面向对象程序的分析与设计
Object-Oriented Analysis and Design

Lecture 8

OOD

Design Pattern (I)

Prof. S. Xu

Contents

- Review: Domain Model
- GRASP

Review: Domain Model

Process of Domain Modelling

1. Based on the use case description, Using noun-extraction method to find the candidates for domain concepts
2. remove those duplicates
3. find those terms which are too abstract (which might be the attributes for other concepts
4. remove those terms which are out side of system boundary
5. find those terms which are too simple (which could be the attributes of other concepts.
6. draw the domain diagram and find the relationships between two of concepts
7. Draw the multiplicities of association relationship

Noun Extraction: A Library Example

The library contains books and journals. It may have several copies of a given book. Some of the books are reserved for short-term loans only. All others may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

The system must keep track of when books and journals are borrowed and returned and enforce the rules.

Noun Extraction: A Library Example

The library contains books and journals. It may have several copies of a given book. Some of the books are reserved for short-term loans only. All others may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

The system must keep track of when books and journals are borrowed and returned and enforce the rules.

Candidate Classes -

Library	the name of the system
Book	
Journal	
Copy	
ShortTermLoan	event
LibraryMember	
Week	measure
MemberOfLibrary	repeat
Item	book or journal
Time	abstract term
MemberOfStaff	
System	general term
Rule	general term

Relations between Classes

Book	is an	Item
Journal	is an	Item
Copy	is a	copy of a Book
LibraryMember		
Item		
MemberOfStaff	is a	LibraryMember

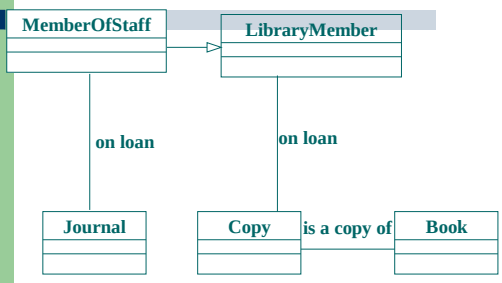
Is Item needed?

Relationships

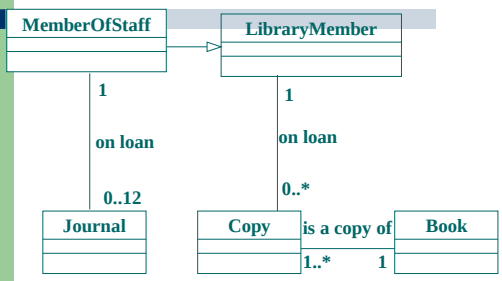
LibraryMember	borrows	Copy
LibraryMember	returns	Copy
MemberOfStaff	borrows	Journal
MemberOfStaff	returns	Journal

Item not needed yet.

Domain Model



Domain Model



GRASP

Grasp: Design Objects with Responsibilities

General Responsibility Assignment Software Patterns (GRASP)

13

2015/11/4

GRASP

The GRASP?

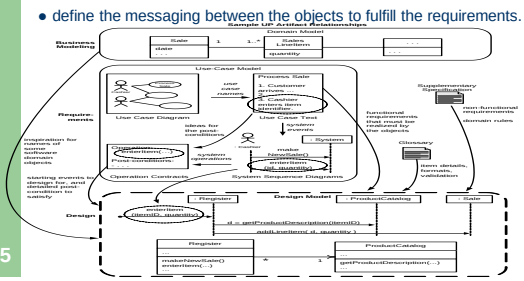
- It is a methodical approach including a set of principles and patterns to help us understand essential object design and apply design reasoning in a methodical way.

14

2015/11/4

GRASP

- What is involved in 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.



15

Responsibility-Driven Design (RDD)

- A popular way of thinking about the design of software objects and also larger-scale components is in terms of
  - **responsibilities**,
  - **roles**, and
  - **collaborations**.

- **Basically, these responsibilities are of the following two types:**
  - **Doing**
  - **Knowing.**

16

RDD

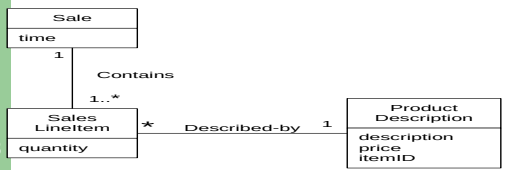
- Doing responsibilities of an object include:
  - doing something itself,
  - initiating action in other objects
- Knowing responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about things it can derive or calculate

17

2015/11/4

RDD

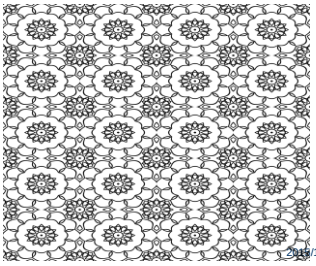
- Responsibilities are implemented by methods that either act alone or collaborate with other methods and objects.
  - For example, the Sale class might define one to know its total: *getTotal*. To fulfill that responsibility, the Sale may collaborate with other objects, such as sending a *getSubtotal* message to each SalesLineItem object asking for its subtotal.



18

Patterns

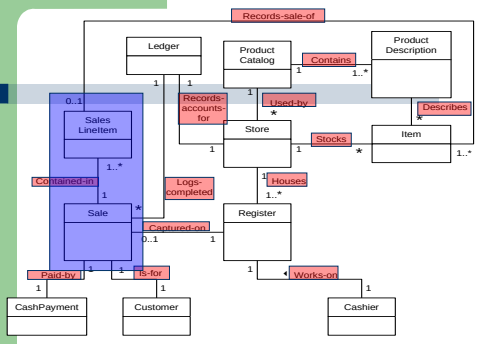
- In OO design, a pattern is a *named description of a problem and solution* that can be applied to new contexts;



A short Example of OOD with GRASP

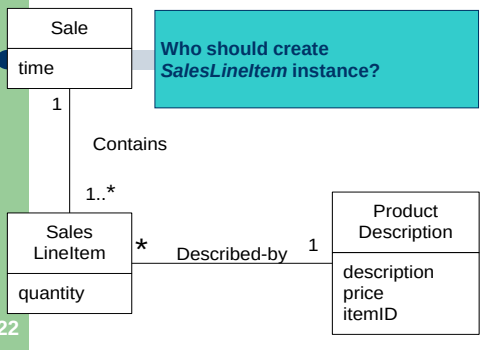
- There are nine GRASP patterns; We learn the following:
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion

Example: NextGen POS



CREATOR

- **Example**



GRASP: Creator

- **Problem: Who creates the *SalesLineItem* object?**
- One of the first problems you have to consider in OO design is:
  - ***Who creates object X?***
- What kind of responsibility?

GRASP: Creator

Definition of the **Creator pattern**

- **Name**: Creator
- **Problem**: Who creates an A?
- **Solution**: (this can be viewed as advice)
  - Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):
    - B "contains" or compositely aggregates A.
    - B records A.
    - B closely uses A.
    - B has the initializing data for A.

GRASP: Creator

- **Problem: Who creates the *SalesLineItem* object?**
- This is a doing responsibility.
  - Sale contains a set of SalesLineItems

25 2015/11/4

CREATOR

- **Example**

This assignment of responsibilities requires that a *makeLineItem* method be defined in Sale.

Once again, the context in which we considered and decided on these responsibilities was while drawing an interaction diagram. The method section of a class diagram can then summarize the responsibility assignment results.

26

Creator

27 2015/11/4

Review

Big Data video

GRASP: Information Expert

- Problem: In the NextGEN POS application, some class needs to know the grand total of a sale.
- The pattern Information Expert (often abbreviated to Expert) is one of the most basic responsibility assignment principles in object design.
- ***Who should be responsible for knowing a grandTotal?*** Of course, this is a knowing responsibility, but Expert also applies to doing.

29 2015/11/4

GRASP: Information Expert

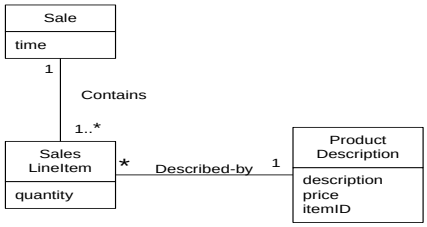
- **Name:** Information Expert
- **Problem:** What is a basic principle by which to assign responsibilities to objects?
- **Solution:** (advice):

  Assign a responsibility to the class that has the information needed to fulfill it.

30 2015/11/4

Information Expert

- *Who should be responsible for knowing the grand total of a sale?*
- Do we look in the Design Model (first) or the Domain Model (second) to analyze the classes that have the information needed?



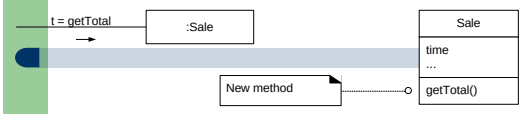
31

Information Expert

- What information do we need to determine the grand total?
- We need to know about all the ***SalesLineItem*** instances of a sale and the sum of their subtotals.
- A Sale instance contains these; therefore, by the guideline of Information Expert, ***Sale*** is a suitable class of object for this responsibility.

32

Information Expert



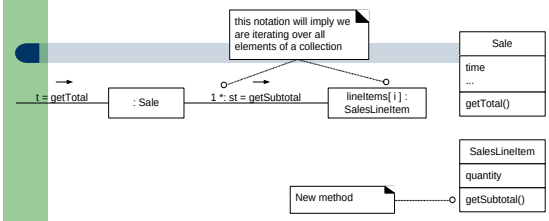
33

Information Expert

- Not done yet!
- What information do we need to determine the line item subtotal? *SalesLineItem.quantity* and *ProductDescription.price*.
- The *SalesLineItem* knows its quantity and its associated *ProductDescription*; therefore, by Expert, *SalesLineItem* should determine the subtotal;
- So *Sale* should send *getSubtotal* messages to each of the *SalesLineItems* and sum the results.

34

Information Expert



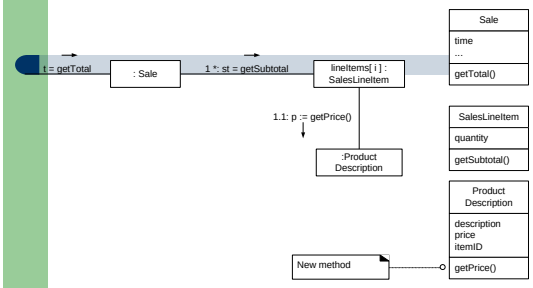
35

Information Expert

- To fulfill the responsibility of knowing and answering its subtotal, a *SalesLineItem* has to know the product price.
- The *ProductDescription* is an information expert on answering its price; therefore, *SalesLineItem* sends it a message asking for the product price.

36

Information Expert



37

Information Expert

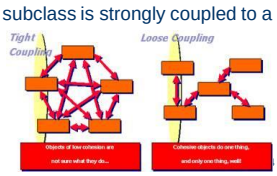
- In conclusion, for sale's total, we assigned three responsibilities to three design classes of objects as follows.

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

38

GRASP: Low coupling

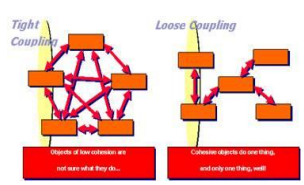
- **Coupling** is a measure of how strongly one element is connected to, or depends on other elements.
- If there is coupling or dependency, then when the depended-upon element changes, the dependent may be affected.
  - For example, a subclass is strongly coupled to a superclass.



39

GRASP: Low coupling

- **Pattern Name: Low Coupling**
- **Problem: How to reduce the impact of change?**
- **Solution:**
  Assign a responsibility so that coupling remains low.
  Use this principle to evaluate alternatives (and support reuse)



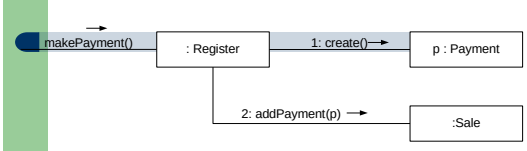
40

Low Coupling

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

41

Low Coupling

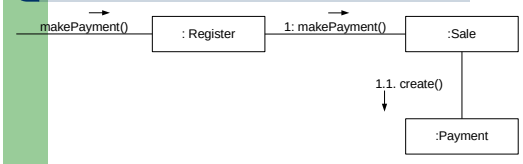


- This assignment of responsibilities couples the Register class to knowledge of the Payment class.

42

Low Coupling

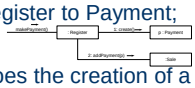
- An alternative



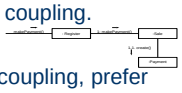
43

Low Coupling

- In both cases we assume the *Sale* must eventually be coupled to knowledge of a *Payment*.
- Design 1, in which the Register creates the *Payment*, adds coupling of Register to Payment;



- Design 2, in which 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.

44

Low Coupling

- Low Coupling supports the design of classes that are more independent, which reduces the impact of change.

45

GRASP: Controller

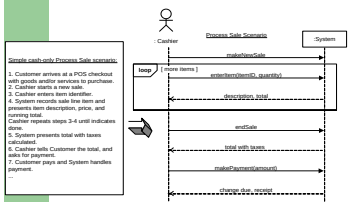
- A simple layered architecture has a UI layer and a domain layer, among others.
- From the Model-View Separation Principle, we know the UI objects should ***not*** contain application or "business" logic such as calculating a payment.
- Therefore, once the UI objects pick up the mouse event, for example, they need to delegate (forward the task to another object) the request to domain objects in the domain layer.

46

Controller

Problem

- What first object beyond the UI layer receives and coordinates ("controls") a ***system operation***?
- A **controller** is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.



Controller

Solution

- Assign the responsibility to a class representing one of the following choices:
  - Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem.
  - Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session.

48

Controller

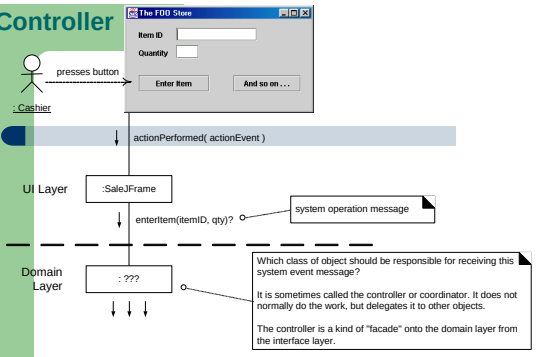
- During analysis, system operations may be assigned to the class *System* in some analysis model, to indicate they are system operations.

System
endSale() enterItem() makeNewSale() makePayment() . . .

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

49

Controller

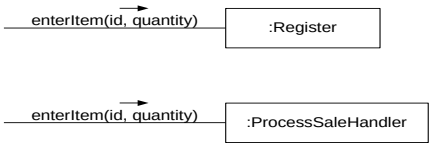


Who should be the controller for system events such as enterItem and endSale?

50

Controller

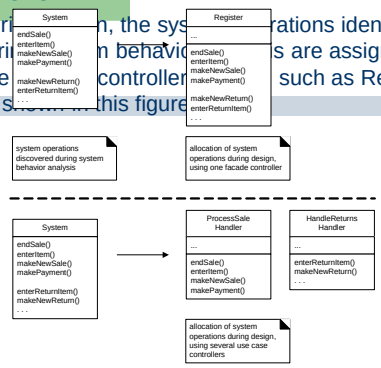
- By the Controller pattern, here are some choices:
  - Represents the overall "system," "root object," device, or subsystem.
    - *Register, POSSystem*
  - Represents a receiver or handler of all system events of a use case scenario.
    - *ProcessSaleHandler, ProcessSaleSession*



51

Controller

- Duri n, the sys rations identified duri n behavi s are assigned to one controller such as Register, as s this figure



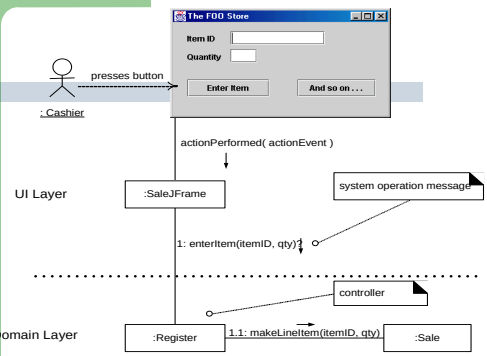
52

Controller – Two Choices

- Facade controllers (representing overall system, device, or a subsystem) are suitable when there are not "too many" system events.
- With use case controller, then you will have a different controller for each use case.
  - The NextGen application contains use cases such as Process Sale and Handle Returns, then there may be a ProcessSaleHandler class and so forth
  - when there are many system events across different processes.

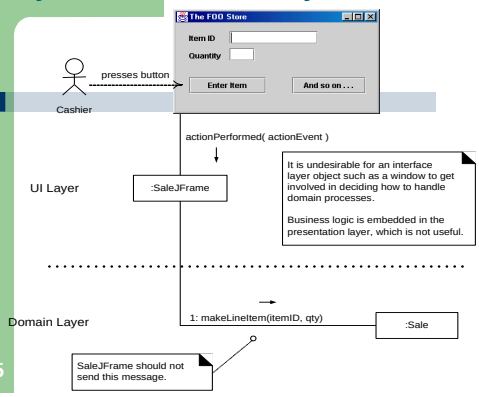
53

UI Layer Does Not Handle System Events



54

UI Layer Does Not Handle System Events



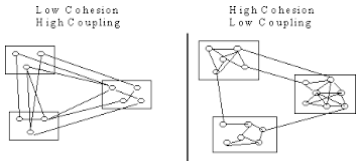
55

- What is wearable device?

High Cohesion

Problem

- Cohesion is *a measure of how strongly related and focused the responsibilities of an element are.*
- An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. (These elements include classes, subsystems, and methods.)



57

GRASP: High Cohesion

- **Name:** High Cohesion
- **Problem:** How to keep objects focused, and understandable, and as a side effect, support Low Coupling?
- **Solution:** (advice)
  Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

58

High Cohesion

- A class with low cohesion does many unrelated things or does too much work. Such classes suffer from the following problems:
  - hard to comprehend
  - hard to reuse
  - hard to maintain

59

High Cohesion

Example

- Assume we 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

60

High Cohesion

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.

It is okay now; 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.

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

61

High Cohesion

This design delegates the payment creation responsibility to the Sale supports higher cohesion in the Register.

It supports both high cohesion and low coupling, it is desirable.

62

High Cohesion

Discussion

- Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions
- It is an evaluative principle that a designer applies while evaluating all design decisions.
- High cohesion exists ***when the elements of a component (such as a class) "all work together to provide some well-bounded behavior".***

63

Applying GRASP to OOD

- **There are nine GRASP patterns:**
  - ***Creator***
  - ***Controller***
  - **Pure Fabrication**
  - ***Information Expert***
  - ***High Cohesion***
  - **Indirection**
  - ***Low Coupling***
  - **Polymorphism**
  - **Protected Variations**

64

Review

- What is false about design patterns?

A: A design pattern can decrease time to market and increase quality.
B: A design pattern is a map from object-oriented design to specific language implementation.
C: A design pattern contains well-defined constructs for specific types of problems.
D: Design patterns can be applied to software problems, and organizational problems.

Review

- One of the following design patterns is to evaluate the responsibility assignment given by other patterns

a) Information Expert	b) Creator
c) Controller	d) High Cohesion

Review

- A well modularized design consists of objects that are:
  - A strongly coupled
    B uncohesive
    C highly cohesive and loosely coupled
    D polymorphic

Design patterns

a.) Are design patterns a form of reuse?
If so, what do they enable reusing?

b.) How are design patterns different from algorithms and data structures?

Review

- What is GRASP?
- G_____R_____A_____S_____P

Review

- Explain what the **Information Expert pattern** is and the **questions** and the **solution** it solves.

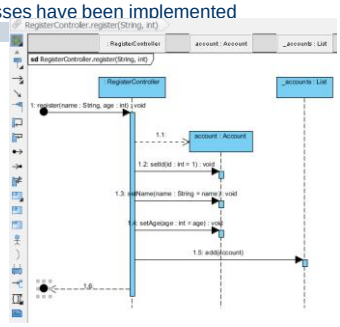
Name: Information Expert

Problem:

Solution:

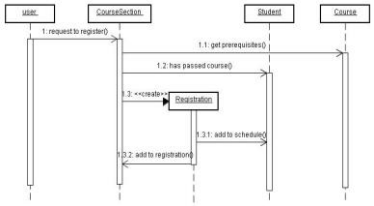
Review

- Based on the following sequence diagram, implement the class **RegisterController** as much as you can (including data members and methods). You can assume that **Account and List** classes have been implemented



Review

- Based on the following sequence diagram, implement the class **CourseSection** as much as you can (including data members and methods). You can assume that **Registration, Student, Course** classes have been implemented



- What are future wearable devices?