Software Engineering

Professor Dr. Safa Amir Najim
Computer Information System Dept.
College of CS and IT
University of Basrah
2019-2020

Software design based on GRASP principles

Chapter 7

Design Process

- After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.

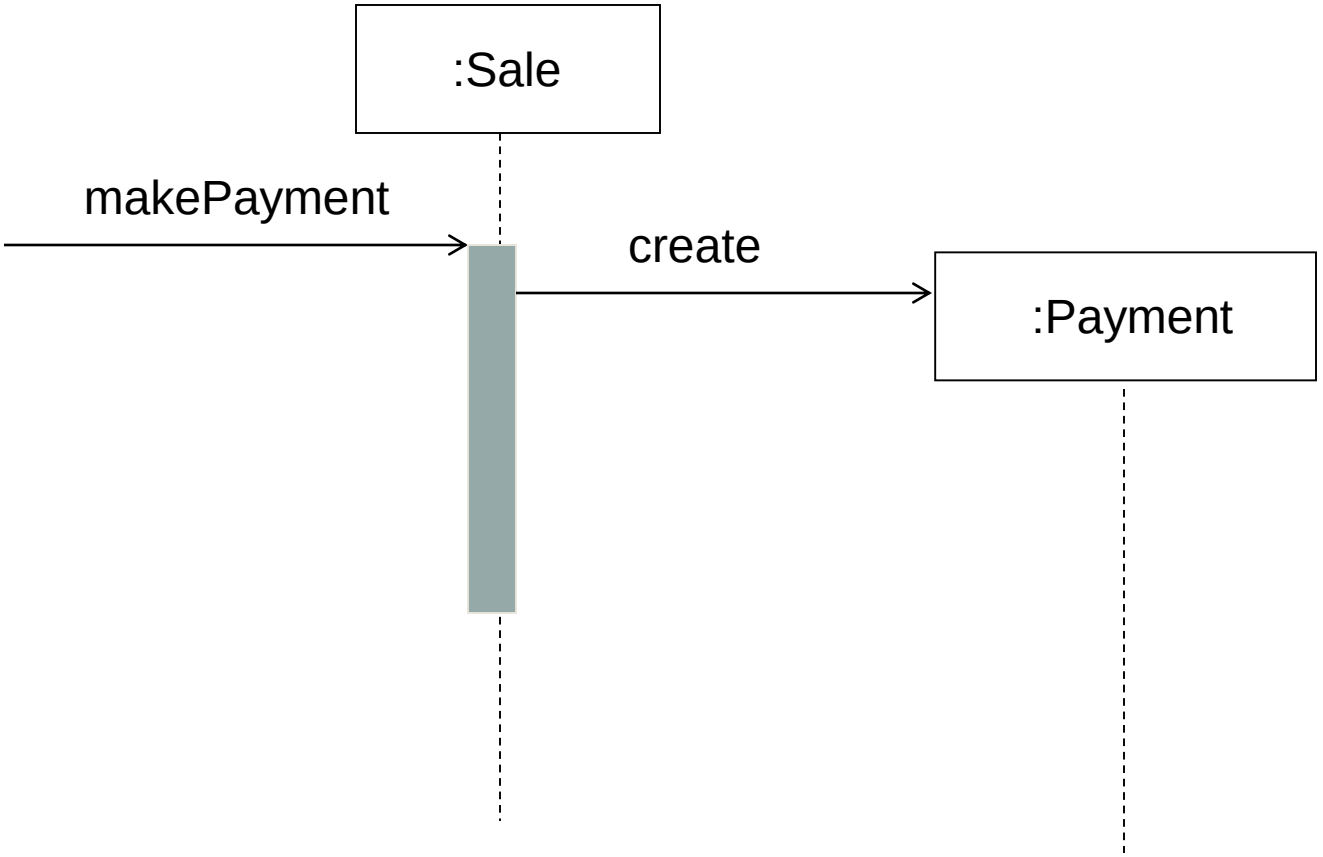
What's GRASP pattern

- GRASP is a General Responsibility Assignment Software Patterns.
- This approach to understand and use design principles base on patterns of assigning responsibilities.
- The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.

What is responsibility

- Doing:
  - Doing something itself, such as creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects.
- Knowing:
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate

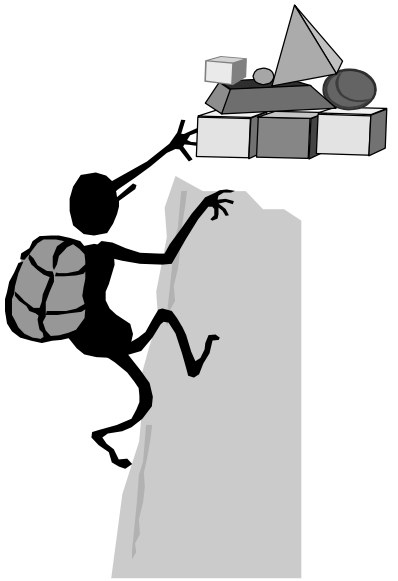
Responsibilities and methods



makePayment implies Sale object has a responsibility to create a *Payment* object

GRASP Patterns

- *Creator*
- *Information Expert*
- *Low Coupling*
- *High Cohesion*
- *Controller*



Creator

Creator principle

- ***Problem***: Who creates an ***a*** object
- ***Solution***: Assign class **B** the *responsibility* to create an instance of class **A** if one of these is true

  *B "**contains** or **aggregate** "A*

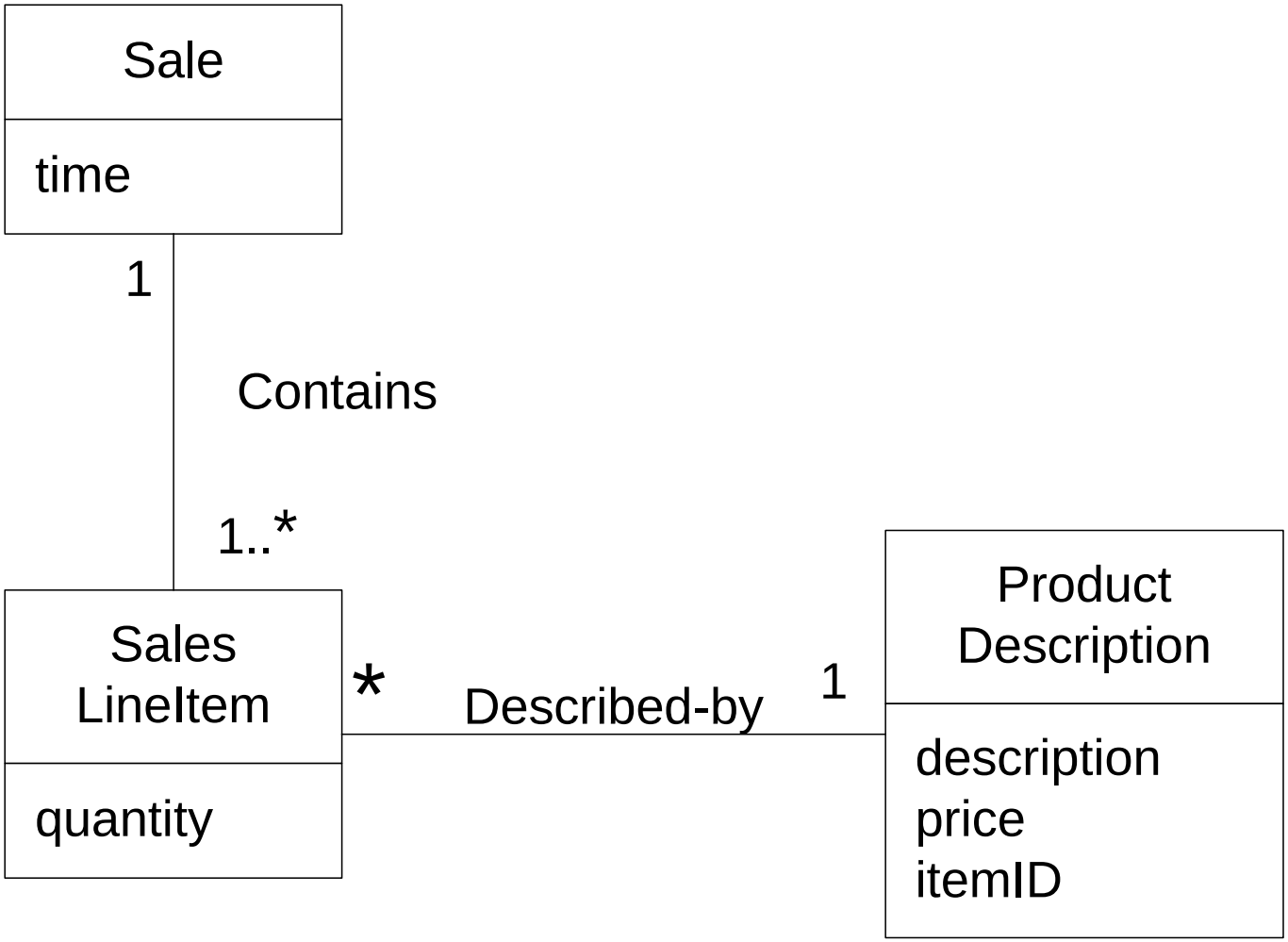
  *B "**records**"A*

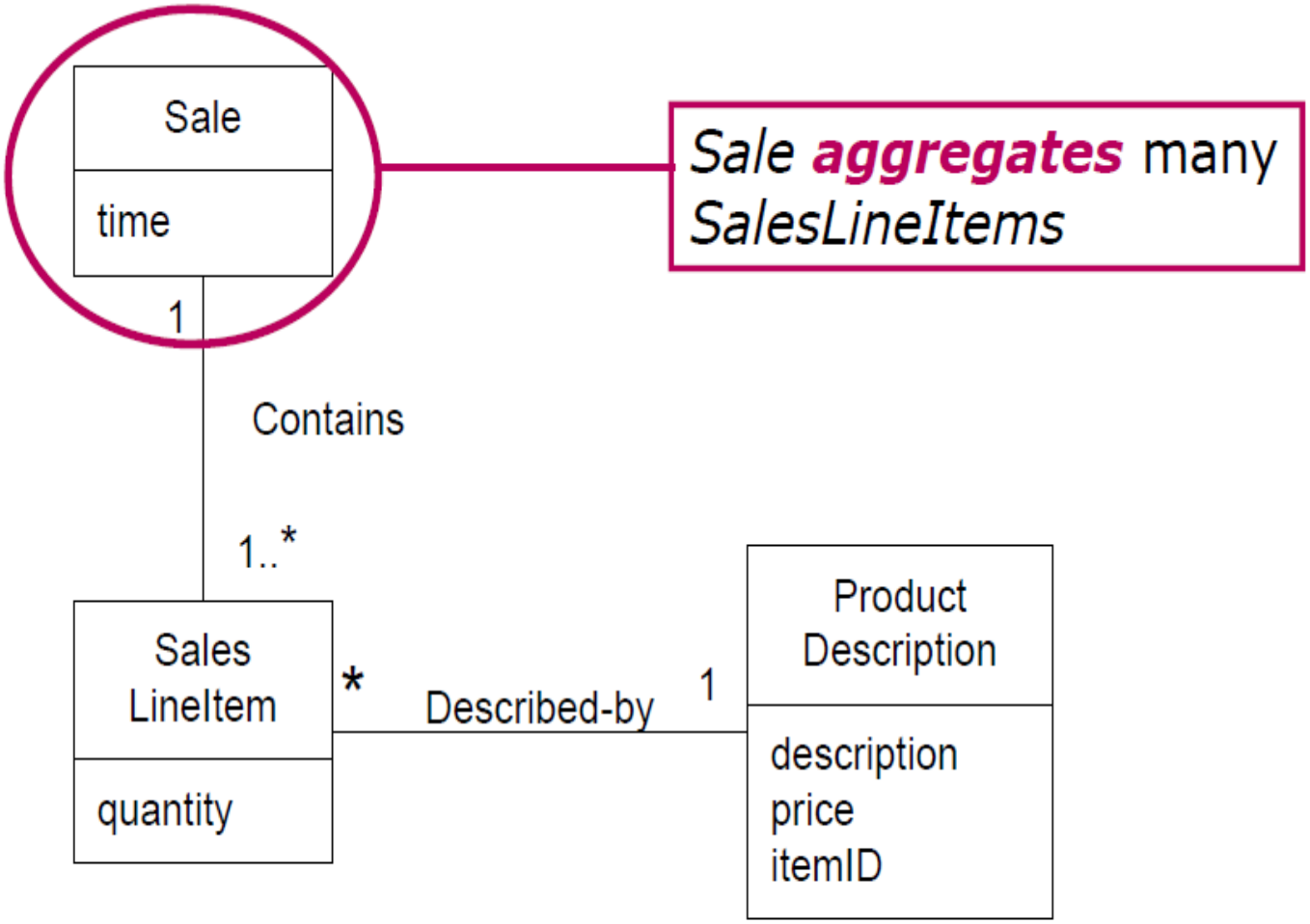
  *B "**closely uses**"A*

  *B " has the **Initializing** data for "A*

Problem

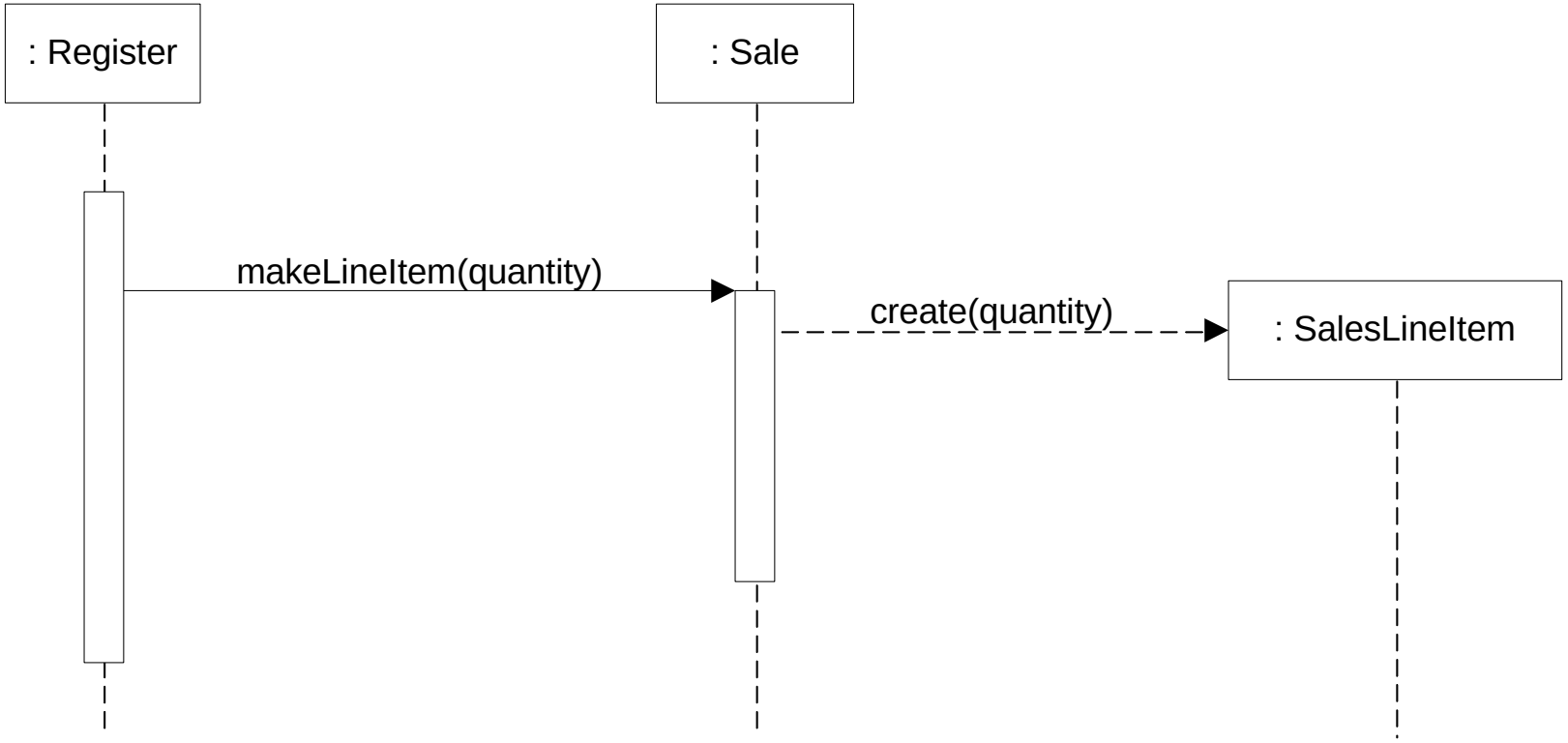
Who should create a SalesLineItem?





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

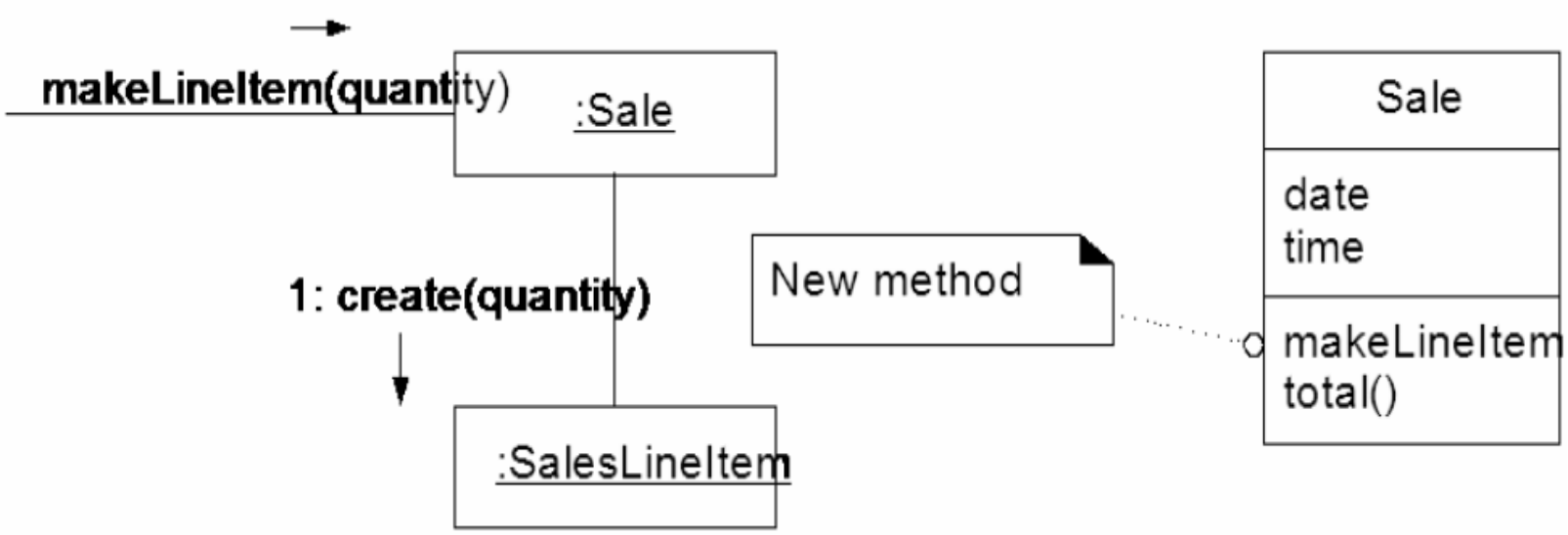
Creating a *SalesLineItem*



Sale objects are given a responsibility to create *SaleLineItem*.

The responsibility is invoked with a *makeLineItem* message

Creating a *SalesLineItem*



Information Expert

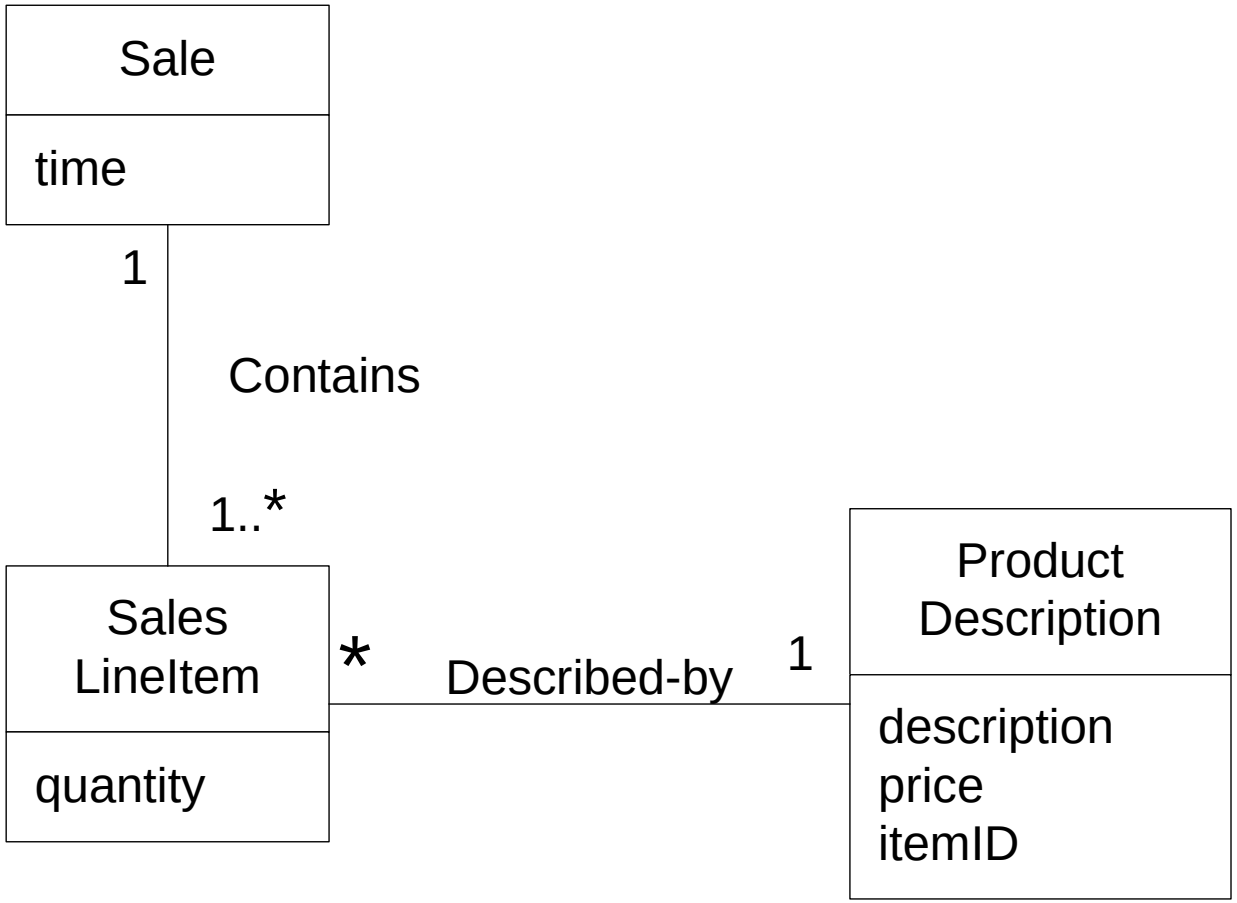
Information Expert

Problem **:** What is a basic principle by which to assign responsibilities to objects?

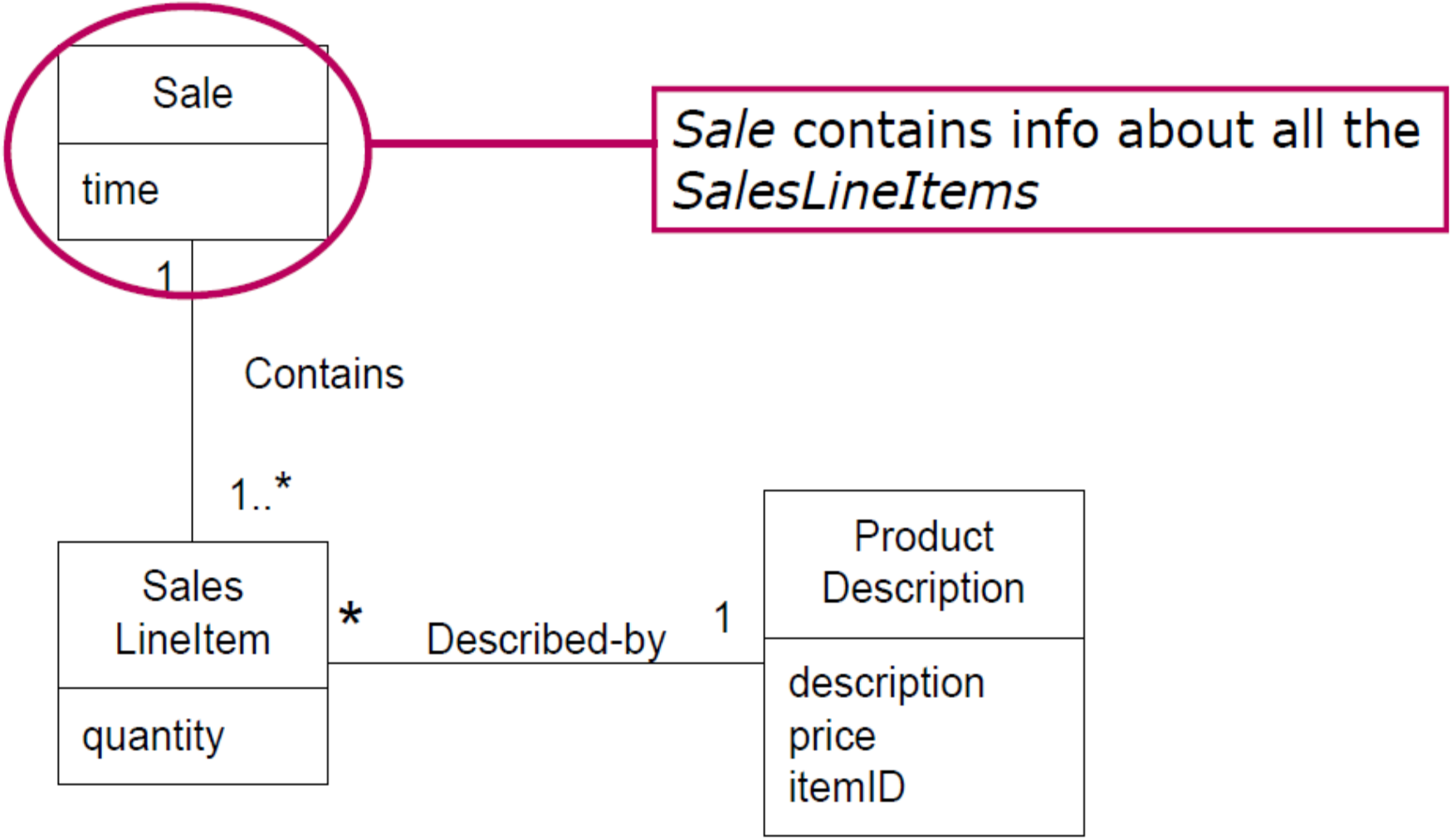
Solution (advice) : Assign a responsibility to the *information expert* , that is the class with the ***information necessary*** to fulfill the responsibility.

"Objects do things related to the information they have."

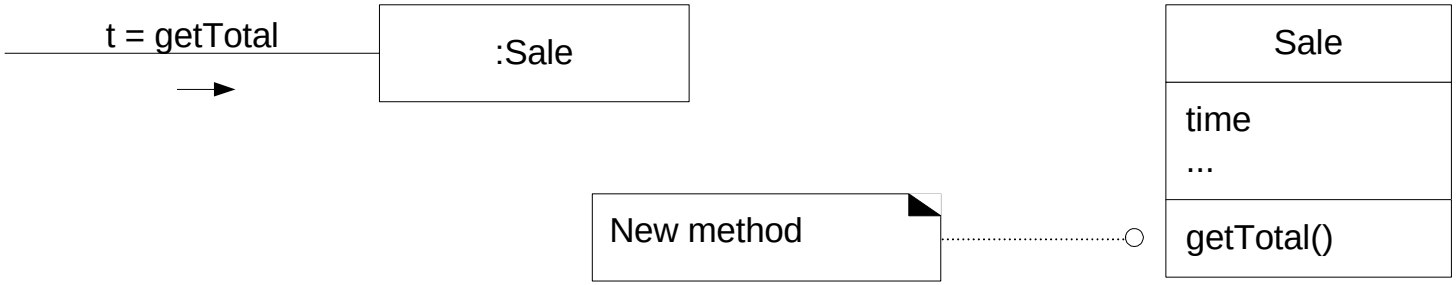
*Who should be responsible for **knowing/getting the grand total** of a sale?*



Who is responsible for ***knowing the grand total*** *of a sale?*



Partial interaction and class diagrams

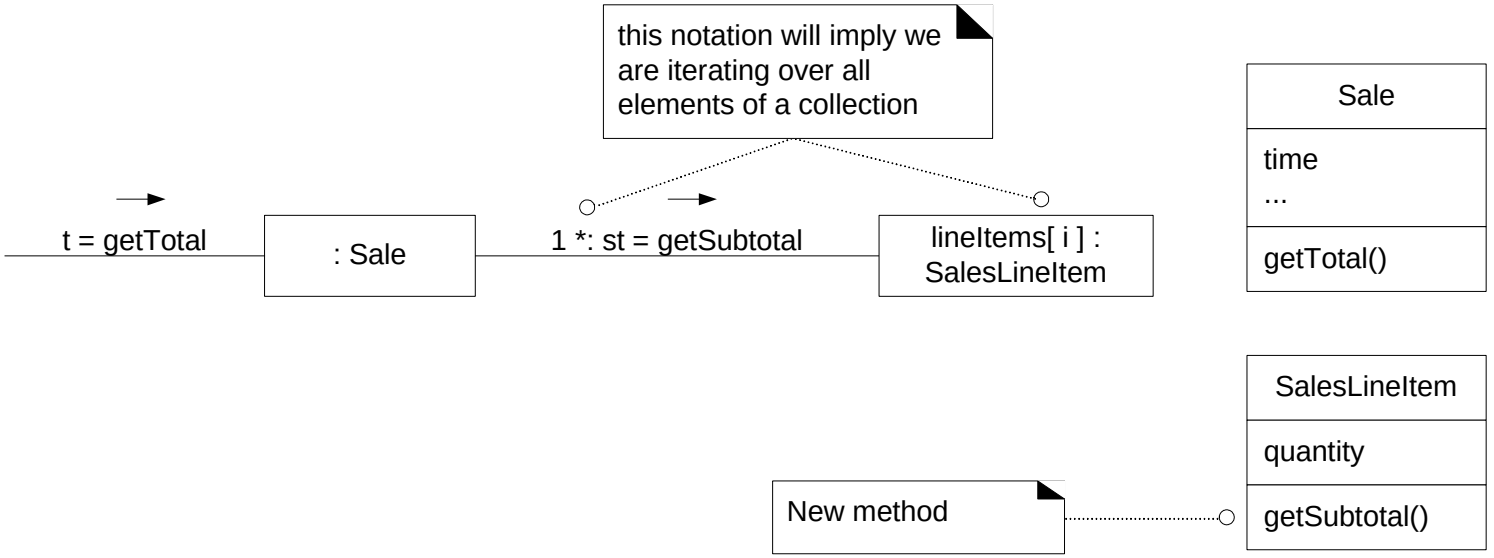


- *Add a Sale class to the Design Model.*
- *Express responsibility of knowing the total of a sale with the method named* ***getTotal****.*

What information do we need to know to determine the line item subtotal?

Sale knows about neighbours (associations), SaleLineitems who is responsible for knowing its subtotal

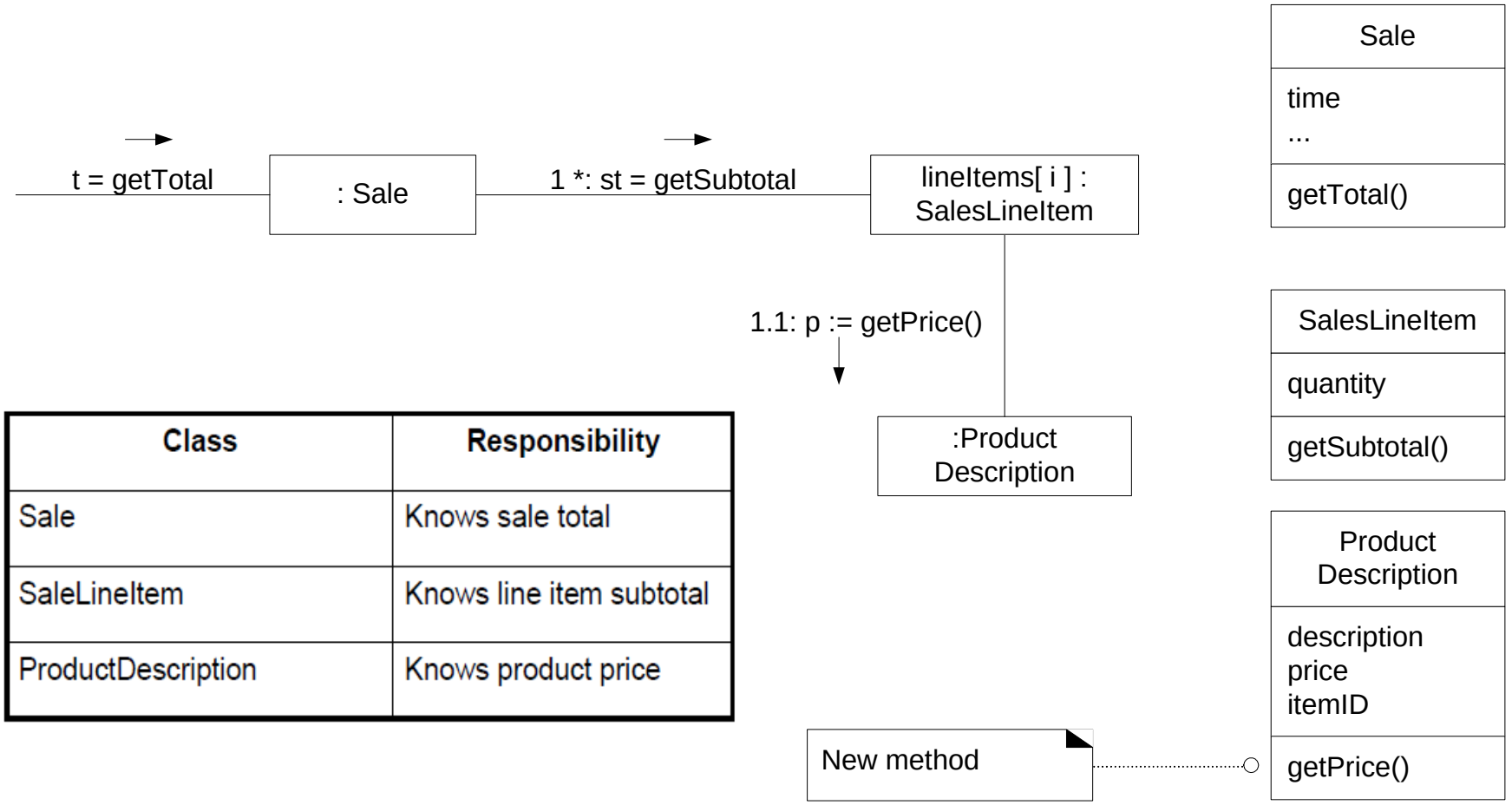
SalesLineItem is Expert for Subtotal



How does the SalesLineItem find out the product price?

SaleLineItem knows about neighbours (ProductDescription) to get the price.

ProductDescription is Expert for Price



Class	Responsibility
Sale	Knows sale total
SaleLineItem	Knows line item subtotal
ProductDescription	Knows product price

"Partial" information experts **collaborate** to fulfill the responsibility.

Low Coupling Principle

"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.

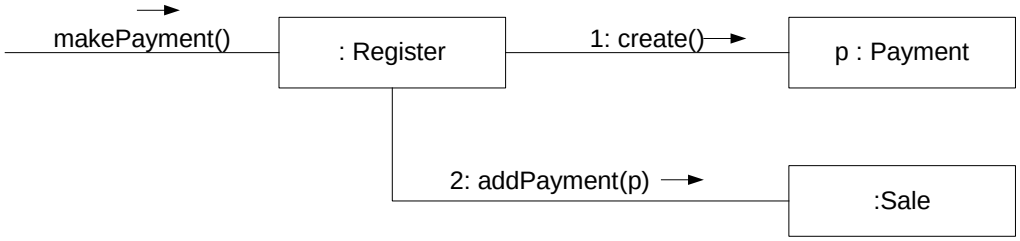
Coupling is **a measure of how strongly** one class is

- **connected to,**
- **has knowledge of**, or
- **relies upon** other classes.

What is a coupling ?

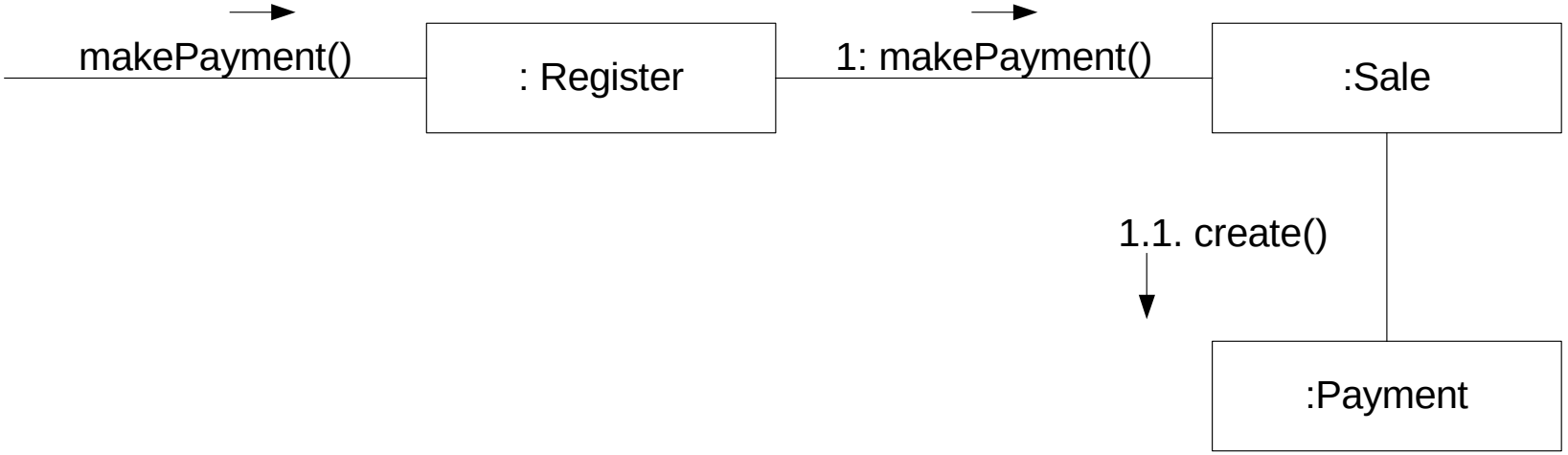
Coupling between classes is dependency of one class on another class

What is the problem if *Register* creates *Payment*



- ***Register*** **is coupled to both** ***Sale*** **and** ***Payment*****.**

What will be happen if *Sale* creates *Payment* ?



- Assuming that the ***Sale*** must eventually be ***coupled*** to knowledge of a ***Payment***, having *Sale* create the *Payment* does not increase coupling.

NB : Low Coupling and Creator may suggest different solutions.

High Cohesion

High Cohesion

A class with ***low cohesion*** does too much unrelated work and are:

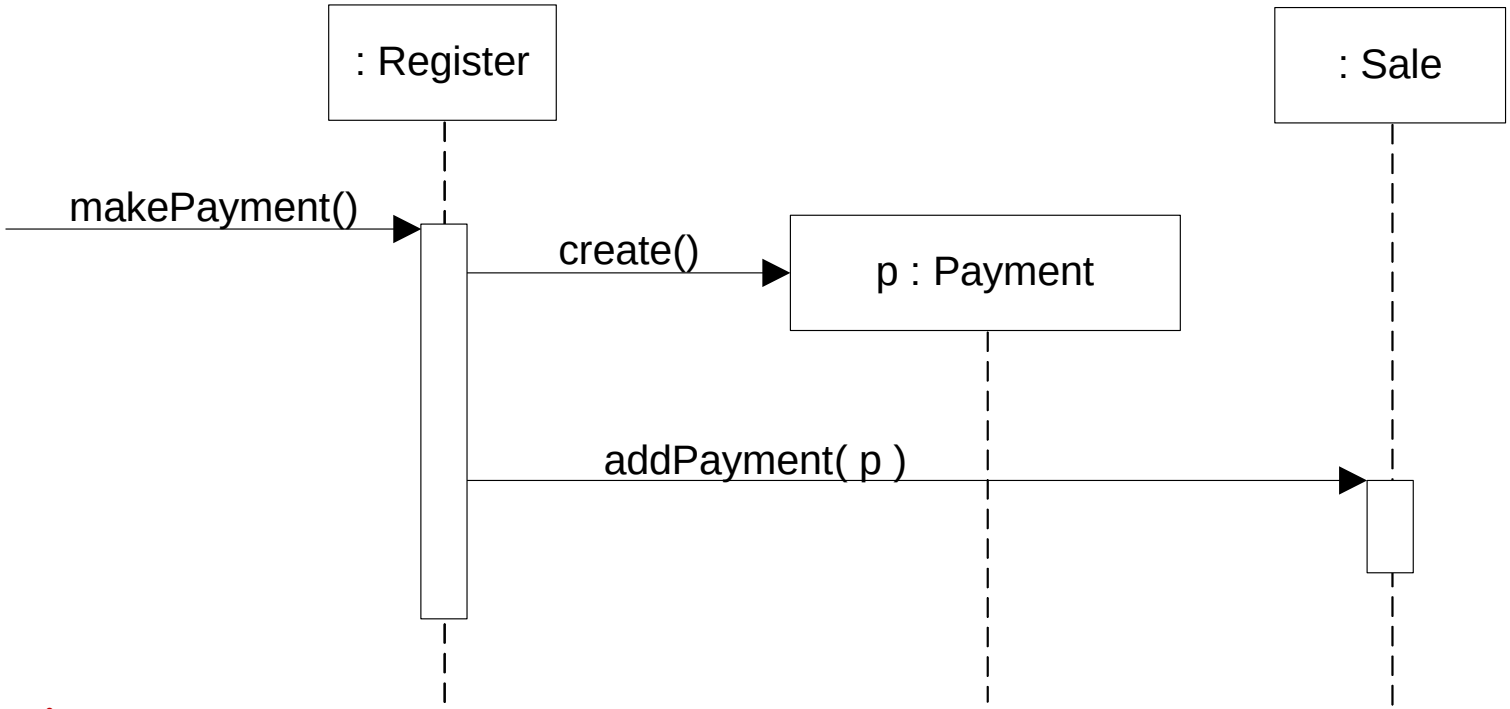
- Hard to comprehend
- Hard to reuse.
- Hard to maintain.
- Delicate and constantly affected by change

Cohesion is a measure of *how strongly related the responsibilities of an element (classes, subsystems) are.*

High Cohesion

- **Problem**
  - How to keep complexity manageable?
- **Solution**
  - Assign a responsibility so that cohesion remains high

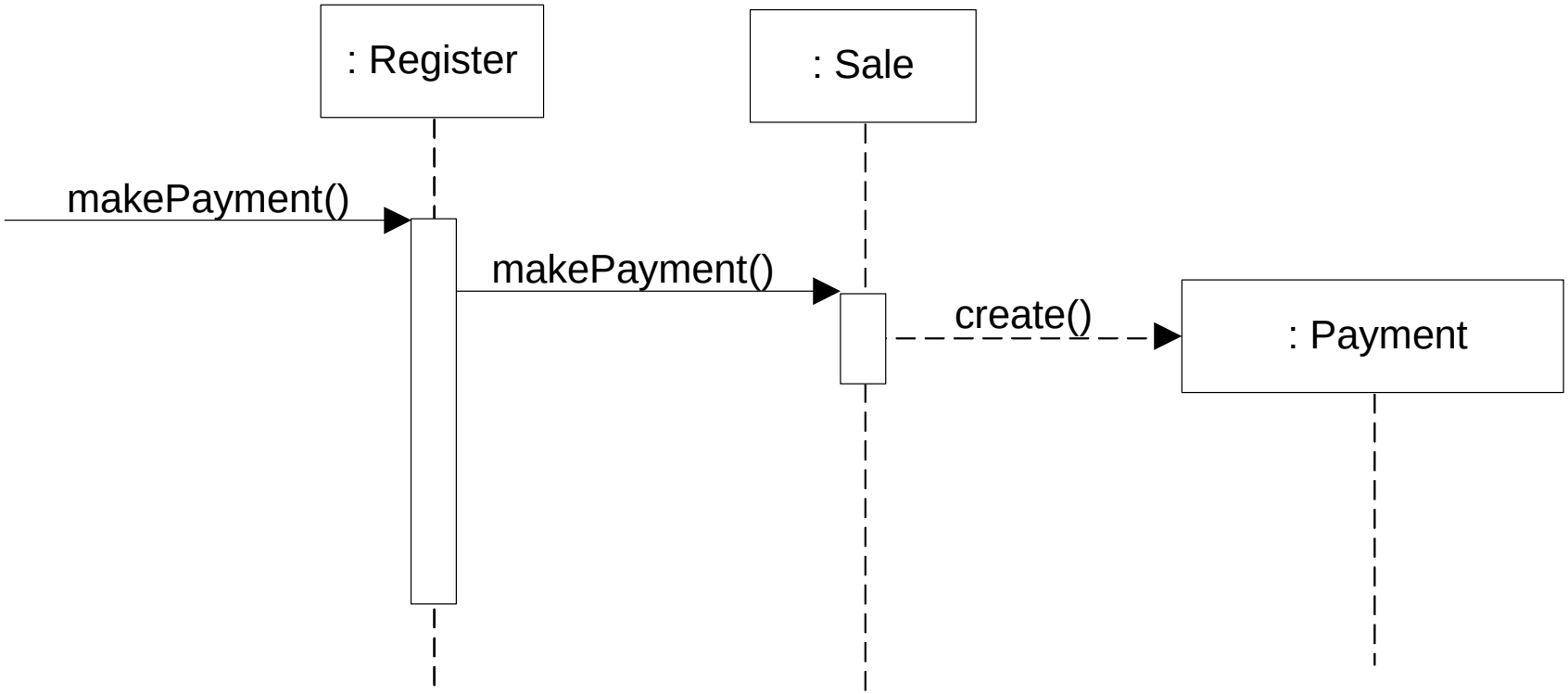
Reduced cohesion of *Register(creator pattern)*



Low cohesion:

***Register* is taking part of the responsibility for fulfilling "*makePayment*" operation and many other unrelated responsibility (50 system operations all received by *Register*). Thus, it will become burden with tasks and become incoherent**

Better solution
Higher Cohesion and Lower Coupling



***Solution*:**

Delegate the *payment* creation responsibility to "*Sale*" to support high cohesion

Controller Pattern

Controller Pattern

UI layer does not contain any business logic

Problem:
How to connect UI layer to the business logic layer?

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.

Controller Pattern

What first object beyond the UI layer receives and coordinates ("controls") a system operation message?

- ***Solution:*** *Assign the responsibility to a class that represents one of the following options:*

Options for Control Responsibility

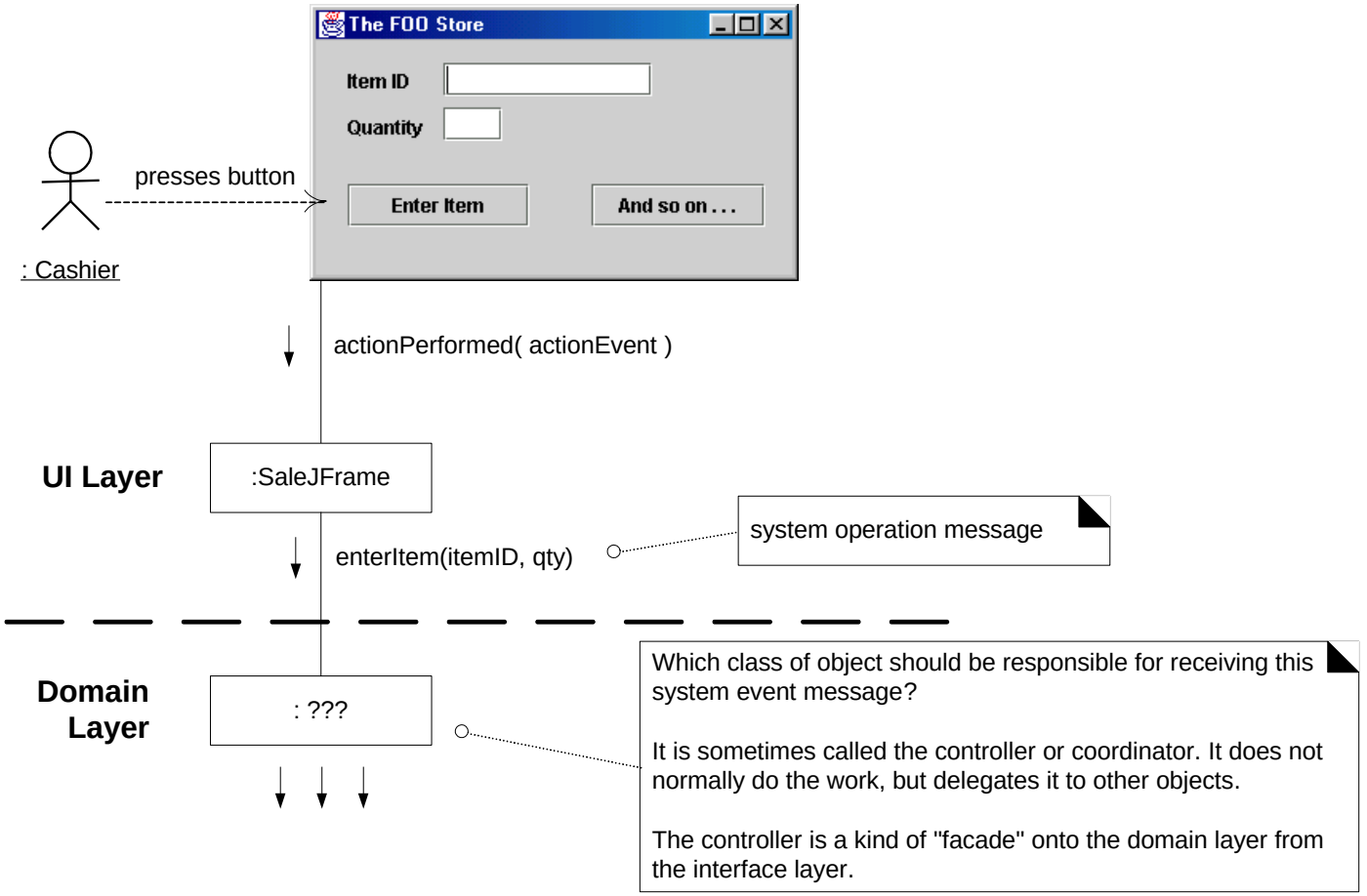
1. ***Representing the overall system or a root object.***

   e.g., an object called ***System or Register***

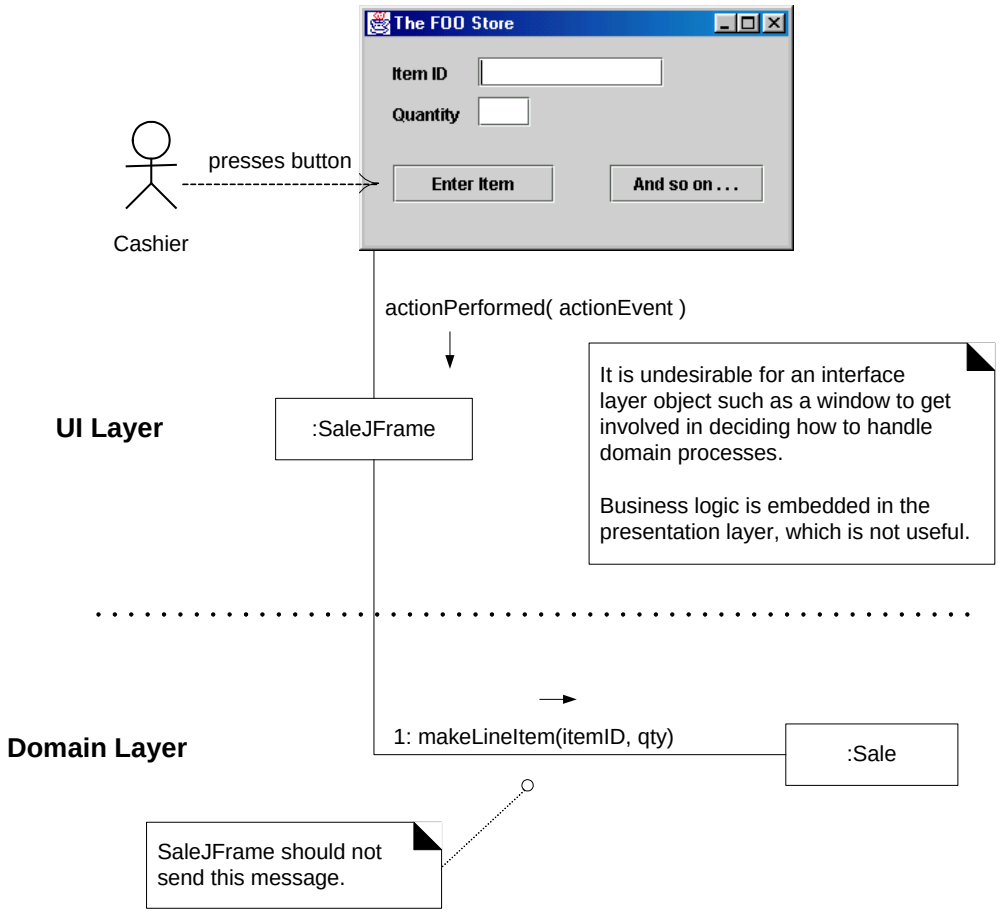
   Suitable when there are not too many system events or when UI cannot choose between multiple controllers.

2. ***A controller for each use case***

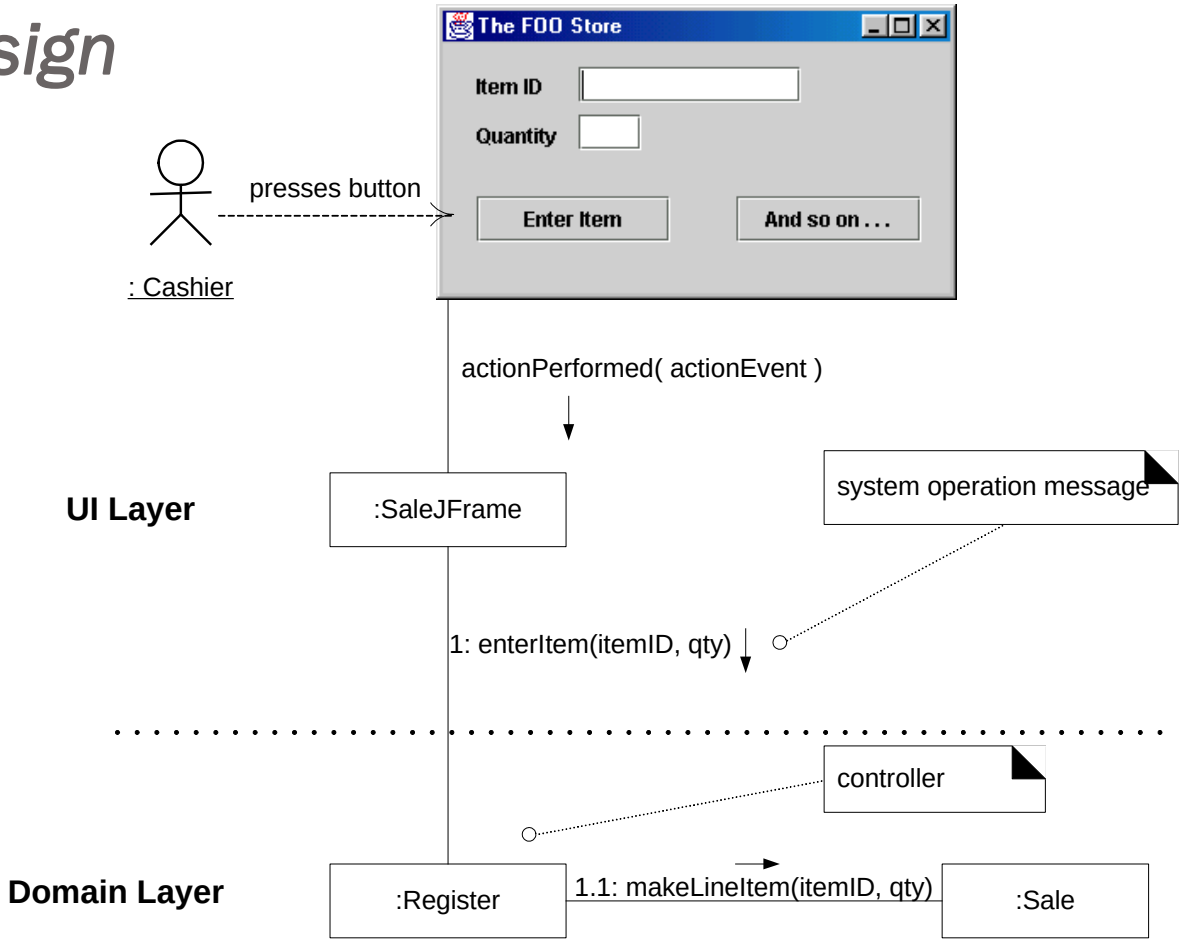
What should be Controller for enterItem?



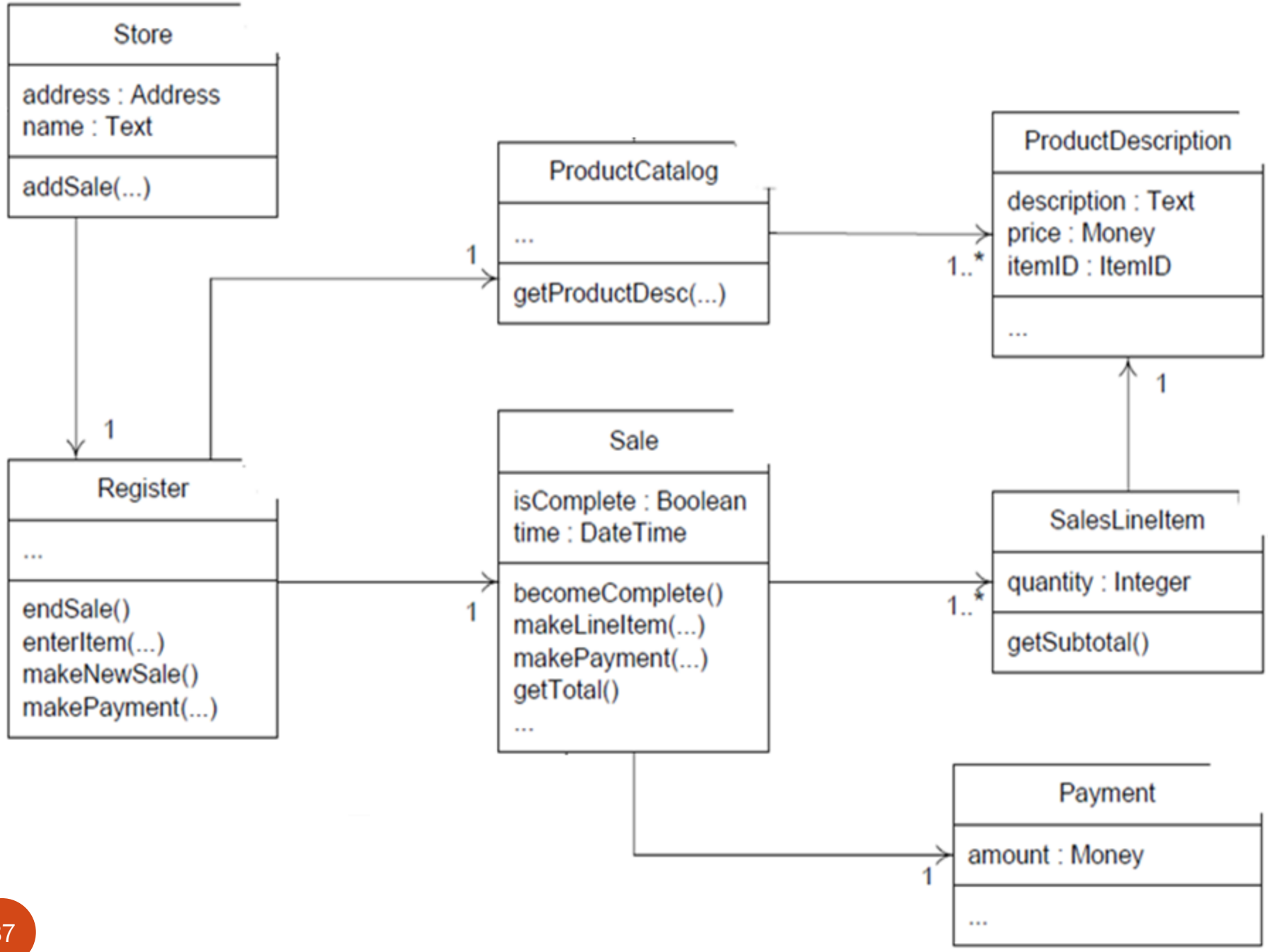
Bad Design


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
Cashier
presses button
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
It is undesirable for an interface layer object such as a window to get 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.

Good Design



Controller should delegate the work that needs to be done to other objects.


Store
address : Address
name : Text
addSale(...)
ProductCatalog
...
getProductDesc(...)
ProductDescription
description : Text
price : Money
itemID : ItemID
...
Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)
Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
...
SalesLineItem
quantity : Integer
getSubtotal()
Payment
amount : Money
...
1
1
1
1..*
1
1..*
1