design 2

- The Sale does the creation of a Payment, **does not increase the coupling**.
- Purely from the point of view of coupling, prefer Design 2 because it maintains overall lower coupling.
- In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and High Cohesion. 实际当中，不要独立考虑一个方面的原则



第二种设计
没有增加耦合

通常的耦合关系

Common Forms of Coupling from X to Y

- X has an attribute that refers to a Y instance, or Y itself. 属性
- X object calls on services of a Y object. 调用服务.
- X has a method that references an instance of Y, or Y itself, by any means. These typically include a parameter or local variable of type Y, or the object returned from a message being an instance of Y. 方法使用
- X is a direct or indirect subclass of Y. 子类
- Y is an interface, and X implements that interface. 实现类.

Discussion

子类强耦合于父类.

- A subclass is strongly coupled to its superclass.
  - Suppose objects must be stored persistently in a relational or object database.
  - Creating an abstract superclass called PersistentObject.
  - Advantage: automatic inheritance of persistence behavior.
  - Disadvantage: it highly couples domain objects to a particular technical service and mixes different architectural concerns.
- Classes that are inherently generic and high reuse should have low coupling. 通用的、高复用的类应该具有低耦合性

Discussion

- Contraindications: High coupling to stable elements and to pervasive elements is seldom a problem.
  - J2EE application can safely couple itself to the Java libraries (java.util.*), because they are stable and widespread. 高耦合于稳定的元素一般不会有大问题.
- Focus on the points of realistic high instability or evolution. 所以应该关注不稳定和变动元素的依赖性(耦合性)
- Benefits at this point
  - not affected by changes in other components
  - simple to understand in isolation
  - convenient to reuse

GRASP原则4:

Controller

- A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.
- System operations were first explored during the analysis of SSD. These are the major input events upon our system
- For example, the “End Sale” button in a POS system or the “Spell Check” button on a word processor.
- This is a delegation pattern

Controller

- **Problem:** What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- **Solution:** Assign responsibility to a class with one of the following:
  - Represents the overall “system,” a “root object,” a device the software is running within, or a major subsystem (***façade controller***)
  - Represents a use case scenario within which the system event occurs, often called <UseCaseName>Handler (**Usecase controller**)
- Window, View, and Document are not on the list

a design pattern.

我们不将Controller引入UI层中！

Common layers in an information system logical architecture

- GUI windows
- reports
- speech interface
- HTML, XML, XSLT, JSP, Javascript, ...

UI

UI
(AKA Presentation, View)

用户界面
表示层

应用相关性增高

- handles presentation layer requests
- workflow
- session state
- window/page transitions
- consolidation/transformation of disparate data for presentation

Application
(AKA Workflow, Process, Mediation, App Controller)

应用层
控制层

依赖方向

dependency

more app specific

- handles application layer requests
- implementation of domain rules
- domain services (POS, Inventory)
- services may be used by just one application, but there is also the possibility of multi-application services

Domain
(AKA Business, Application Logic, Model)

领域层 Model

- very general low-level business services used in many business domains
- CurrencyConverter

Business Infrastructure
(AKA Low-level Business Services)

业务逻辑层

- (relatively) high-level technical services and frameworks
- Persistence, Security

Technical Services
(AKA Technical Infrastructure, High-level Technical Services)

技术服务

- low-level technical services, utilities, and frameworks
- data structures, threads, math, file, DB, and network I/O

Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)

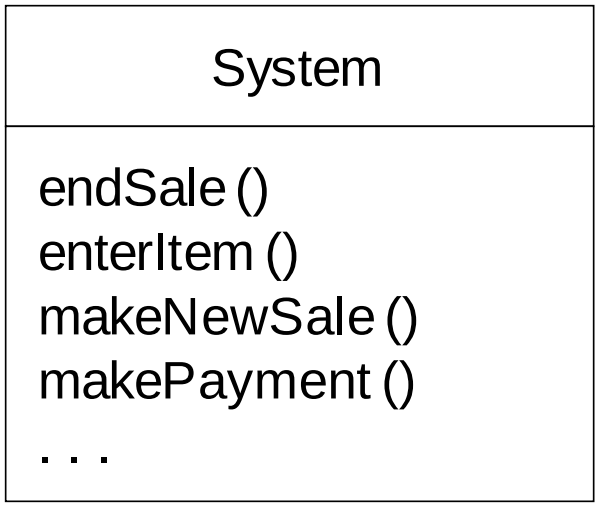
基础服务层

width implies range of applicability

可应用性范围

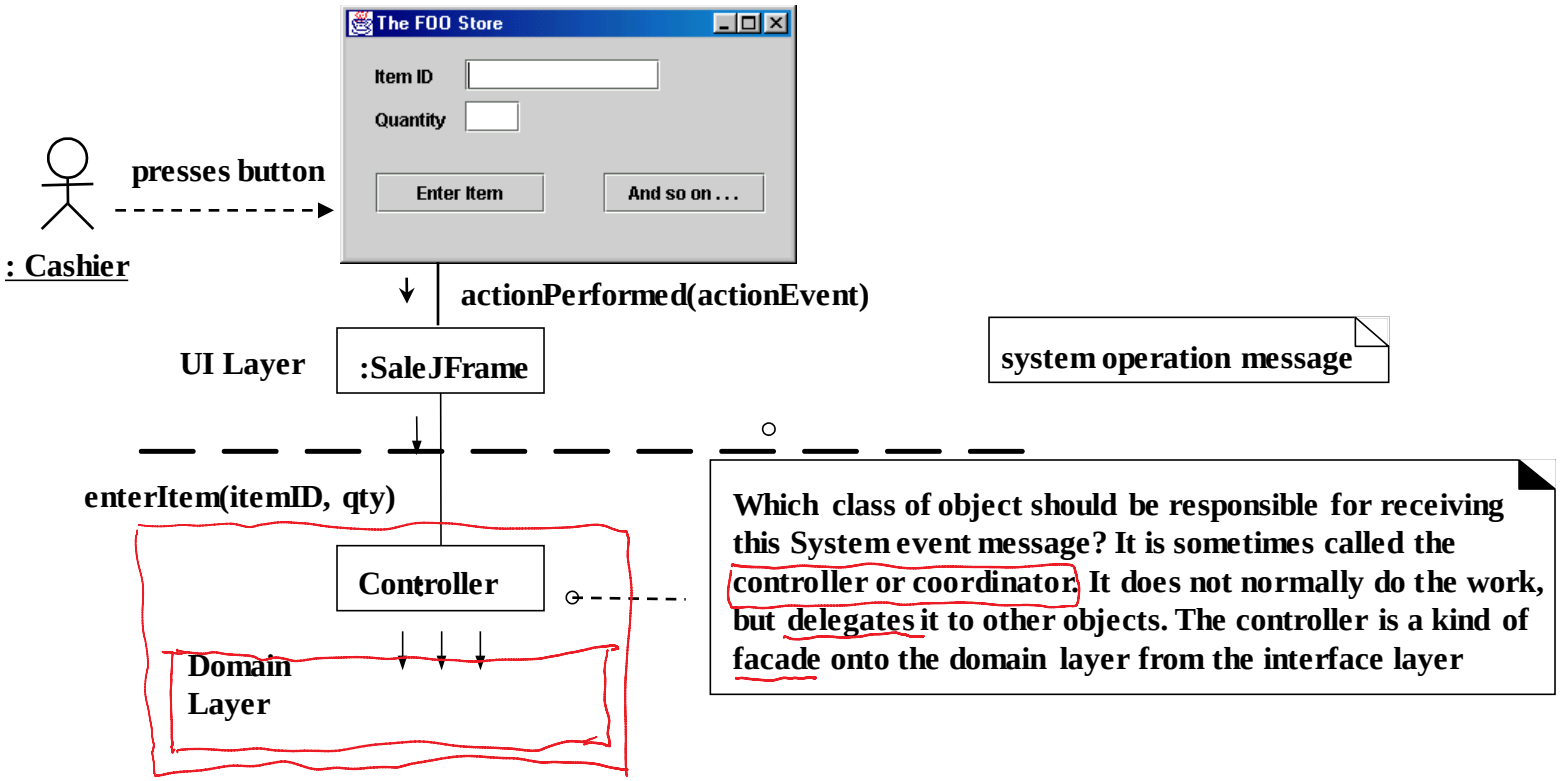
NextGen POS Example

- Some system operations shown.
- This model shows the system itself as a class (which is legal and sometimes useful when modeling).



NextGen POS Example

- During analysis, system operations may be assigned to the class System.
- During design, a controller class is assigned the responsibility for system operations



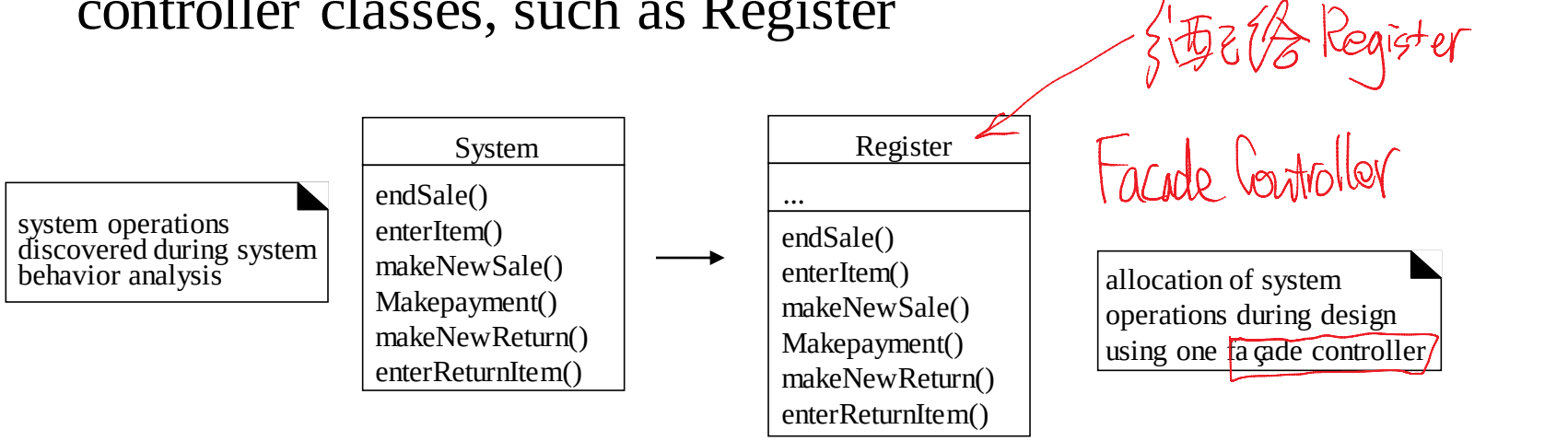
NextGen POS Example

- Who should be the controller for system events such as enterItem and endSale?
- By the Controller pattern
  - Represents the overall "system," "root object," device, or subsystem: **Register, POSSystem**
  - Represents a receiver or handler of all system events of a use case scenario: **ProcessSaleHandler, ProcessSaleSession**
- A Register (POS Terminal) is a specialized device with software running in it.
- The interaction diagrams:

NextGen POS Example

设计 Controller类

- During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such as Register



把它给 Register

Facade Controller

设计多个 Controllers

一个 usecase

一个 usecase

use case Controller

Controller is a Delegation Pattern

- The UI layer shouldn't contain application logic.
- UI layer objects must delegate work requests to domain layer.
- In the domain layer, the Controller pattern summarizes choices, make for the domain object delegate that receives the work requests.
- The controller is ***a kind of façade(门面)*** into the domain layer from the UI layer.

Controller是Domain层的facade，面向UI层.

(Controller可以独立为层：Application层)

Which Controller to Use

- Façade Controller
  - Suitable when there are not too many system events
  - This could be an abstraction of the overall physical unit, such as PhoneSwitch, CashRegister, etc.
- Use Case Controller
  - When using a façade controller leads to low cohesion and high coupling
  - When the code in façade controller gets too big

Bloated Controller 胖控制器

- There is only one and it handles too many events
- The controller performs the work rather than delegating it
- Controller has many attributes and maintains significant system or domain info
- Bloated Controller has low cohesion unfocused and handling too many areas of responsibility

修正膨控制器

Cures for Bloated Controller

- Add more controllers, a system does not have to need only one.
- Instead of facade controllers, employ use case controllers.
  - Use case controllers in an airline reservation system: MakeReservationHandler, ManageSchedulesHandler, ManageFaresHandler
- Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects.

Benefits

- Increased potential for reuse and pluggable interfaces
  - Application logic is not handled/bound in the interface layer, it can be replaced with a different interface.
  - Delegating a system operation responsibility to a controller supports the reuse of the logic in future applications.
- Opportunity to reason about the state of the use case.
  - To ensure that system operations occur in a legal sequence, or we want to be able to reason about the current state of activity and operations within the use case.

GRASP原则5：

High Cohesion

- Cohesion is a measure of how strongly related the responsibilities of an element are
- A class with low cohesion does many unrelated things.
  - It is hard to understand, hard to reuse, hard to maintain, and delicate
  - constantly affected by change
- In POS system, if Register creates the payment, this is less cohesive than if Sale does it

High Cohesion

- A class with high cohesion has a relatively small number of highly-related methods
- It collaborates with other objects
- Modular design: a system has been decomposed into a set of cohesive and loosely coupled modules

High Cohesion Pattern

- **Problem:**
  How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling? 如何使对象内聚,易理解,易管理,从而也低耦合?
- **Solution:**
  - Assign a responsibility so that cohesion remains high.
  - Use this to evaluate alternatives.

POS Example

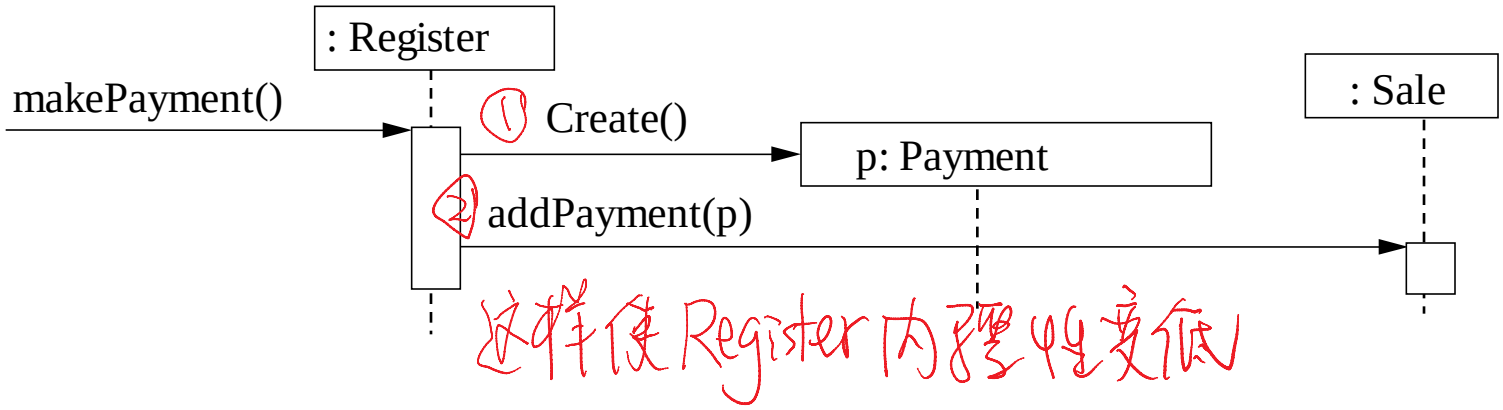
- To create a (cash) Payment instance and associate it with the Sale. What class should be responsible for this?

创建Payment实例，并关联于Sale

POS Example

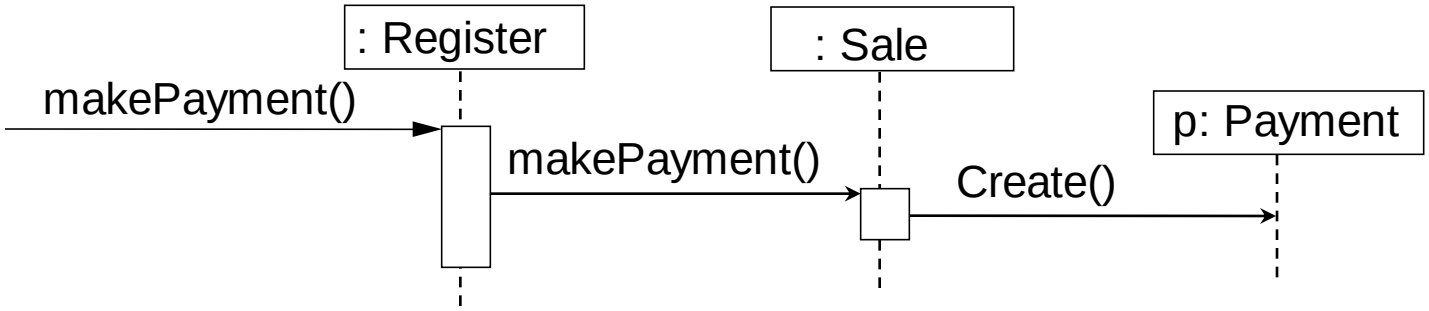
- **Design 1**
  - Since Register records a Payment in the real-world domain, the Creator pattern suggests Register for creating the Payment.
  - The Register instance could send an addPayment message to the Sale, passing along the new Payment as a parameter
  - To places the responsibility for making a payment in the Register.

  这Register (by Creator)
- To continuous make the Register class responsible for doing most of the work related to more system operations
  - it will become increasingly with tasks and become incohesive.



POS Example

- **Design 2:**
  Delegating the payment creation responsibility to the Sale supports higher cohesion in the Register.
  - supports both high cohesion and low coupling.
- In practice, the level of cohesion alone can't be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.



Discussion

- **Very low cohesion**: A class is responsible for many things in very different functional areas.
  - The responsibilities should be split into a family of classes.
- **Low cohesion:** A class has responsibility for a complex task in one functional area.
  - The class should split into a family of lightweight classes.
- **Moderate cohesion:** A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.
- **High cohesion:** A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

Contraindications

Lower cohesion is with distributed server objects.

- Because of overhead and performance implications associated with remote objects and remote communication, it is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations.
- Instead of a remote object with three fine-grained operations setName, setSalary, and setHireDate, there is one remote operation, setData, which receives a set of data. This results in fewer remote calls and better performance.

Benefits

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.