SOFTWARE ENGINEERING INSTITUTE
华东师范大学软件学院

Object oriented Analysis &Design
面向对象分析与设计

Lecture_12 GRASP

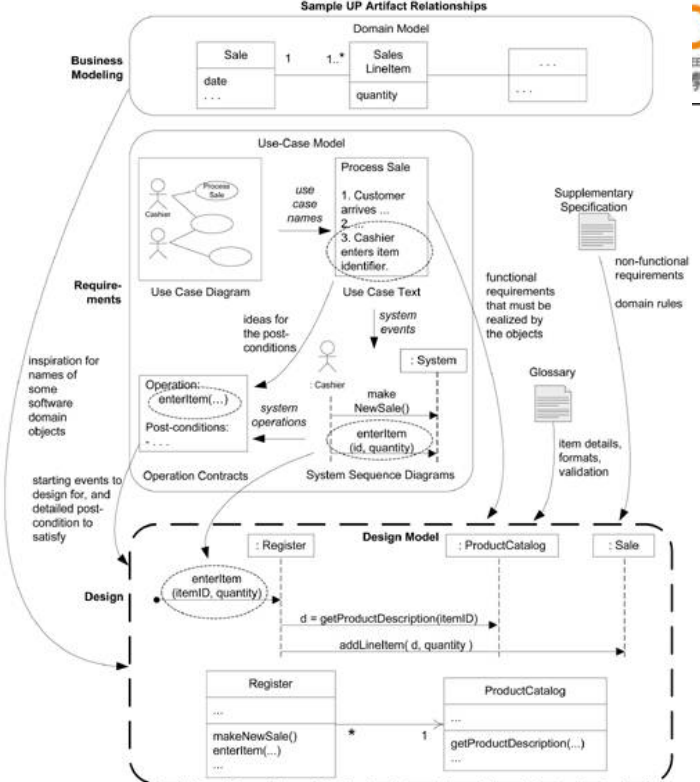
主讲: 陈小红

日期:

掌握内容：

- 熟练使用5种GRASP原则
- 会进行设计结果的评价

Artifact Influences



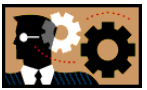
Design

Supplementary Specs
Glossary
Data dictionary
Use Case under development, with System Sequence Diagram (System operations) and Contracts
Conceptual Model

Design

Interactions, class diagram, etc
Packages

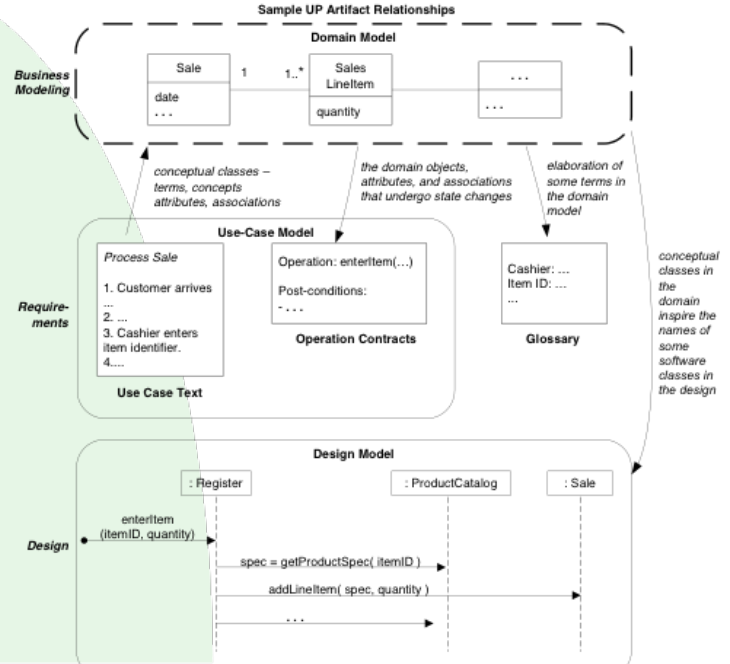
How?



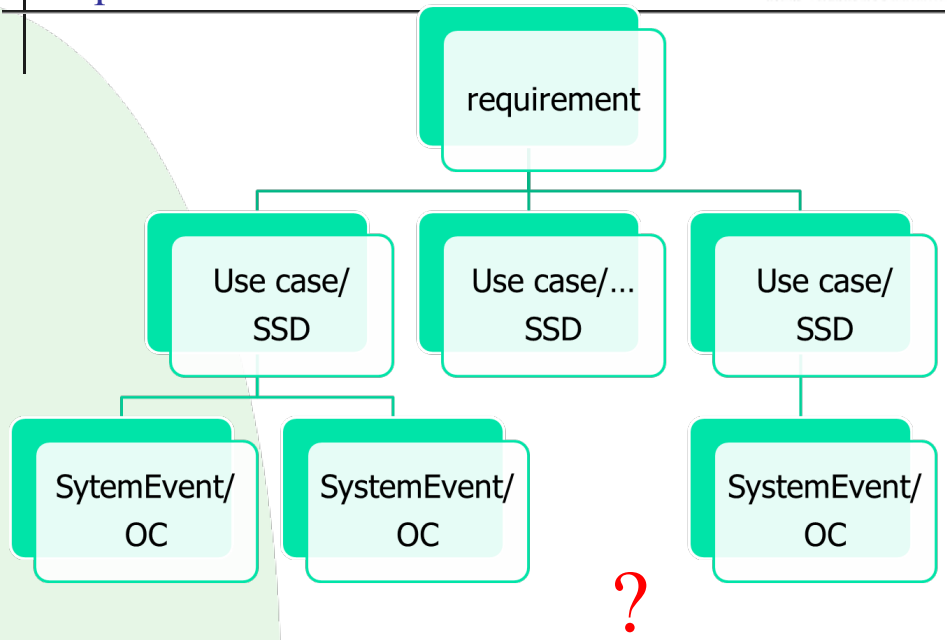
OOD

- After identifying your requirements and creating a domain model,
- then add methods to the appropriate classes, and define the messaging between the objects to fulfill the requirements

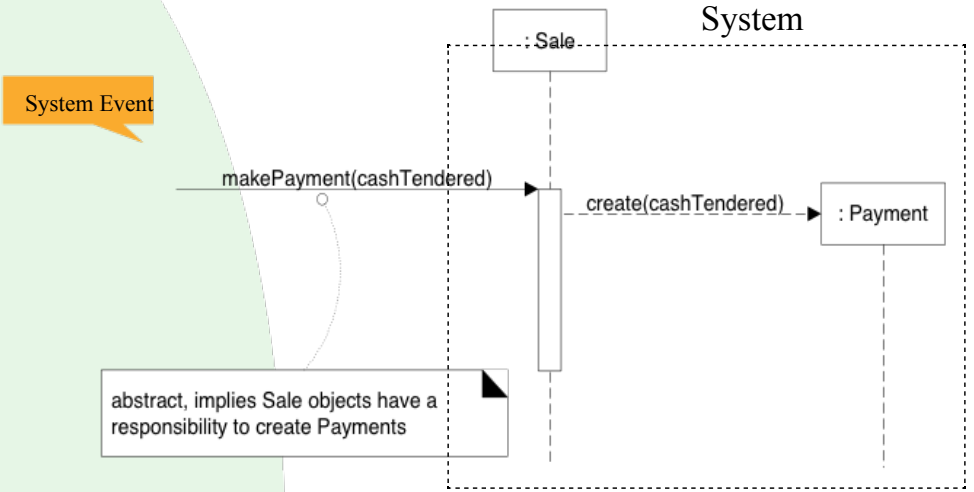
Review-where are we?



Requirement?



- OO System
- - objects collaborating to respond to System Events



- How to judge whether it is finished or not?
  - Operation Contract
  - Input /Output

How to assign?

- In the process

  Refer to domain model

  to assign task to objects and make them collaborating with each other

- How to evaluate whether it is a good design?
  - High cohesion
  - Low coupling

Cohesion，内聚：模块内的操作之间联系紧密的程度

Coupling，耦合：两个子模块之间联系的强度

Back to OOD
Realisation ways

- The problem is
  - Above advice is too general to be handled
  - Deciding what methods belong to where and how objects should interact carries consequences and should be undertaken seriously
- responsibility-driven design (RDD)
  - thinking about how to assign responsibilities to collaborating objects

Ch16. Responsibilities

- Responsibilities are an abstraction
- Software objects do not have responsibilities, they have methods.
  - The responsibility for persistence
    - Large-grained responsibility
  - The responsibility for tax calculation
    - More fine-grained responsibility
- Concretely, implemented with classes, their methods and in terms of how they collaborate with other objects.
- Responsibilities are a useful metaphor

Responsibility-Driven Design (RDD)

- In RDD when designing objects we ask questions like
  - What are the responsibilities of this object?
  - What does it collaborate with?

Basic Principles of Object Design --- GRASP Principles

- What are the guiding principles to help assign responsibilities?
- These principles are captured in the GRASP patterns
  - General Responsibility Assignment Software Patterns
  - Very basic and broad principles of object design

Basic Responsibilities

- These responsibilities are of the following two types: doing and knowing
  - Doing (行为职责)
    - Do it yourself
      - such as creating an object or doing a calculation
    - initiating action in other objects
    - controlling and coordinating activities in other objects
  - Knowing (认知职责)
    - knowing about private encapsulated data
    - knowing about related objects
    - knowing about things it can derive or calculate

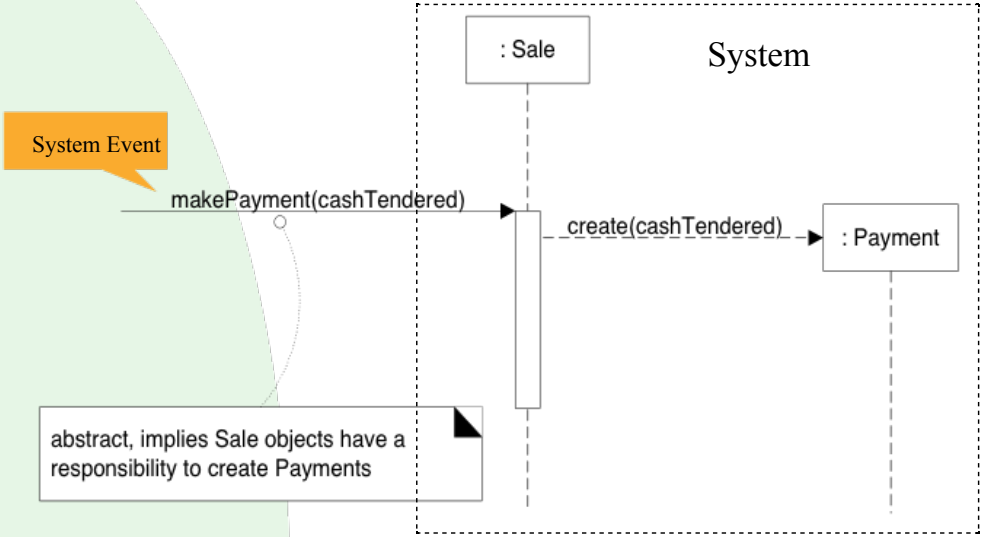
Basic Responsibilities

- Responsibilities are assigned to classes of objects during object design.
  - For example, we may declare that
    - "a Sale is responsible for creating SalesLineItems" (a doing), or
    - "a Sale is responsible for knowing its total" (a knowing).
- Relevant responsibilities related to "knowing" are often inferable from the domain model, because of the attributes and associations it illustrates.
- The translation of responsibilities into classes and methods is influenced by the granularity of the responsibility.
  - E.g. The responsibility
    - "to provide access to relational databases"
  - may involve dozens of classes and hundreds of methods, packaged in a subsystem.
  - By contrast, the responsibility "to create a Sale" may involve only one or a few methods.

Methods Implement Responsibilities

- OO System
  - objects collaborating to respond to System Events
- Responsibilities are often considered during the creation of interaction diagrams.
  - Next Figure indicates that Sale objects have been given a responsibility to create Payments, which is invoked with a makePayment message and handled with a corresponding makePayment method

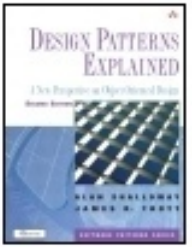
Methods Implement Responsibilities



Ch17. GRASP – Designing Objects with Responsibilities

SOFTWARE ENGINEERING INSTITUTE
華東師范大學软件学院

- **How?**
  - **Apply Patterns and principles of responsibility assignments**
    - **GoF (Gang of Four patterns)**
      - **Famous 23 patterns**
    - **GRASP (General responsibility assignment Software pattern)**
      - **A set of (9) principles**
    - **Others**
      - **OCP, …**



Design Patterns Explained A New Perspective on Object-Oriented Design Second Edition

By Alan Shalloway, James R. Trott

Patterns

•Experienced object-oriented developers (and other software developers) build up a repertoire ['repətwɑ:] of both general principles and idiomatic solutions that guide them in the creation of software.

• These principles and idioms, if codified in a structured format describing the problem and solution, and given a name, may be called patterns. For example, here is a sample pattern:

Pattern Name: Information Expert
Problem It Solves: What is a basic principle by which to assign responsibilities to objects? Solution:
Assign a responsibility to the class that has the information needed to fulfill it.

Patterns

•In object technology, a pattern is a named description of a problem and solution that can be applied to new contexts;

• ideally, it provides advice in how to apply it circumstances, and considers the forces and trade- patterns provide guidance for how responsibilities assigned to objects, given a specific category of pro

•Most simply, a pattern is a named problem/soluti can be applied in new contexts, with advice on how in novel situations and discussion of its trade-off's.

Alexander, C., The Timeless Way of Building建筑的永恒之道，New York: Oxford University Press, 1979

It is important to realize that the Gang of Four did not CREATE the patterns described in the book. Rather, they IDENTIFIED patterns that already existed within the software community, patterns that reflected what had been learned about high-quality designs for specific problems

GRASP Patterns

- **GRASP**
  - General Responsibility Assignment Software Patterns
  - **Understanding and being able to apply the ideas behind GRASP while coding or while drawing interaction and class diagrams enables developers new to object technology needs to master these basic principles as quickly as possible; they form a foundation for designing OO systems**(对象技术初学者在编码或绘制交互图和类图时，应该理解并应用GRASP的基本思想，以便尽快地掌握这些基本原则，它们是设计OO系统的基础)
- The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable ways.

GRASP Patterns

GRASP patterns describe fundamental principles of object design and responsibility assignment. The following is a list of GRASP patterns:

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

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Mini Exercise 1

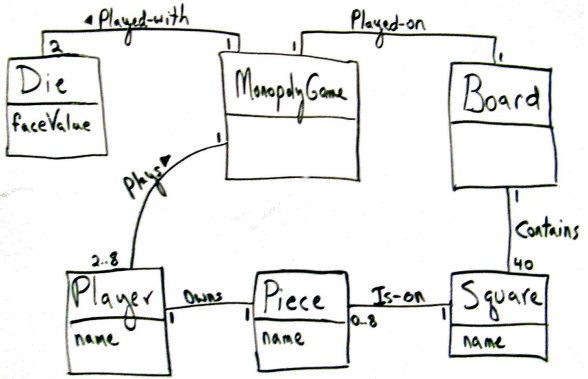


Figure 17.3. Monopoly iteration-1 domain model

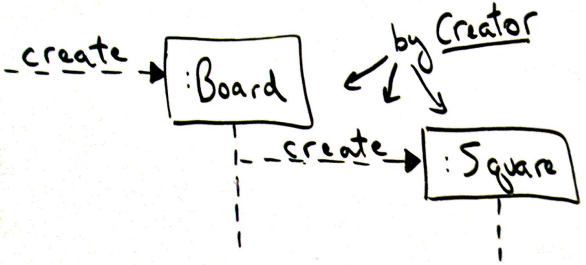
- Who will create Square object in Monopoly ? Why?
- Thinking steps
  - Have no design model, so start with Domain Model
  - LRG: Low representational gap — build design model (dynamic and static)

GRASP rule1: Creator (创建者)

- Name: Creator
- Problem:
  - Who should ... ng a new instance of some class?
- Solution:
  - Assign class ... eate an instance of class A if one of these ... *etter)* :
    1. B "contai ... ates A
    2. B records ...
    3. B closely ...
    4. B has the ... will be passed to A when it is created (B is ...
  - 如果有一个以 ... 选 聚集 或包含A的类
- Note:
  - B and A refer to software objects, not domain model objects

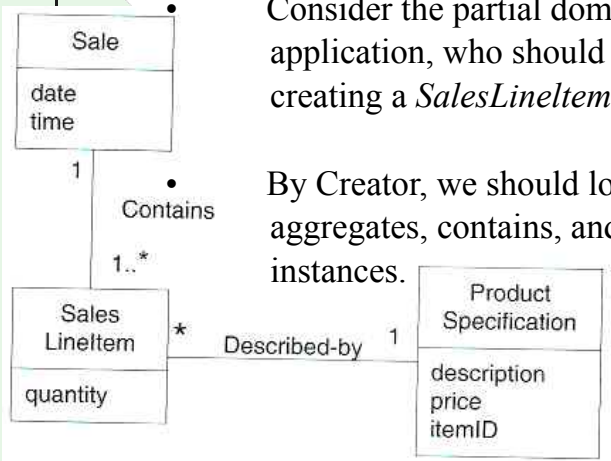
Creator : example Monopoly

- **Board create Square**
  - **Because of 'Board' contains 'Square'**



Who create 'Board'?

Creator : POS example



- Consider the partial domain model : In the POS application, who should be responsible for creating a *SalesLineItem* instance?

- By Creator, we should look for a class that aggregates, contains, and so on, *SalesLineItem* instances.

Creator : POS example

•Since a *Sale* contains (in fact, aggregates) many *SalesLineItem* objects, the Creator pattern suggests that *Sale* is a good candidate to have the responsibility of creating *SalesLineItem* instances.

•This leads to a design of object interactions as shown :

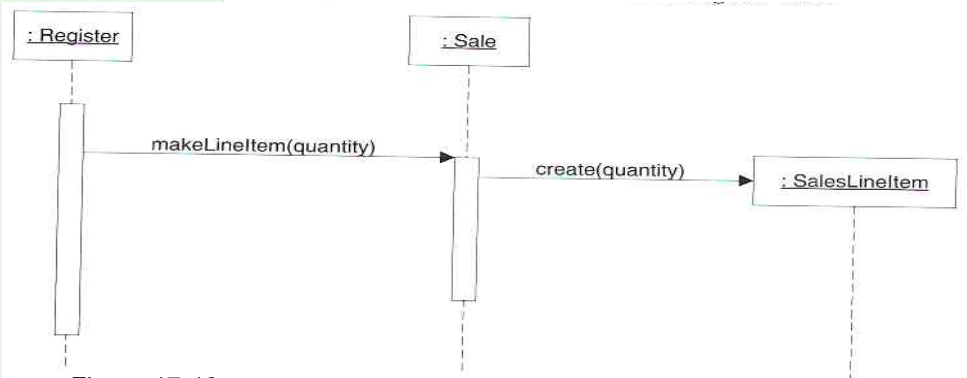


Figure 17-13

Creator — When Not to Use

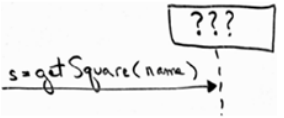
- When want to reuse existing instances for performance purposes (caching)
- conditionally creating an instance from one of a family of similar classes based upon some external property value
- GOF pattern
- Delegate responsibility further down
- Other more complex situations

Creator — Benefits

- Existing associations means created class is in any case visible to creator
- High cohesion
- Does not increase coupling

Mini Exercise 2

- Given a key, which object can tell me about
  - …Square in Monopoly



  - …Total money of a sale in POS

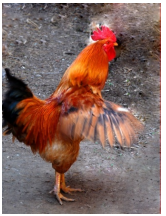
- Note:
  - this is a knowing responsibility

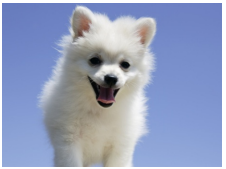
GRASP rule2: Information Expert

- Name: Information Expert(信息专家)
- Problem:
  - What is a general principle of assigning responsibility to objects?
- Solution:
  - Assign responsibility to the class that has the information necessary to fulfill responsibility

Information Expert — ”Real World” Analogy

- “Do It Myself” [Coad] or “Animation” [Larman] principle
  - objects are “alive” and can do things themselves
- **“鸡鸣狗盗”**
- Ultimate knowledge may require co-operation





Information Expert

- Information
  - an object's own state,
  - about other objects, the world around an object,
  - information the object can derive,
  - and so forth
- Answer to Mini exercise 2

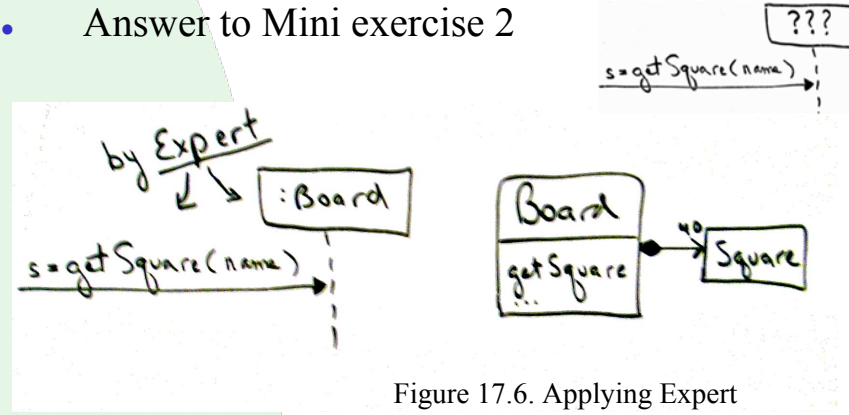


Figure 17.6. Applying Expert

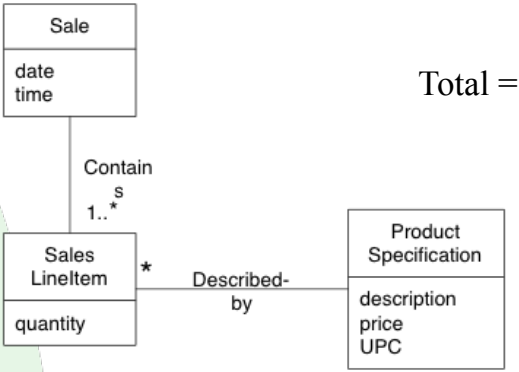
Information Expert — How to?

- 1. Clearly state the responsibility
- 2. Look in Design Model for relevant classes
- 3. Else look in Domain Model and create design classes

Information Expert — Sale Total Example

- Who should know the grand total (总计) of a sale?

Tell me the total money?



$$\text{Total} = \sum \text{price}_i * \text{qty}_i$$

- Product has information about Price so it is expert for that (getPrice method)
- SalesLineItems has information about Product and Quantity so it is expert for PriceTotal (getPriceTotal method)
- Sale has information SalesLineItems with associated PriceTotal so it is expert for SaleTotal (getSaleTotal method)

Information Expert — Sale Total

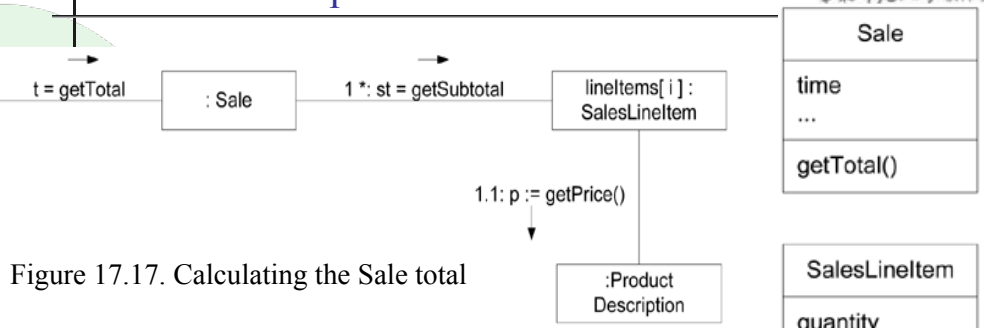


Figure 17.17. Calculating the Sale total

getSubTotal == getPriceTotal

getTotal == getSaleTotal

Information Expert — Sale Total

Hence responsibilities assign to the 3 classes.

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

Partial Information Expert

- Fulfilling responsibility may require information spread across different classes
  - Partial information experts collaborate in task
    - — getSaleTotal example
- Interact through messages (methods) to share the work

Information Expert — Benefits

- Encapsulation
  - objects support their own information
  - supports low coupling
- Behavior is distributed across classes
  - supports high cohesion

Contraindications:

Database?

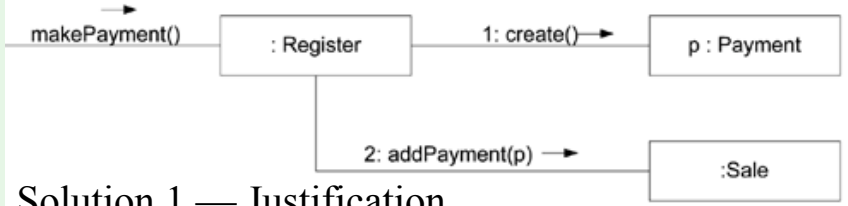
Mini-Exercise 3

- Who creates the Payment?
- makePayment method invoked in Register
- A Payment has to be associated with the Sale
- Who creates the Payment instance?
- By creator □ Register, Sale are candidates

Mini-Exercise 3

- Solution 1

Figure 17.18. Register creates Payment.

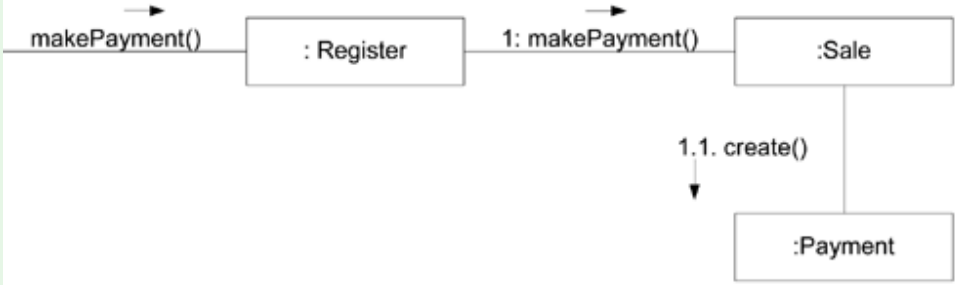


- Solution 1 — Justification
  - Register records Payment
    - Creator pattern
  - addPayment method passes p instance of Payment as parameter
  - Register coupled to Payment class

Mini-Exercise 3

- Solution 2

Figure 17.19. Sale creates Payment.



GRASP rule3： Low Coupling (!)

- Name: Low Coupling
- Problem:
  - How to support low dependency, low change impact and increased reuse?
- Solution:
  - Assign responsibility so coupling remains low.
    - Use this principle to evaluate alternatives
    - All other things being equal prefer the low coupling solution
- Note: Information Expert encourages Low Coupling
- Why most secret service (spy) is one-way contact? 为什么特务工作都是单线联系?

Relation with the other patterns

•It can't be considered in isolation from other patterns such as Expert and High Cohesion, but rather needs to be included as one of several design principles that influence a choice in assigning a responsibility.

Coupling

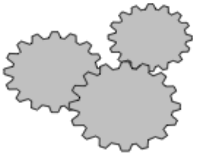
- Measure of how strongly one element is:
  - 1. connected to
  - 2. has knowledge of
  - 3. relies on

another element

(耦合：一个元素与其它元素的联接、感知以及依赖的程度的度量)

What is coupling?

- (For Example, in Java),
- common forms of coupling from X to Y include:
- X has an attribute that refers to Y.
- X calls on services of Y.
- X has a method that references Y (parameter, local variable, or return value).
- X is a direct or indirect subclass of Y.
- Y is an interface, and X implements that interface.



Coupling

- Problems with High Coupling
  - Forced local changes because of changes in related class
    - If there is coupling or dependency, then when the depended-upon element changes, the dependant may be affected.
  - Harder to understand in isolation
  - Harder to reuse — drags in more classes

High Coupling

- A subclass is strongly coupled to its superclass.
  - The decision to derive from a superclass needs to be carefully considered since it is such a strong form of coupling.

Importance

Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider.

It is an evaluative principle that a designer applies while evaluating all design decisions.

How to Evaluate Low Coupling

- There is no absolute measure of when coupling is too high.

- What is important is that a developer can gauge(评估) the current degree of coupling, and assess if increasing it will lead to problems.

- In general, classes that are inherently very generic in nature, and with a high probability for reuse, should have especially low coupling.

Low Coupling-To What Degree

- The extreme case of Low Coupling is when there is no coupling between classes.
  - This is not desirable
    - because a central metaphor of object technology is a system of connected objects that communicate via messages
    - Too little coupling means we aren't a "collaborating community of objects"
  - it yields a poor design
    - because it leads to a few incohesive, bloated, and complex active objects that do all the work, with many very passive zero-coupled objects that act as simple data repositories.
- Some moderate degree of coupling between classes is normal and necessary
  - to create an object-oriented system in which tasks are fulfilled by a collaboration between connected objects

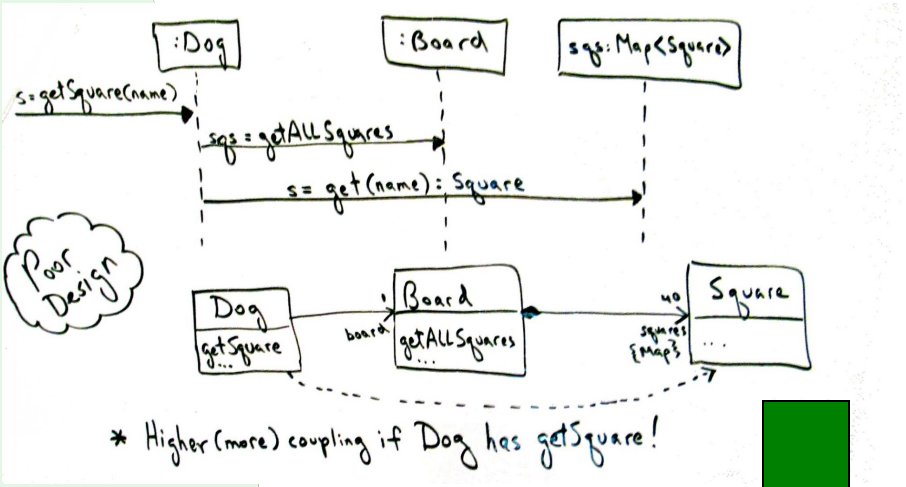
Low Coupling — When Not To

- High coupling to stable elements and to pervasive elements is seldom a problem.
  - e.g. language libraries
  - For example, a Java J2EE application can safely couple itself to the Java libraries (java.util, and so on)
  - because they are stable and widespread.

- Low Coupling Benefits
  - Not affected by changes in other components
  - Simple to understand in isolation
  - Convenient to reuse

Example : Low Coupling

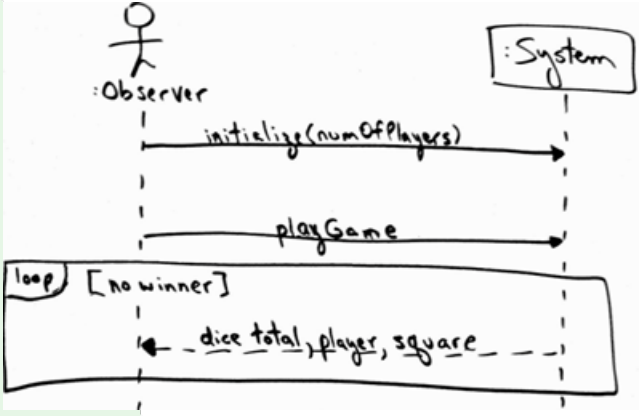
- **Why not assign getSquare to Dog (i.e., some arbitrary other class)?**
- **Minimal coupling will reduce change (dependency)**



Mini Exercise 4

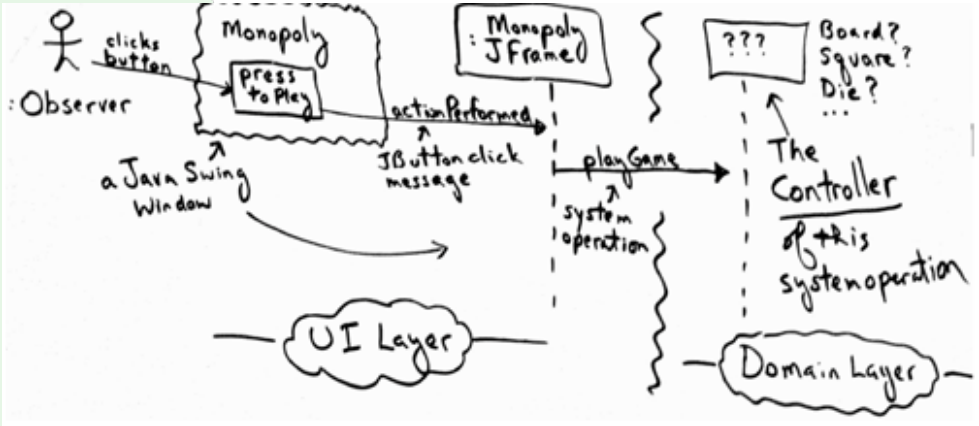
- For Monopoly game, Which object starts the game?
- Understanding the Problem
  - SSD — boundary between the User and SUD (system under development)
  - UI layer “catches” the request
  - The request is a system operation — public interface
  - Model-View Separation principle says UI must not contain business logic
  - Problem: to which domain layer object should the UI pass the system operation ?

- Figure 17.8. SSD for the Monopoly game.
- Note the playGame operation



Play game

- **Assign responsibility to receive the system operations by an object which**
  - **Representing System/subsystem/device or**
  - **Handling just this use case**

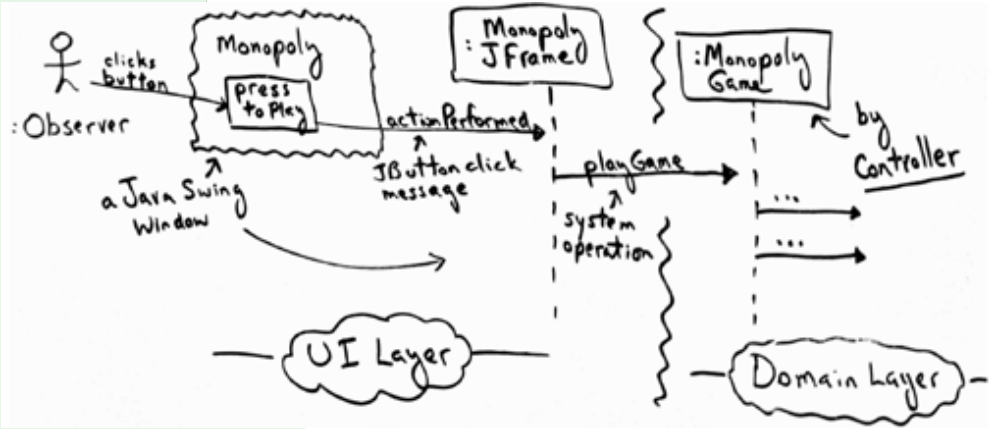


GRASP rule4: Controller

- Name: Controller
- Problem:
  - What first object beyond the UI layer receives and co-ordinates (controls) a system operation
- Solution:
  - Assign the responsibility to a class representing one of the following choices:
  - 1. Facade(外观) Controller :
    - represents the overall system, a root object, a device that the object is running within, or a major sub-system
  - 2. Use Case or Session Controller :
    - represents a use case scenario within which the system event occurs

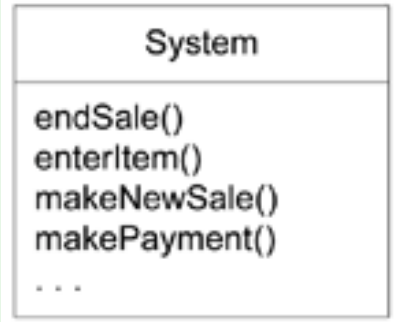
Mini Exercise 4 — Solution

- The use case that the playGame system operation occurs within is called Monopoly Game.
- Thus, a software class such as PlayMonopolyGameHandler



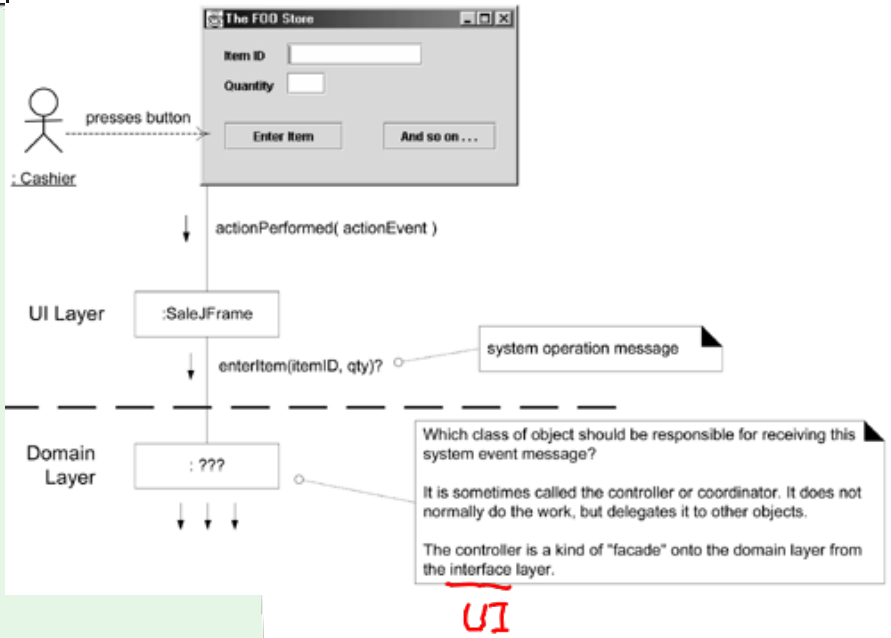
Controller — NextGen POS Example

- Figure 17.20. Some system operations of the NextGen POS application



Conceptual representation — what is the class that handles these operations?

Figure 17.21. What object should be the Controller for enterItem?



Controller Example for POS

- (Solution) options
  - Facade: Register, POSSystem
  - Session: ProcessSaleHandler, ProcessSaleSession

Figure 17.22. Controller choices

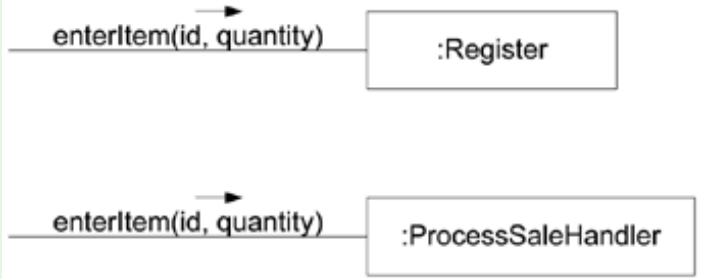
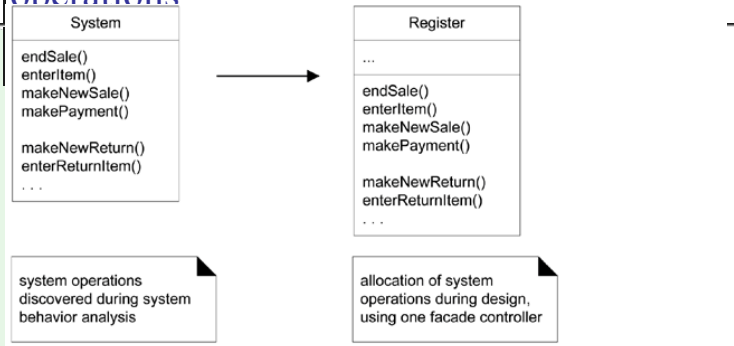


Figure 17.23. Allocation of system operations



Controller — Observations

- Delegation pattern
- External input events may come from human actors or other systems
  - Facade — the "face" of the domain layer to the world
    - Ex: Register
  - Handler — deals with a clear cut part of the issue事物一个明确的组成部分
    - Ex: ProcessSale

Note : Session — conversation where information needs to be retained between steps

Controller — Facade

- Facade外观模式：
  - Provide a consistent interface for a set of interfaces in subsystem为子系统中的一组接口提供一个一致的界面
  - “cover” over the other layers of the application
  - Abstraction of an overall physical unit
    - — Register, PizzaShop
  - The entire software system
    - — POSSystem
  - Abstraction of some other overall system or sub-system concept
    - — MonopolyGame
- Suitable for
  - relatively small systems
  - and/or system with limited number of system operations
  - in message handling system when can’t direct messages to alternative controllers
    - — Internet application servers

Session Controller — Use Case

- Pure Fabrication
  - — ProcessSaleHandler is not a domain concept in Domain Model
- When assigning to facade may lead to high coupling or low cohesion (“Bloat”)
- Many system operations across different processes
- Conceptually easier to understand and build
- Session Controllers Naming conventions:
  - <UseCaseName> Handler or
  - <UseCaseName> CoOrdinator or
  - <UseCaseName>Session
  - Use same controller class for all system operations in the use case scenario
  - Session is a type of conversation between the actor and the SUD

Controller vs. UI

- UI should not have responsibility for fulfilling system operations
- System operations handled in domain layer
- 本书把 应用逻辑层 又称为 架构的 领域层（ref ch13.6)
- Controller is responsible for forwarding messages.

Controller — Benefits

- Allows for easy change of UI and/or alternative UI
- Allows for reuse of domain layer code (UI is usually application specific)
- Helps ensure sequence of operation which may differ from UI input
- Can reason about system state — UI does not preserve state

Bloated(臃肿的) Controllers — The Problem

- Low cohesion — unfocused and handling too many areas of responsibility:
  - When have a facade controller handling all of many system events
  - When the controller performs many of the system operations instead of delegating
  - When the controller has many attributes (much information) about the system
    - which should be distributed to or duplicates from elsewhere

Bloated Controllers — The Solution

- Add more controllers
  - — "session controller" instead of facade
- Design the controller so that it delegates operations to other objects
- High Cohesion is itself a GRASP principle

GRASP rule 5：High Cohesion

- Name：High Cohesion
- Problem:
  - How to keep objects focused, understandable and manageable, and as a side effect support Low Coupling?
- Solution:
  - Assign responsibility so cohesion remains high.
- Dosage(用法):
  - Used as an evaluation tool

Cohesion Defined

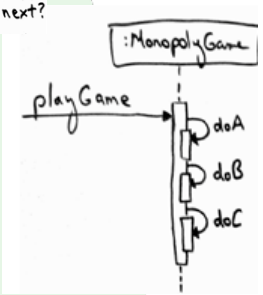
- measure of how strongly related and focused the responsibilities of an object are
  - Cohesion,内聚：程序中的操作之间联系紧密的程度
  - Cf: Coupling，耦合：两个子模块之间联系的强度
- Goal:
  - highly related responsibilities
  - small number of responsibilities
- Judge：?
  - An object with 100 methods in 2000 SLOC
    SLOC: 代码行数source lines of code
  - Other object with 10 method in 200 SLOC
  - Are you sure the big object can complete task without reference to outside object? (which will increase coupling)

Cohesion Defined-Reason

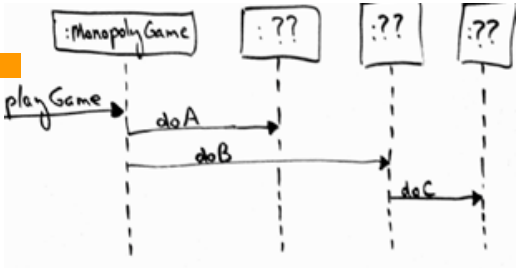
- An element has high cohesion
  - with highly related responsibilities,
  - and which does not do a tremendous amount of work
- A class with low cohesion
  - does many unrelated things,
  - or does too much work.
  - Such classes are undesirable.
- Low cohesion classes often represent
  - a very large grain of abstraction,
  - or have taken on responsibilities that should have been delegated to other objects.

High Cohesion: Monopoly example

- Based on the Controller decision, we are now at the design point
  - two contrasting design approaches worth considering



Poor (Low) Cohesion in the MonopolyGame object



Better

Low Cohesion — The Problem

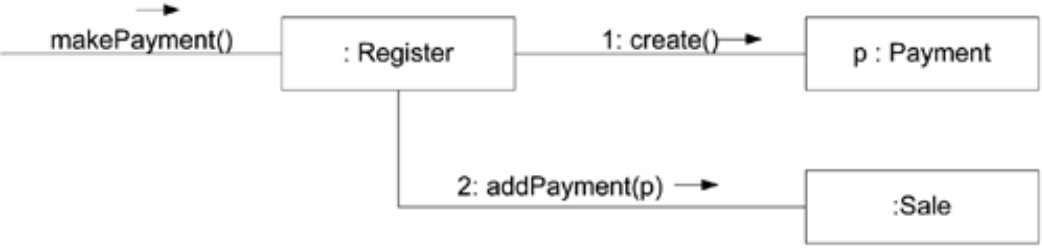
- Hard to understand
- Hard to rescue
- Hard to maintain
- Subject to constant need to change (usually high coupling)

High Cohesion — NextGen POS Example

- The same example problem used in the Low Coupling pattern can be analyzed for High Cohesion. Assume we have a need to create a (cash) Payment instance and associate it with the Sale.
- What class should be responsible for this?

- Since 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

High Cohesion — NextGen POS Example



This assignment of responsibilities places the responsibility for making a payment in the *Register.* The *Register* is taking on part of the responsibility for fulfilling the *makePayment* system operation.

In this isolated example, this is acceptable;
but if we continue to make the *Register* class responsible for doing some or most of the work related to more and more system operations, it will become increasingly burdened with tasks and become incohesive.

High Cohesion — NextGen POS Example

Imagine that there were fifty system operations, all received by *Register*. If it did the work related to each, it would become a "bloated" incohesive object.

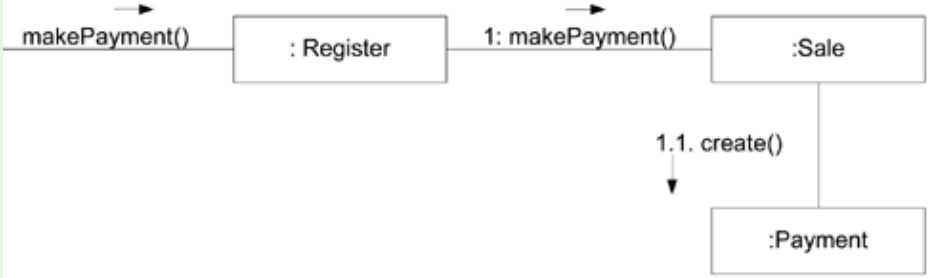
The point is not that this single *Payment* creation task in itself makes the *Register* incohesive, but as part of a larger picture of overall responsibility assignment, it may suggest a trend toward low cohesion.

And most important in terms of developing skills as an object designer, regardless of the final design choice, the valuable thing is that at least a developer knows to consider the impact on cohesion.

High Cohesion — NextGen POS Example

•By contrast, as shown in next Figure , the second design delegates the payment creation responsibility to the *Sale,* which supports higher cohesion in the *Register.*

•Since the second design supports both high cohesion and low coupling, it is desirable.



Delegating to Sale creates greater cohesion in Register

Things to remember about high cohesion

- 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.

- Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider.
- **It is an evaluative principle that a designer applies while evaluating all design decisions.**

High Cohesion — When Not to

- One case is the grouping of responsibilities or code into one class or component to simplify maintenance by one person.
  - For example, suppose an application contains embedded SQL statements that by other good design principles should be distributed across ten classes, such as ten "database mapper" classes.
  - Now, it is common that only one or two SQL experts know how to best define and maintain this SQL. The software architect may decide to group all the SQL statements into one class, so that it is easy for the SQL expert to work on the SQL in one location.
- Another case for components with lower cohesion is with distributed server objects. (Performance issues demand grouping a set of operations together)
  - 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.

High Cohesion-benefits

- As a rule of thumb, a class with high cohesion:
- has a relatively small number of methods, with highly related functionality,
- and does not do too much work.
- It collaborates with other objects to share the effort if the task is large.

- A class with high cohesion is advantageous because it is relatively easy to maintain, understand, and reuse.
- The high degree of related functionality, combined with a small number of operations, also simplifies maintenance and enhancements.

High Cohesion a real-world analogy

- The High Cohesion pattern has a real-world analogy.
- It is a common observation that if a person takes on too many unrelated responsibilities-especially ones that should properly be delegated to others-then the person is not effective.
- This is observed in some managers who have not learned how to delegate. These people suffer from low cohesion; they are ready to become "unglued".

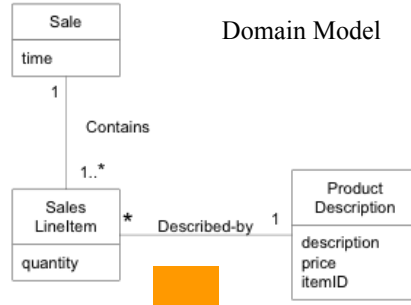
- Coupling and cohesion are truly fundamental principles in design, and should be appreciated and applied as such by all software developers.

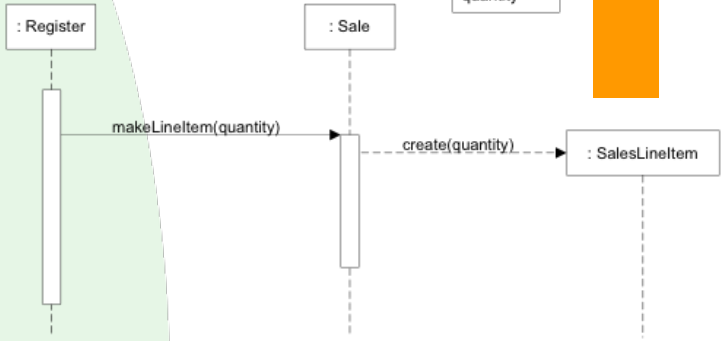
- GRASP Review / Summary

POS: Creator Example

In the NextGen POS application, who should be responsible for creating a **SalesLineItem** instance?
-- who aggregates, contains, and so on, SalesLineItem ?

Domain Model

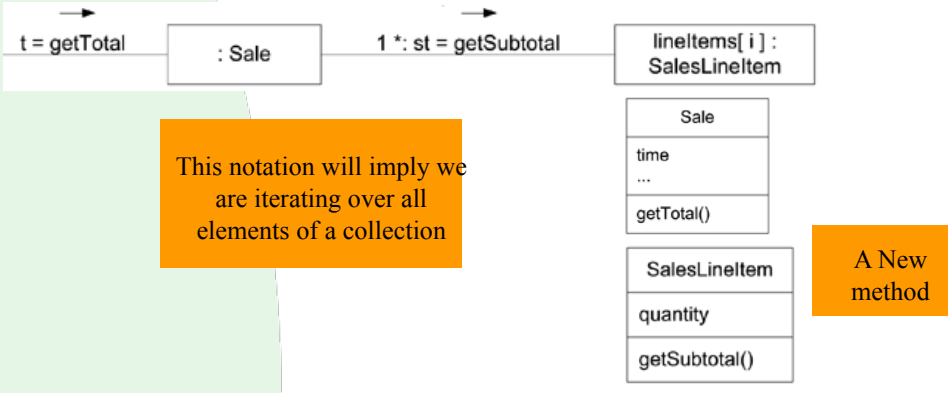




- NextGEN POS application, some class needs to know the grand total of a sale.
- First Step: Start assigning responsibilities by clearly stating the responsibility
  - Who should be responsible for knowing the grand total of a sale?
- Second Step: Where to find the class
  - Design Model first
  - Domain Model Second
- Sale knows
  - All the SalesLineItem instances of a sale and the sum of their subtotals

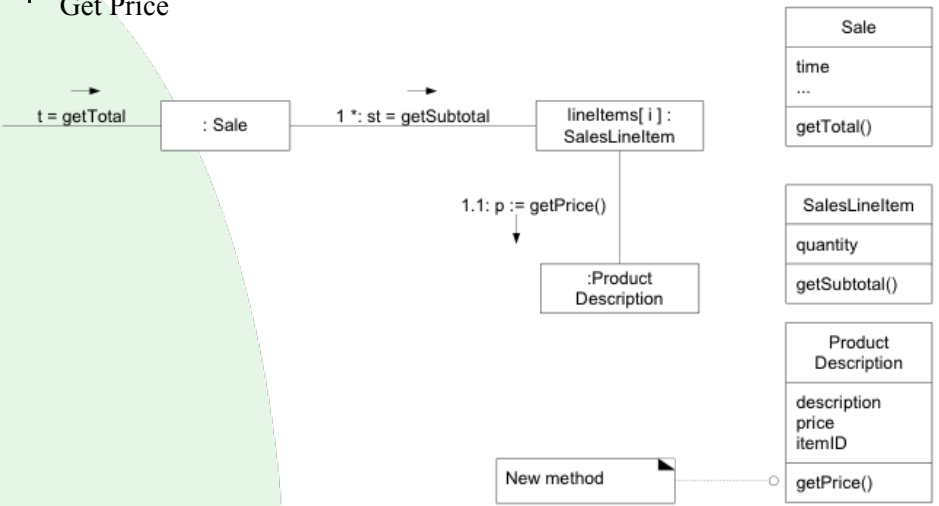
POS: Expert Example

How to determine the line item subtotal?



POS: Expert Example cont.

Get Price

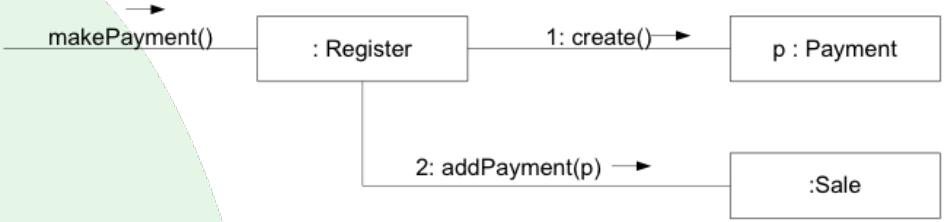


POS: Low Coupling Example bad and good

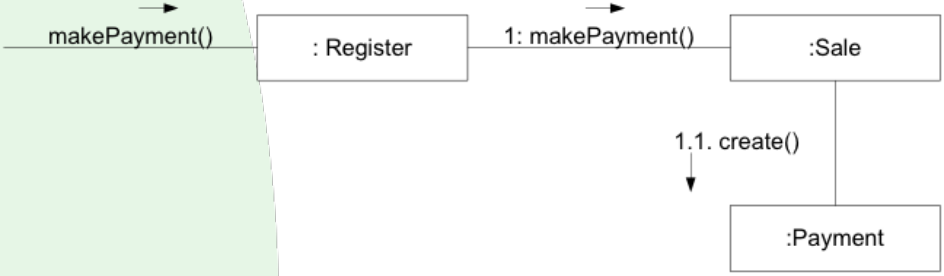
- Assume we need to create a Payment instance and associate it with the Sale. What class should be responsible for this?

POS: Low Coupling Example bad and good

the Creator pattern suggests Register as a candidate for creating the Payment

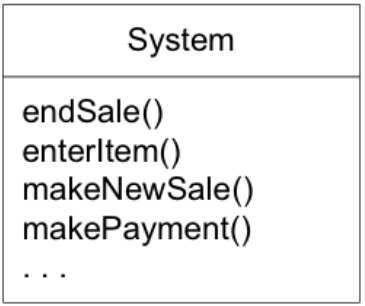


supports Low Coupling



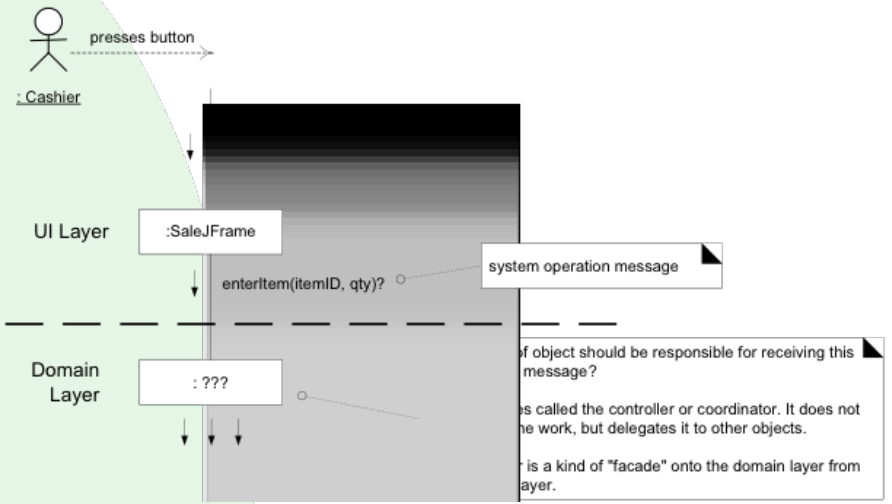
POS: Controller Example

- NextGen application contains several system operations



- during design, a controller class is assigned the responsibility for system operations
-

POS: Controller Example

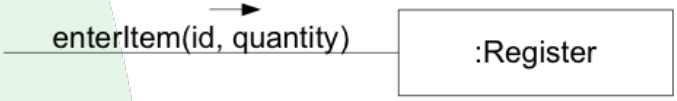


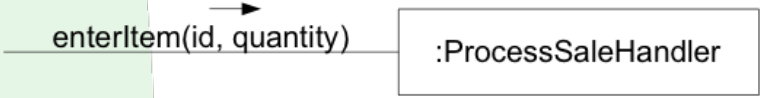
POS: Controller Example - choices

Who should be the controller for system events such as enterItem and endSale?

some choices

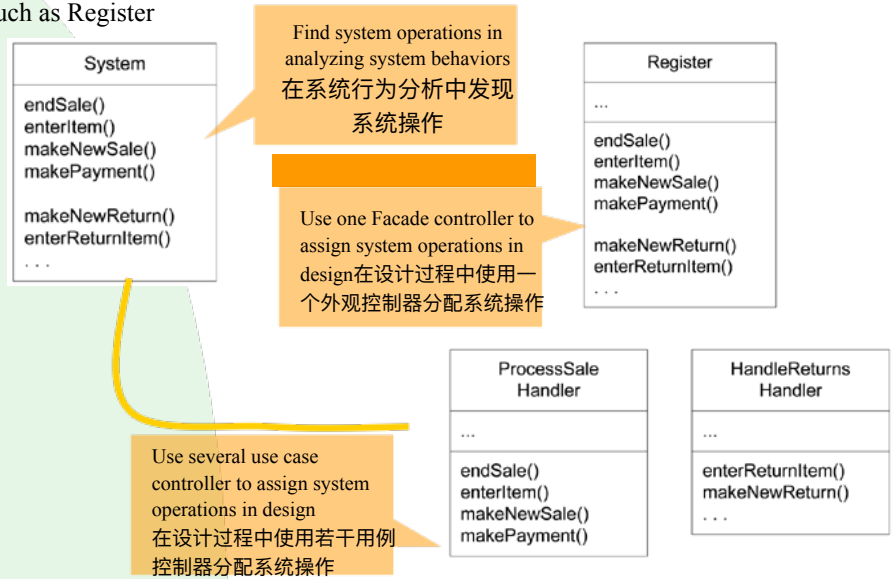
Represents the overall "system," "root object," device, or subsystem.	Register, POSSystem
Represents a receiver or handler of all system events of a use case scenario	ProcessSaleHandler, ProcessSaleSession



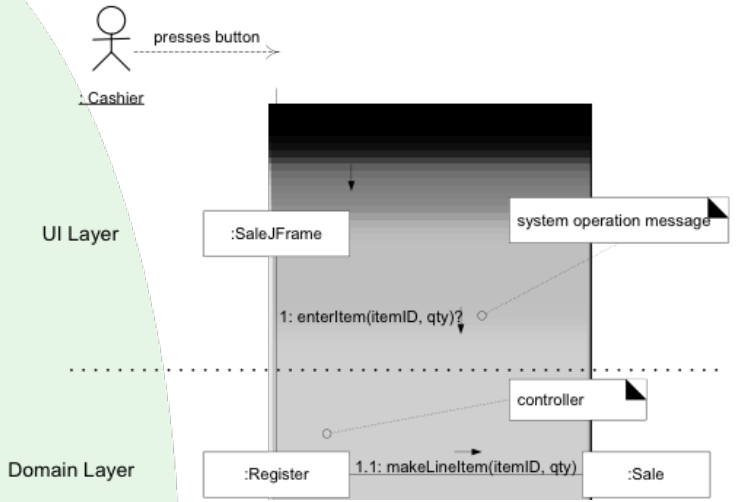


POS: Controller Example - choices

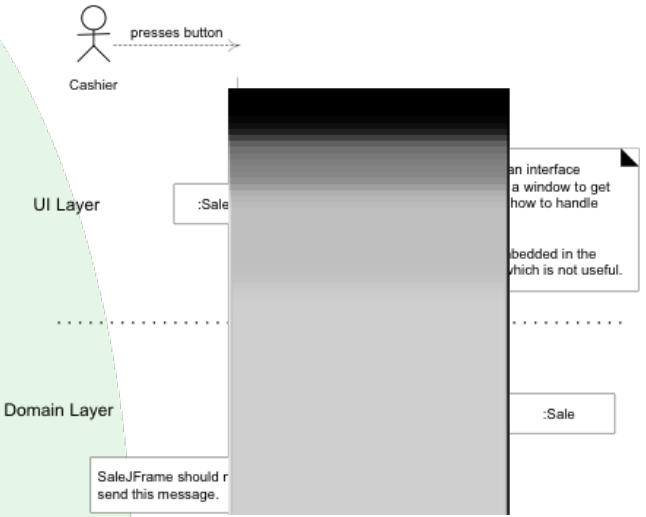
During design, the system operations are assigned to one or more controller classes, such as Register



POS: Controller Example – final choice (conformity to physical entity)



POS: UI Thin Layer should not handle System Operations (bad example)

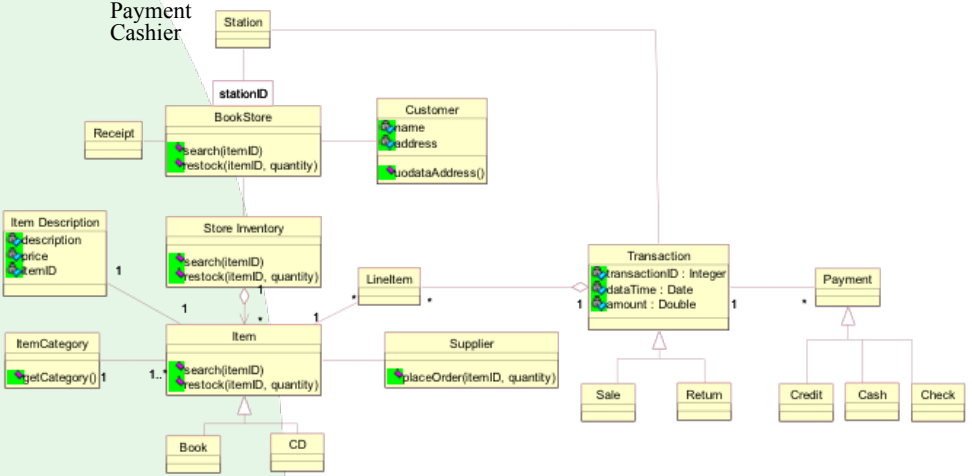


Exercise

Which class should be responsible for creating the Transaction class?

Select any 1 option:

- LineItem
- Customer
- Station
- Payment
- Cashier

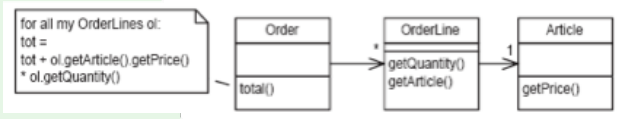


Exercise

A well designed Object Oriented System is : (Single select)

A. Is loosely cohesive and loosely coupled.
B. Is tightly cohesive and loosely coupled.
C. Is tightly cohesive and tightly coupled.
D. Is loosely cohesive and tightly coupled.

Consider the following design.
How would the introduction of a subtotal() method in OrderLine improve the design?
(a) It enhances the coherence of Article.
(b) It reduces the coupling of Order.
(c) It reduces the coherence of Order.
(d) It enhances the coupling of Article.



- Homework
  - Review : Read Text book Ch17~18
  - Preparation: Read Text book Ch21 “refactoring”
- End