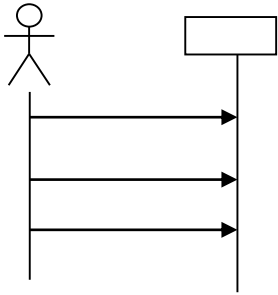


Interaction Modeling



*Please read
Chapter 8*

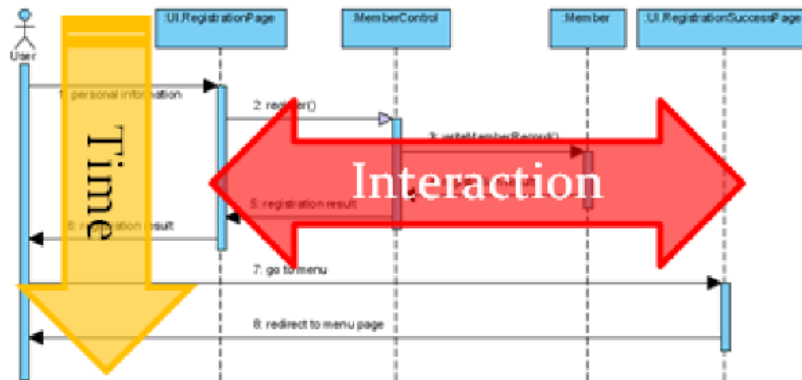
Dr. Abdelkarim Erradi

Dept. of Computer Science & Engineering

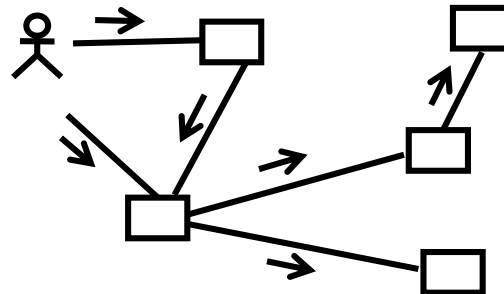
QU

Outline

- Sequence Diagram



- Communication Diagram



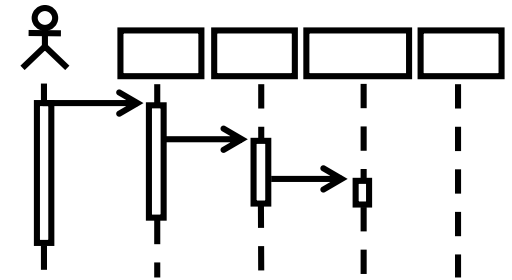
Interaction Diagrams

- Interaction diagrams are used to model the **dynamic aspects** of a system
- Dynamic aspects of the system
 - Messages moving among objects
 - Flow of control among objects
 - Sequences of events
- The main UML diagram to model interactions is the Sequence Diagram

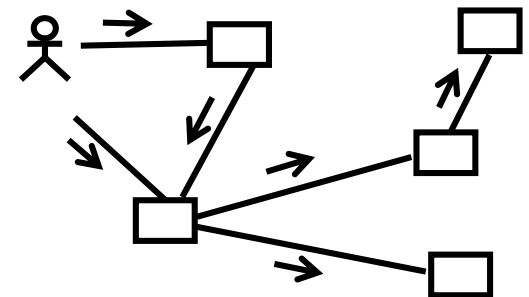


Interaction Diagrams

- Design Sequence Diagram
 - **Time oriented view** emphasize the time ordering of the interactions. The diagram shows:
 - The **objects participating** in the interaction.
 - The **sequence of messages exchanged**.
- Communication Diagram
 - Shows how the objects related to each other
 - Emphasize the **structural organization** of the objects participating in interactions:
 - The objects participating in the interaction.
 - **Links between the objects**.
 - Messages passed between the objects.



Sequence Diagrams



Communication Diagrams

System-Level Sequence Diagrams

System Sequence Diagrams

- Use case scenarios describe how external actors interact with the system...
 - The actor generates system events to a system, requesting some system operation to handle the event.
- A System Sequence Diagram (SSD) is a diagram that shows, *for one particular scenario of a use case*:
 - the **events that the actor generates**,
 - **their order**
 - the **system response** to such events.
- The **system is regarded as a black box** and the functionalities are expressed from a user's perspective

SSD Example – Process Sale Scenario

Message with parameters to pass data to the system

System as a black box

A loop indicates any recurring events

Process Sale Scenario

: Cashier

:System

makeNewSale

loop

[more items]

enterItem(itemID, quantity)

description, total

endSale

total with taxes

Actor (cashier)

System

Cashier starts a new sale

Displays a transaction entry area.

Cashier enters item identifier

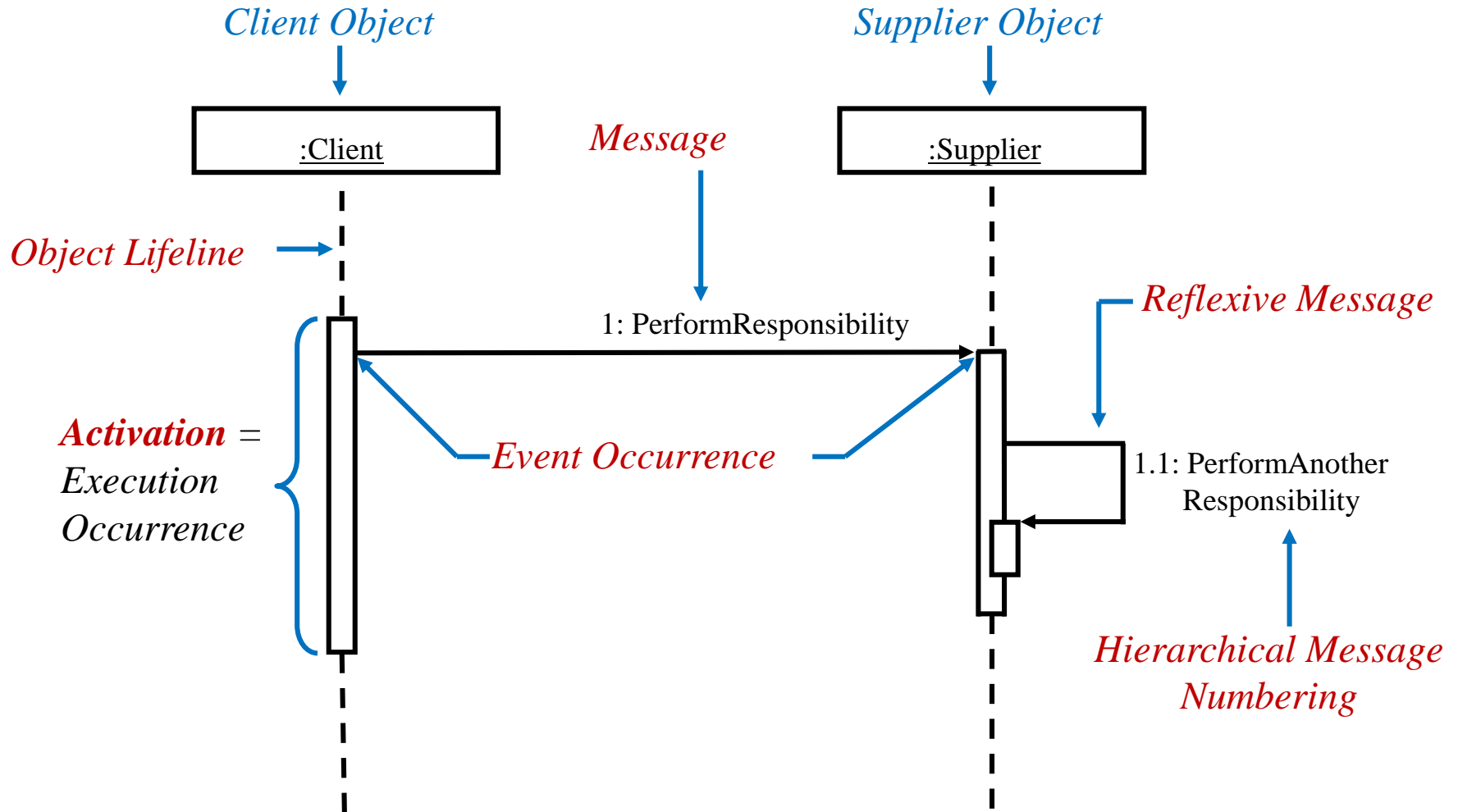
System creates a line item record and retrieves and presents item description (including price) and running total for the transaction.

The above step is repeated until the cashier signals that the transaction is complete.

The system calculates total with taxes and presents the results.

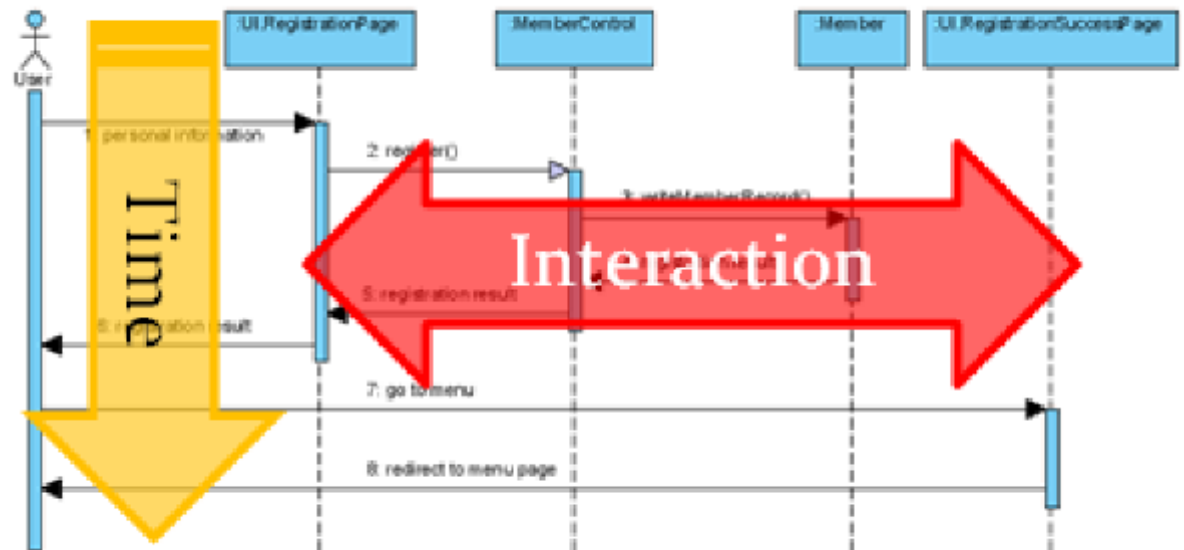
Return Value associated with previous message; The return line is optional if nothing is returned

The Anatomy of Sequence Diagrams

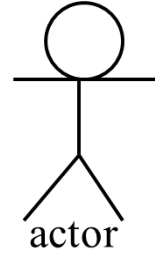


The Anatomy of Sequence Diagrams (cont.)

- DSD Visualize interactions between objects
 - Interactions shown in horizontal direction
 - Time in shown in vertical direction
- Activation is a rectangle on the lifeline, it indicate the time where execution takes place on that object.



SSD Elements



- **Actor:** An Actor is modeled using the ubiquitous symbol, the stick figure.
- **Lifeline:** The Lifeline identifies the existence of the object over time. The notation for a Lifeline is a vertical dotted line extending from an object.
- **System** behaves as “Black Box”.

A rectangular box with a double border. Inside the box, the text ":System" is written, with "System" underlined.

 - In the design phase we will simply opened up the black box to show details of the object interactions to handle the actor requests
- **Message:** Messages, modeled as horizontal arrows indicating the interactions between the actor and the system.


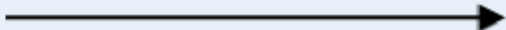

messageName(argument)



Message Types

- Message, often a method call, carry information from the actor to the system (and vice versa)
 - A message is numbered and labelled and can have an argument list and a return value.

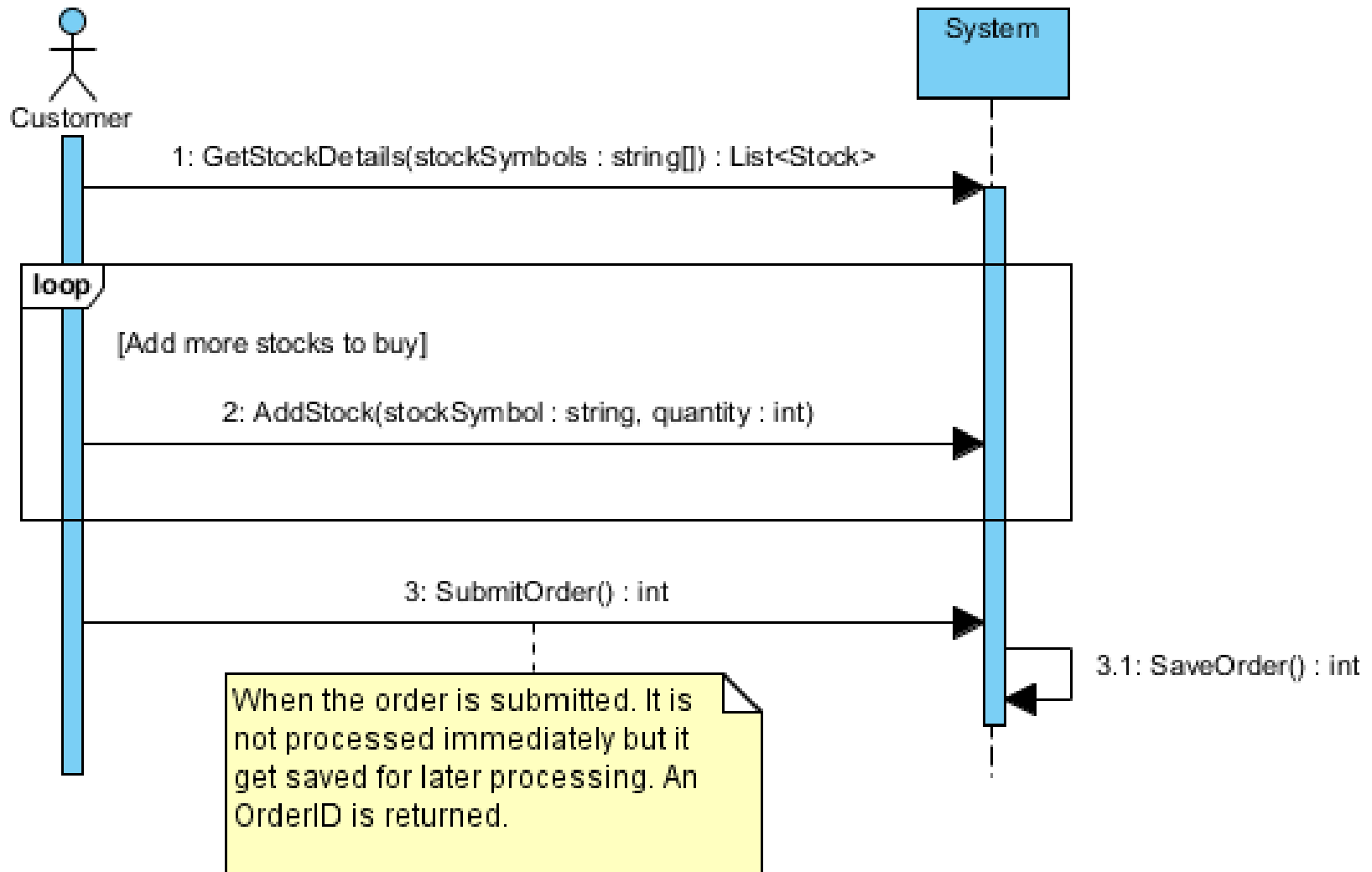
=> Ignore the UI as they are often designed separately. UI is simply the visual representation of the exchanged objects

Message Type	Notation
Asynchronous	
Synchronous	
Return	

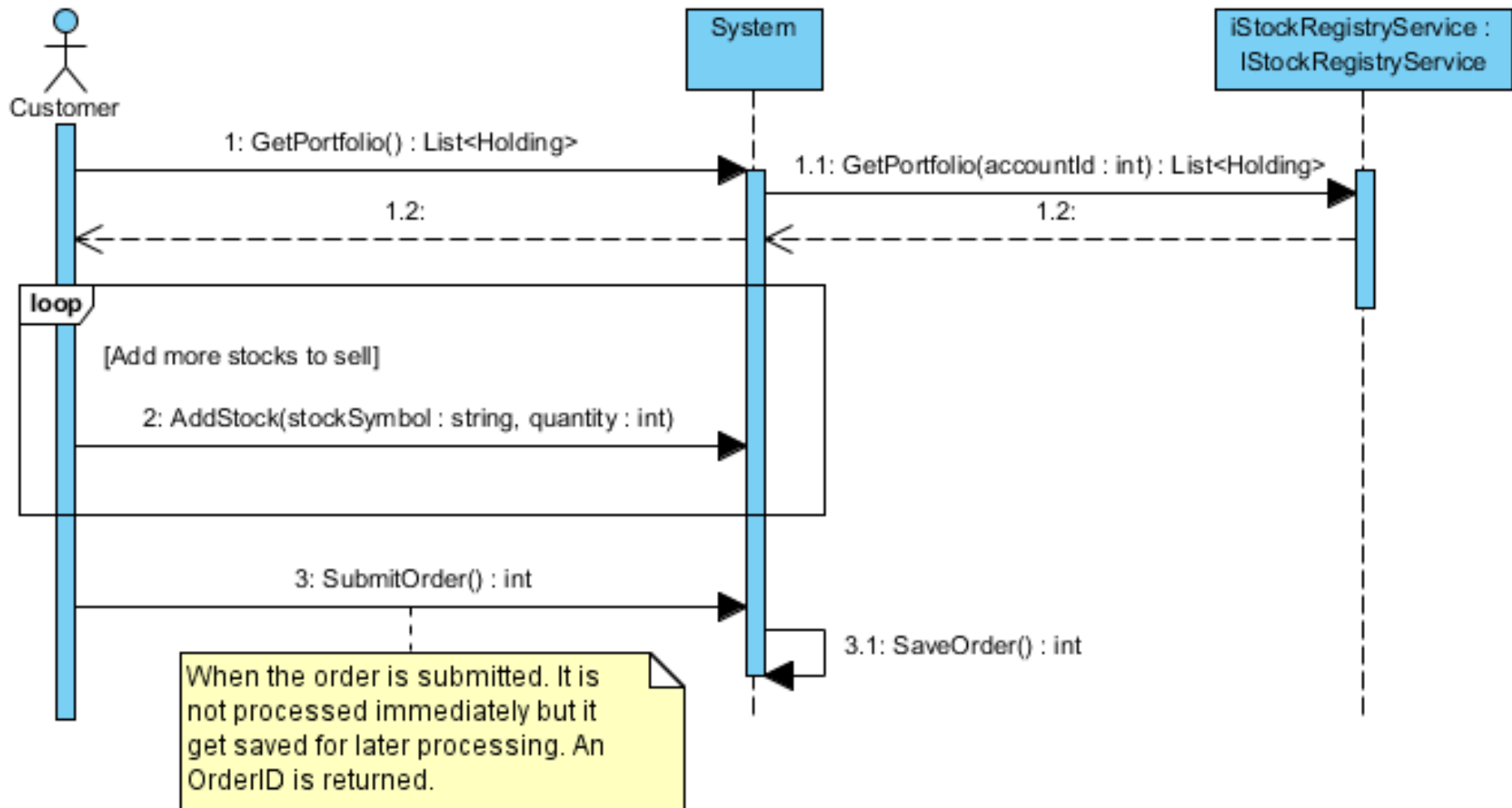
Drawing System Sequence Diagrams

- Select a Use Case scenario to model
 - One diagram per scenario
- Add lifelines for the System and each involved Actor
- Add interactions between the Actors and the System using arrows
- Name the arrows using message names and parameters
 - Keep it simple, no need to provide all parameters at this point

Buy Stocks SSD



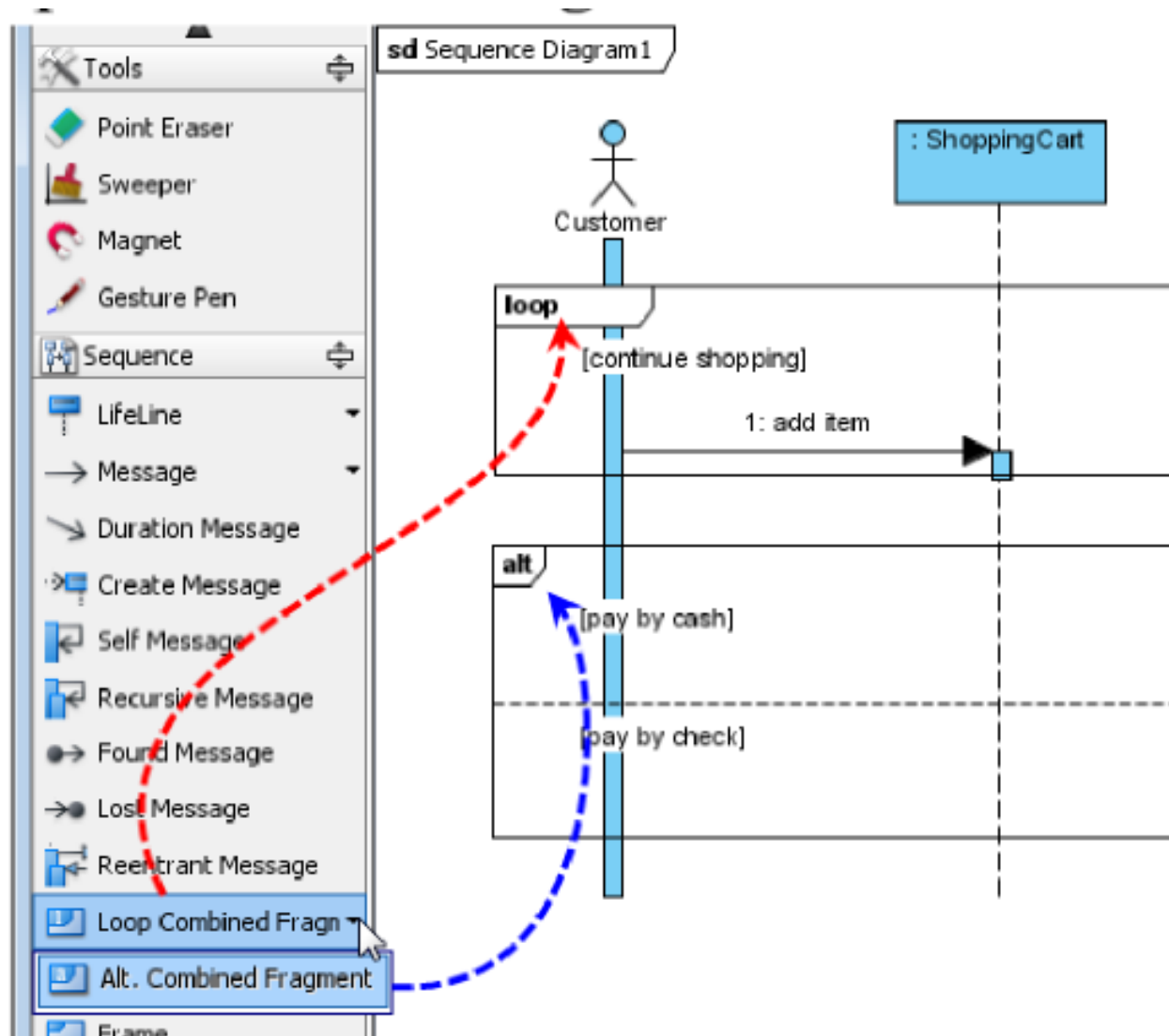
Sell Stocks SSD



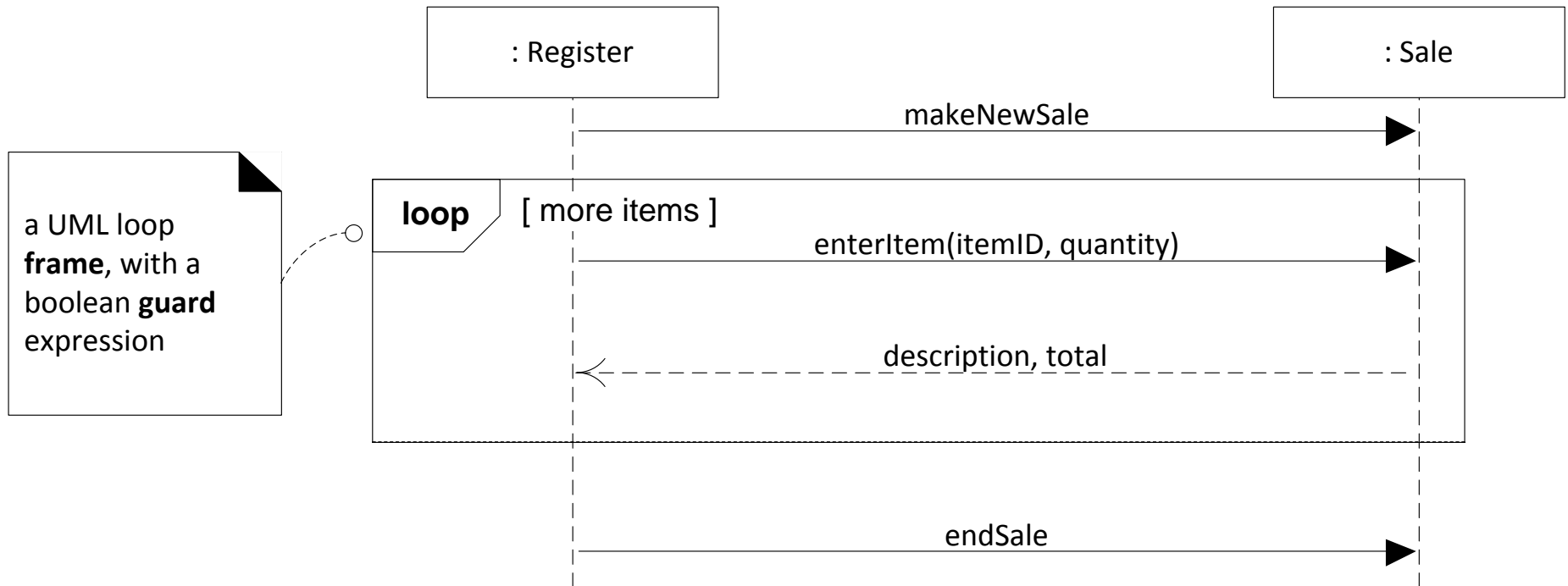
Sequence Diagram Common Operators

- Alternative fragment (denoted “**alt**”) models **if...then...else constructs**.
 - Guard condition specify the true case for the execution of the interaction
- Option fragment (denoted “**opt**”) models **switch constructs**.
 - Guard condition specified for each case.
- **Loop** fragment encloses a series of messages which are repeated.
 - Guard condition specify the lower and upper limit of the loop.
- “**ref**” refers to an interaction defined on another diagram.
- Parallel fragment (denoted “**par**”) models concurrent processing.

Example of using Alt and Loop

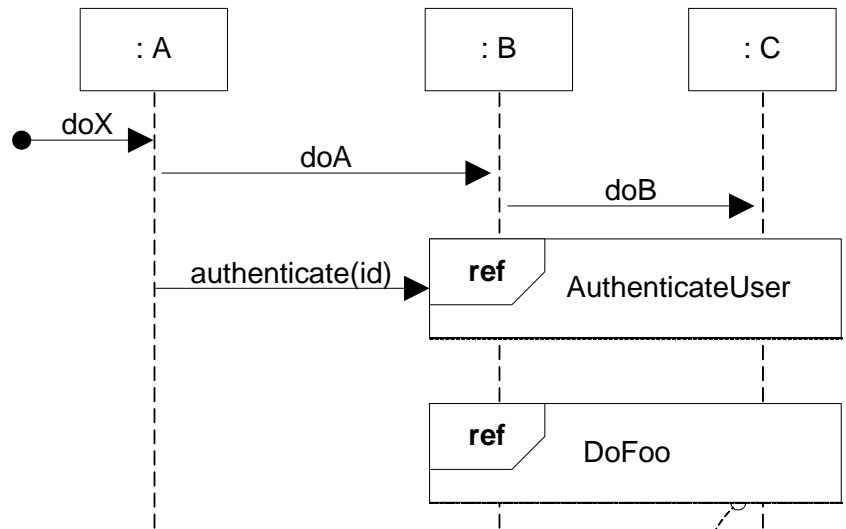


Sequence diagram with loop



Avoid returns in sequence diagrams unless they add clarity.

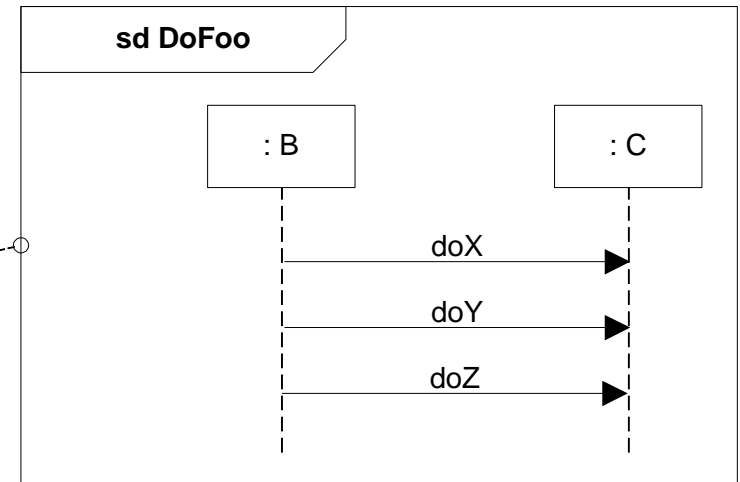
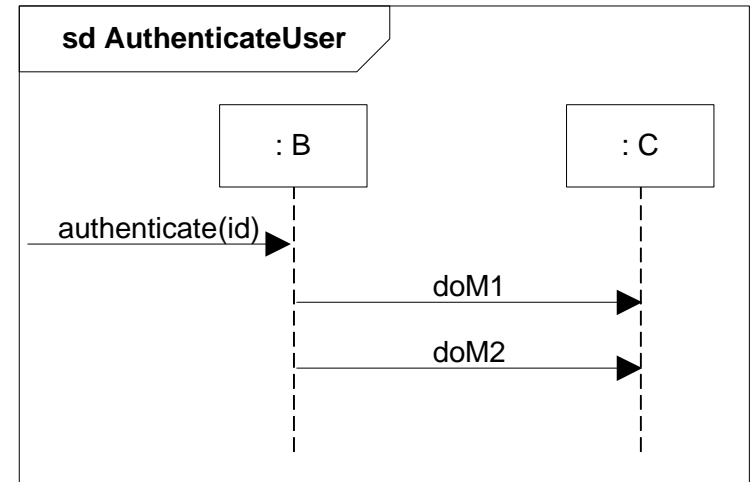
Referencing another SD



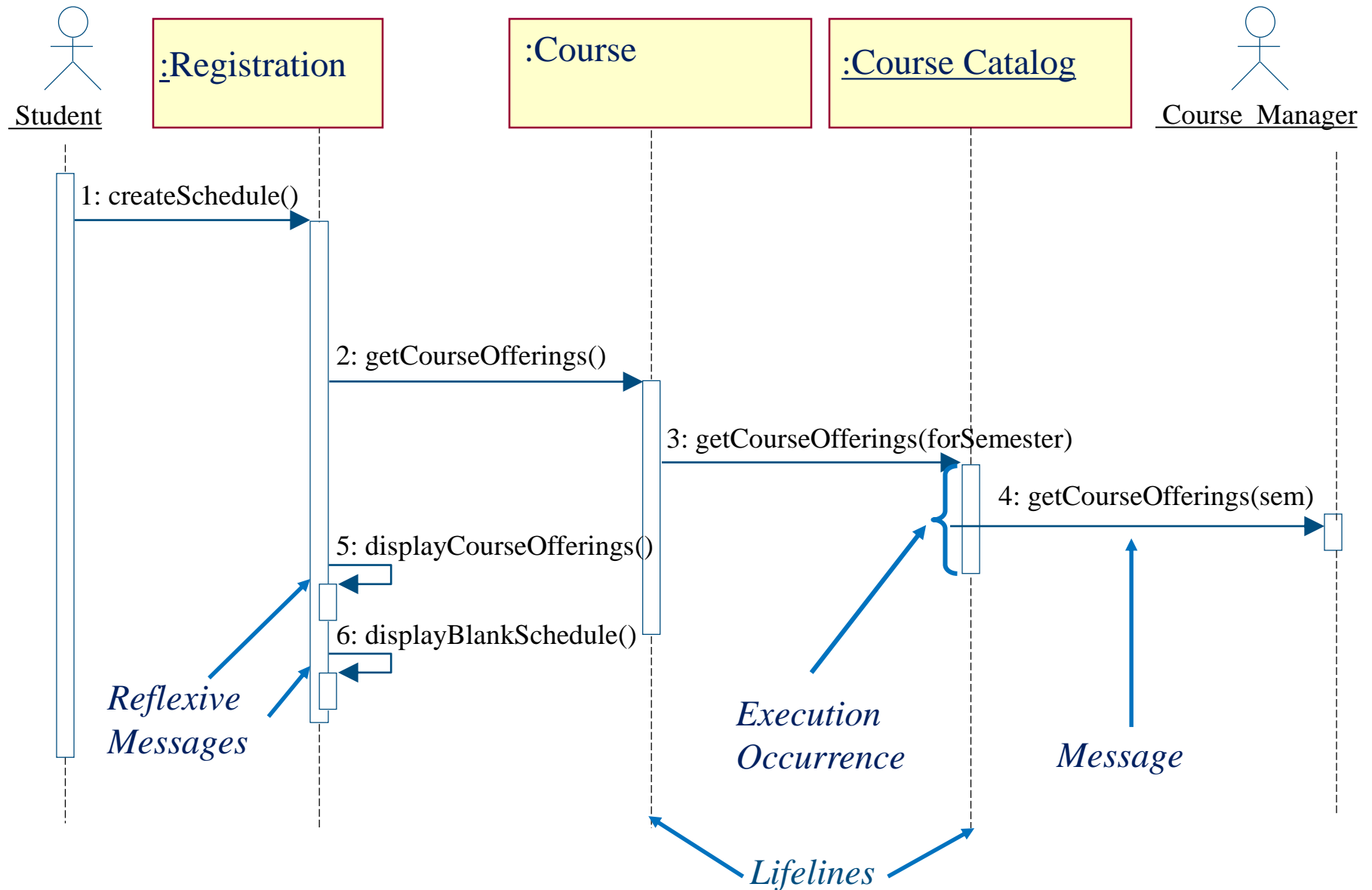
interaction occurrence

note it covers a set of lifelines

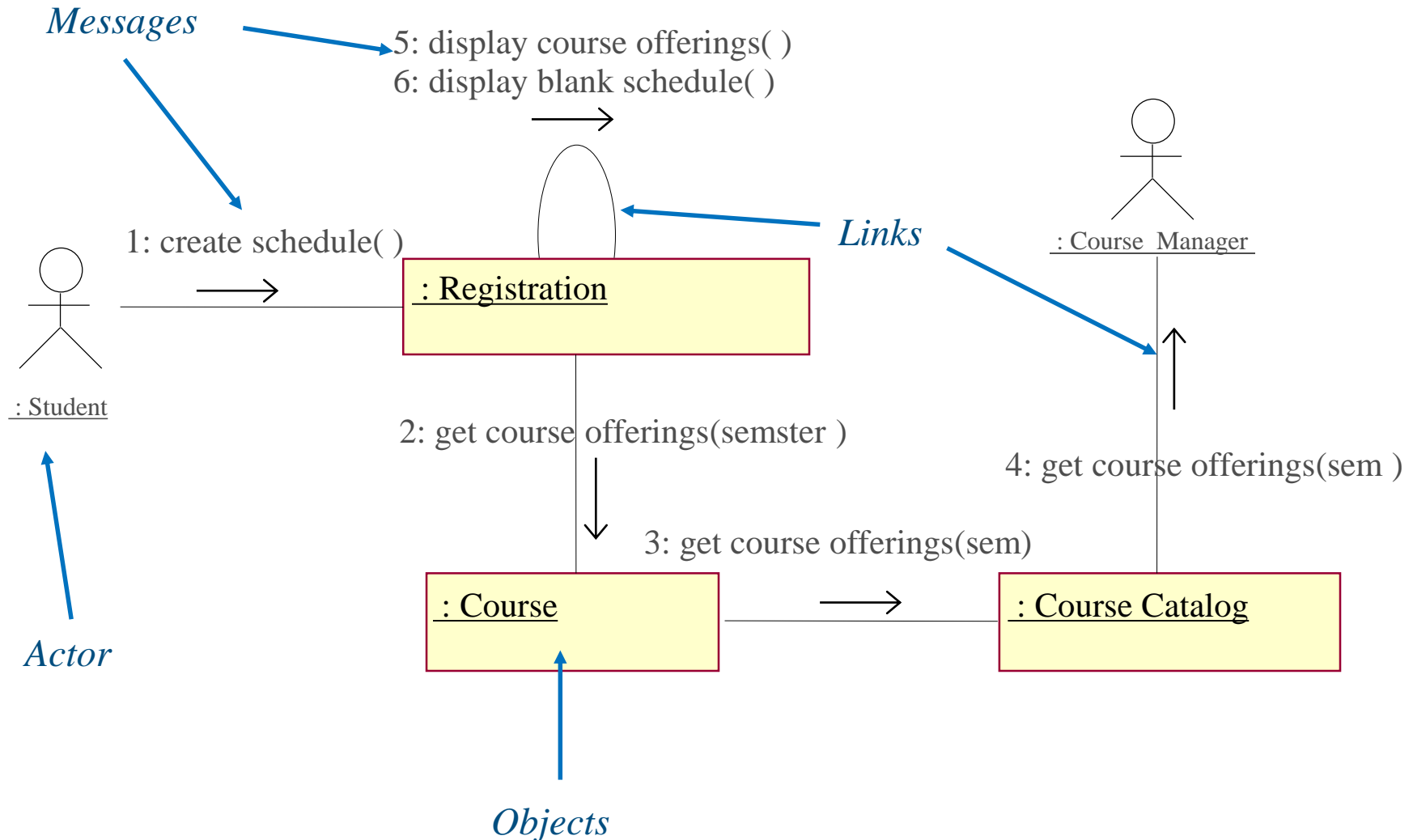
note that the sd frame it relates to has the same lifelines: B and C



Sequence Diagram for Register for Courses Use Case



Communication Diagram Contents: Links and Messages



Design Sequence Diagram vs. Communication Diagram

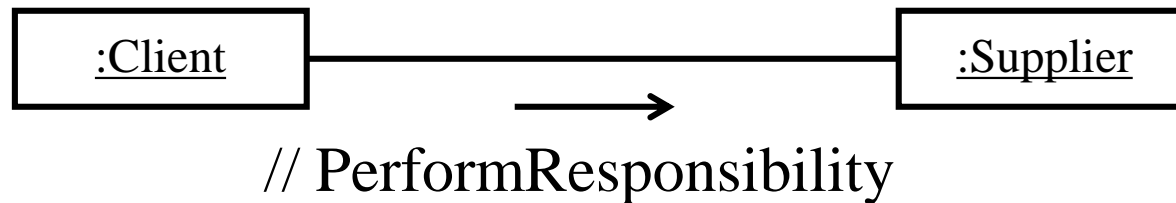
- Semantically equivalent: Can convert one diagram to the other without losing any information
 - Model the dynamic aspects of a system
 - Model a use-case scenario
- Sequence diagrams
 - **Time-oriented:** better for visualizing overall flow
- Communication diagrams
 - **Message-oriented:** useful for *validating* class diagrams
 - Better for visualizing all of the effects on a given object

Relationship between classes and Sequence Diagram

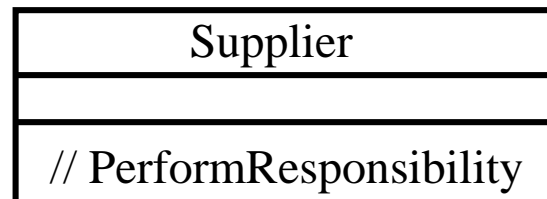
Reminder

- Sequence diagram messages are methods of in the supplier class!

Sequence Diagram



Class Diagram

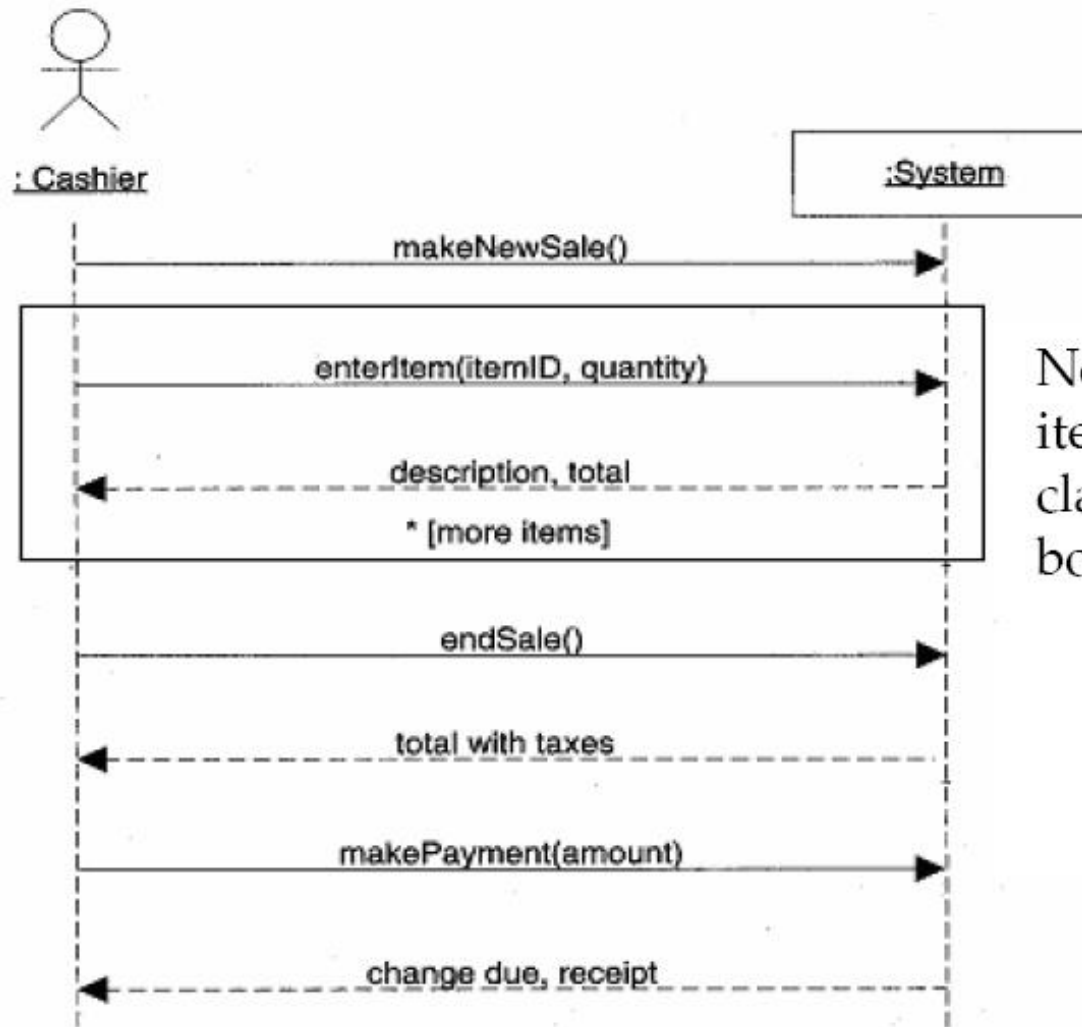


Assigning Responsibilities to Classes using GRASP Principles

Well-known Pattern Families

- **GRASP** = **G**eneral **R**esponsibility **A**ssignment
Software ~~**P**atterns~~ **P**inciples
 - Describe fundamental principles for **assigning responsibilities to classes** and for **designing interactions between classes**
 - GRASP try to formalize "common sense" in object oriented design.
- We will focus on the following GRASP principles:
 - Information Expert
 - Creator
 - Controller
 - Low Coupling
 - High Cohesion

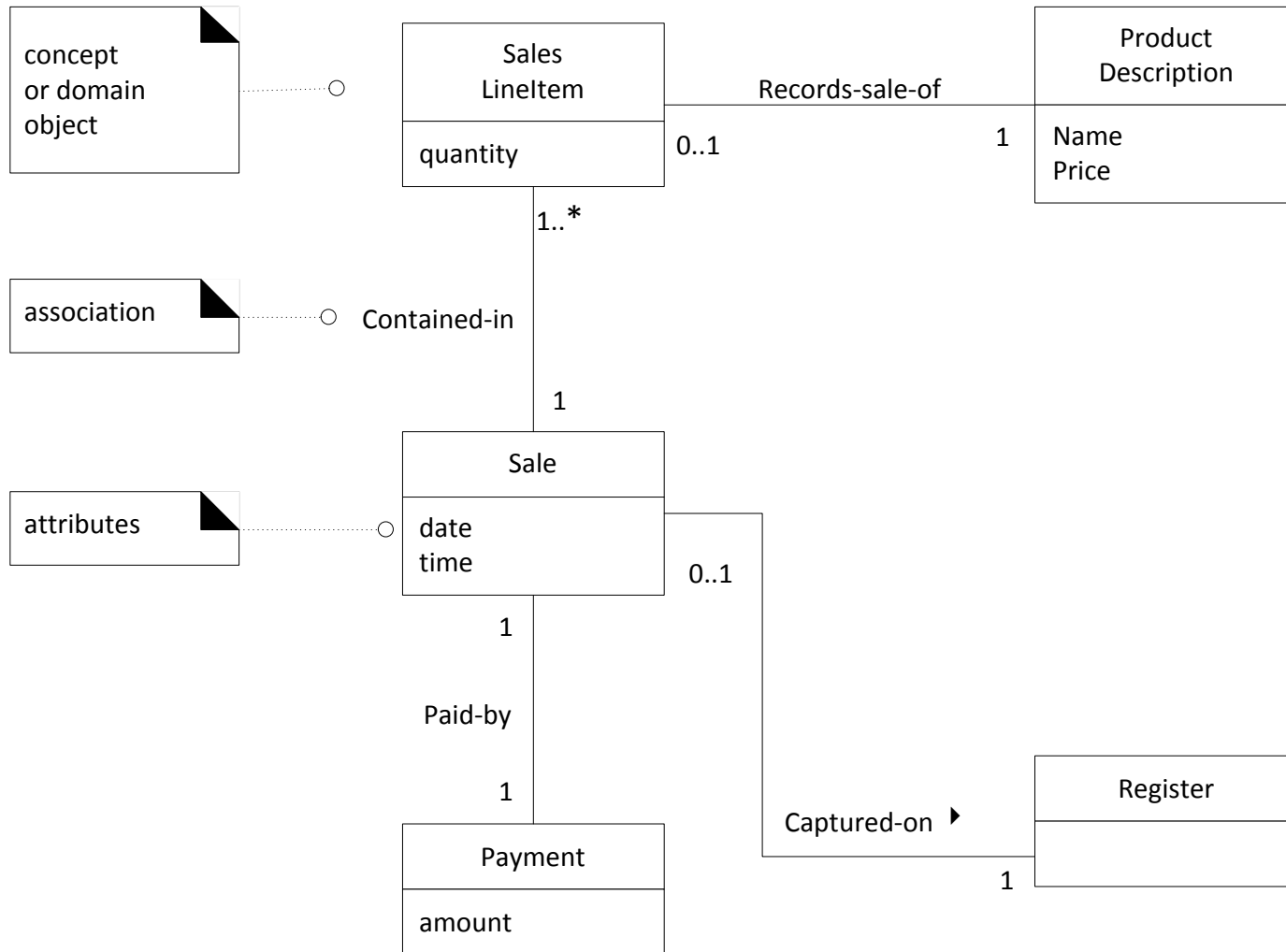
Example we will use to discuss GRASP



Note: The ***[...]** is an iteration marker and clause indicating the box is for iteration.

System Sequence Diagrams for the Process Sale use case

Domain Model for Process Sale



Use Domain model as source of inspiration of the design classes

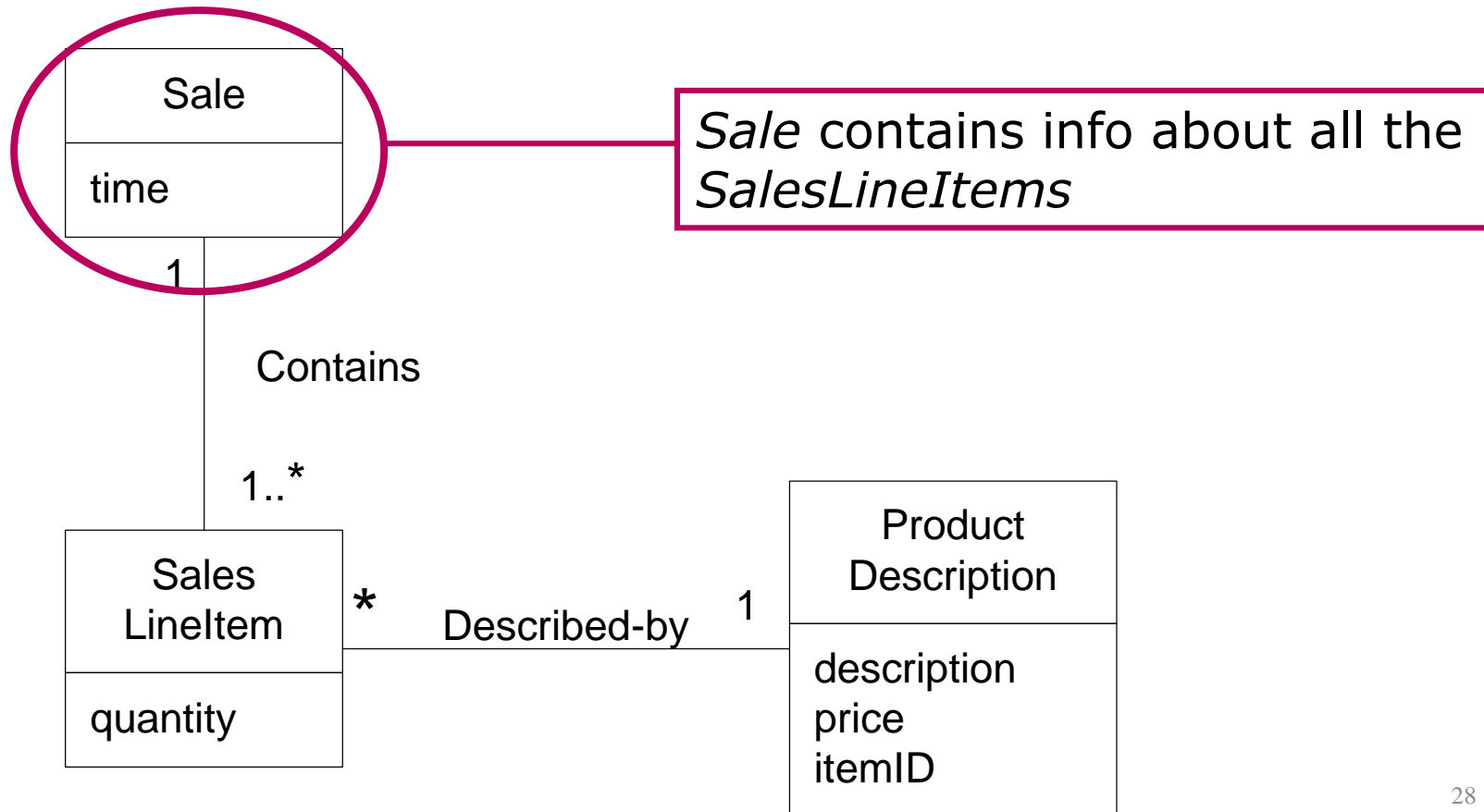
The “Information Expert” Principle

- 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

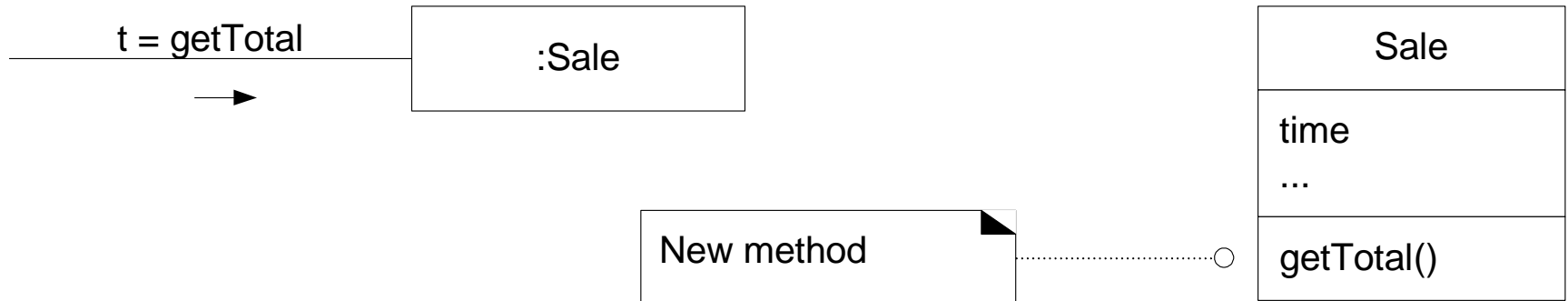
“Objects do things related to the information they have.”

Information Expert Pattern

- Problem
 - Who should be responsible for getting the total of a sale?

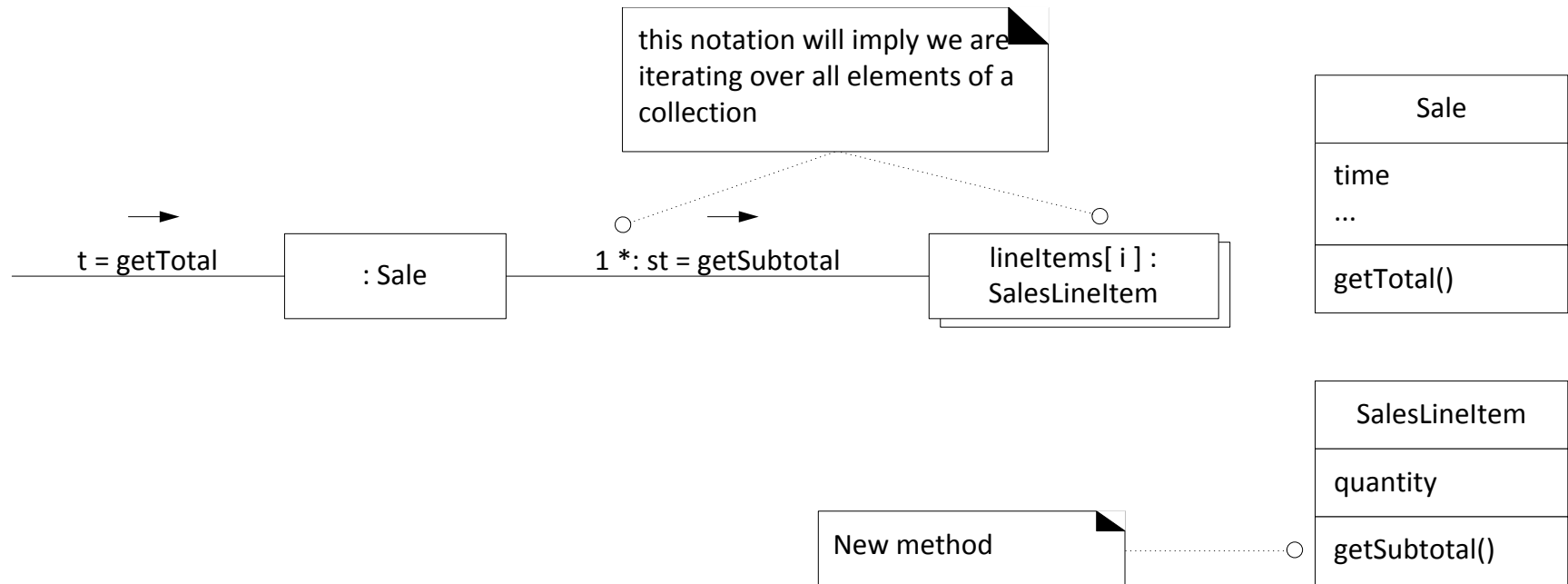


Information Expert Pattern (cont'd)



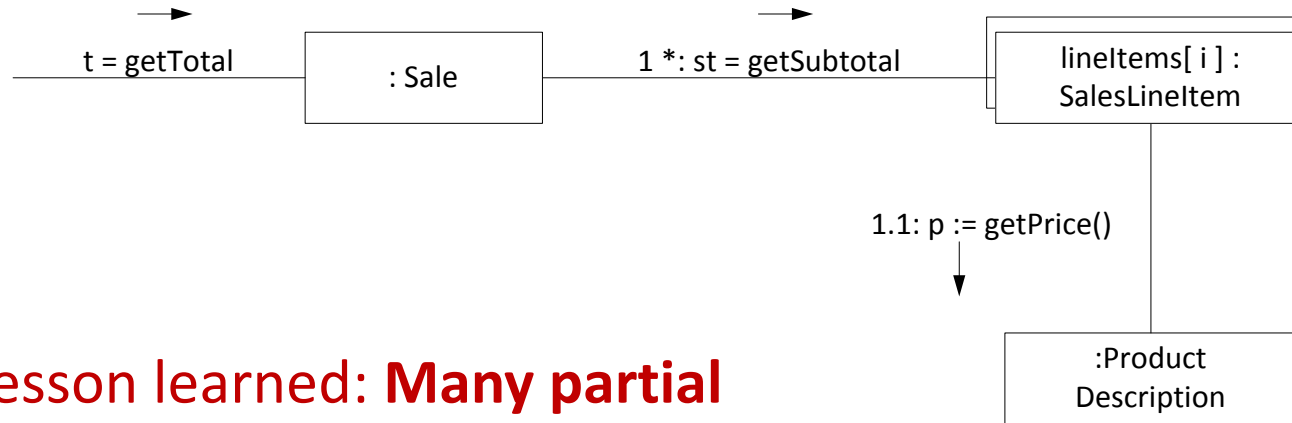
- What information do we need to determine the grand total of a `Sale`?
 - All the `SalesLineItem` instances
 - Sum of their subtotals
- A ***Sale*** instance contains these
 - By “Information Expert”, it is a suitable class for computing the grand total
- Since we said that `Sale` is responsible for getting the total it needs a **`getTotal()`** method

Information Expert Principle (cont'd)



- Sale calculates its total by adding up the subtotals from each of its SalesLineItems
- Sale has to request a subtotal from each SalesLineItem.
- Therefore we add a **getSubtotal()** method to SalesLineItem

Information Expert (cont'd)



Lesson learned: **Many partial information experts collaborate in a task**

Sale
time ...
getTotal()

SalesLineItem
quantity
getSubtotal()

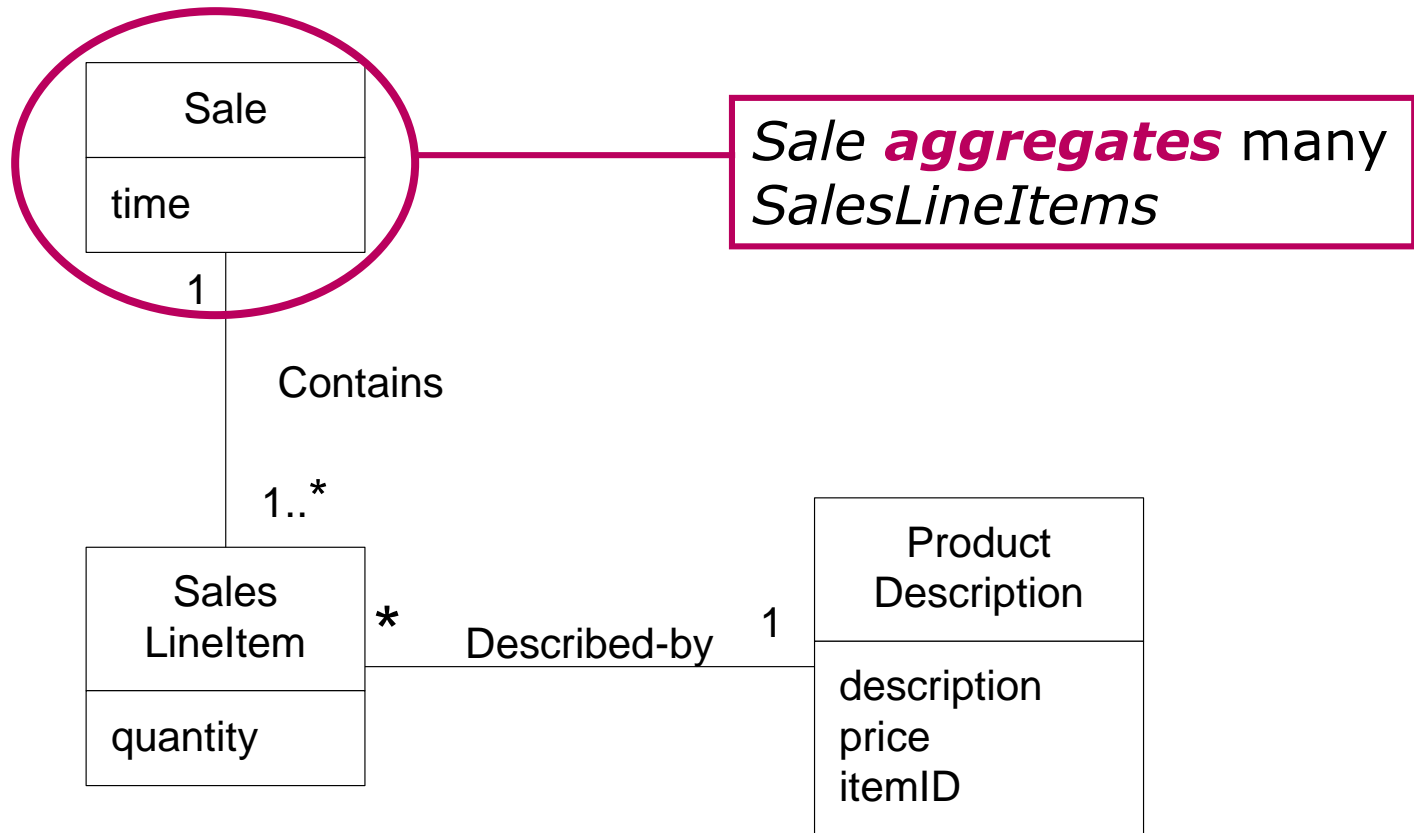
Product Description
description price itemID
getPrice()



- To calculate the subtotal the *SalesLineItem* needs to get the unit price. *SalesLineItem* asks *ProductDescription* to return the price.
- Since *ProductDescription* is the Information Expert of the unit price, we add a **getPrice()** method to the *ProductDescription*

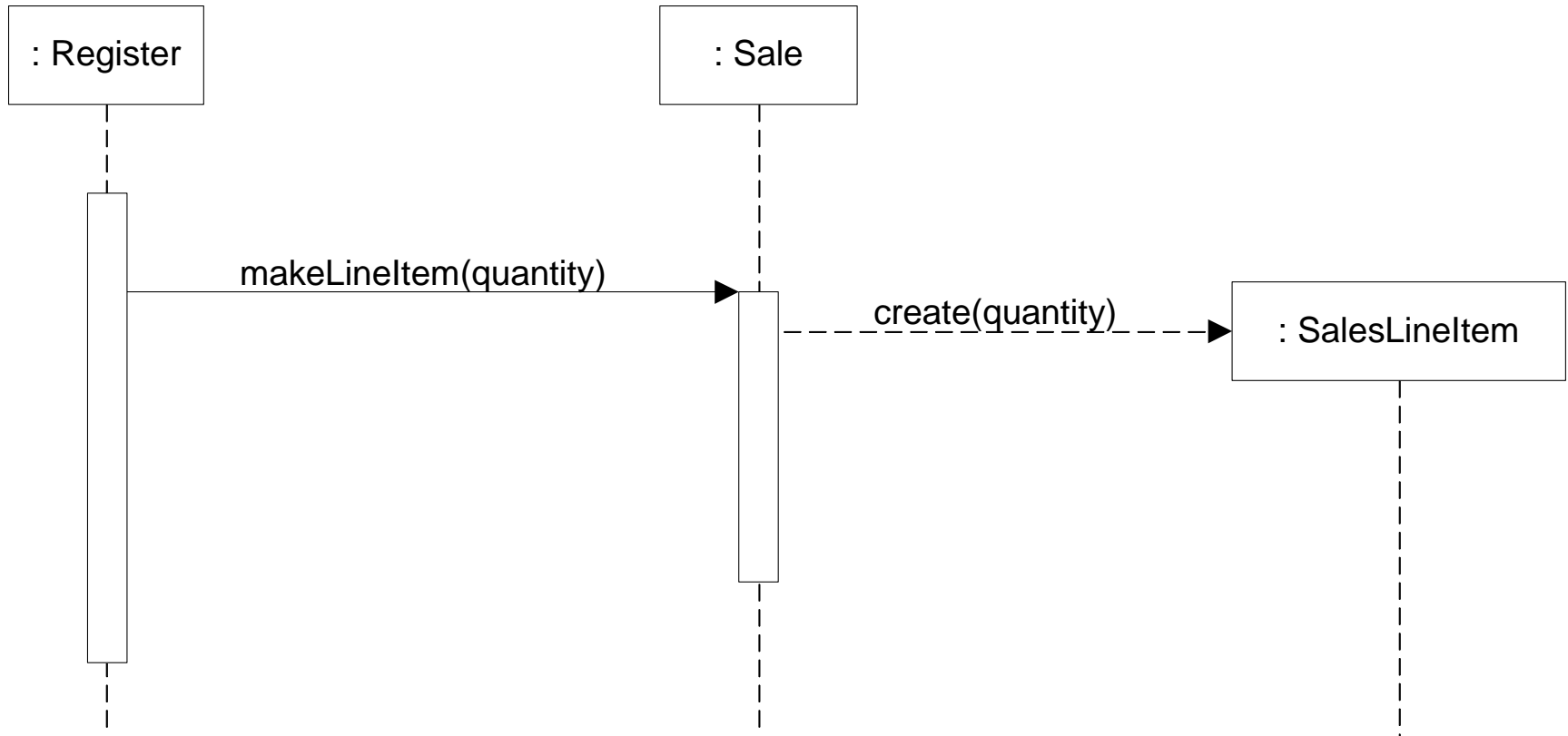
Creator Pattern

- Problem
 - Who should create a SalesLineItem?



Sale **aggregates** a list of LineItems → it's a good candidate to create these.

The Creator Pattern



- Sale needs a `makeLineItem(quantity)`.
- Why does register tell Sale object to create the `SalesLineItem`?
Let Sale do it because the Sale **contains** the List of `SalesLineItem`
=> Creator Pattern

A more detailed look at “Creator”

- **Problem:** Who should be responsible for creating a new instance of some class?
- **Solution:** B should create an instance of A if
 - B “contains” or **aggregates** A
 - B records A
 - **B has the initializing data for A** that will be passed to A when it is created.
 - *Often initiation is done using a constructor with parameters*
 - *E.g. a Payment instance, when created needs to be initialized with the Sale total.*
 - *Sale class knows Sale total. Good candidate for creating Payment.*
 - B closely uses A
- If more than one of the above applies, **prefer a class B which aggregates or contains A.**

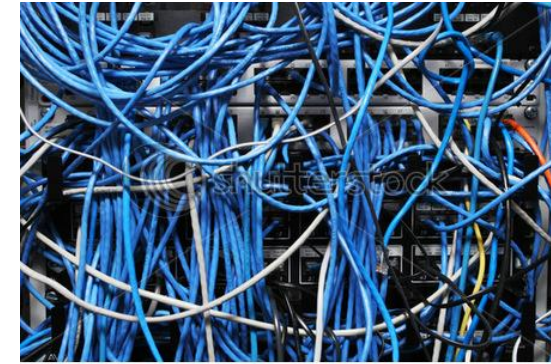
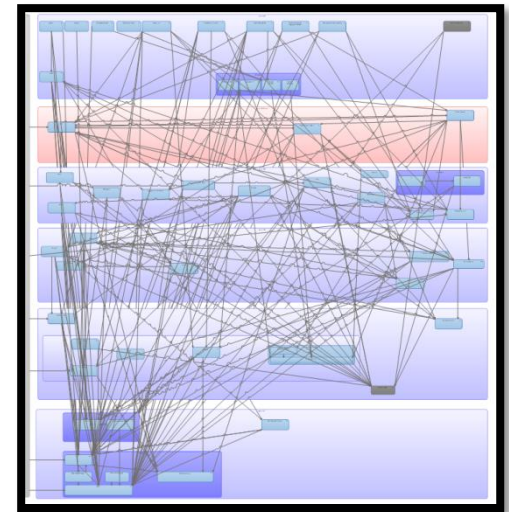
The “Low Coupling” Principle

- Problem: How to support low dependency, **low change impact**, and **increased reuse**?
- Solution: **Assign responsibilities so that *coupling* remains low**. Use this principle to evaluate alternatives.

- *Coupling* is a measure of how strongly one class is
 - **connected to,**
 - **has knowledge of, or**
 - **relies upon**other classes.

Why is a class with high (or strong) coupling bad?

- Constantly affected by changes in related classes
- Harder to understand in isolation
- Hard to maintain
- Harder to re-use
 - Because it requires the presence of classes that it depends on



“Low Coupling” Principle

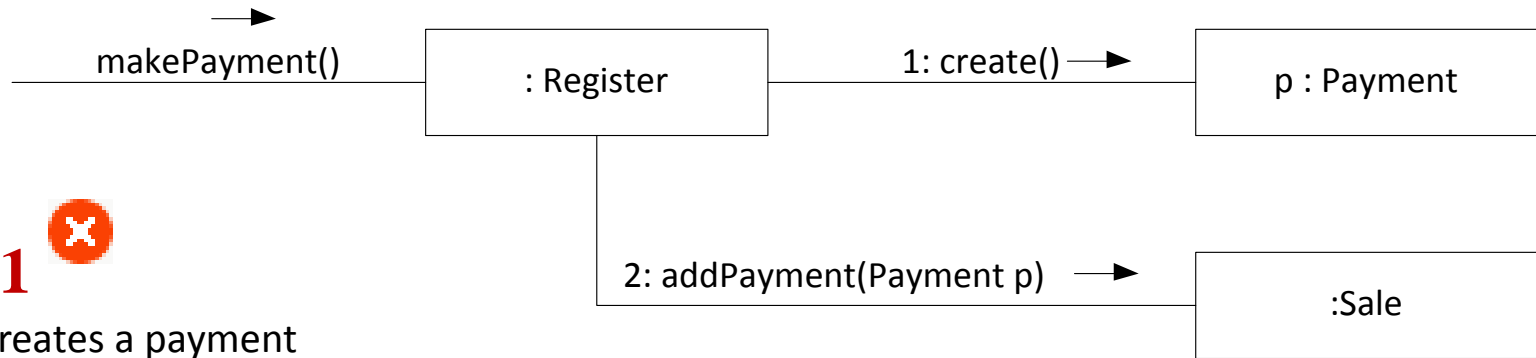
- Problem Solved
 - Who should create a Payment and associate it with a Sale?
- Solution (advice)
 - By Expert
 - Let the Register create the Payment and associate it with the Sale
 - **Register and sale are now coupled to Payment**
 - By Low Coupling Register pass Payment creation on to Sale
 - **Only sale is coupled to Payment**

Two alternative responses to “Who creates Payment?”

Design Option 1



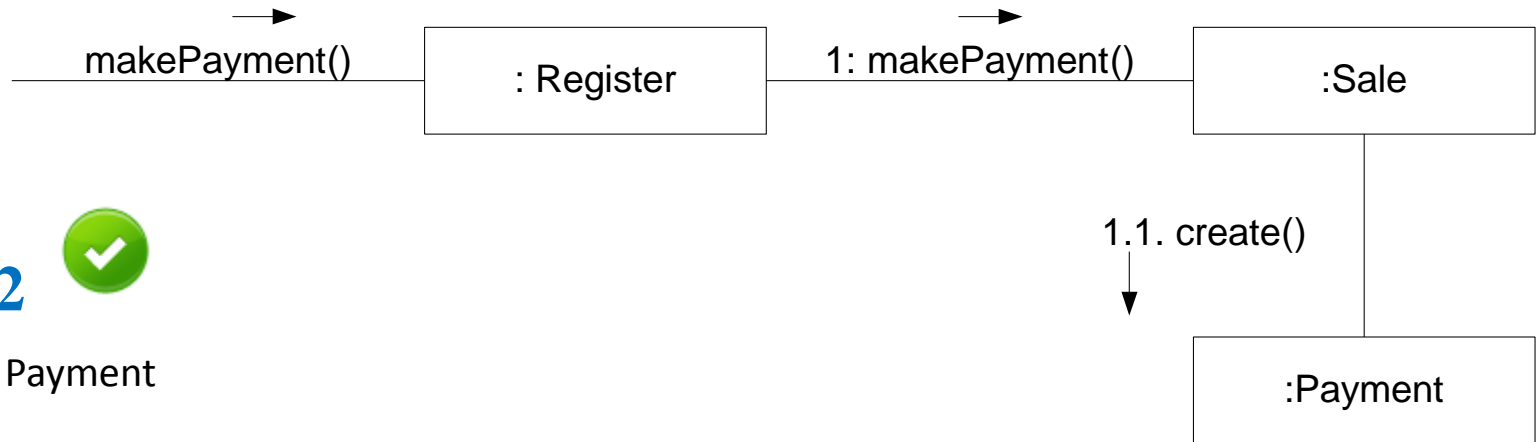
1. Register creates a payment
2. Payment is added to Sale



Design Option 2



Sale creates Payment



Option 1: Both Sale & Register are **coupled** to Payment.

Option 2: **Only necessary relationships are made**

=> **coupling has been lowered.**

Benefits of Low Coupling


- It reduces time, effort and defects involved in modifying software
- Classes are more
 - Independent
 - Easier to reuse
 - Easier to understand
 - Easier to maintain
- Coupling to stable class libraries is more acceptable
 - Such as Java libraries (more stable)
- Avoid coupling to frequently changes classes
 - Can result in frequent changes to your classes
- Avoid coupling to APIs for proprietary software
 - Can result in vendor lock in

The “High Cohesion” Principle

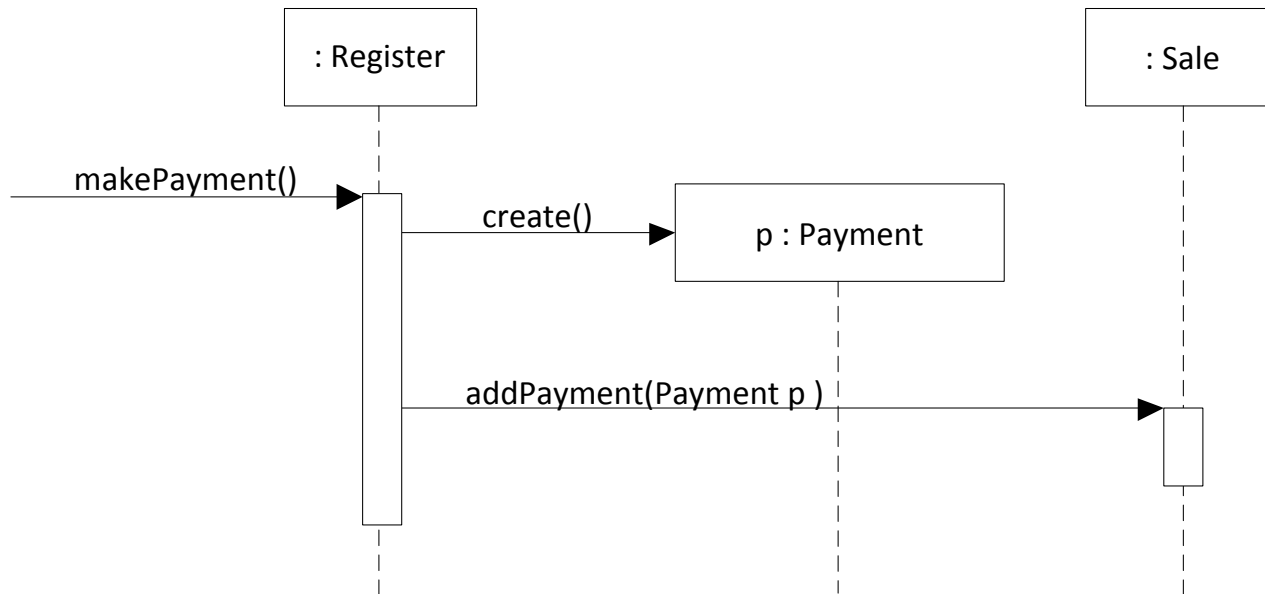
- Problem: How to keep objects focused, understandable, and manageable?
- Solution (advice): Assign responsibilities so that **cohesion remains high**. Use this principle to evaluate alternatives.

Cohesion measures the degree to which the tasks performed by a single class are **functionally related**.

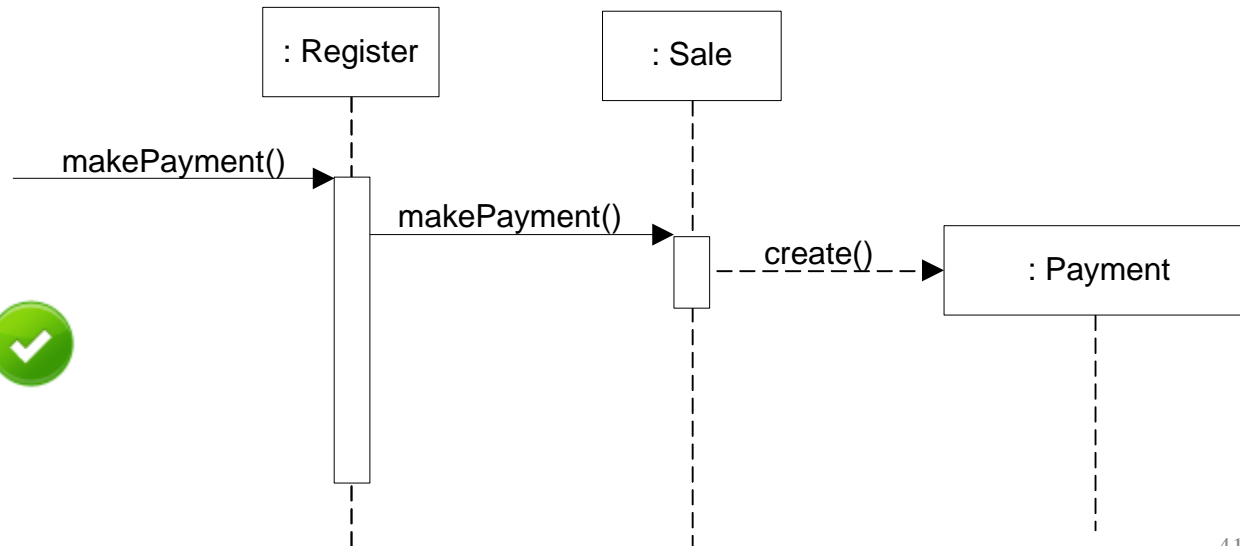
=> **Single Responsibility Principle**

 A class with high cohesion has a **relatively small number of methods, with highly related functionality, and does not do ‘too much’ work.**

High Cohesion Pattern



Danger:
Register is
doing **too**
much work ❌



Better Design

Register **delegates**

Lower Coupling

Higher Cohesiveness



High Cohesion Details

- Cohesion:
 - How functionally related are the operations of a software element?
 - How much work is a software element doing?
- Example:
 - The “Big” class: 100 methods, 2000 lines of code
 - The “Small” class 10 methods, 200 lines of code.
 - “Big” is probably **covering many different areas of responsibility**
 - Examples: database access AND random number generation
 - Big has less focus or **functional cohesion** than small
- An object that has **too many different kinds of responsibilities probably has to collaborate with many other objects**
 - Low cohesion → High coupling, Both bad.

Benefits of High Cohesion

- Benefits:
 - Clarity and ease of comprehension of the design is increased.
 - Likelihood of reuse is increased
 - Maintenance and enhancements are simplified.
 - Low coupling is often supported.
- Disadvantages of low cohesion are:



- » Increased difficulty in understanding classes.
- » Increased difficulty in maintaining a system, because application changes affect multiple classes, and because changes in one class require changes in related classes.
- » Increased difficulty in reusing a class because most applications won't need the arbitrary set of operations provided by a class.

Controller Pattern

- Problem: What first object beyond the UI layer **receives and coordinates** ("controls") a system operation?
- Solution: The **controller** is the first object beyond the UI layer that should be responsible for receiving and coordinating actions to handle user requests.

=> **Apply MVC pattern**

Controller Pattern Guidelines

- Controller **delegates handling events to other objects** in the domain.
 - It coordinates or controls the activities to realize a UC
 - Like managers: they boss people around but don't do much work themselves 😊 => **mainly coordinate between objects**
- Controller **maintains information about the state of the use case**
(e.g., controller maintains the shoppingCard)
- Helps identify out-of-sequence operations
 - E.g., makePayment before endSale

The MVC pattern is intended to allow each part to be changed independently of the others.

How MVC Works

Controller

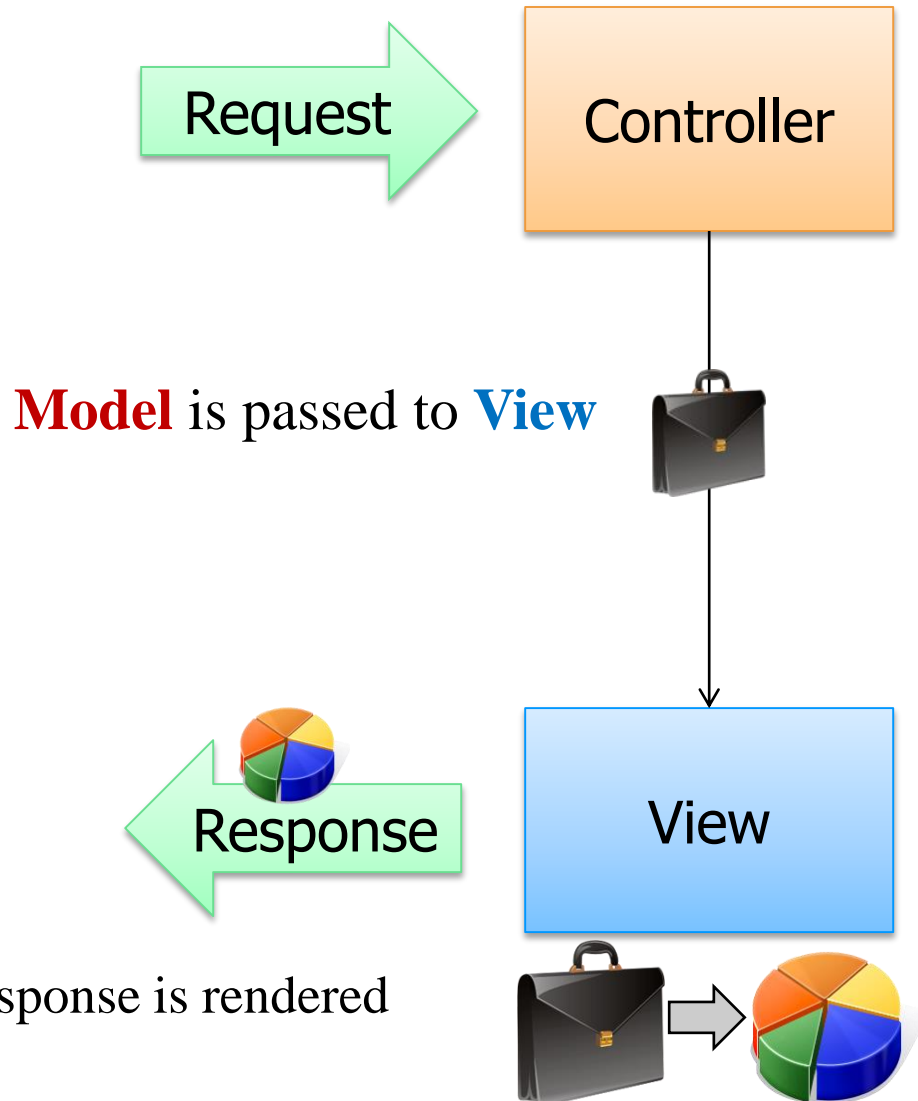
- Incoming request directed to **Controller**
- A controller accepts input from the user and **instructs the model to perform actions** based on that input

e.g. the controller adds an item to the user's shopping cart

- Model is then passed to the View

View

View transforms Model into appropriate output format



Model-View-Control Architecture (MVC)

Model

- Holds the application data and implements the application logic

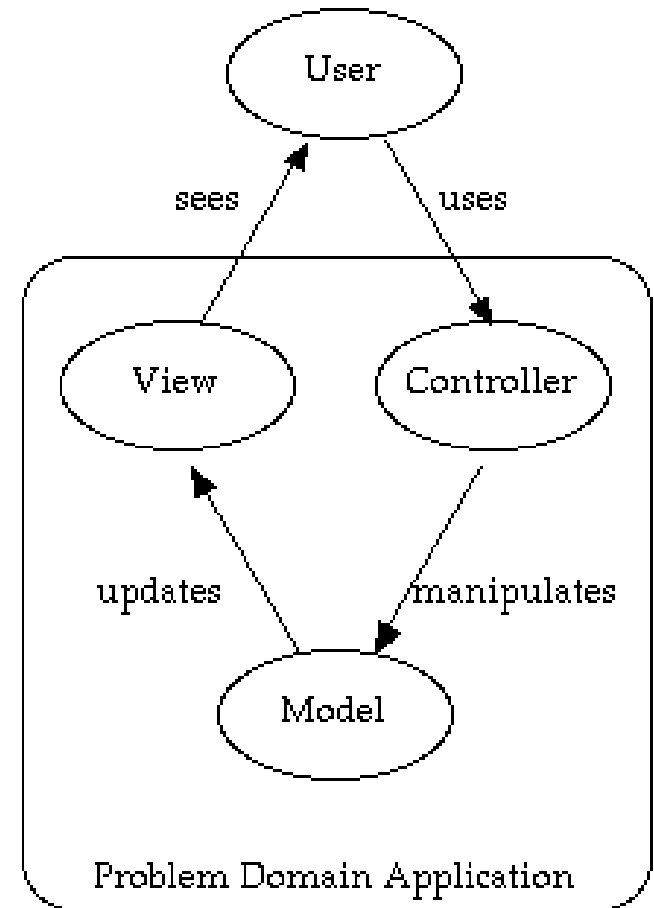
Know how to carry out specific tasks such as processing a new subscription

View

- The View provides a visual representation of the model.

Controller

- Handles to user input (mouse movement, clicks, keystrokes, etc.)
- Process data and communicate with the Model to save state e.g.: delete row, insert row, etc.
- Coordination logic is placed in the controllers



Advantages of MVC

- ***Separation of concerns***
 - Views, controller, and model are separate components. This allows modification and change in each component without significantly disturbing the other.
 - Computation is not intermixed with Presentation. Consequently, code is cleaner and easier to understand and change.
- **Flexibility**
 - The view component, which often needs changes and updates to keep the users continued interests, is separate
 - The UI can be completely changed without touching the model in any way
- **Reusability**
 - The same model can be used by different views (e.g., Web view and mobile view)
- **Disadvantages:**
 - Heavily dependent on a framework and tools that support the MVC architecture (e.g. ASP.Net MVC, Ruby on Rails)

MVC is widely used and recommended particularly for interactive web-applications.

Example Controllers

- An e-commerce application may have 2 controllers:
 - **ProductCatalogController** handles
 - Get Categories
 - Get Product in Category X
 - **ShoppingCartController** handles
 - Get Items in Shopping Cart
 - Add Item to Shopping Cart
 - Checkout
- => This keep the controllers **highly cohesive**

Conclusion

- After class diagrams, sequence diagrams are the most widely used diagrams in UML.
- It is impossible to model all possible interactions within a system.
- Only model those interactions that are interesting or shed light on important aspects of the system.
- A system of even modest complexity may require several interaction diagrams
- Apply GRASP and other design patterns to improve your design