



# BITS Pilani

**BITS Pilani**  
Pilani Campus

Avinash Gautam  
Department of Computer Science and Information Systems

# GRASP: Designing Objects with Responsibilities

Chapters 17

*Applying UML and Patterns*

Craig Larman

# Learning Objectives

---

- Learn about design patterns
- Learn how to apply nine GRASP patterns
- You've learned about static class diagrams and dynamic interaction diagrams
- UML is just notation; now you need to learn how to make effective use of the notation
- UML modeling is an art, guided by principles

# What are patterns? [.1]

---

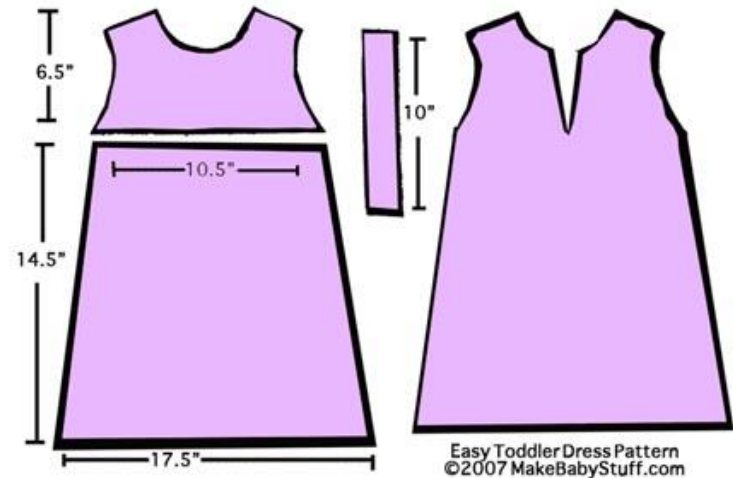
- A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

# Design and dress patterns

We can relate this definition to dress patterns ...

I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern.

Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself



# Patterns in engineering



- How do other engineers find and use patterns?
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - Should software engineers make use of patterns? Why?
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture design

# Definitions

---

- Alexander: “A pattern is a recurring solution to a standard problem, in a context.”
- Larman: “In OO design, a pattern is a named description of a problem and solution that can be applied in new contexts; ideally, a pattern advises us on how to apply the solution in varying circumstances and considers the forces and trade-offs.”

# What are patterns? [..2]

---

- Principles and solutions codified in a structured format describing a problem and a solution
- A named problem/solution pair that can be applied in new contexts
- It is advice from previous designers to help designers in new situations



---

The idea behind design patterns is simple:

Write down and catalog common interactions between objects that programmers have frequently found useful.

Result:

Facilitate reuse of object-oriented code between projects and between programmers.

# Characteristics of Good patterns

---



- It solves a problem
- It is a proven concept
- The solution isn't obvious
- It describes a relationship
- The pattern has a significant human component

**Name :** It must have a meaningful name.

**Problem:** A statement of the problem.

**Context:** This tells us the pattern's applicability.

**Forces:** A description of the relevant forces and constraints and how they interact/conflict with one another..

**Solution:** Static relationships and dynamic rules describing how to realize the desired outcome.

**Consequences:** Implications( good and bad) of using the solution.

**Examples:** One or more sample applications of the pattern .

Which class, in the general case is responsible?

- You want to assign a responsibility to a class
- You want to avoid or minimize additional dependencies
- You want to maximise cohesion and minimise coupling
- You want to increase reuse and decrease maintenance
- You want to maximise understandability
- .....etc.

# GRASP patterns

## General Responsibility Assignment Software Patterns



- Expert
- Creator
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Law of Demeter

Problem:

What is the most basic principle by which responsibilities are assigned in object-oriented design?

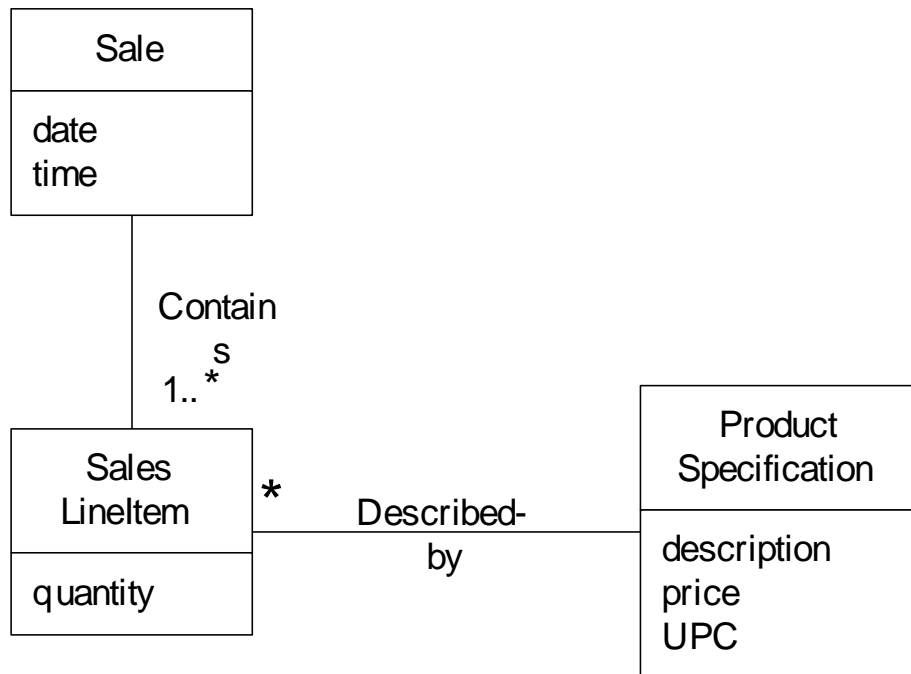
Solution:

Assign a responsibility to the class that has the information necessary to fulfil the responsibility.

# Expert : Example



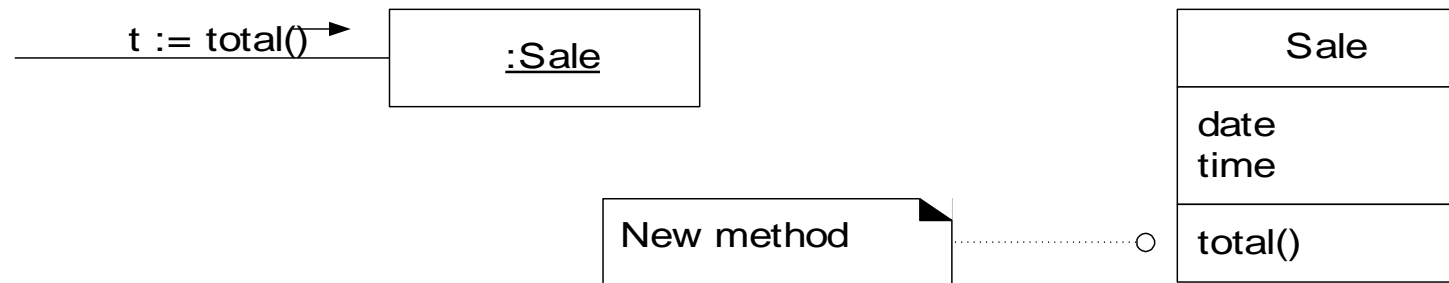
Who is responsible for knowing the grand total of a sale in a typical Point of Sale application?



## Expert : Example

Need all *SalesLineItem* instances and their subtotals. Only *Sale* knows this, so *Sale* is the information expert.

Hence

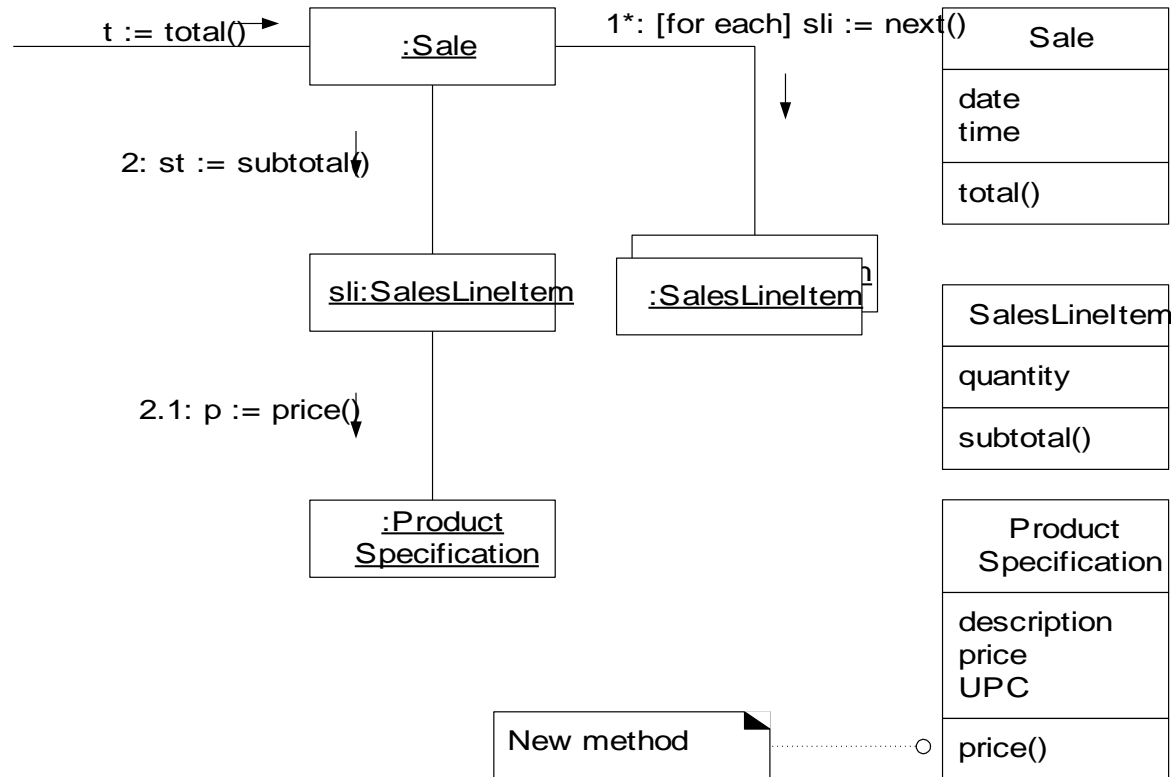




# Expert : Example



But subtotals are needed for each line item (multiply quantity by price).  
By Expert, *SalesLineItem* is expert, knows quantity and has association with *ProductSpecification* which knows price.



# Expert : Example



Hence responsibilities assign to the 3 classes.

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

- Maintain encapsulation of information
- Promotes low coupling
- Promotes highly cohesive classes
- Can cause a class to become excessively complex

## Problem:

Assign responsibility for creating a new instance of some class?

## Solution:

Determine which class should create instances of a class based on the relationship between potential creator classes and the class to be instantiated.

Who has responsibility to create an object?

By creator, assign class B responsibility of creating instance of class A if

*B aggregates or contains A objects*

*B records instances of A objects*

*B closely uses A objects*

*B has the initializing data for creating A objects*

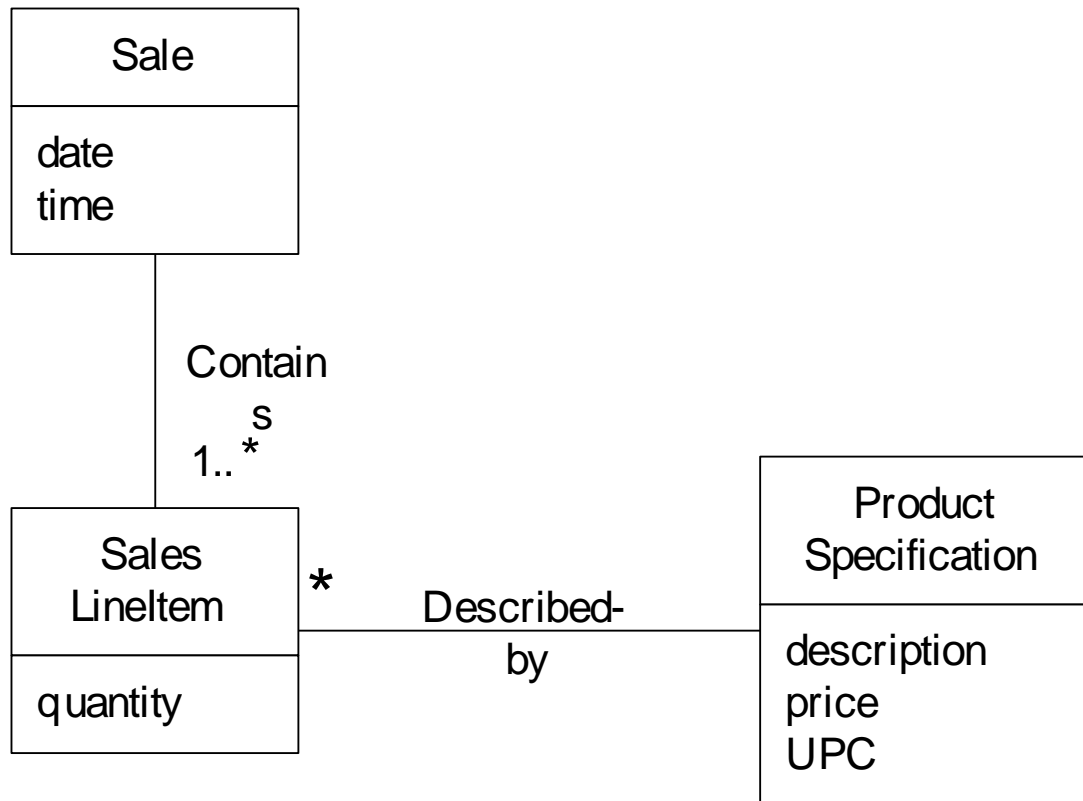
where there is a choice, prefer

*B aggregates or contains A objects*

## Creator : Example

Who is responsible for creating *SalesLineItem* objects?

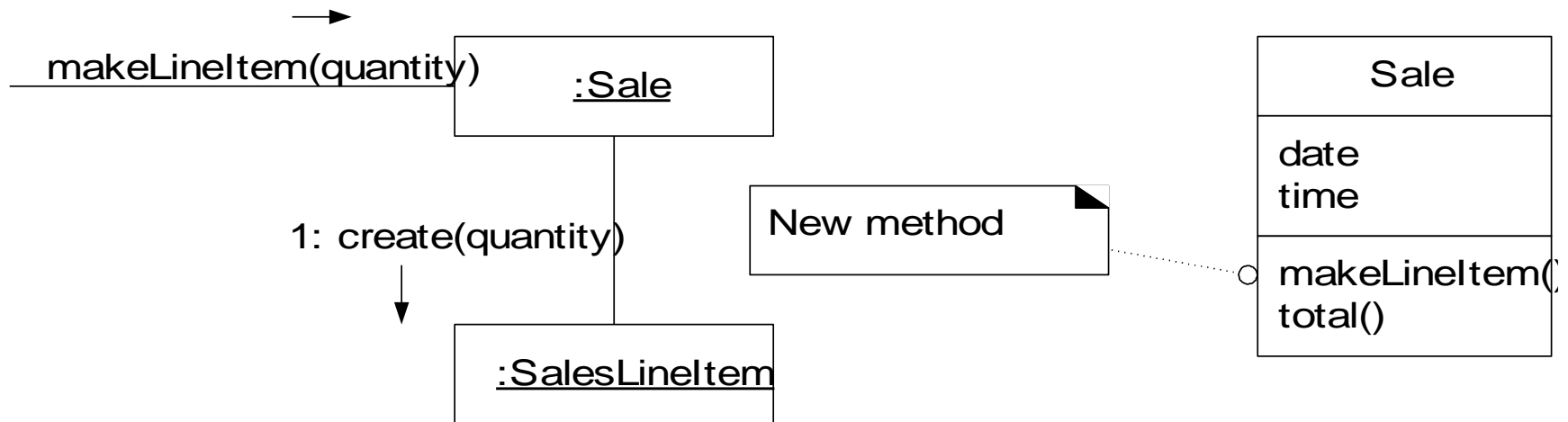
Look for a class that aggregates or contains *SalesLineItem* objects.



# Creator : Example

Creator pattern suggests Sale.

Collaboration diagram is



# Creator

---

- Promotes low coupling by making instances of a class responsible for creating objects they need to reference
- By creating the objects themselves, they avoid being dependent on another class to create the object for them



# Low Coupling

---

## Problem:

To support low dependency and increased reuse?

## Solution:

Assign responsibilities so that coupling remains low.

In object oriented languages, common form of coupling from TypeX to TypeY include:

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- TypeX has a method which references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.

# Low coupling

---

- Classes are easier to maintain
- Easier to reuse
- Changes are localised

## Low Coupling

---

How can we make classes independent of other classes?

changes are localised

easier to understand

easier to reuse

Who has responsibility to create a *payment*?

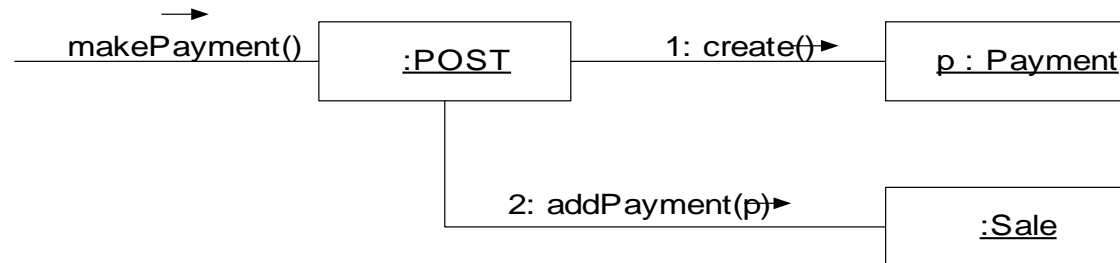
Payment

POST

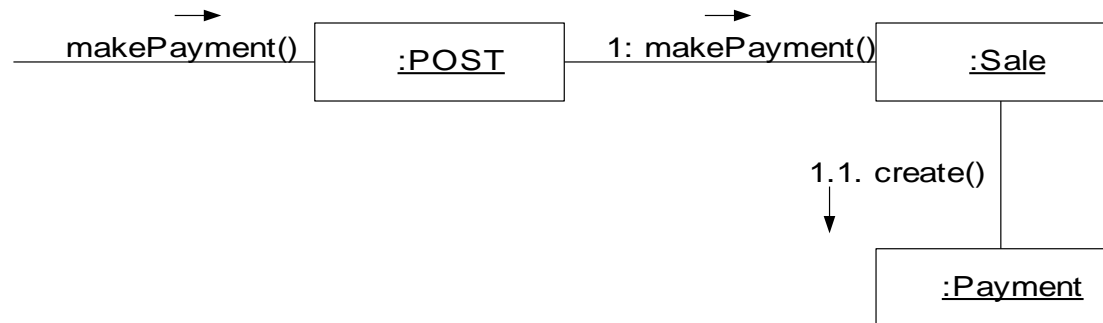
Sale

## Two possibilities:

### 1. Post



### 2. Sale



Low coupling suggests *Sale* because *Sale* has to be coupled to *Payment* anyway (*Sale* knows its *total*).

# High Cohesion

---

Problem:

To keep complexity manageable?

Solution:

Assign responsibilities so that cohesion remains high.

## Some examples:



- Very Low Cohesion: A Class is solely responsible for many things in very different functional areas
- Low Cohesion: A class has sole responsibility for a complex task in one functional area.
- High Cohesion. A class has moderate responsibilities in one functional area and collaborates with classes to fulfil tasks.

# High cohesion

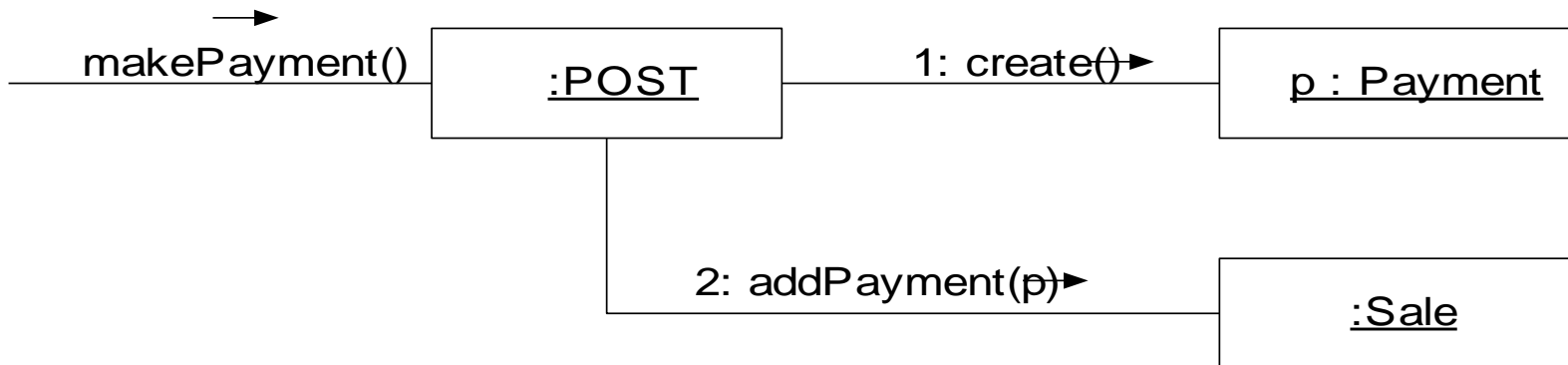
---

- Classes are easier to maintain
- Easier to understand
- Often support low coupling
- Supports reuse because of fine grained responsibility



Who has responsibility to create a *payment*?

## 1.Post



looks OK if *makePayment* considered in isolation, but adding more system operations, *Post* would take on more and more responsibilities and become less cohesive.

Giving responsibility to *Sale* supports higher cohesion in *Post*, as well as low coupling.



## Problem:

To assign responsibility for handling a system event?

## Solution:

If a program receive events from external sources other than its graphical interface, add an event class to decouple the event source(s) from the objects that actually handle the events.

The Controller pattern provides guidance for generally acceptable choices.

Assign the responsibility for handling a system event message to a class representing one of these choices:

1. The business or overall organization (a façade controller).
2. The overall "system" (a façade controller).
3. An animate thing in the domain that would perform the work (a role controller).
4. An artificial class (Pure Fabrication representing the use (a use case controller).

## Benefits:

Increased potential for reuse. Using a controller object keeps external event sources and internal event handlers independent of each other's type and behaviour.

Reason about the states of the use case. Ensure that the system operations occurs in legal sequence, or to be able to reason about the current state of activity and operations within the use case.

# Controller : Example

---

System events in Buy Items use case

`enterItem()`

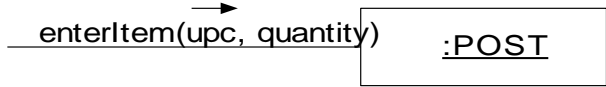
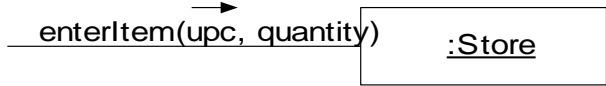
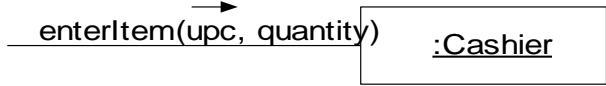
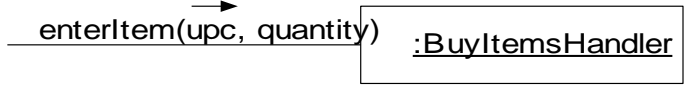
`endSale()`

`makePayment()`

who has the responsibility for *enterItem()*?

# Controller : Example

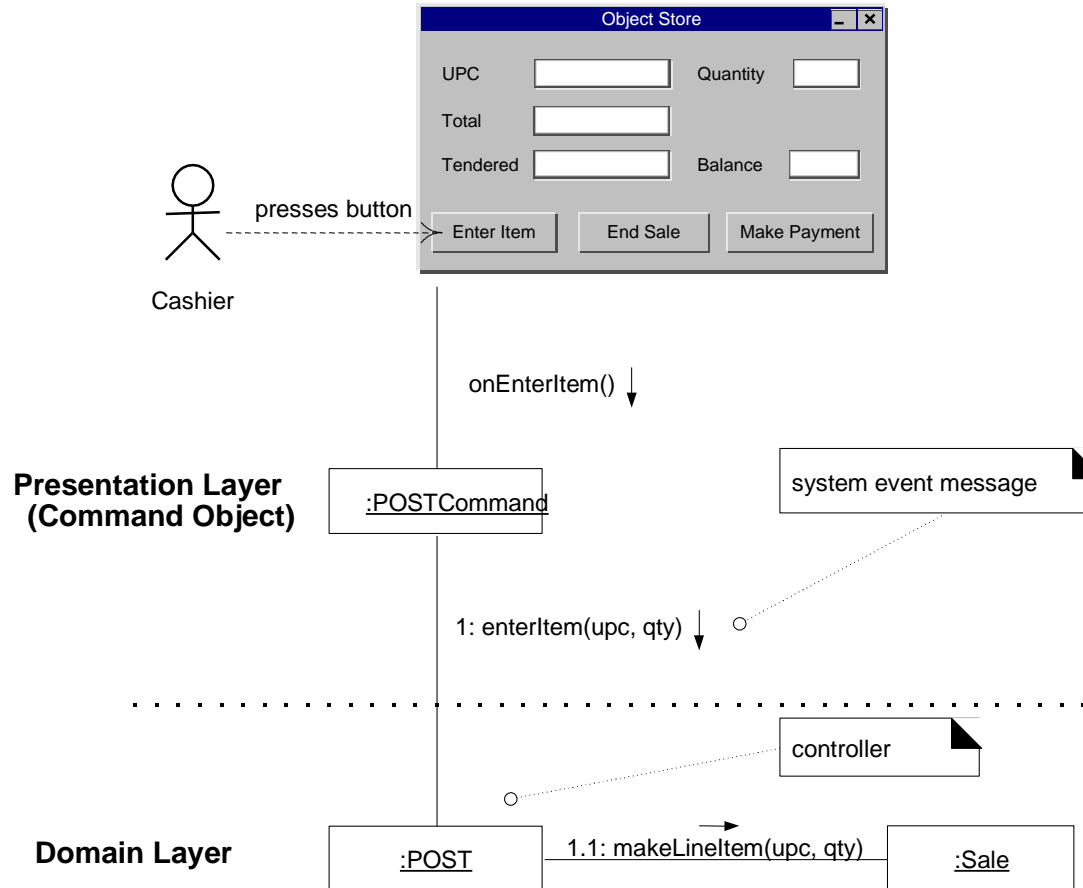
By controller, we have 4 choices

the overall system	Post	
the overall business	Store	
someone in the real world who is active in the task	Cashier	
an artificial handler of all system events of a use case	BuyItemsHandler	

The choice of which one to use will be influenced by other factors such as cohesion and coupling

# Good design

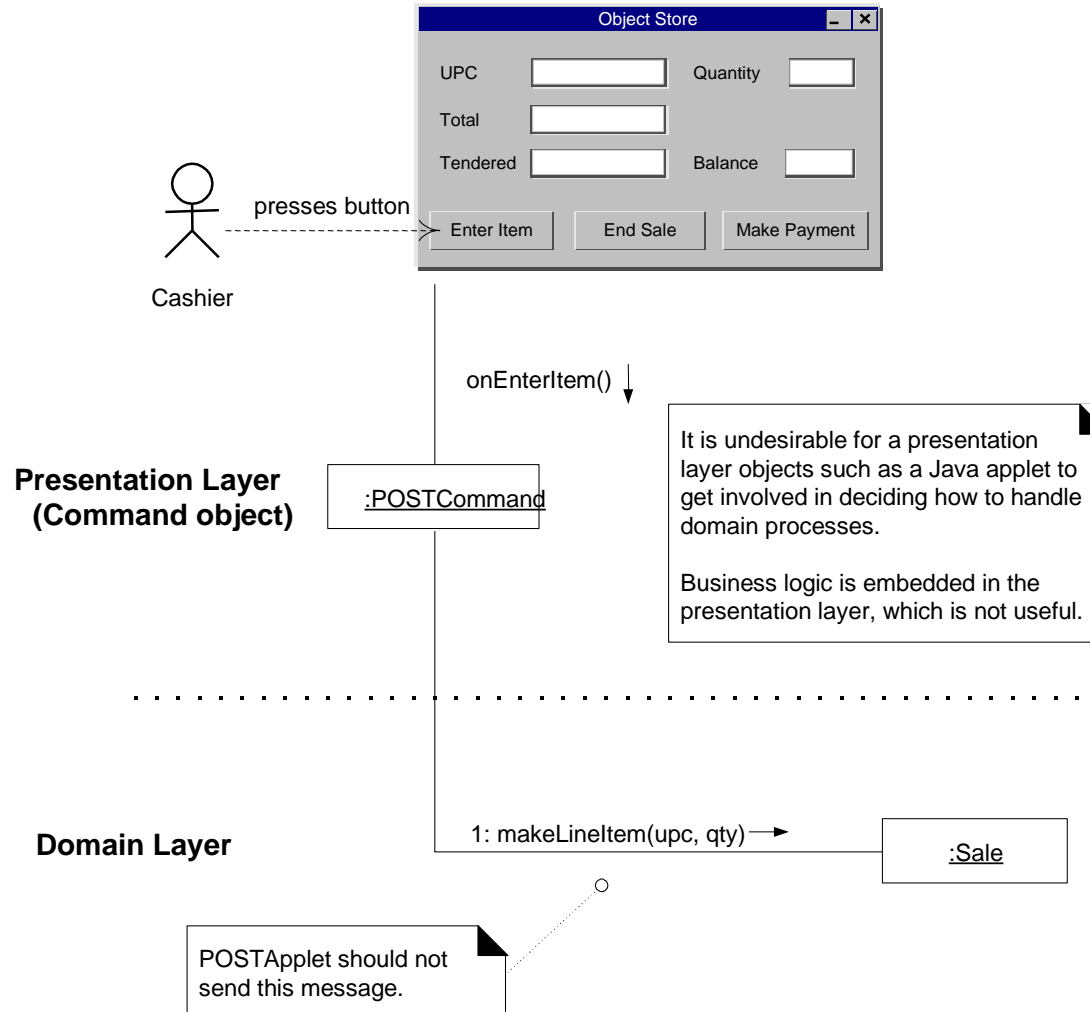
- presentation layer decoupled from problem domain





# Bad design

- presentation layer coupled to problem domain



# Controller

---

- Using a controller object keeps external event sources and internal event handlers independent of each other's type and behaviour
- The controller objects can become highly coupled and uncohesive with more responsibilities

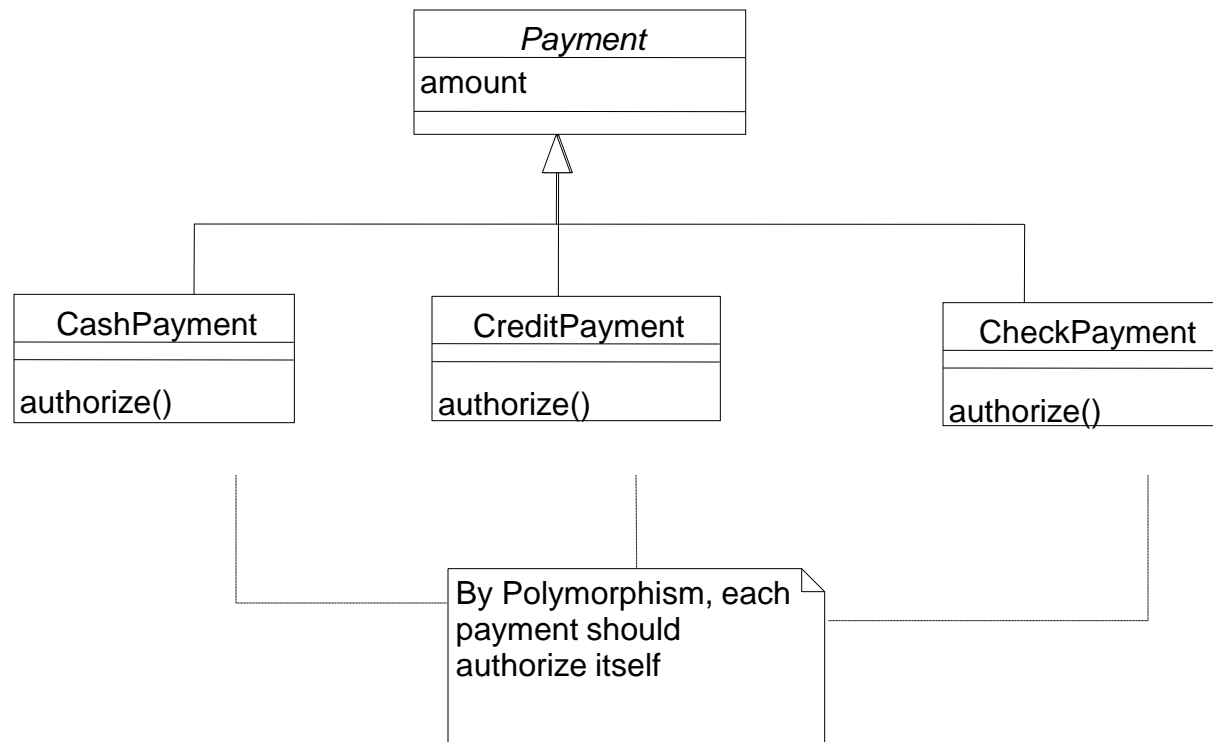
Problem:

To handle alternatives based on types?

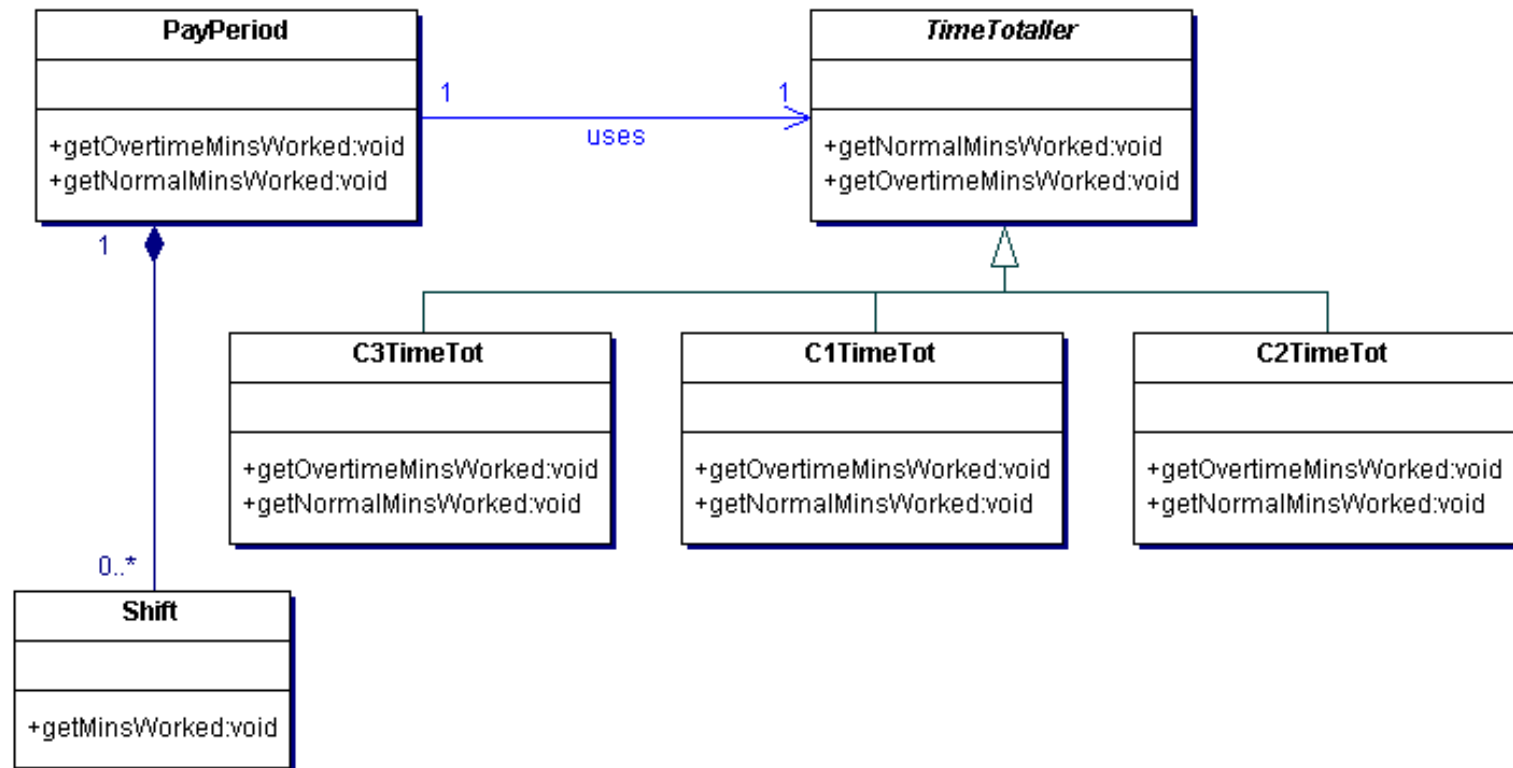
Solution:

When alternate behaviours are selected based on the type of an object, use polymorphic method call to select the behaviour, rather than using if statement to test the type.

# Polymorphism : Example



## Example : Polymorphism



# Polymorphism

---

- Easier and more reliable than using explicit selection logic
- Easier to add additional behaviours later on
- Increased the number classes in a design
- May make the code less easier to follow

# Pure Fabrication

---

Problem:

To not violate High Cohesion and Low Coupling?

Solution:

Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain, in order to support high cohesion, low coupling, and reuse.

## Benefits:

High cohesion is supported because responsibilities are factored into a class that only focuses on a very specific set of related tasks.

Reuse potential may be increased because of the presence of fine grained Pure Fabrication classes.

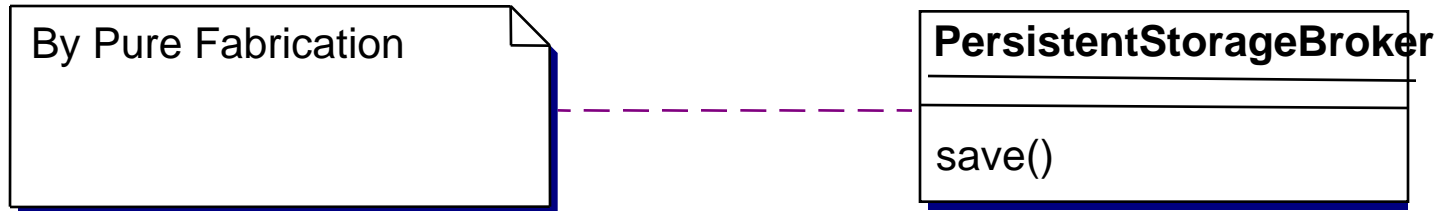


## Example

Suppose, in the point of sale example, that support is needed to save Sale instances in a relational database. By Expert, there is some justification to assign this responsibility to Sale class. However.

- The task requires a relatively large number of supporting database-oriented operations and the Sale class becomes incohesive.
- The sale class has to be coupled to the relational database increasing its coupling.
- Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the Sale class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

## Pure Fabrication : Example



- The Sale remains well design, with high cohesion and low coupling
- The PersistentStorageBroker class is itself relatively cohesive
- The PersistentStorageBroker class is a very generic and reusable object

# Pure Fabrication

---

- Preserves low coupling and high cohesion of classes
- Improve reusability of classes

## Problem:

To avoid direct coupling?

To de-couple objects so that Low coupling is supported and reuse potential remains high?

## Solution:

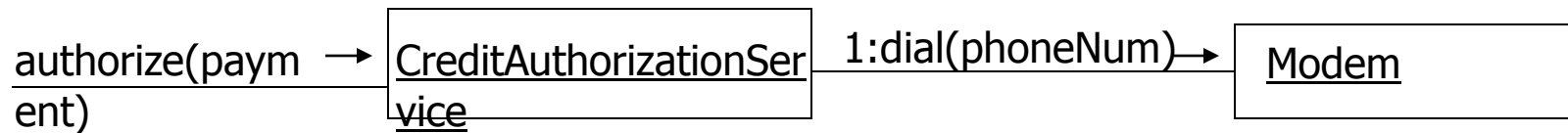
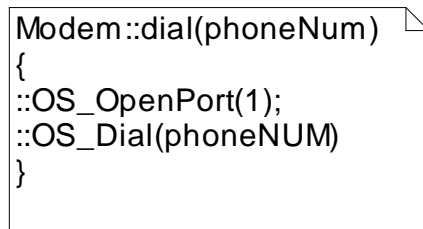
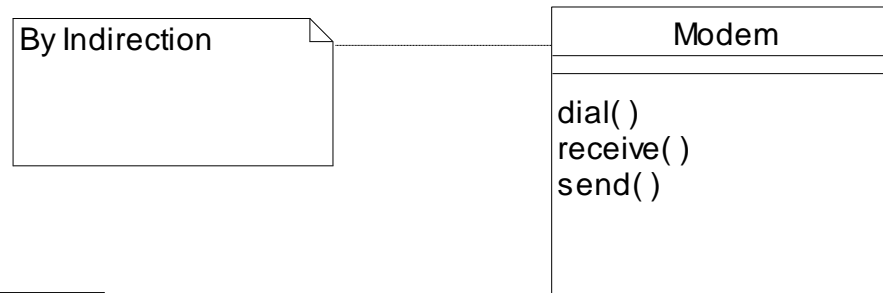
Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

## Example : PersistentStorageBroker

---

The Pure fabrication example of de-coupling the *Sale* from the relational database services through the introduction of a *PersistentStorageBroker* is also an example of assigning responsibilities to support Indirection. The *PersistentStorageBroker* acts as a intermediary between the *Sale* and database

# Indirection : Example



# Indirection

---

- Low coupling
- Promotes reusability

## Problem:

To avoid knowing about the structure of indirect objects?

## Solution:

If two classes have no other reason to be directly aware of each other or otherwise coupled, then the two classes should not directly interact.



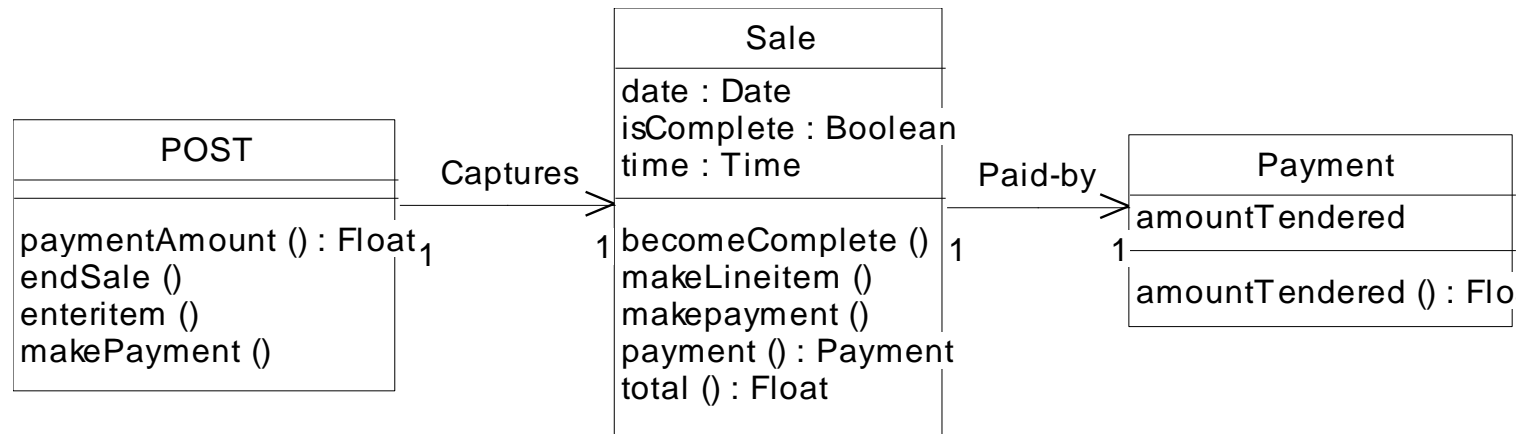
# Law of Demeter

---

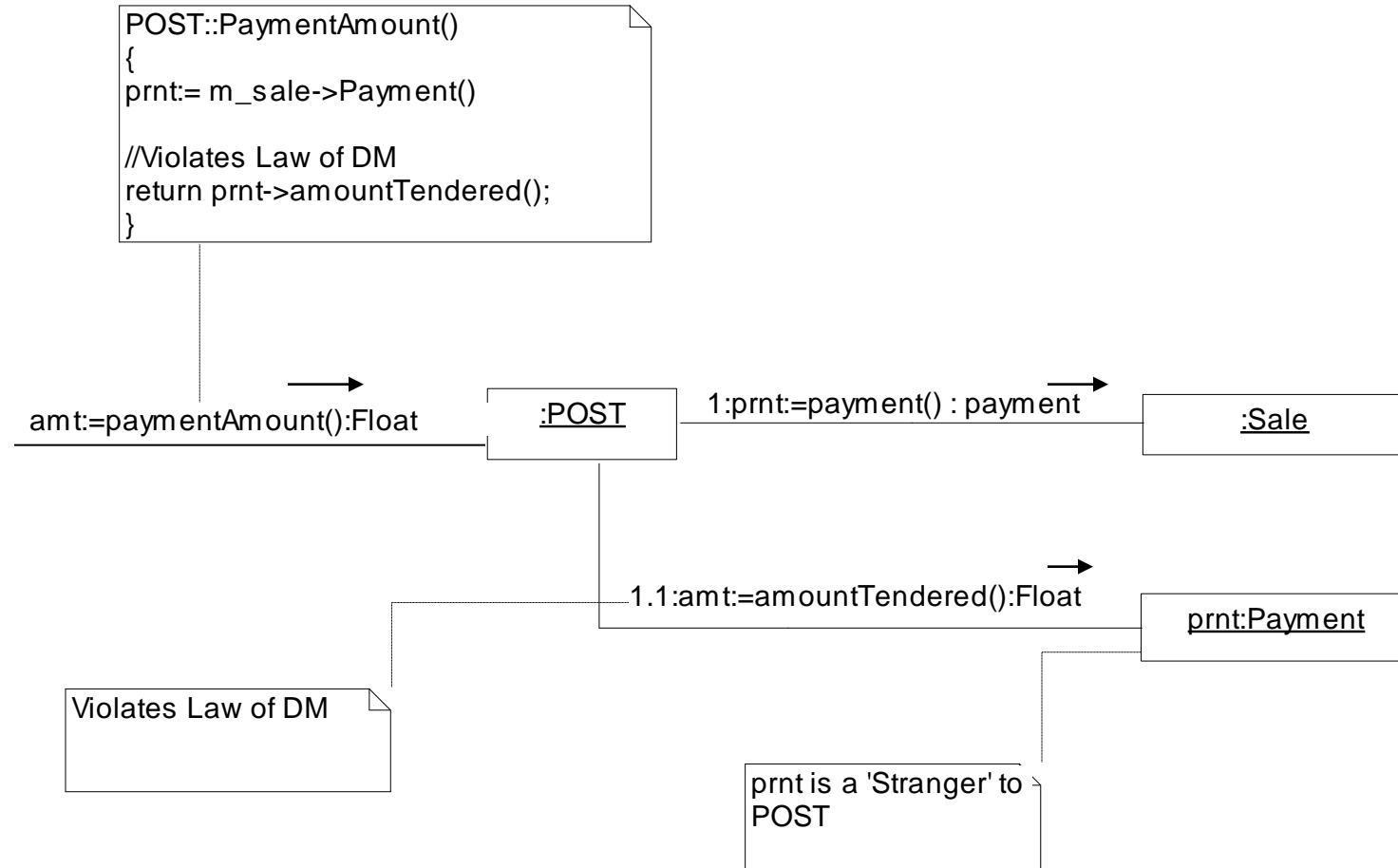
It states that within a method, messages should only be sent to the following objects:

- The *this* object (or *self*)
- A parameter of the method
- An attribute of *self*
- An element of a collection which is an attribute of *self*
- An object created within the method

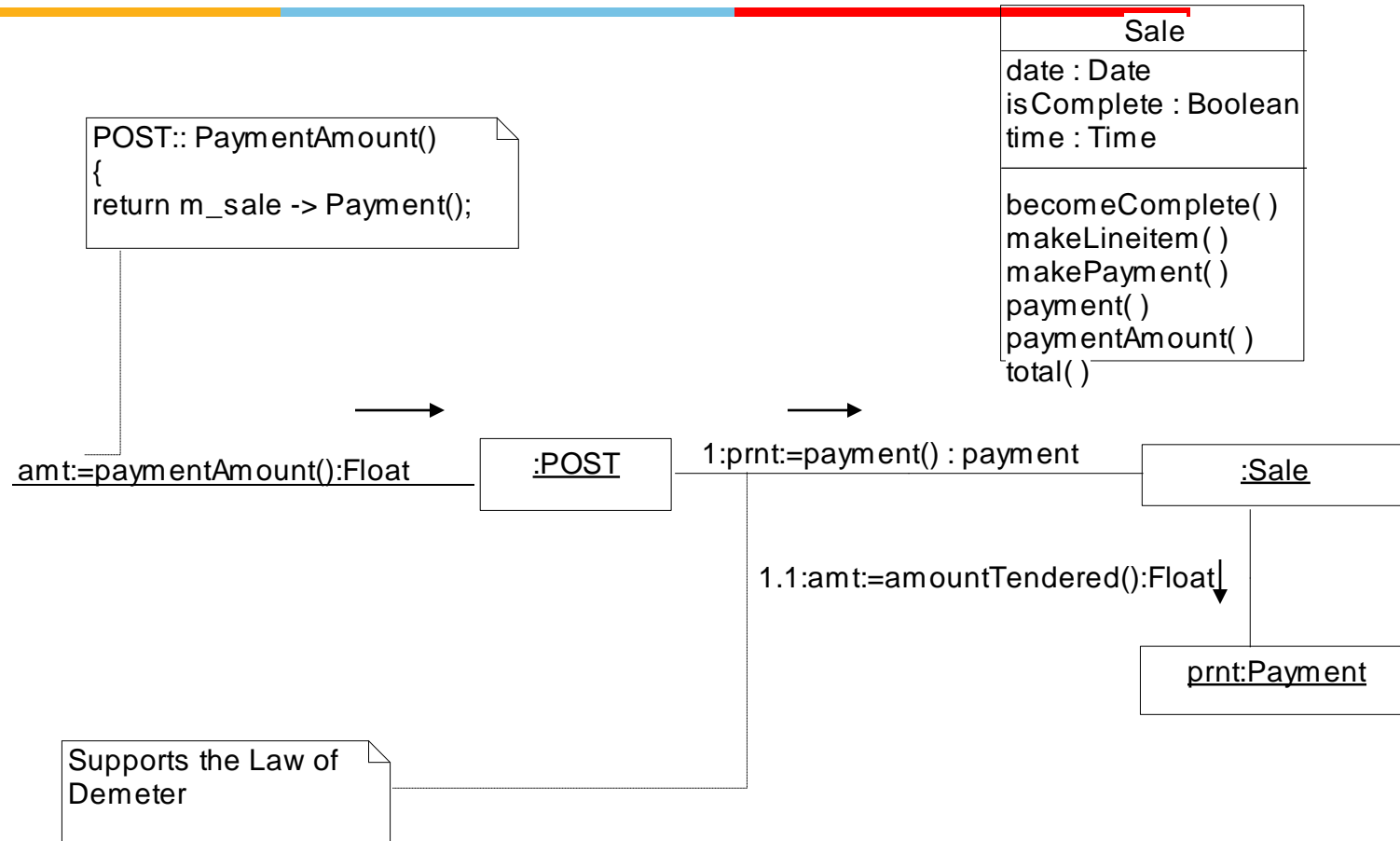
# Law of Demeter : Example



# Violates Law of Demeter : Example



# Support Law of demeter

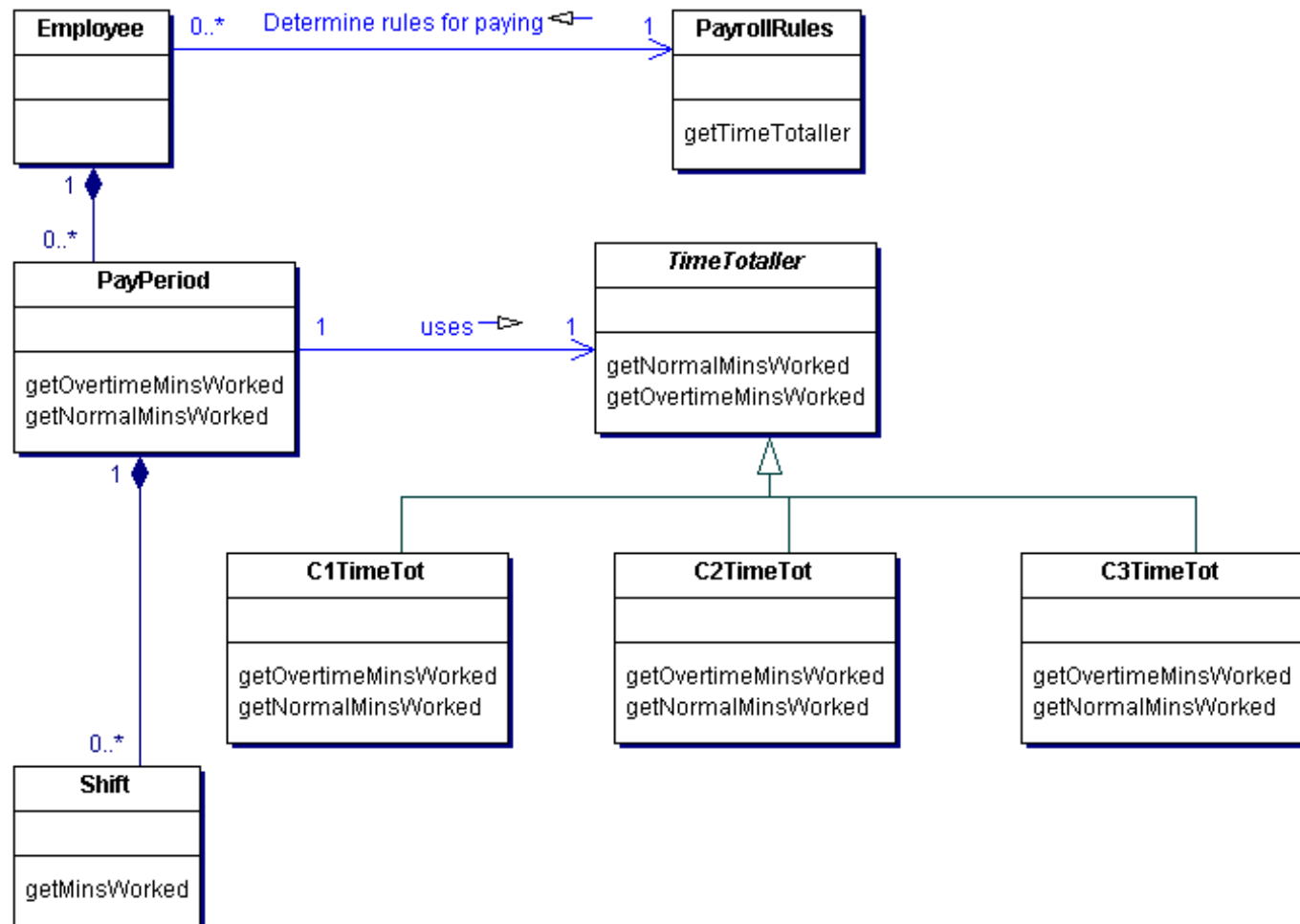


# Law of Demeter

---

- Keeps coupling between classes low and makes a design more robust
- Adds a small amount of overhead in the form of indirect method calls

## Law of Demeter – Time totalling example



## Time totalling example

Employee - Instances of the Employee class represent an employee.

PayrollRules – The rules for paying an employee vary with the laws that apply to the location where the employee works. Instances of the PayrollRules class encapsulate the pay rules that apply to an employee.

PayPeriod – Instances of the Payperiod class represent a range of days for which an employee is paid in the same pay slip.

Shift – Instances of the Shift class represent ranges of time that the employee worked.

TimeTotaller – The Timetotaller class is an abstract class that the PayPeriod class uses to break the total hours worked during a pay period into normal and overtime minutes.

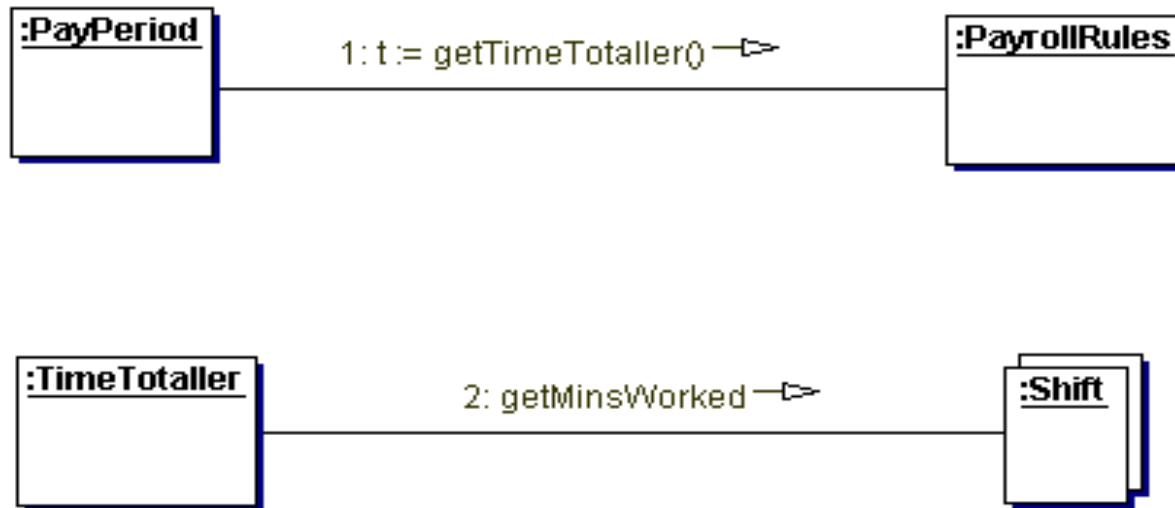
C1TimeTot,C2TimeTot,C3TimeTot – Concrete subclasses for different location of TimeTotaller that encapsulate the rules for breaking total minutes worked into normal and overtime minutes worked.

The following interaction must occur:

- The pay period must become associated with an instance of the subclass of TimeTaller appropriate for the employee when the PayPeriod object is created.
- The TimeTaller object must be able to examine each shift in the pay period to learn the number of minutes worked in each shift.



## Bad time-totalling collaboration



PayPeriod class has no reason to know anything about the PayrollRules class

For TimeTaller to have direct access to the collection of shifts that it needs  
implies violation of the Shift class's encapsulation of how it aggregates  
Collection of shifts -- resulting in higher level of coupling

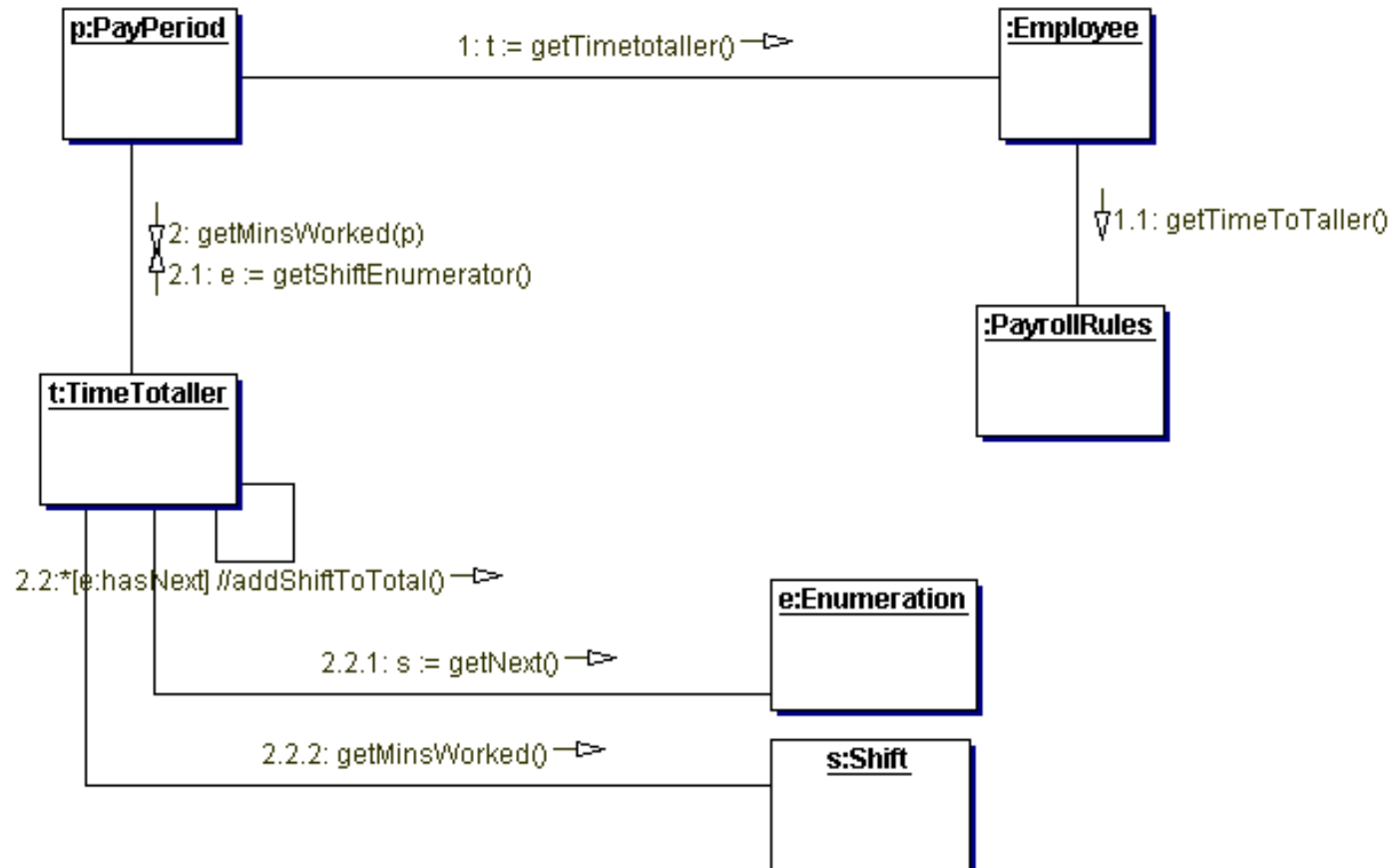
# Good time-totalling collaboration

---

To preserve the level of cohesion and coupling a less direct interaction may be used.

This is done as shown by the following collaboration diagram and the creation of additional methods.

## Good time-totalling collaboration



## Law of Demeter – Time totalling example with added operations

