Advanced Java

LESSON 6: SOLID (OBJECT-ORIENTED DESIGN)

Creative Commons

Attribution 4.0 International (CC BY 4.0)

This workforce product was funded by a grant awarded by the U.S. Department of Labor's Employment and Training Administration. The product was created by the grantee and does not necessarily reflect the official position of the U.S. Department of Labor. The U.S. Department of Labor makes no guarantees, warranties, or assurances of any kind, express or implied, with respect to such information, including any information on linked sites and including, but not limited to, accuracy of the information or its completeness, timeliness, usefulness, adequacy, continued availability, or ownership. This is an equal opportunity program. Assistive technologies are available upon request and include Voice/TTY (771 or 800-947-6644).

Agenda

- Architecture Matters!
- Intro to OOA/D
- GRASP
- SOLID
- CRC Card Activity

Architecture Matters!

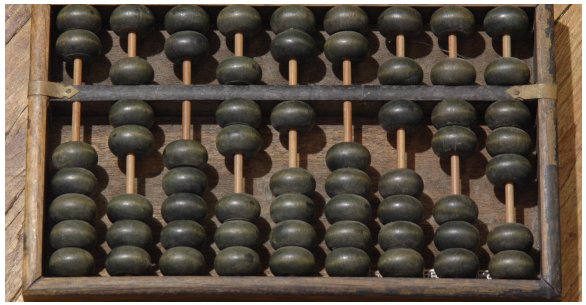
Architecture Matters

Sometimes BAD architecture is easy to spot:

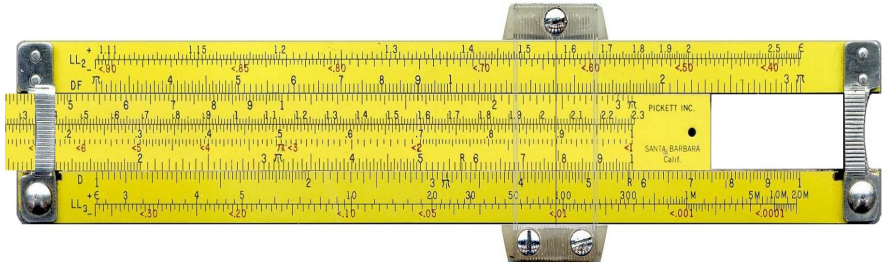
Architecture Matters

We will learn to architect OOP Software that uses **abstraction** to improve flexibility and overall quality, and to easily adapt to changing requirements.

Ancient Chinese Abacus



Slide ruler 1968



Calculator 1972

Modern Calculator

Architecture Matters

Abstraction

- Abstractions are generic terms, like a category of objects, that represent specific types of objects that have something in common
- iPhone Calculator, Handheld Calculator, Slide Rule and Abacus are all specific types of “Calculator” objects

Architecture Matters

Abstraction

- **Hard (or "strong") Dependencies** on specific types of objects are inflexible and causes problems when needs change
- Better to depend on **abstractions, which allow soft (or "weak") dependencies**. This allows greater flexibility via substitution of different specific types that are based on the abstraction

Architecture Matters

Good software architecture means:

- Flexible, non-rigid*
  (easily adapts to change)
- Not Fragile*
  (edits don't break other things; code runs without errors)
- Portable*
  (can reuse some of the code in other projects)

* NOTE: no program can be 100% compliant;
  no code can be 100% compliant

Architecture Matters

Every time you touch the code there's a cost:

- to WRITE the code
- to TEST the code
- to DEBUG the code
- to DOCUMENT the code
- To DEPLOY the code

Architecture Matters

What we build needs to be Flexible

- Hard Dependencies:
  - Make it impossible to edit code without breaking other parts of the program (fragile)
  - Make portability impossible
  - Make flexibility impossible
  - Make code revisions very expensive due to the above

Architecture Matters

What we build needs to be Flexible

Architecture Matters

What we build needs to be Flexible

- Works with ANY standard electrical device
- Has an industry standard interface (3-prong plug)
- Works with several other wires with an adapter

Architecture Matters

Definition Flexible

Characterized by a ready capability to adapt to

new, different or changing requirements

Intro to OOA/D

Intro to OOA/D

What is OOA/D?

Object-Oriented Analysis & Design

Are You a Carpenter or an Architect?

Owning a hammer
and knowing how to use it …

Does not make you an Architect!

Intro to OOA/D

Knowing the Java language and being able to write clever code …

Does not make you a skilled object-oriented software Architect!

Intro to OOA/D

Traditional Software Development

The software development process has been profoundly screwed up!

And the proof is …

15% of all Information Technology projects get cancelled outright *

Cost: > $38 billion per year!

> $17 billion per year are spent on cost overruns!

* According to the Standish Group, which conducts an annual, industry-wide survey.

Intro to OOA/D

Traditional Software Development

Those projects that are finally released contain just 52% of the features customers asked for!

Projects are chronically late – only about 18% hit deadline …

… and are consistently, flawed!

* According to the Standish Group, which conducts an annual, industry-wide survey.

Intro to OOA/D

Traditional Software Development

Estimates of the number of bugs contained in shipped products run from one defect in every 1,000 lines of code to one defect in every 100!

* According to research conducted by Wired magazine.

A Fortune 500 Company once spent $250 million repairing and installing fixes to 13,000 customer-reported flaws. That translates to a stunning $19,000 per defect!

* According to Watts Humphrey in his book, "A Discipline for Software Engineering"

Intro to OOA/D

Every time you touch the code there's a cost:

- to WRITE the code
- to TEST the code
- to DEBUG the code
- to DOCUMENT the code
- To DEPLOY the code

Intro to OOA/D

Object-oriented Analysis & Design can help to solve these problems.

- In a study conducted by MIT, software projects developed with modern OOA/D methodologies showed **60%-90% improvements** in **time**, **quality** and **cost** over traditional development methods.

GRASP Principles

How to Design Objects

The hard step: moving from analysis to design

How to do it?

- Design principles (Larman: “patterns”) – an attempt at a methodical way to do object design
- Larman:

“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.”

Object Oriented Design

Larman (continued):

"Ouch! Such vague advice doesn't help us, because deep principles and issues are involved. Deciding what methods belong where and how objects should interact carries consequences and should be undertaken seriously. Mastering OOD – and this is its intricate charm – involves a large set of soft principles, with many degrees of freedom. It isn't magic – the patterns can be named (important!), explained, and applied. Examples help. Practice helps. . . ." (p. 271)

Object Oriented Design

Fundamental activity in OOD:

Identifying Classes and Assigning responsibilities to objects!

Responsibility – an obligation required of an object

Two types of responsibilities:

- Doing
- Knowing

Responsibilities

Two types of responsibilities:

1. Doing:
   - creating an object
   - doing a calculation
   - initiating action in other objects
2. Knowing:
   - knowing about private encapsulated data
   - knowing about related objects
   - knowing about things it can derive or calculate

Note: “knowing” often found in domain model, e.g. attributes and associations

Responsibilities

Responsibilities are more general than methods

- Methods are implemented to fulfill responsibilities
- Example: Sale class may have a method to know its total but may require interaction with other objects
- What are the principles for assigning responsibilities to objects?
- Assigning responsibilities initially arises when drawing interaction diagrams . . .

Responsibility-Driven Design (RDD)

- RDD is a metaphor for thinking about OO software design.
- Metaphor – software objects thought of as people with responsibilities who collaborate to get work done
- An OO design as a *community of collaborating responsible objects*

GRASP Principles

General responsibility assignment software patterns (or **principles**)

GRASP principles – a learning aid for OO design with responsibilities

Pattern – a *named* and *well-known* problem/solution pair that can be applied in new contexts, with advice on how to apply it in new situations and discussion of its trade-offs, implementations, variations, etc.

GRASP Principles

9 GRASP principles:

1. Information Expert
2. Creator
3. Low Coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

Information Expert

Problem: What is a general principle for assigning responsibilities to objects?

Solution: Assign a responsibility to the information expert, that is, the class that has the information necessary to fulfill the responsibility.

Example: (from POS system) Who should be responsible for knowing the grand total of a sale?

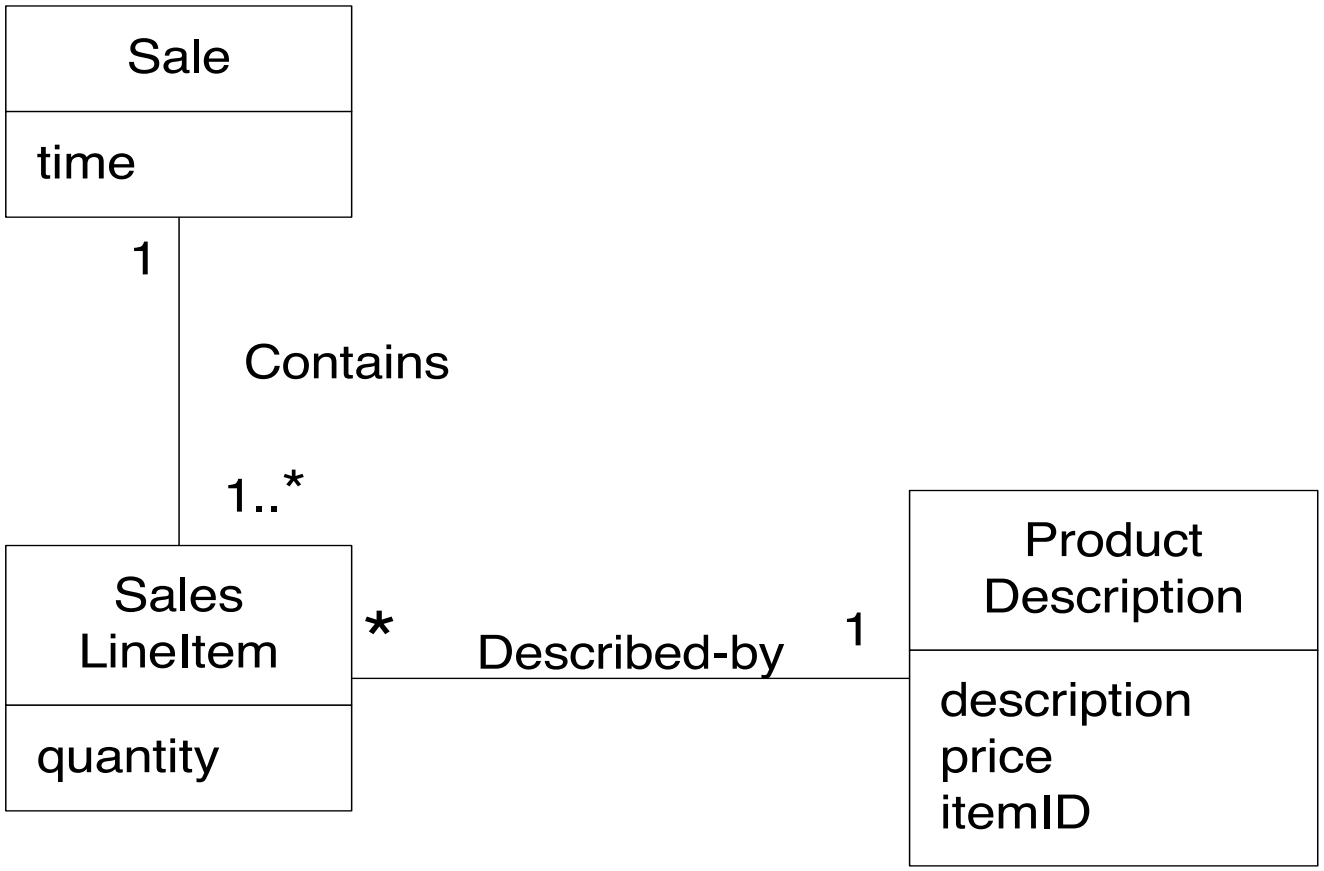
Information Expert

Example: (from POS system) Who should be responsible for knowing the grand total of a sale?

Start with domain model and look for associations with Sale (Fig. 17.14)

What information is necessary to calculate grand total and which objects have the information?

Information Expert



From domain model: “Sale Contains SalesLineItems” so it has the information necessary for total . . .

Information Expert

Example (cont.): Who should be responsible for knowing the grand total of a sale?

Summary of responsibilities:

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

Information Expert

- Information Expert => “objects do things related to the information they have”
- Information necessary may be spread across several classes => objects interact via messages
- Animation principle – in real world a sale is inanimate – it just stores data, it is not active
- In object-oriented world these objects become animated
- Instead of having something done to them they do it themselves
- Information Expert is not the only pattern so just because an object has information necessary doesn’t mean it will have responsibility for action related to the information

Example: who is responsible for saving a sale in a database

- Problems if it is given to Sale (will violate other principles: cohesion, coupling)

Creator

Problem: Who should be responsible for creating a new instance of a class?

Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:

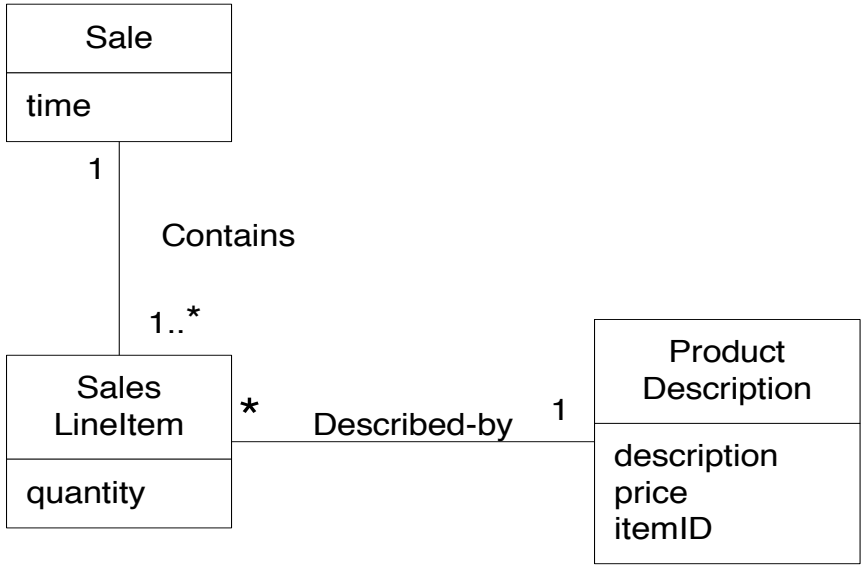
- B contains A objects
- B records instances of A objects
- B closely uses A objects
- B has the initializing data that will be passed to A when it is created.

Example: (from POS system) Who should be responsible for creating a new SalesLineItem instance?

Creator

Example: (from POS system) Who should be responsible for creating a new SalesLineItem instance?

Start with domain model:

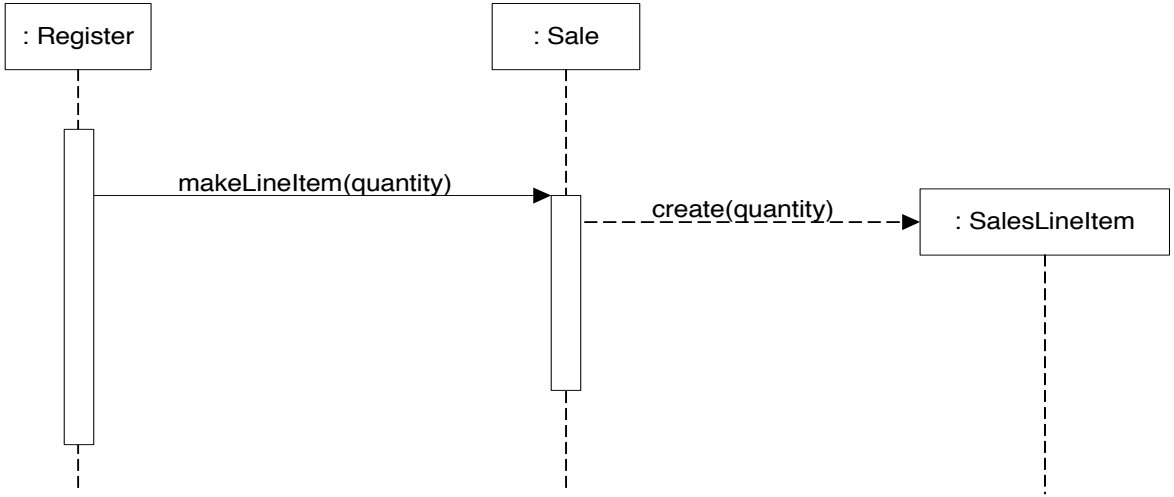


Creator

Example: (from POS system) Who should be responsible for creating a new SalesLineItem instance?

Since a Sale contains SalesLineItem objects it should be responsible according to the Creator pattern

Note: there is a related design pattern called Factory for more complex creation situations



Low Coupling

Problem: How to support low dependency, low change impact, increased reuse?

Solution: Assign a responsibility so coupling is low.

Coupling – a measure of how strongly one element is connected to, has knowledge of, or relies on other elements

Low Coupling

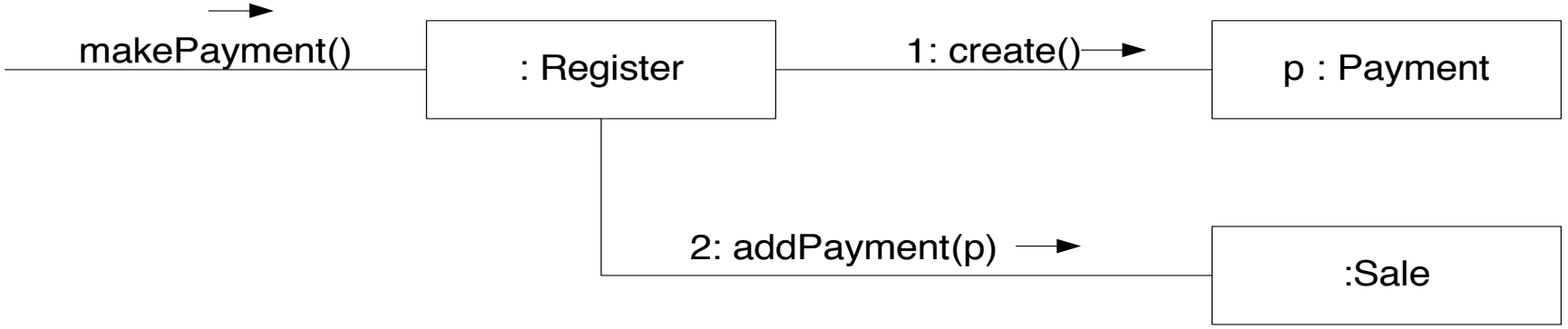
A class with high coupling relies on many other classes – leads to problems:

- Changes in related classes force local changes
- Harder to understand in isolation
- Harder to reuse

Low Coupling

Example (from POS system):

- Consider Payment, Register, Sale
- Need to create a Payment and associate it with a Sale, who is responsible?
- Creator => since a Register "records" a Payment it should have this responsibility
- Register creates Payment *p* then sends *p* to a Sale => coupling of Register class to Payment class



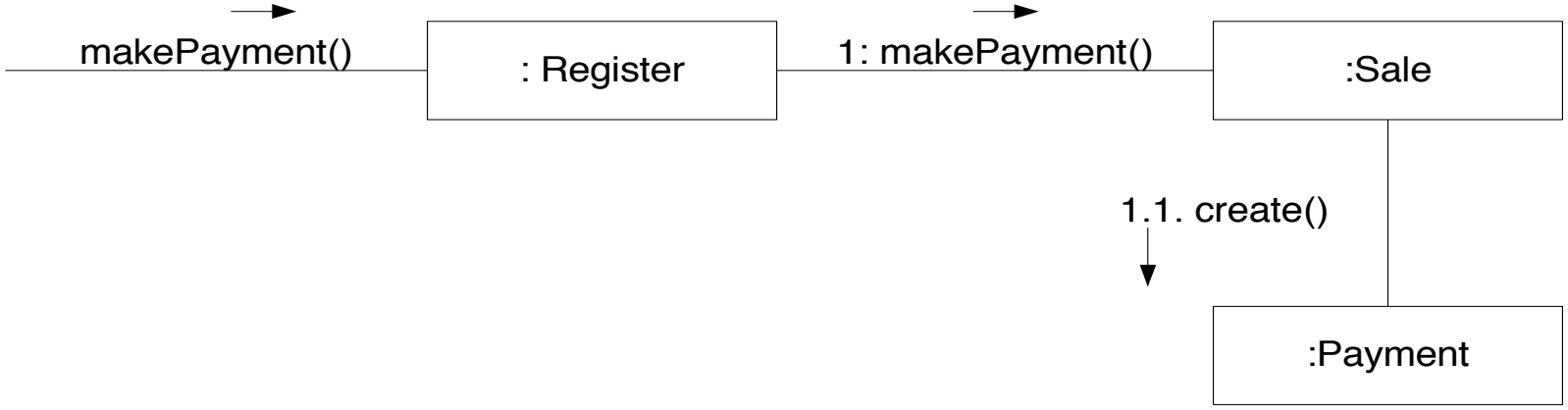
Low Coupling

Example (from POS system):

- Consider Payment, Register, Sale
- Need to create a Payment and associate it with a Sale, who is responsible?

Alternate approach:

Register requests Sale to create the Payment

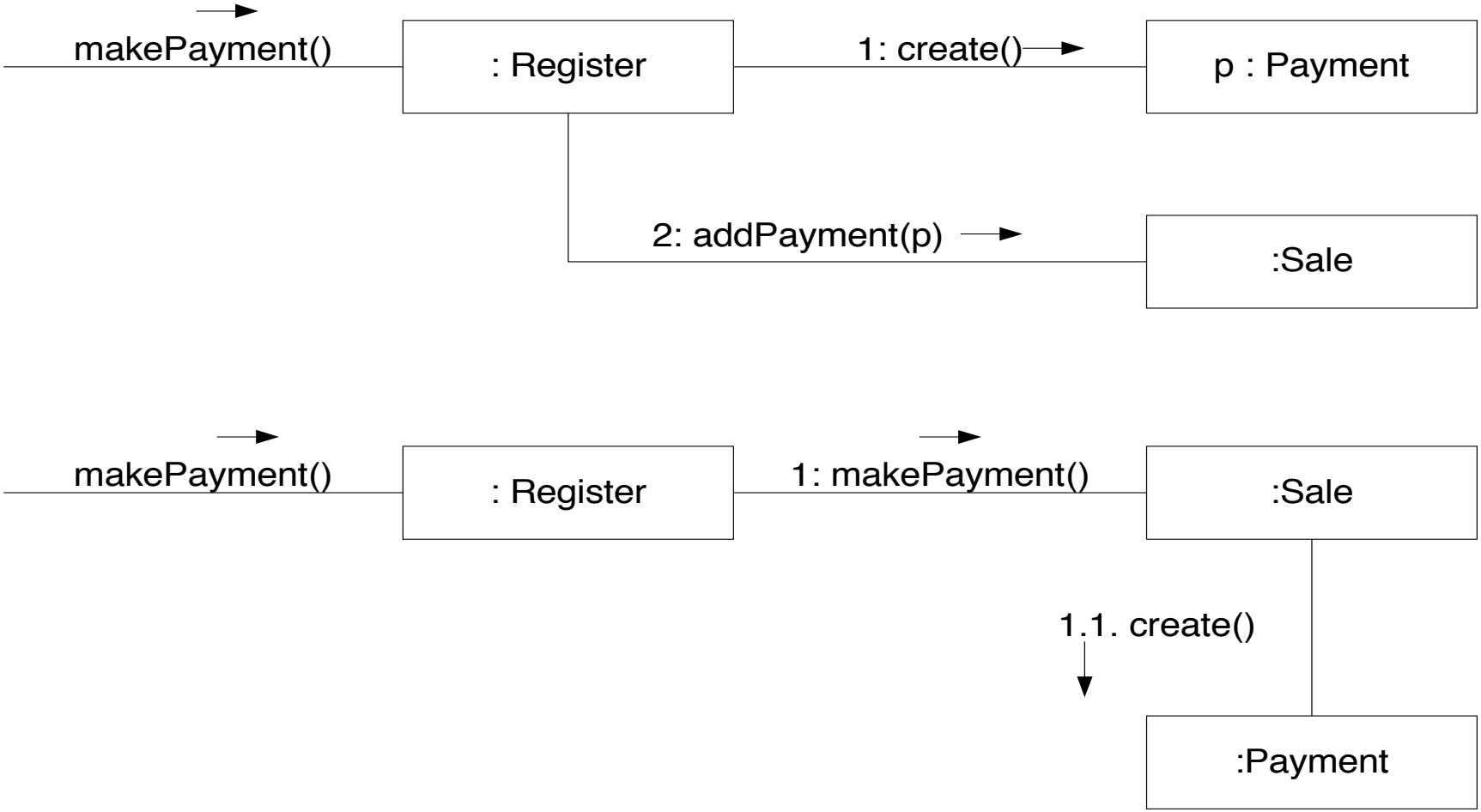


Low Coupling

Consider coupling in two approaches:

- In both cases a Sale needs to know about a Payment
- However a Register needs to know about a Payment in first but not in second
- Second approach has lower coupling

Low Coupling



Low Coupling

Low coupling is an evaluative principle, i.e. keep it in mind when evaluating designs

Examples of coupling in Java (think “dependencies”):

- TypeX has an attribute of TypeY
- TypeX calls on services of a TypeY object
- TypeX has a method that references an instance of TypeY
- TypeX is a subclass of TypeY
- TypeY is an interface and TypeX implements the interface

Note: subclassing => high coupling

Note: extreme of low coupling is unreasonable

Controller

Problem: Who should be responsible for handling a system event? (Or, what object receives and coordinates a system operation?)

Solution: Assign the responsibility for receiving and/or handling a system event to one of following choices:

- Object that represents overall system, device or subsystem (*façade controller*)
- Object that represents a use case scenario within which the system event occurs (a <UseCase>Handler)

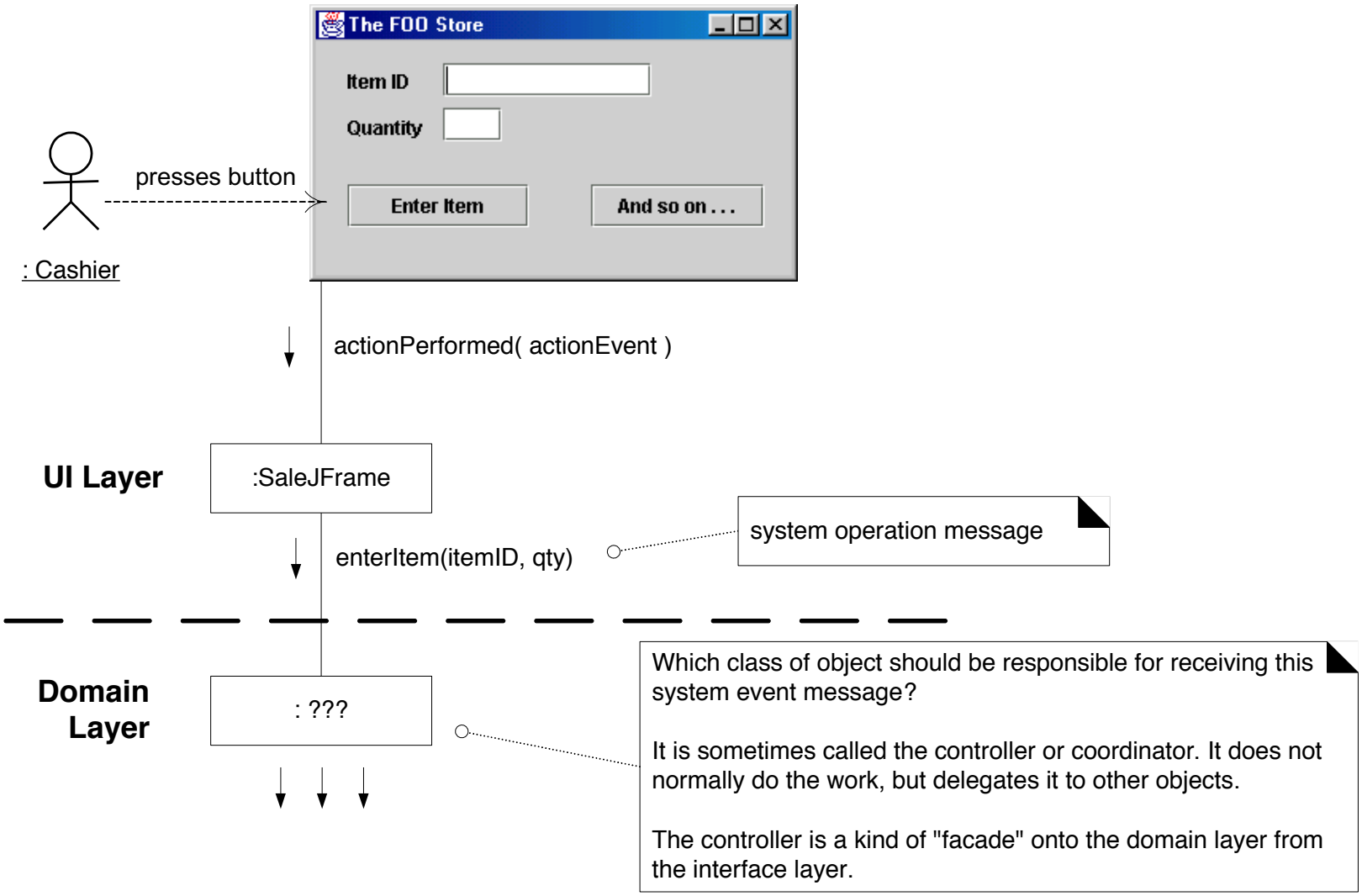
Controller

Input system event – event generated by an external actor associated with a system operation

Controller – a non-UI object responsible for receiving or handling a system event

- During analysis can assign system operations to a class System
- That doesn't mean there will be a System class at time of design
- During design a controller class is given responsibility for system operations

Controller


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.

Controller

- Controller is a *façade* into domain layer from interface layer
- Often use same controller class for all system events of one use case so that one can maintain state information, e.g. events must occur in a certain order
- Normally controller coordinates activity but delegates work to other objects rather than doing work itself
- Façade controller representing overall system – use when there aren't many system events
- Use case controllers – different controller for each use case

Controller

Bloated controller

- Single class receiving all system events and there are many of them
- Controller performs many tasks rather than delegating them
- Controller has many attributes and maintains significant information about system which should have been distributed among other objects

Important note: interface objects should not have responsibility to fulfill system events (recall model-view separation principle)

High Cohesion

Problem: How to keep complexity manageable?

Solution: Assign the responsibility so that cohesion remains high.

Cohesion – a measure of how strongly related and focused the responsibilities of an element (class, subsystem, etc.) are

High Cohesion

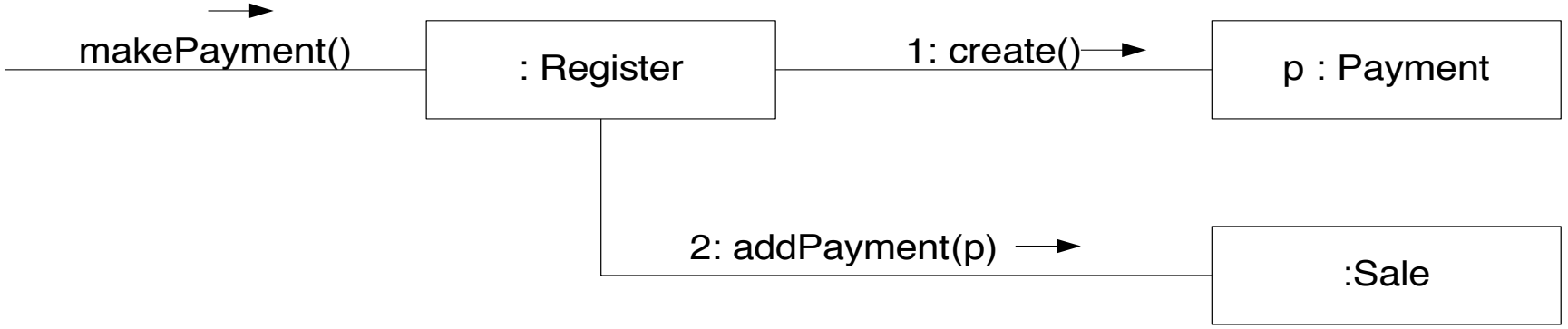
Problems from low cohesion (does many unrelated things or does too much work):

- Hard to understand/comprehend
- Hard to reuse
- Hard to maintain
- Brittle – easily affected by change

High Cohesion

Example: (from POS system) Who should be responsible for creating a Payment instance and associate it with a Sale?

Register creates a Payment *p* then sends addPayment(*p*) message to the Sale



High Cohesion

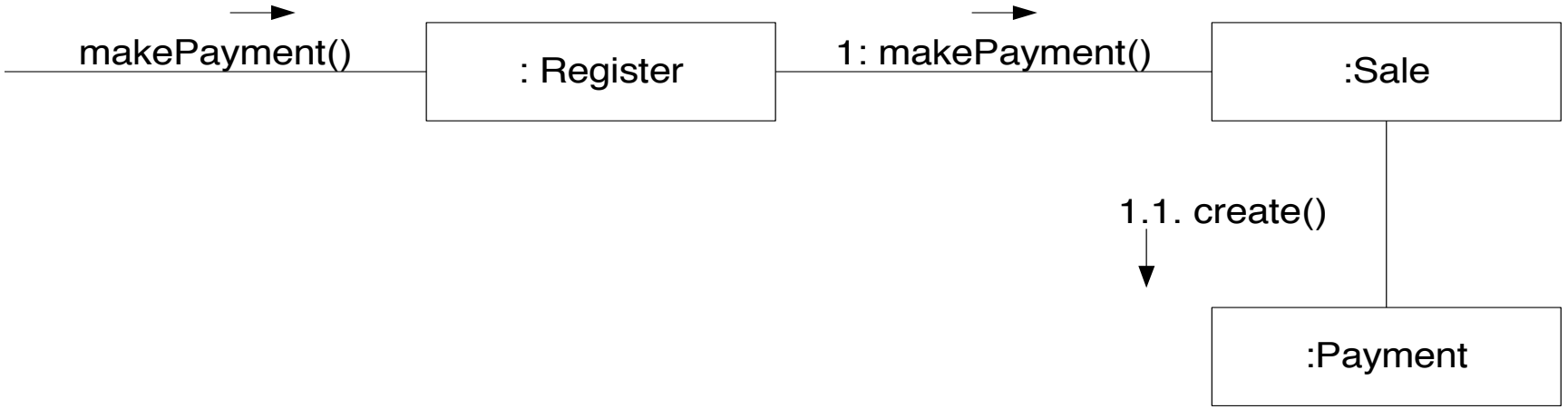
Register is taking on responsibility for system operation makePayment()

- In isolation no problem
- But if we start assigning additional system operations to Register then will violate **high cohesion**

High Cohesion

Example: (from POS system) Who should be responsible for creating a Payment instance and associate it with a Sale?

2. Register delegates Payment creation to the Sale



High Cohesion

- This second approach will lead to higher cohesion for Register class.
- Note: this design supports both low coupling and high cohesion
- High cohesion, like low coupling, is an evaluative principle

High Cohesion

Consider:

- Very low cohesion – a class is responsible for many things in different functional areas
- Low cohesion – a class has sole responsibility for a complex task in one functional area
- 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

High Cohesion

Typically high cohesion => few methods with highly related functionality

Benefits of high cohesion:

- Easy to maintain
- Easy to understand
- Easy to reuse

High Cohesion & Low Coupling

- High cohesion and low coupling pre-date object-oriented design (“modular design” – Liskov)
- Two principles are closely related – the “yin and yang” of software engineering
- Often low cohesion leads to high coupling and vice versa
- There are situations where low cohesion may be acceptable (read p. 318), e.g. distributed server objects

Additional Patterns

6. Polymorphism – Has been covered and will continue to be covered through out the semester
7. Pure Fabrication – A class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion
8. Indirection - The indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view-controller pattern.
9. Protected Variations – We will study this under the **Open/Closed Principle**. Protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism.

SOLID

SOLID

A mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable.

- Single Responsibility
- Open/closed
- Liskov substitution
- Interface segregation
- Dependency inversion

SOLID

Initial	Concept
S	**Single responsibility principle** - a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).
O	**Open/closed principle** - “software entities … should be open for extension, but closed for modification.”
L	**Liskov substitution principle** - “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”
I	**Interface segregation principle** - “many client-specific interfaces are better than one general-purpose interface.”
D	**Dependency inversion principle** - one should “depend upon abstractions, [not] concretions.”

CRC Card Activity

CRC Card Activity

- **CRC-cards** (**C**lass, **R**esponsibility and **C**ollaborator) are a approach to collaborative object-oriented modelling that has been developed as a tool for teaching object-oriented thinking to programmers.
- A **responsibility** is something the objects of a class know or do as a service for other objects. A book object in a library application, for example, might, among other things, be responsible for checking itself out and knowing its title and due date. The responsibilities of the objects of a class are written along the left side of the card.
- A **collaborator** is an object of another class helping to fulfill a specific responsibility. A book object, for example, can only know whether it is overdue, if it also knows the current date. The class names of collaborators are written along the right side of the card.
- The back of the card can be used for a brief description of the class' purpose, comments and miscellaneous details.

CRC Card Activity

CRC-cards for modelling a simple library application

Class: *Book*

Responsibilities	Collaborators
knows whether on loan	
knows due date	
knows its title	
knows its author(s)	
knows its registration code	
knows if late	Date
check out	

Class: *Librarian*

Responsibilities	Collaborators
check in book	Book
check out book	Book, Borrower
search for book	Book
knows all books	
search for borrower	Borrower
knows all borrowers	

Class: *Borrower*

Responsibilities	Collaborators
knows its name	
keeps track of borrowed items	
keeps track of overdue fines	

Class: *Date*

Responsibilities	Collaborators
knows current date	
can compare two dates	
can compute new dates	

CRC Card Activity

Using CRC-Cards

- The CRC-card approach can be roughly divided into two stages;

1. development of an initial CRC-card model and
2. scenario definition and roleplaying.

- In the roleplaying, participants act-out predefined scenarios, much like actors following a script when playing the characters in play. In object-oriented development, the characters of the play are the objects in a software system and the scenarios are hypothetical but concrete situations of system usage.

CRC Card Activity

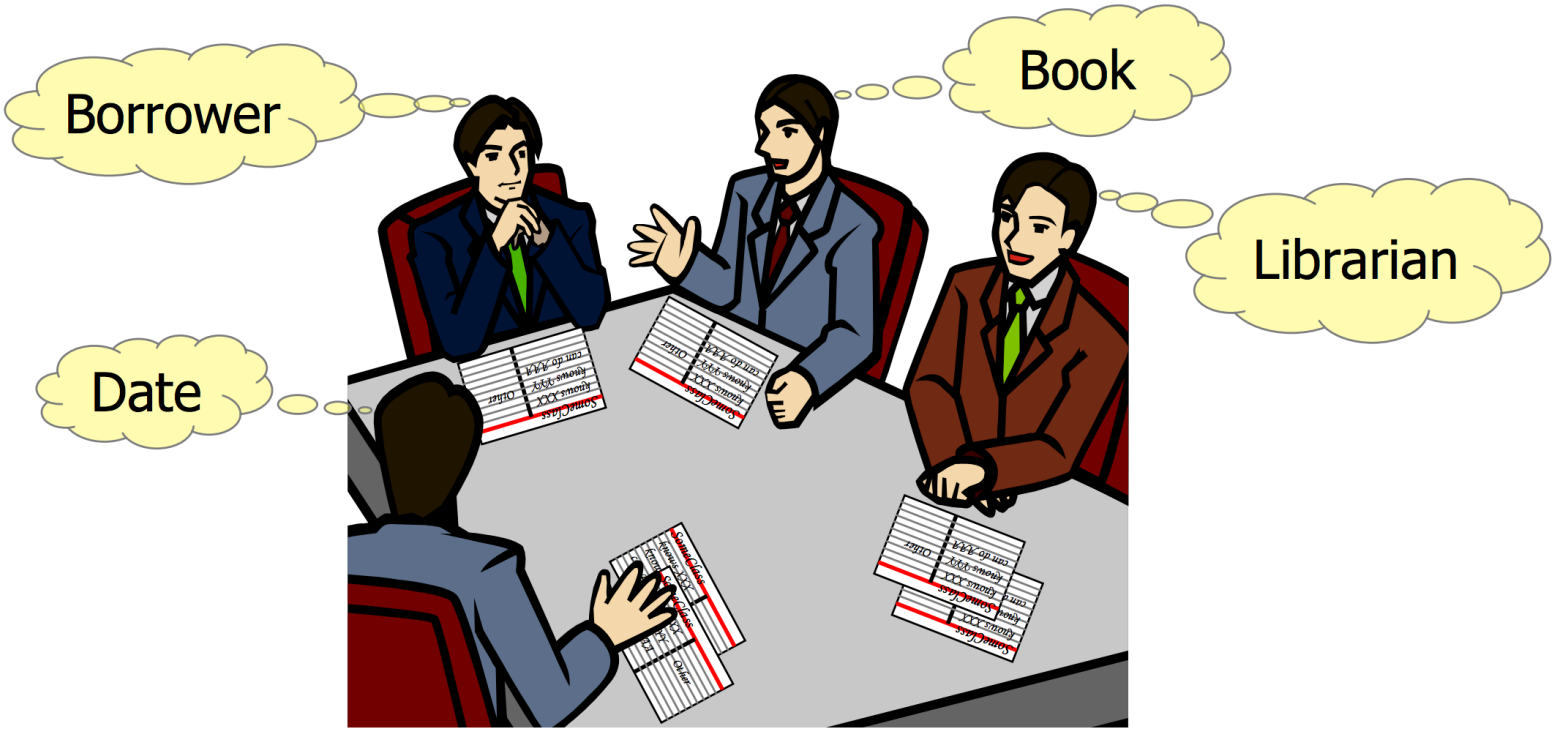
Using CRC-Cards

- The roleplaying helps students learn to think in objects in the following way

1. Students can only control their own assigned role(s) (i.e. CRC-card(s)); this helps them to give up global control
2. Students can only act according to the predefined responsibilities of their role(s); this helps them to focus on design, i.e. distribution of responsibilities.
3. Students collaboratively enact how their system should work; this immerses them in the object-ness of the material.

CRC Card Activity

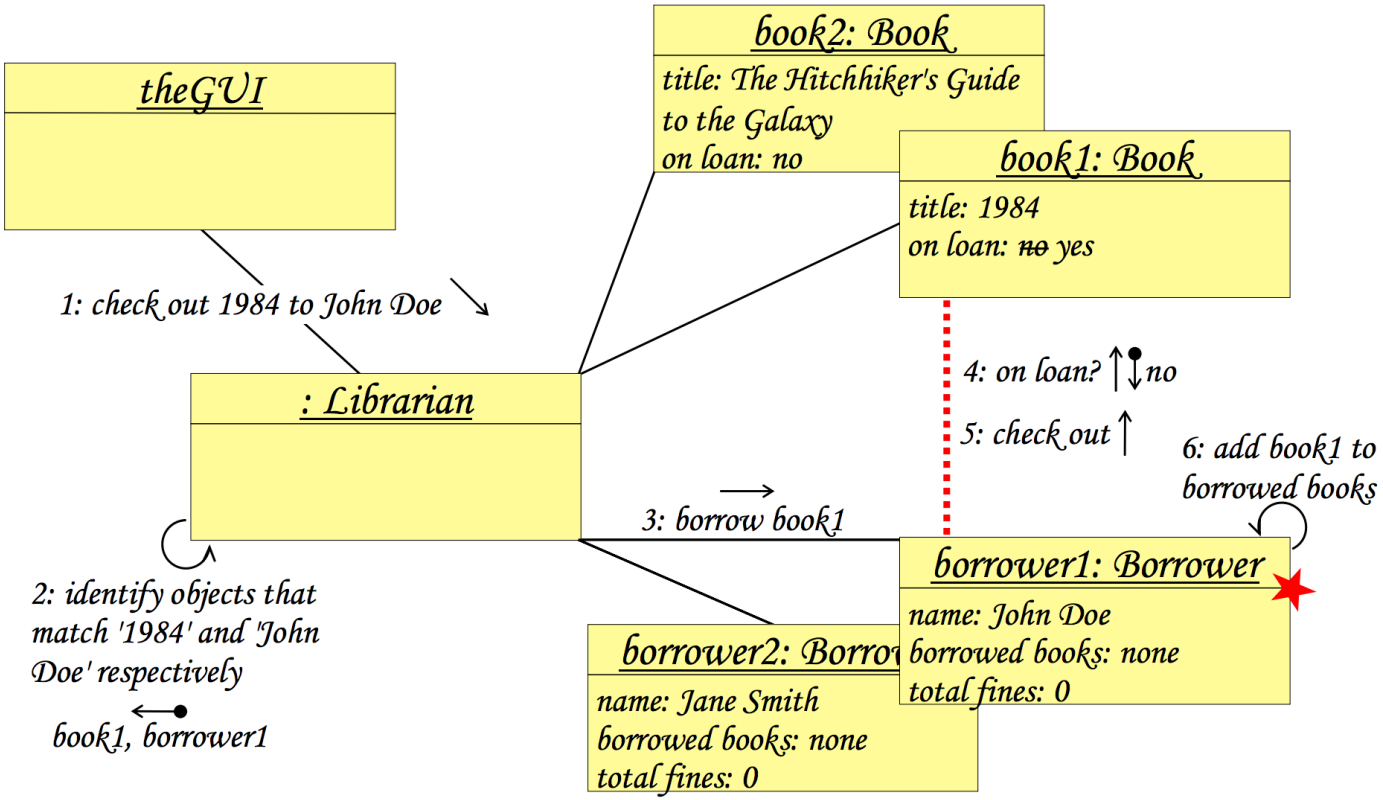
Roleplaying is a collaborative activity



CRC Card Activity

Roleplay Diagrams (RPDs)

Combines elements from UML's object and collaboration diagrams.



In-Class Activity

1. In groups of 4-5, create CRC Cards and a RPD for the following Scenario:

- You will need to design a basic Twitter competitor for a specific user group.
- This app will have the primary features of Twitter plus the required enrollment into specific groups that need to be managed.

2. To submit, take pictures of your cards and upload your RPD as a pdf from Lucid Chart (UML > Activity Diagram). All students will have the same artifacts but still need to submit them.