



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
- 



Object Oriented 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. . . .”



Object Oriented Design

Fundamental activity in OOD:

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



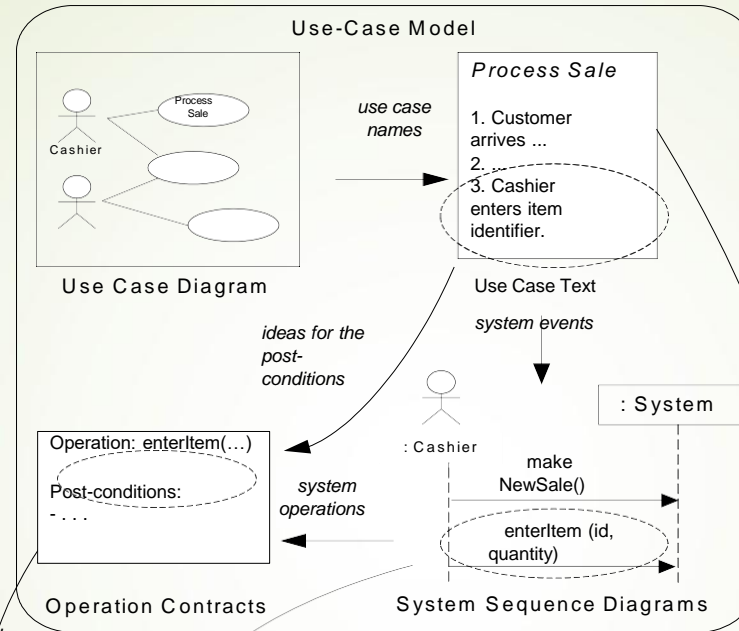
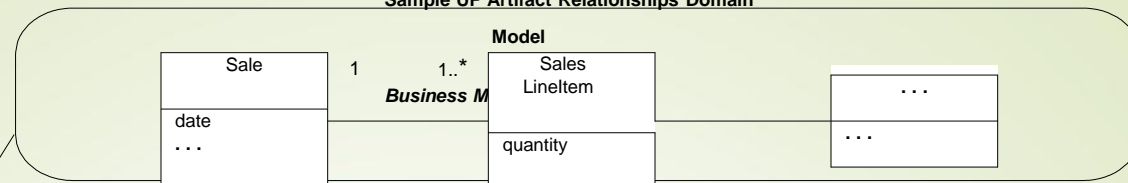
Responsibilities

What are the principles for assigning responsibilities to objects?

Assigning responsibilities initially arises when drawing interaction diagrams . . .



Sample UP Artifact Relationships Domain

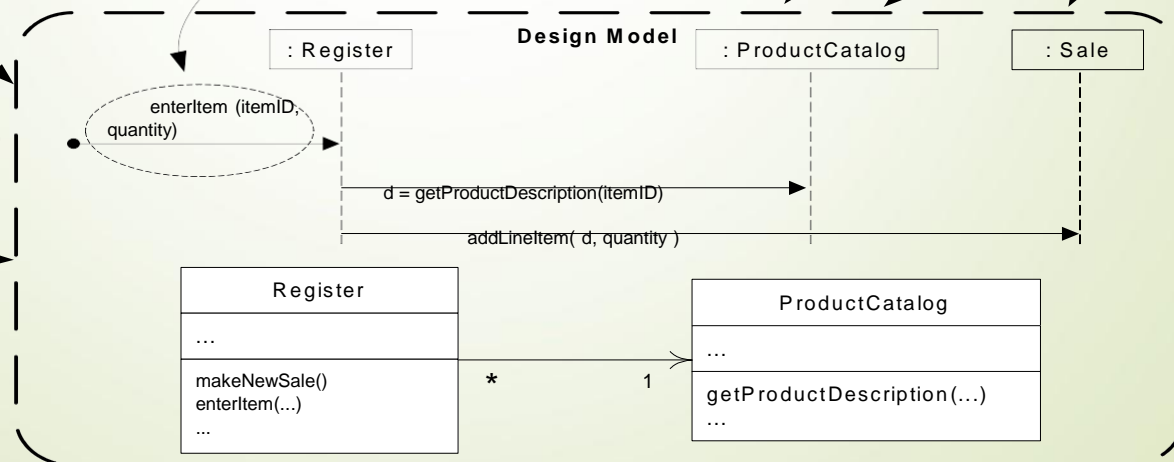


Require-ments

inspiration for names of some software/domain objects

starting events to design for, and detailed post- condition to satisfy

Design



Supplementary Specification

non-functional requirements

domain rules

Glossary

item details, formats, validation

functional requirements that must be realized by the objects



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

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



Summary of OOD

- Assigning responsibilities is important
 - First becomes important in interaction diagrams then in programming
 - Patterns are named problem/solution pairs for identifying principles to be used in assigning responsibilities
- 



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: Who should be responsible for knowing the grand total of a sale?



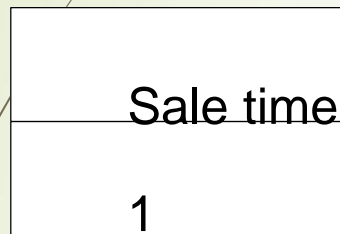


Information Expert

Example: Who should be responsible for knowing the grand total of a sale?

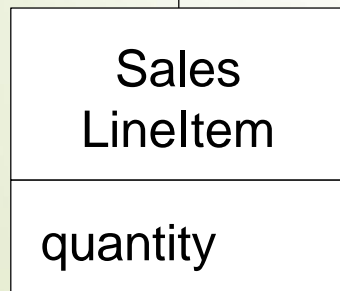
Start with domain model and look for associations with Sale

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



Contains

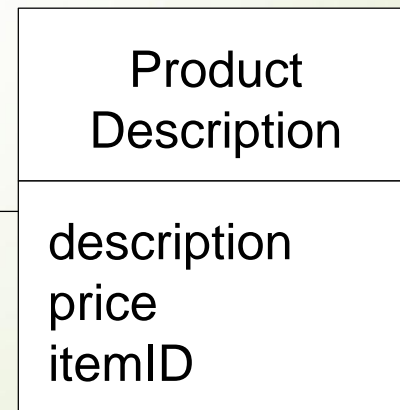
1..*



*

Described-by

1



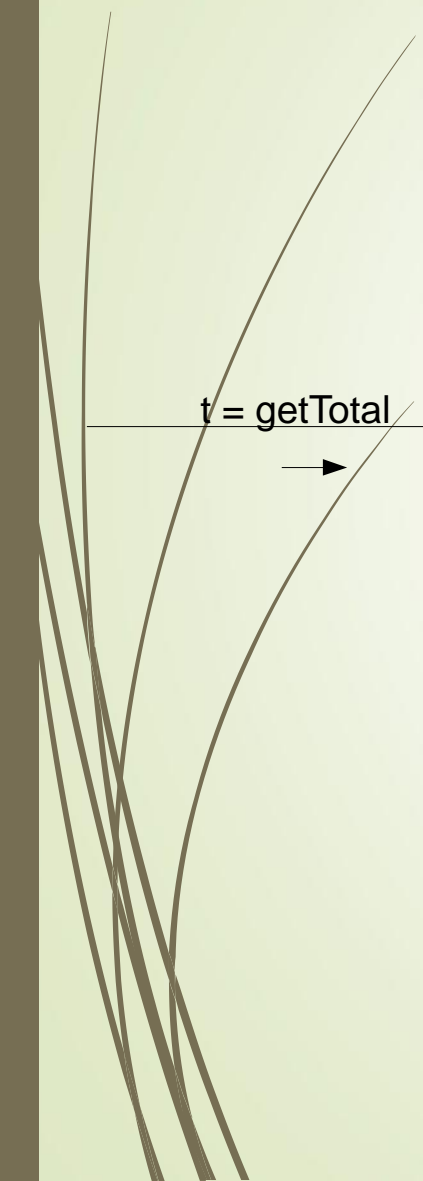


Information Expert

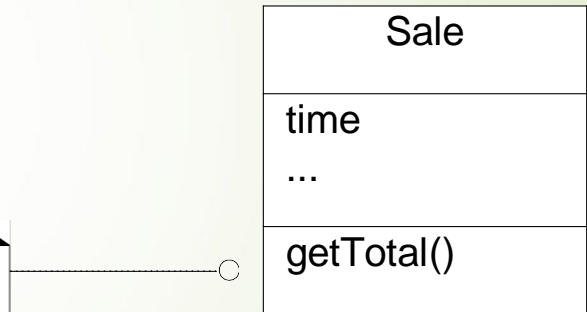
Example (cont.): From domain model:

“Sale Contains SalesLineItems” so it has the information necessary for total . . .





New method





Information Expert

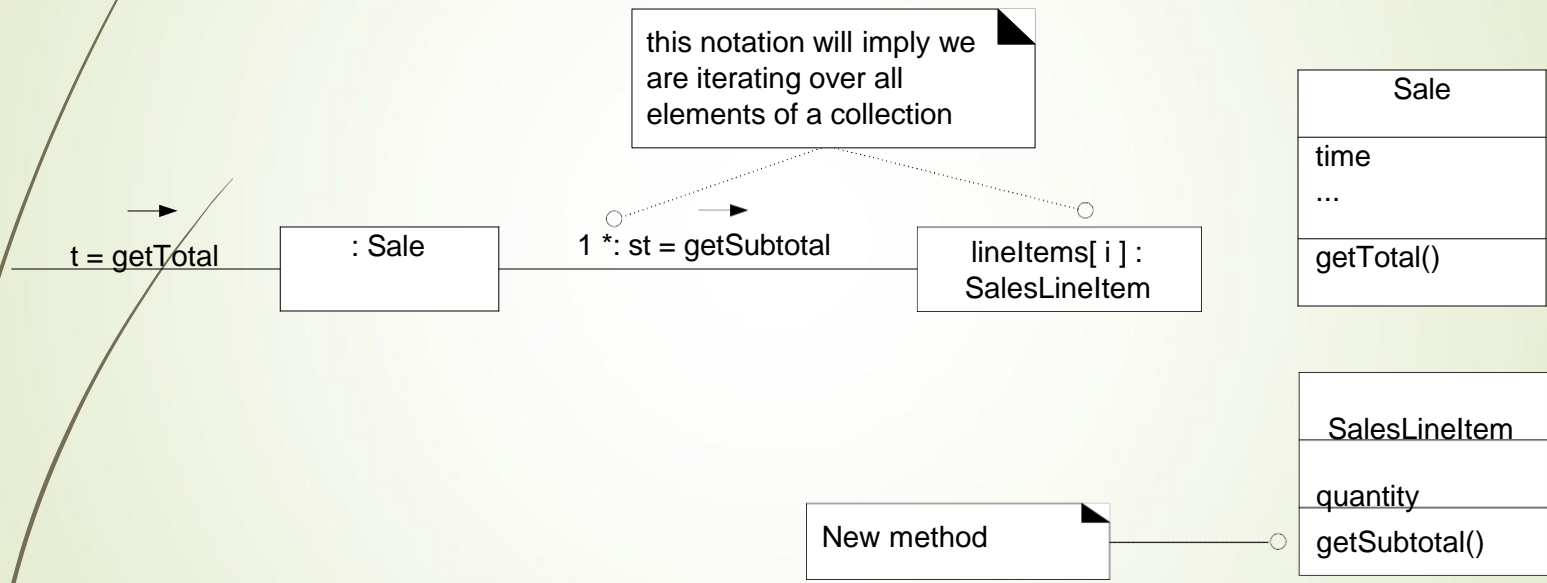


Example (cont.):

How is line item subtotal determined?

quantity – an attribute of SalesLineItem

price – stored in ProductDescription . . .





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
- 



Information Expert

- 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

- 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:

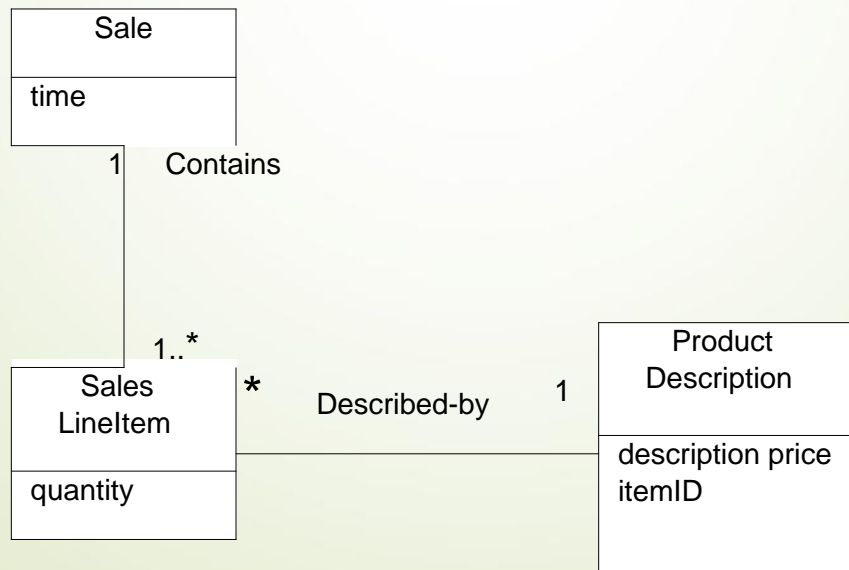
- B contains A objects 
 - B records instances of A objects
 - B closely uses A objects
-
- B has the initializing data that will be passed to A when it is created.

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 (Fig. 17.13)

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



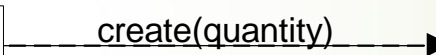
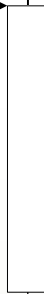
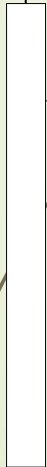
: Register

: Sale

makeLineItem(quantity)

create(quantity)

: SalesLineItem





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

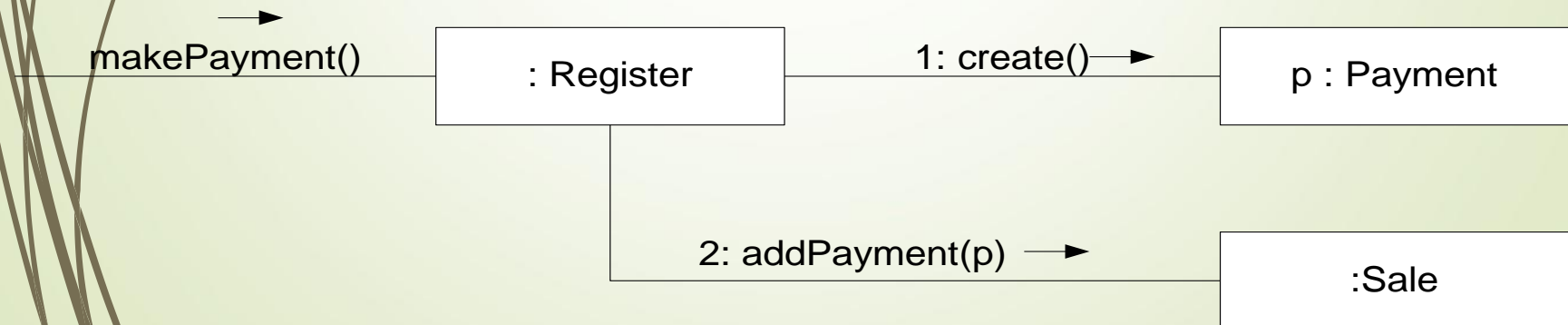
Example (from POS system):

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

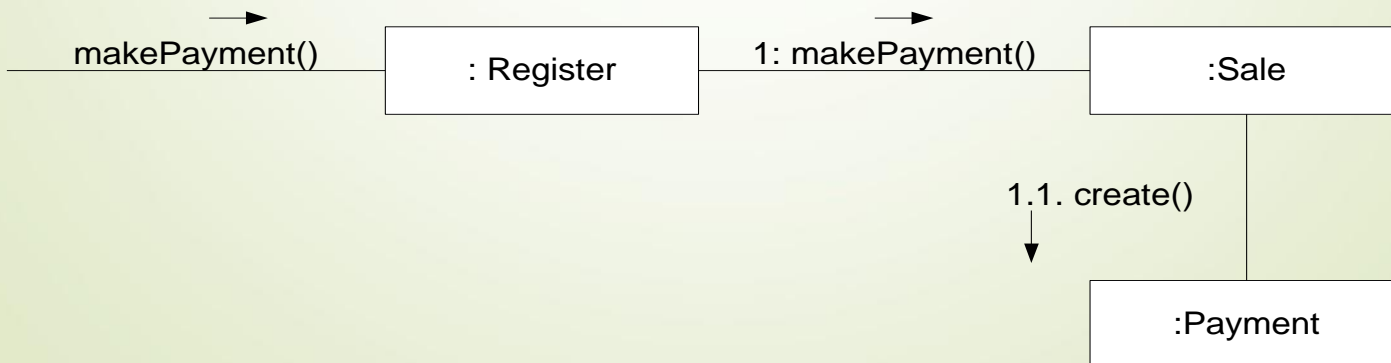
Example (from POS system):

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



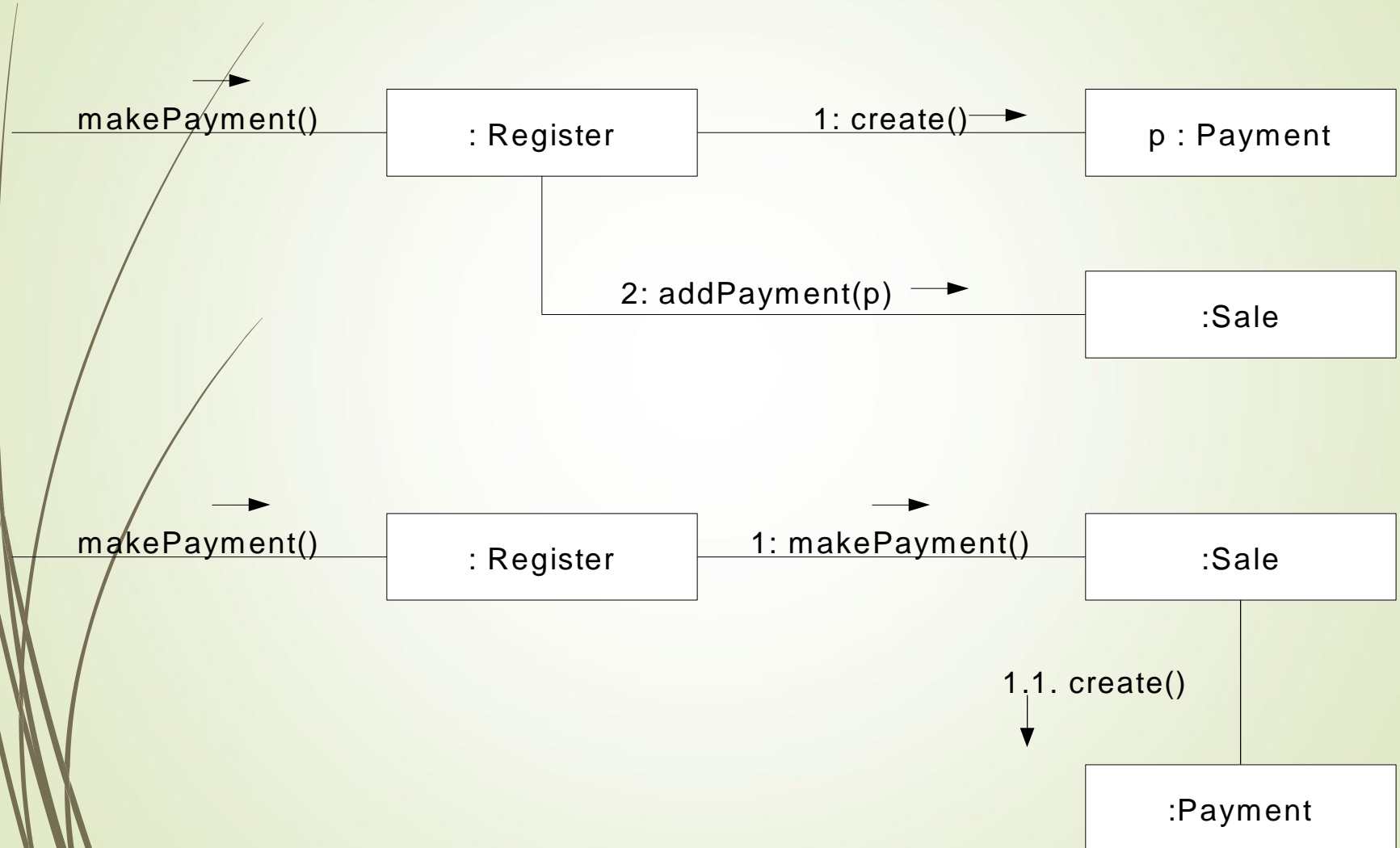


Low Coupling

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

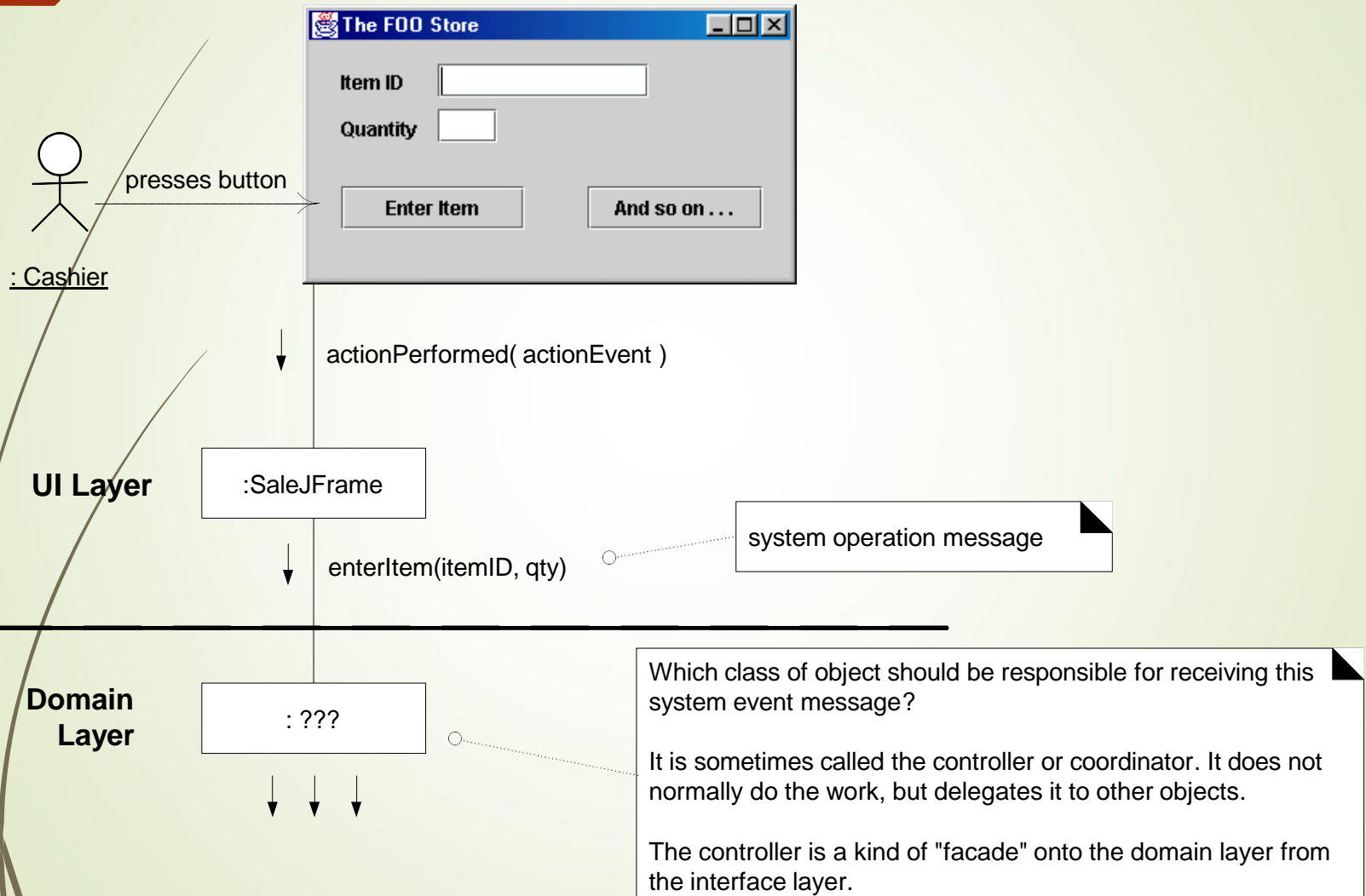




Controller


- 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

A façade controller (Fig. 17.21)







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
- 



Controller

- Façade controller representing overall system – use when there aren't many system events
- Use case controllers – different controller for each use case

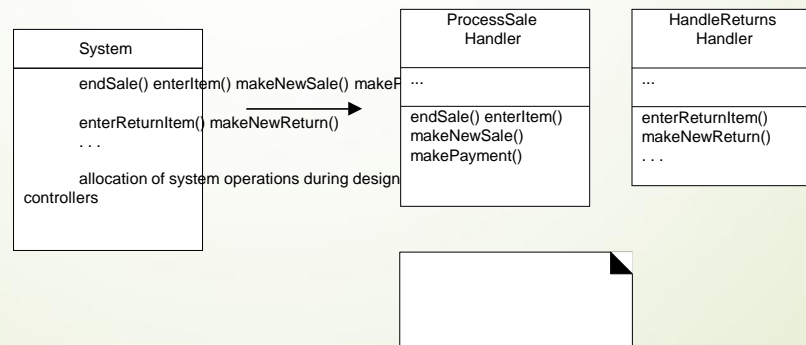
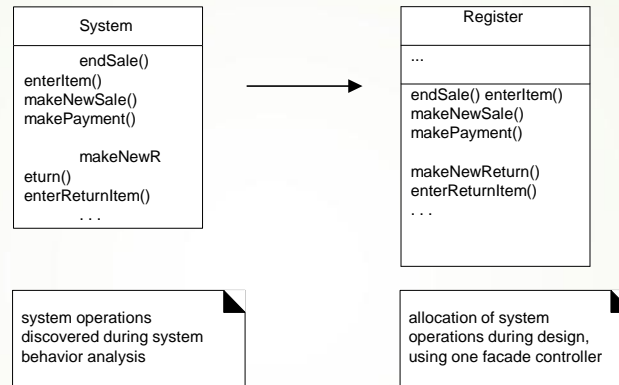


enterItem(id, quantity)

:Register

enterItem(id, quantity)

:ProcessSaleHandler





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



Bloated Controllers

- Controller class is called bloated, if
 - The class is overloaded with too many responsibilities.

Solution – Add more controllers

- Controller class also performing many tasks instead of delegating to other class.

Solution – controller class has to delegate things to others.

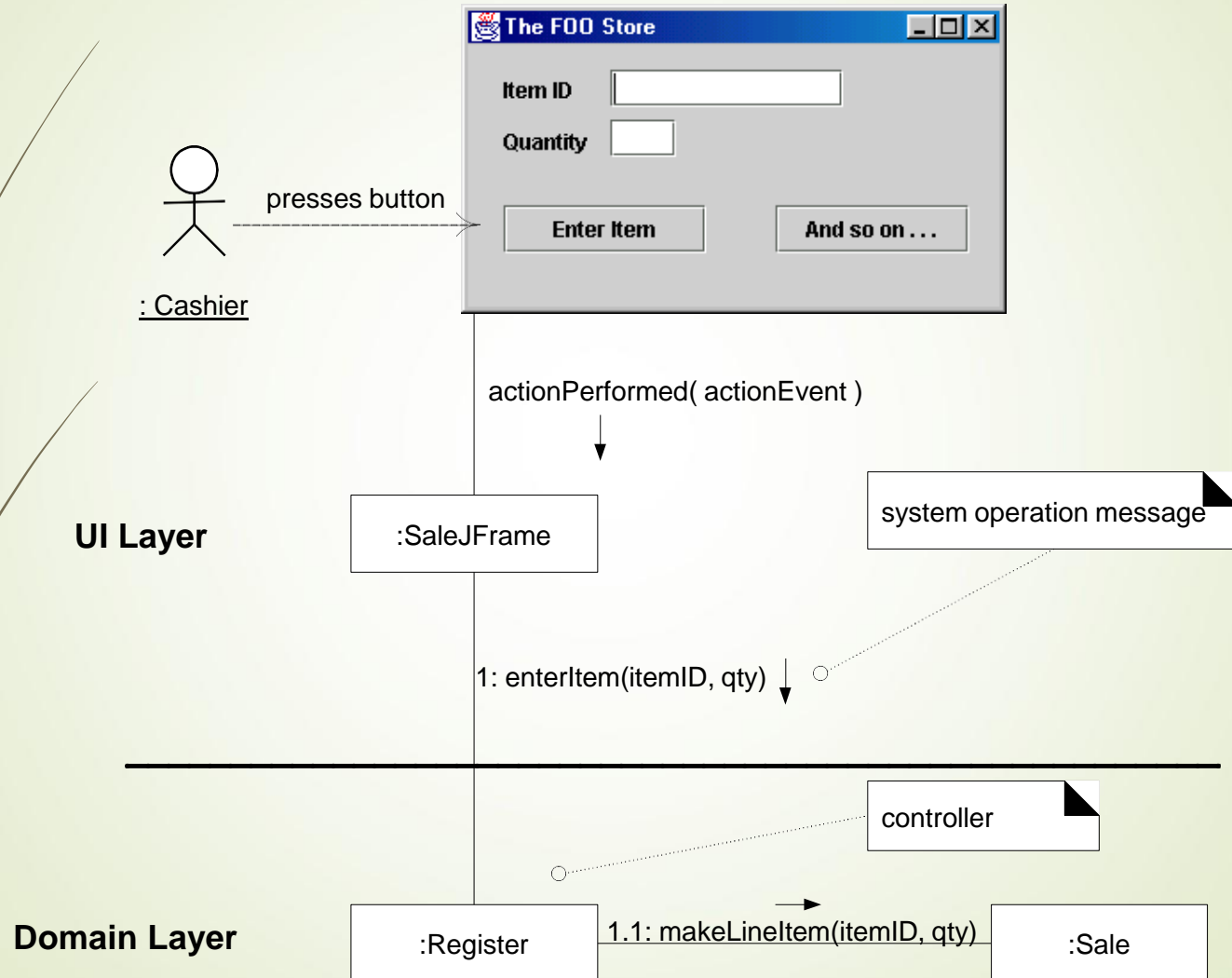


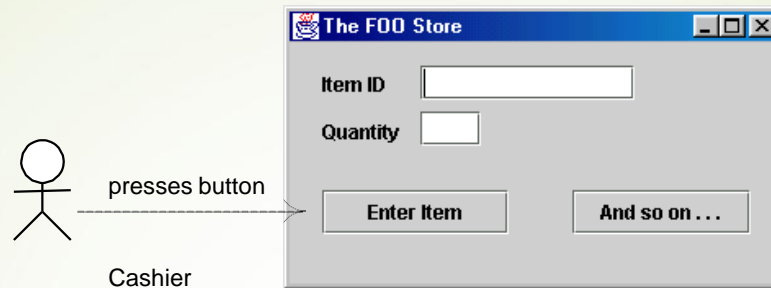
Controller

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

Contrast Fig. 17.24 with 17.25 . . .







presses button

Cashier

actionPerformed(actionEvent)

It is undesirable for an interface layer object such as a window to get

UI Layer

:SaleJFrame

involved in deciding how to handle domain processes.

Business logic is embedded in the presentation layer, which is not useful.

Domain Layer

1: makeLineItem(itemID, qty)

:Sale

SaleJFrame should not send this message.



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

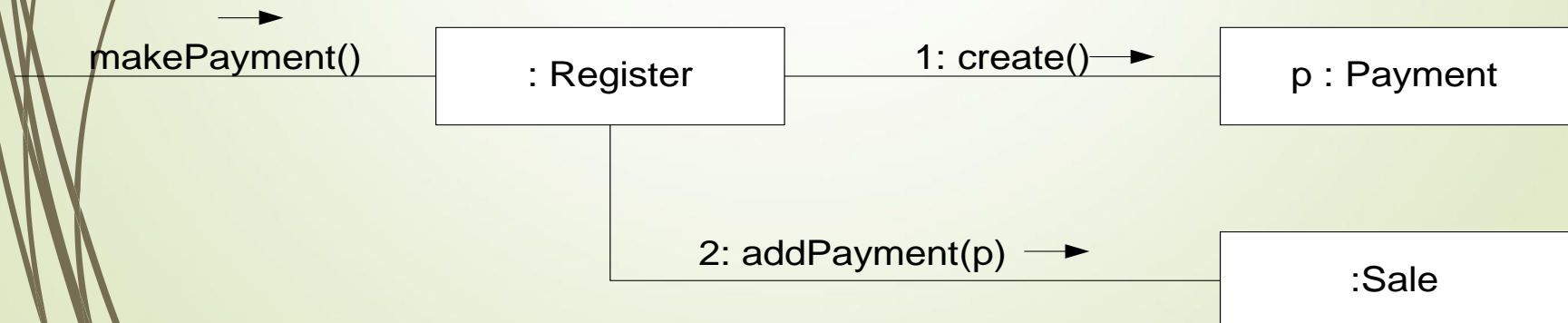
Problems from low cohesion (does many unrelated things or does too much work):

- 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?

1. 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

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

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



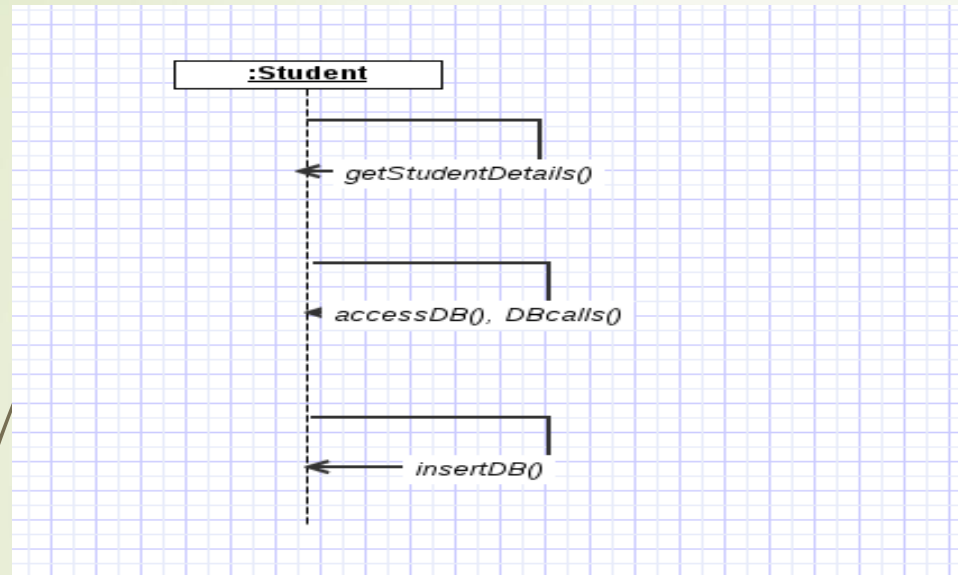
High Cohesion & Low Coupling

- 
- 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

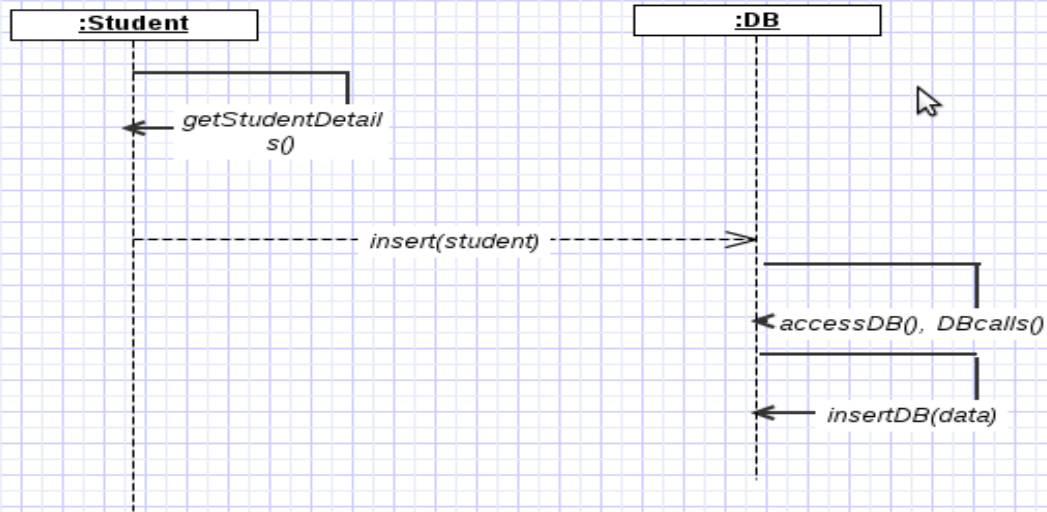
High Cohesion


- How are the operations of any element are functionally related?
- Related responsibilities in to one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
 - Easily understandable and maintainable.
 - Code reuse
 - Low coupling

Example for low cohesion




Example for High Cohesion






High Cohesion Discussion

- Very similar to Low Coupling
 - Often related (but not always)
 - Should be considered in *every* design decision.
- Lower cohesion almost always means:
 - An element more difficult to understand, maintain, or reuse
 - An element more likely to be affected by change
- Low cohesion suggests that more delegation should be used.



Polymorphism

- How to handle related but varying elements based on element type?
 - Polymorphism guides us in deciding which object is responsible for handling those varying elements.
 - Benefits: handling new variations will become easy.
- 

Polymorphism

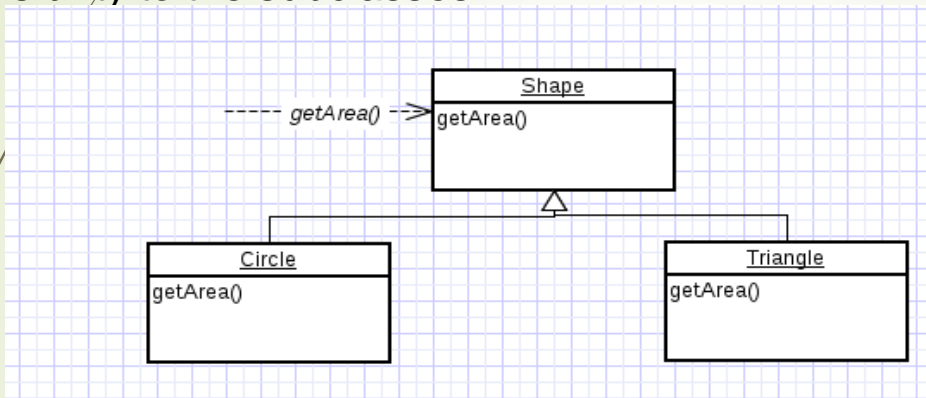
- With respect to implementation, this usually means the use of a super (parent) class or interface
 - Coding to an interface is generally preferred and avoids committing to a particular class hierarchy.
- Code like the following should raise a red flag!

```
Switch creatureType
Case   batType:      print  "Screech!"
Case   cowType:      print  "Moooooooo..."
Case   humanType:    print  "Let's watch
[...]               TV!"
```

- Also see GRASP pattern #8, Protected Variations.

Example for Polymorphism


- the `getArea()` varies by the type of shape, so we assign that responsibility to the subclasses.



- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.



Pure Fabrication

- Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.
 - Provides a highly cohesive set of activities.
 - Behavioral decomposed – implements some algorithm.
 - Examples: Adapter, Strategy
 - Benefits: High cohesion, low coupling and can reuse this class.
- 




Pure Fabrication Discussion

- In other words, getting class concepts from a good domain model or real-life objects \parallel oÛ't always work out well!
- An example of a possible pure fabrication class: `PersistentStorage`
 - May very well not be in the domain model
 - May very well not map to a real-life object
 - But it might be the answer to achieve our goals of low coupling / high cohesion while still having a clear responsibility
- Observe that all of the GoF design patterns are pure fabrications (often of multiple classes)




Example

- 
- Suppose we Shape class, if we must store the shape data in a database.
 - If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
 - So, create a fabricated class DBStore which is responsible to perform all database operations.
 - Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

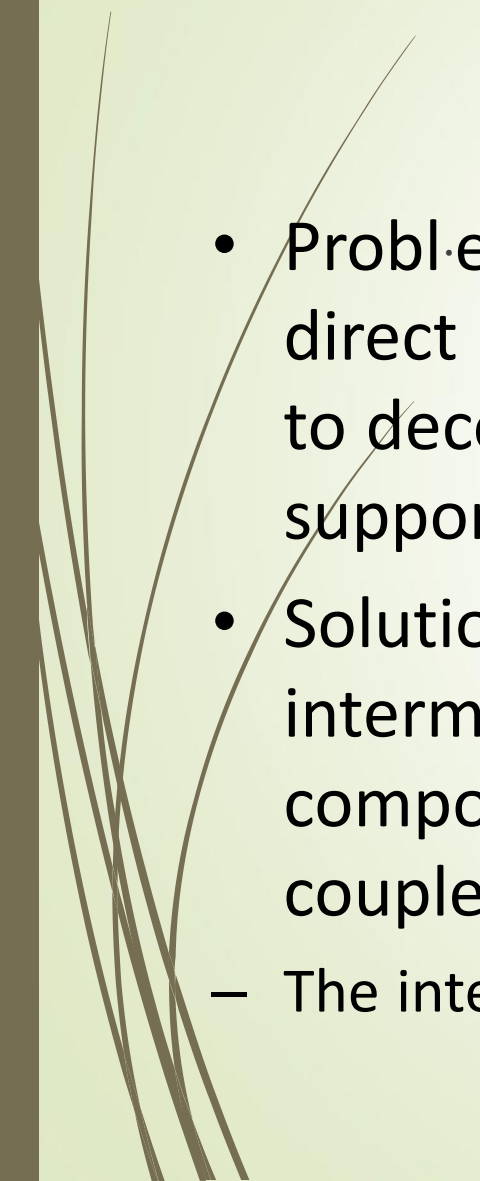


Indirection

- How can we avoid a direct coupling between two or more elements.
 - Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
 - Benefits: low coupling
 - Example: Adapter, Facade, Observer
- 



Indirection

- Problem: Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to decouple objects so that low coupling is supported and reuse potential remains higher?
 - Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.
 - The intermediary creates the indirection.
- 

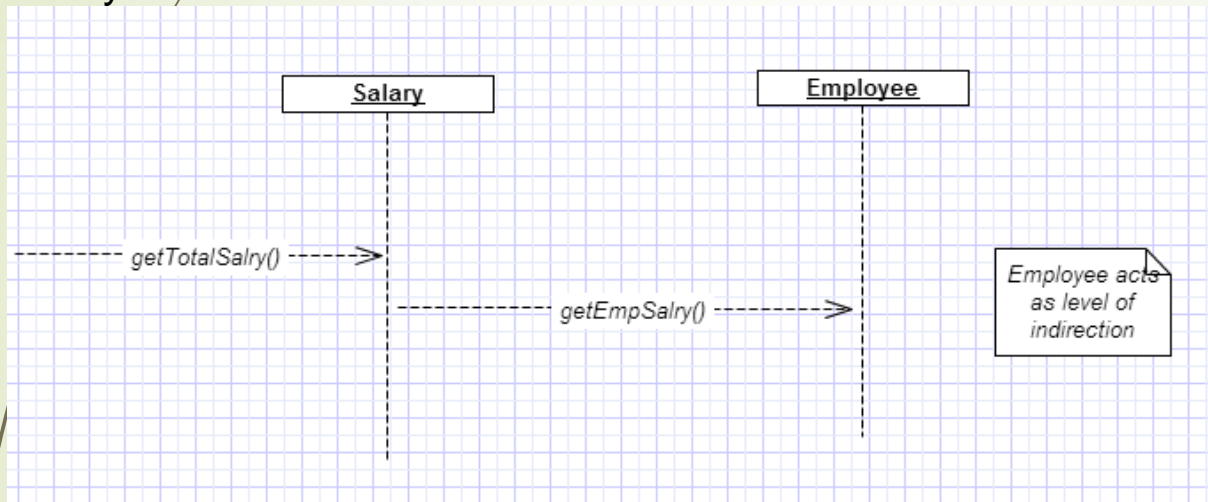


Indirection Discussion

- Often an indirection intermediary is also a pure fabrication.
 - The PersistentStorage example could very well be an indirection between a Sale class and the database.
- The GoF patterns Adapter, Bridge, Façade, Observer, and Mediator all accomplish this.
- The main benefit is lower coupling.

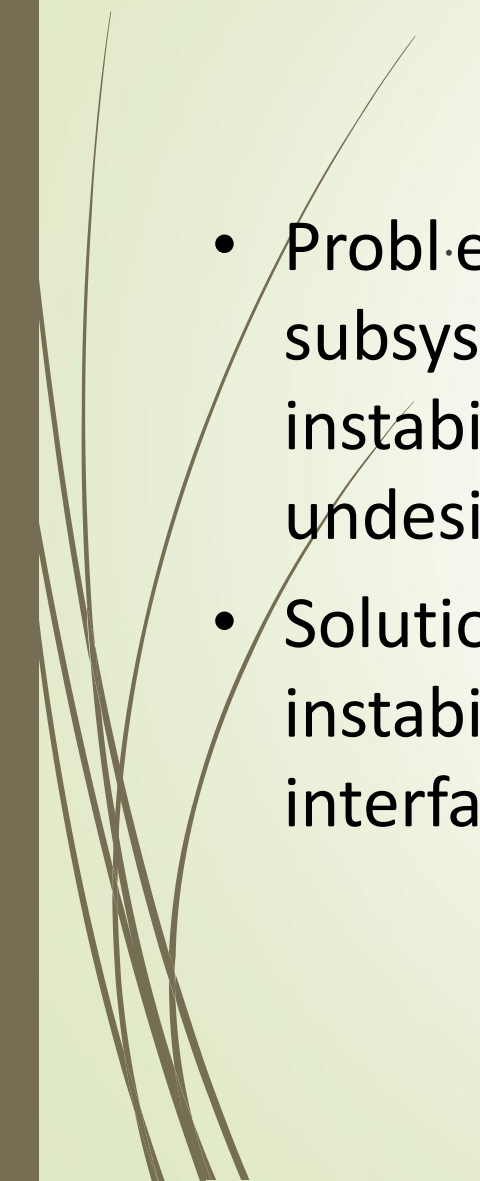
Example for Indirection

- Here polymorphism illustrates indirection
- Class Employee provides a level of indirection to other units of the system.






Protected Variations

- Problem: How to design elements (objects, subsystems, and systems) so that the variations or instability in these elements does not have an undesirable impact on other elements?
 - Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.
- 



Protected Variation

- How to avoid impact of variations of some elements on the other elements.
 - It provides a well defined interface so that there will be no affect on other units.
 - Provides flexibility and protection from variations.
 - Provides more structured design.
 - Example: polymorphism, data encapsulation, interfaces
- 

Protected Variations Discussion

- In the solution Interface is meant in the general sense; but you'll often want to use an interface programming construct (in Java, for example) to implement the solution!
 - Benefits:
 - Easy to extend functionality at PV points
 - Lower coupling
 - Implementations can be updated without affecting clients
 - Reduces impact of change
 - Very similar to the open/closed principle or the concept of information hiding (not the same as data hiding)
 - In Larman's first edition, was the Law of Demeter, but Protected Variations is a more generalized expression
 - Novice developers tend toward brittle designs, intermediate developers tend toward overly fancy and flexible, generalized ones (in ways that never get used). Expert designers choose with insight.

For More Information...

- For more on GRASP, it's hard to beat the depth of Larman's text:
 - Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Third edition, Prentice Hall, 2005.
 - It is well written and there is also substantial material on UML, agile, GoF design patterns, a project-level perspective, and more.
- If GRASP doesn't strike a chord, there are alternative approaches to the general question of "How do I create an OO design using objects?" For example:
 - Wirfs-Brock, Rebecca and McKean, Alan. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley Professional, 2002.
 - Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.