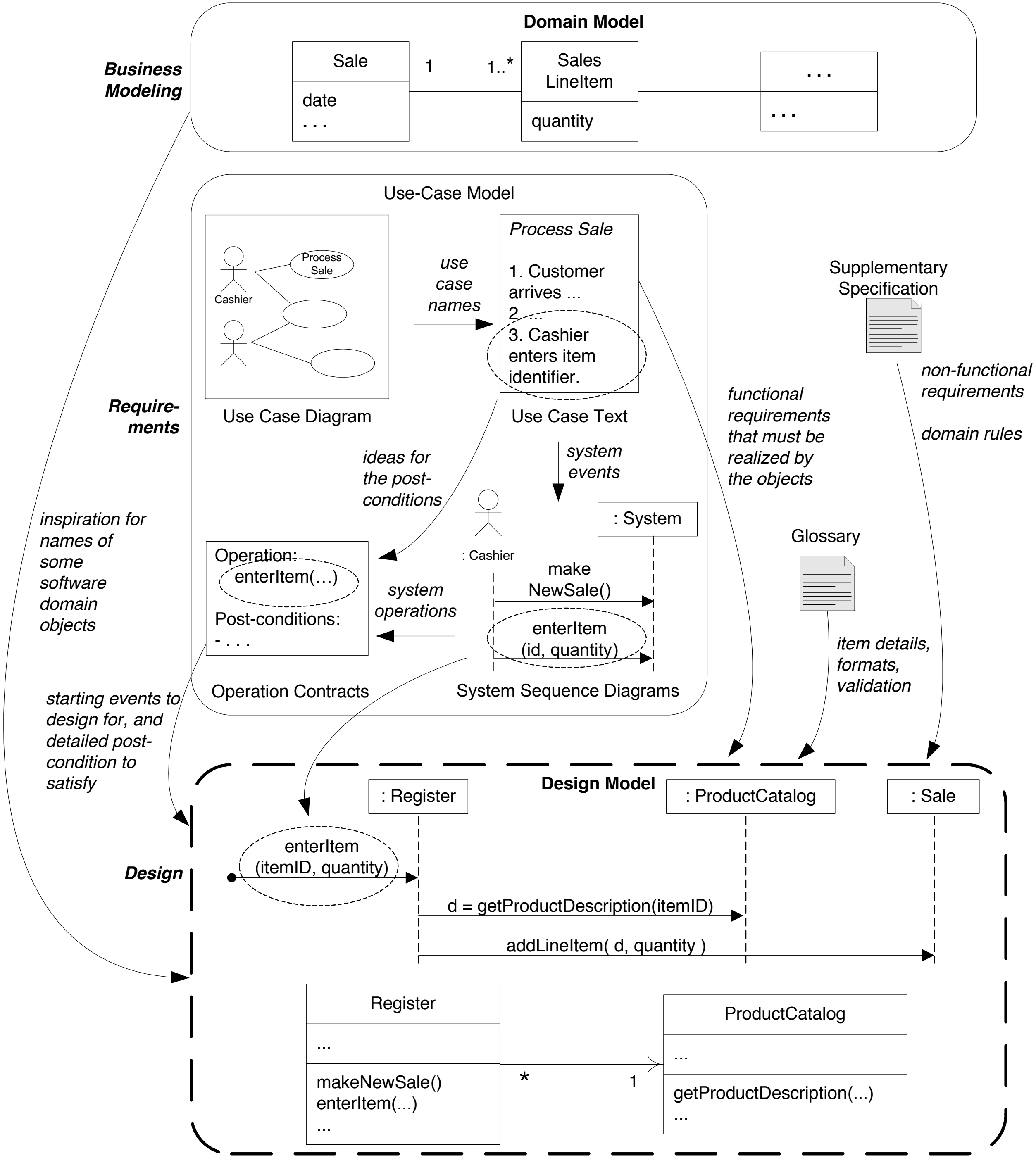
Chapter 17

GRASP Design Patterns:

Designing Objects with Responsibilities

Sample UP Artifact Relationships



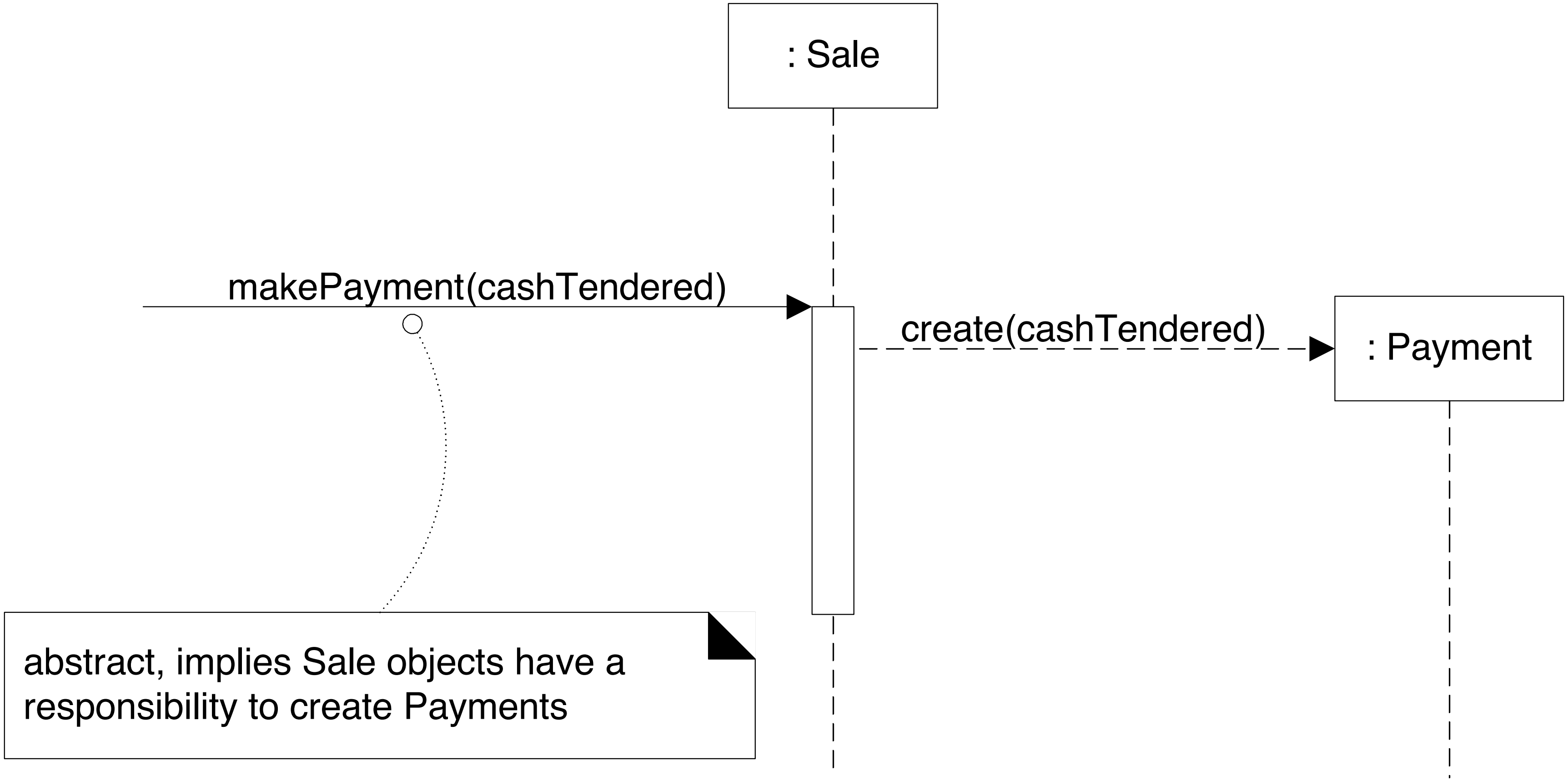
Responsibility-Driven Design (RDD)

- A way of thinking about OOD:
  - In terms of
    - Responsibilities
    - Roles
    - Collaborations
- Common responsibility categories:
  - Doing:
    - Doing something itself:
      - Creating and object or doing a calculation
    - Initiating action in other objects
    - Controlling and coordinating activities in other objects
  - Knowing:
    - Knowing about private data
    - Knowing about related objects
    - Knowing about things it can derive or calculate
- Bigger responsibilities may take several classes
- Guideline:
  - Domain model helps with “knowing”
  - Interaction diagrams help with “doing”

RDD and Collaboration

- Responsibilities implemented by methods
- Some methods act alone and do a job
- Some collaborate with other objects and methods
  - Example: Sale class has getTotal() method
  - Sale and getTotal() collaborate with other objects to fulfill a responsibility
    - SalesLineItem objects and getSubtotal() methods
- RDD: Community of collaborating objects

RDD and Interaction Diagrams



abstract, implies Sale objects have a responsibility to create Payments

- We decide on responsibility assignment when drawing interaction diagrams

Patterns

- Patterns: A repertoire of
  - general principles
  - idiomatic solutions

to guide us in the creation of software

- A pattern: A named and well-known problem/solution pair that
  - Can be applied in new contexts
  - With advice on how to apply it in novel situations
  - With a discussion of its trade-offs, implementations, variations, …
- Names facilitate communication about software solutions
- Some patterns may seem obvious: That’s a good thing!

Pattern Example

- Pattern name

  Information Expert

- Problem:

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

- Solution

  Assign a responsibility to the class that has the information needed to fulfill it

Well-known Pattern Families

- GRASP: General Responsibility Assignment Software Patterns or Principles
- GoF: Gang of Four Design Patterns
  - Very well-known book
  - 23 patterns
  - We'll cover some of these
- We start with GRASP

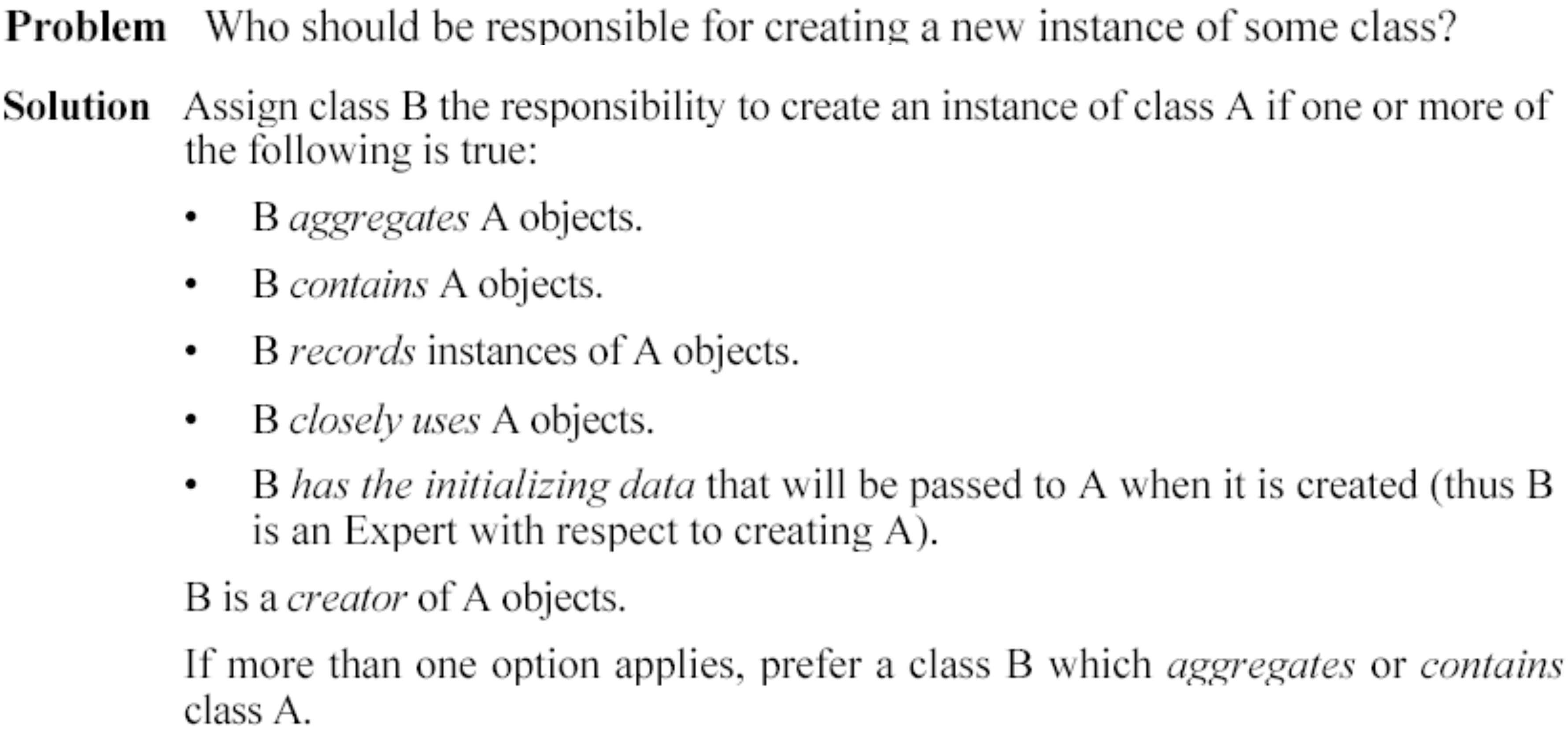
GRASP Patterns

- There are 9 GRASP patterns
- We start with an example (Monopoly) demonstrating the first five
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion

The "Creator" Pattern

- Problem: Who creates the Square object in a Monopoly game?
- Java answer: Any object could
- But what makes for good design?
- Common answer: Make Board create Square objects
- Intuition: Containers should create the things contained in them
- Issue: Wait, but we don't have any classes yet, we only have the domain model!
  - We have to start somewhere, and we look at the domain model for inspiration to pick our first few important software classes

The Creator Pattern



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

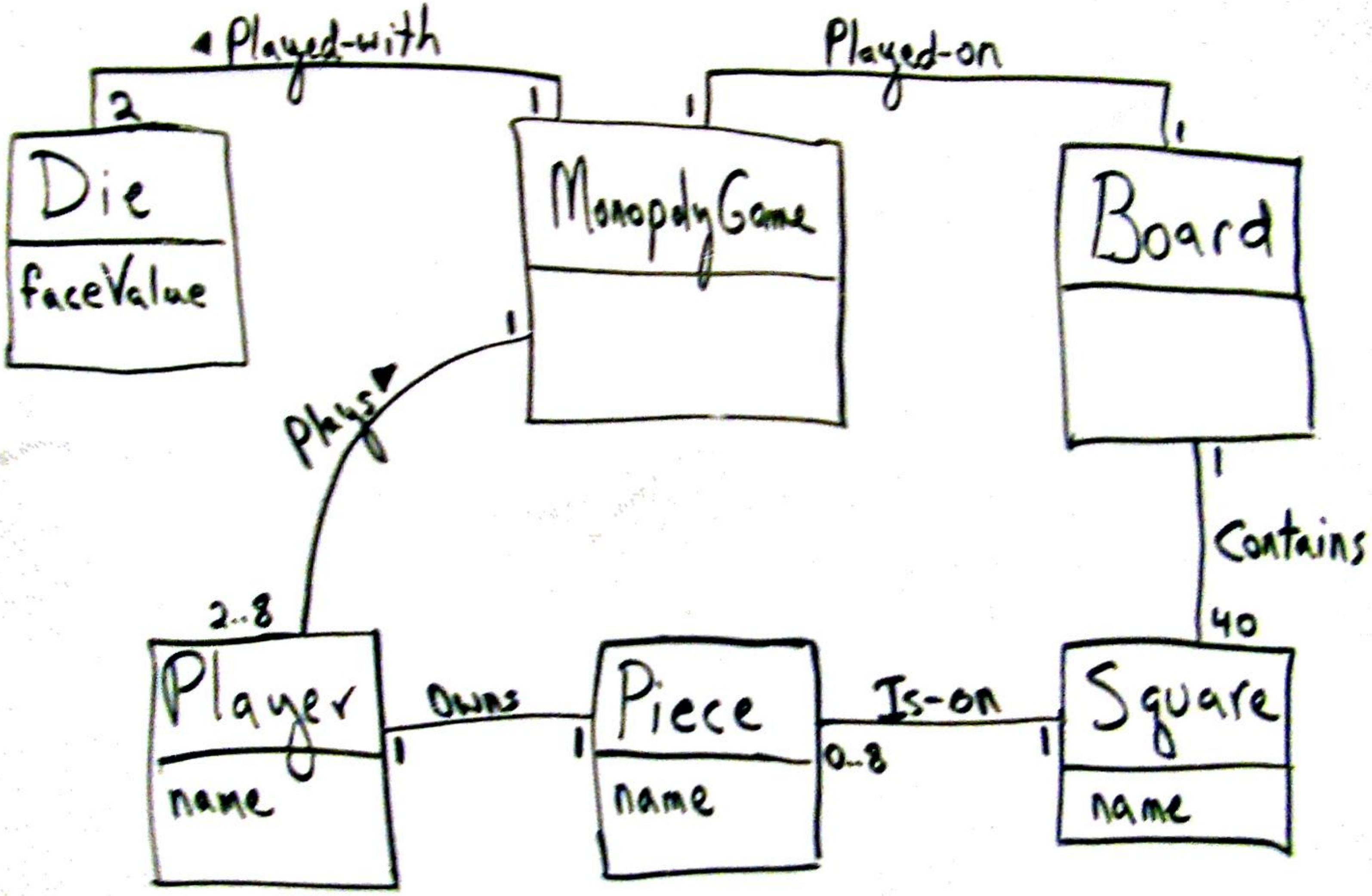
Solution Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B *aggregates* A objects.
- 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 (thus B is an Expert with respect to creating A).

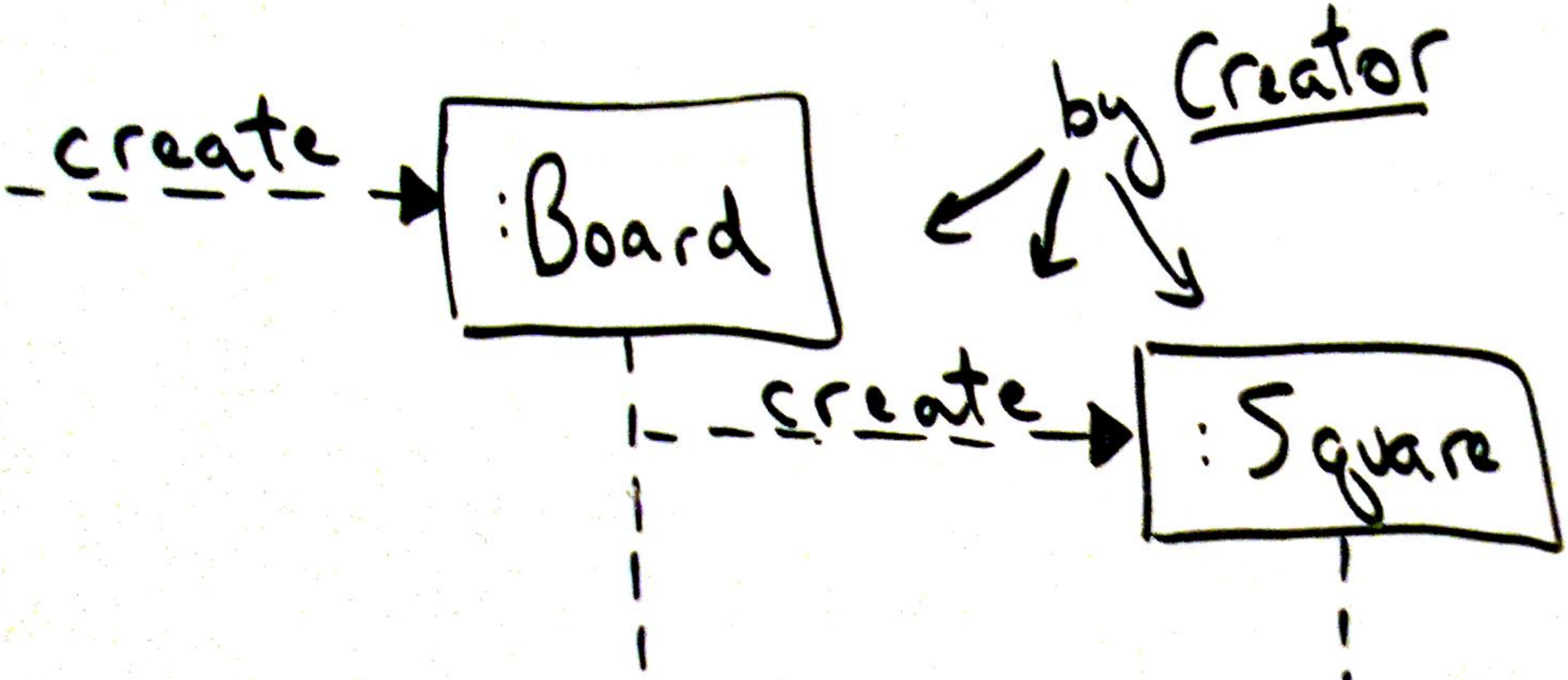
B is a *creator* of A objects.

If more than one option applies, prefer a class B which *aggregates* or *contains* class A.

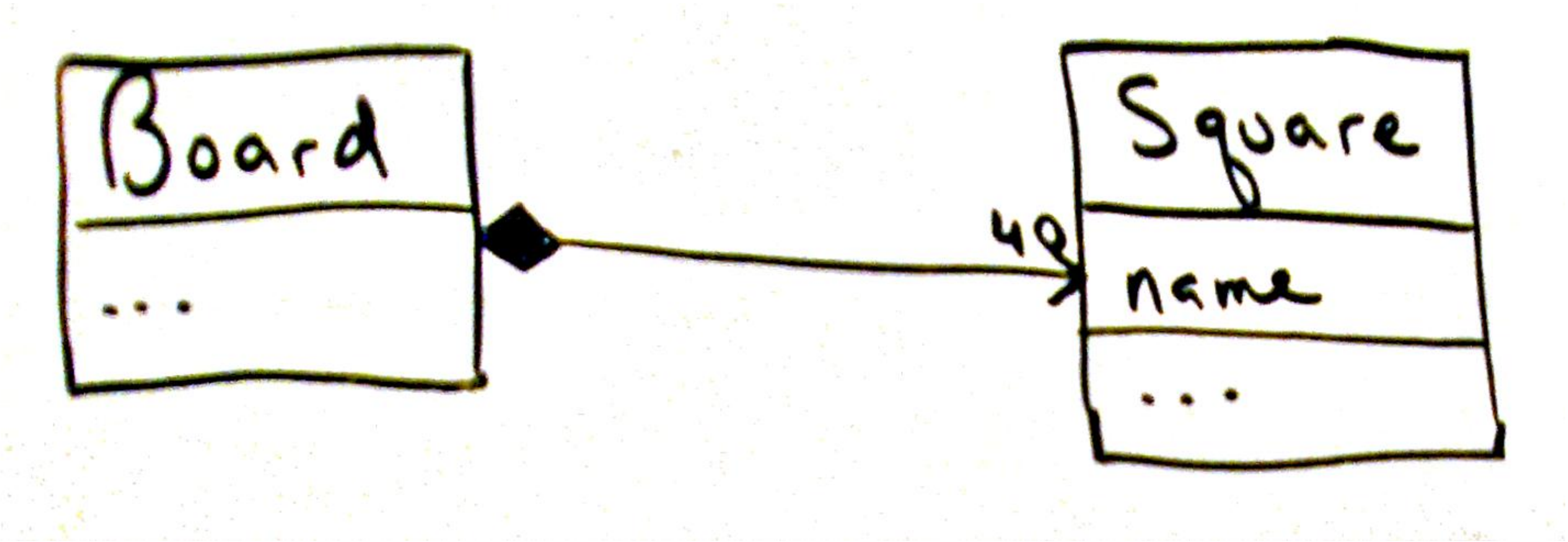
Monopoly Iteration-1 Domain Model



Applying the Creator Pattern in a Dynamic Model



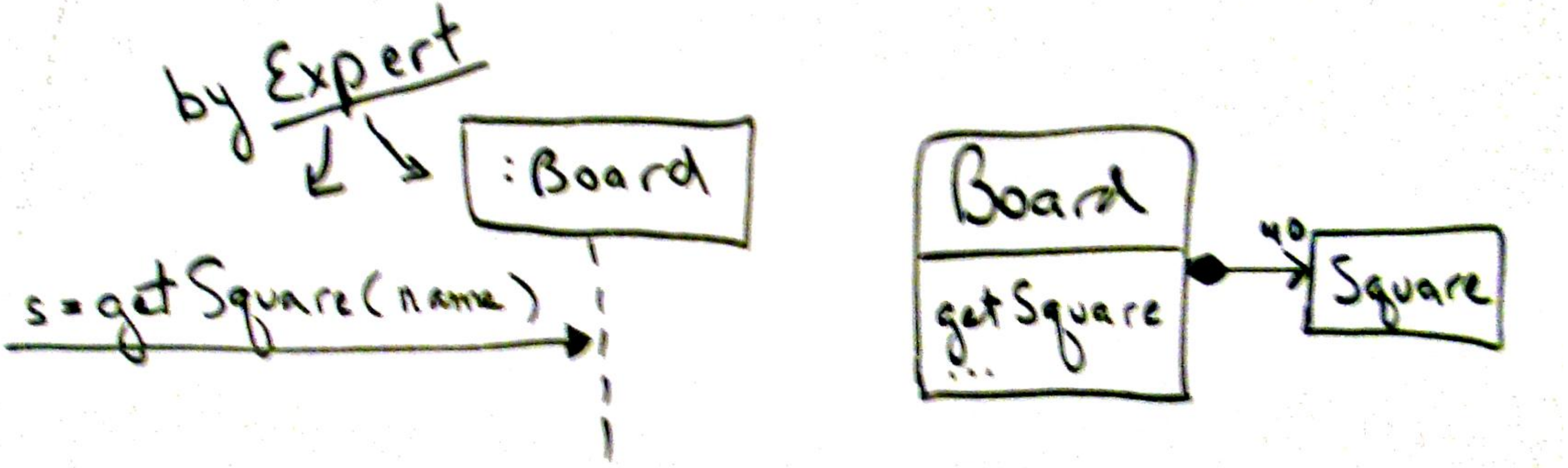
Design Class Diagram Inspires Use of the “Creator” Pattern



The “Information Expert” of “Expert” Pattern

- Problem: Given the name of a Monopoly square, we want to get a reference to the Square object with that name
- Most developers would choose the Board object to do this job
- Intuition:
  - A responsibility needs information for its fulfillment
    - Info about other objects,
    - an object’s own state
    - The world around an object
    - information an object can derive
- The object that will do the job must know all Squares
- Board has the information necessary to fulfill the responsibility

The “Expert” Pattern and UML Diagrams



The “Information Expert” Pattern

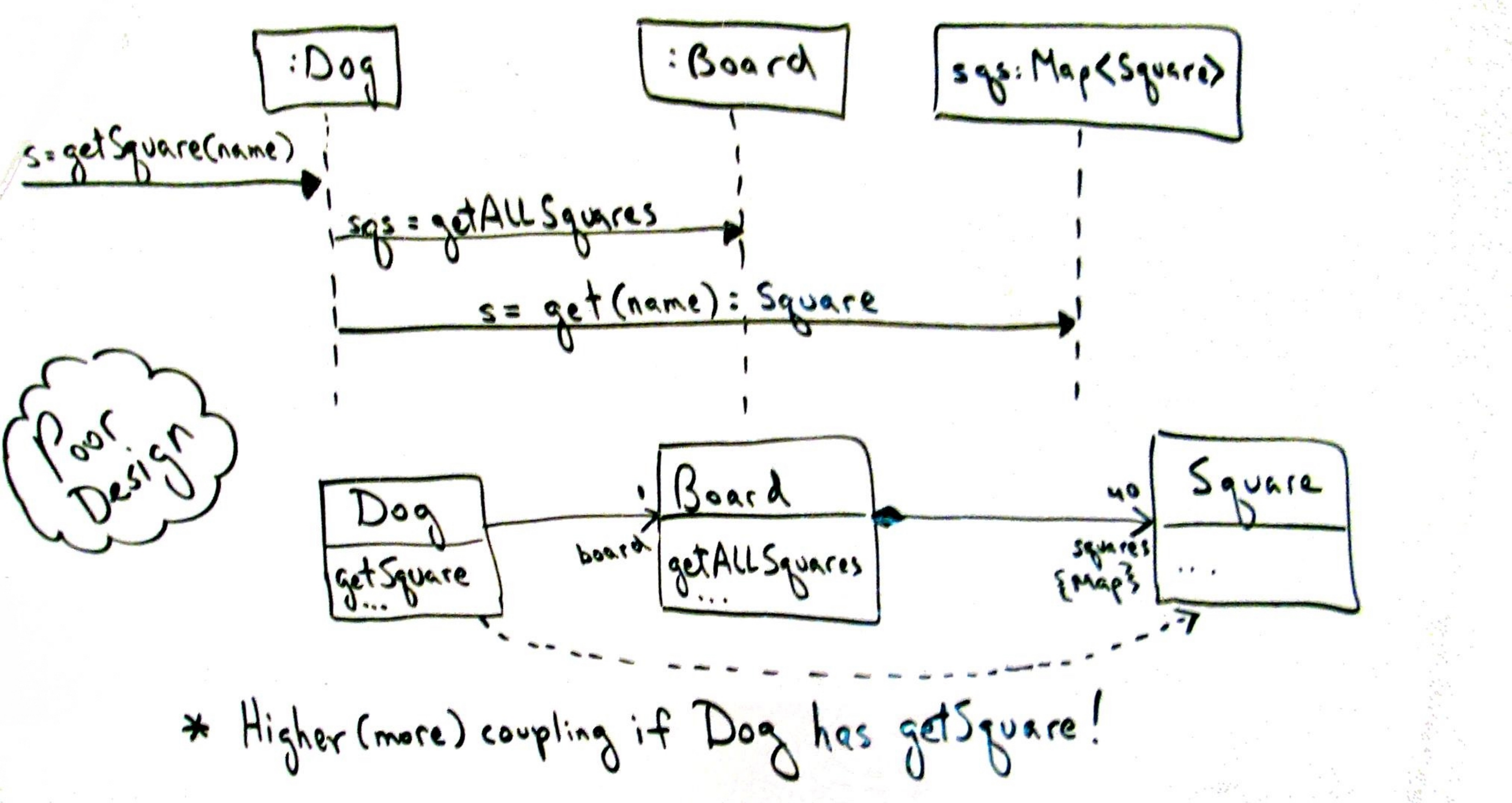
- Name Information Expert or Expert

- 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

Evaluating the Effect of Coupling

- Coupling: A measure of how strongly one element is connected to, has knowledge of, or depends on other elements



The “Low Coupling” Principle

- Name Low Coupling

- Problem: How to reduce the impact of change on software?

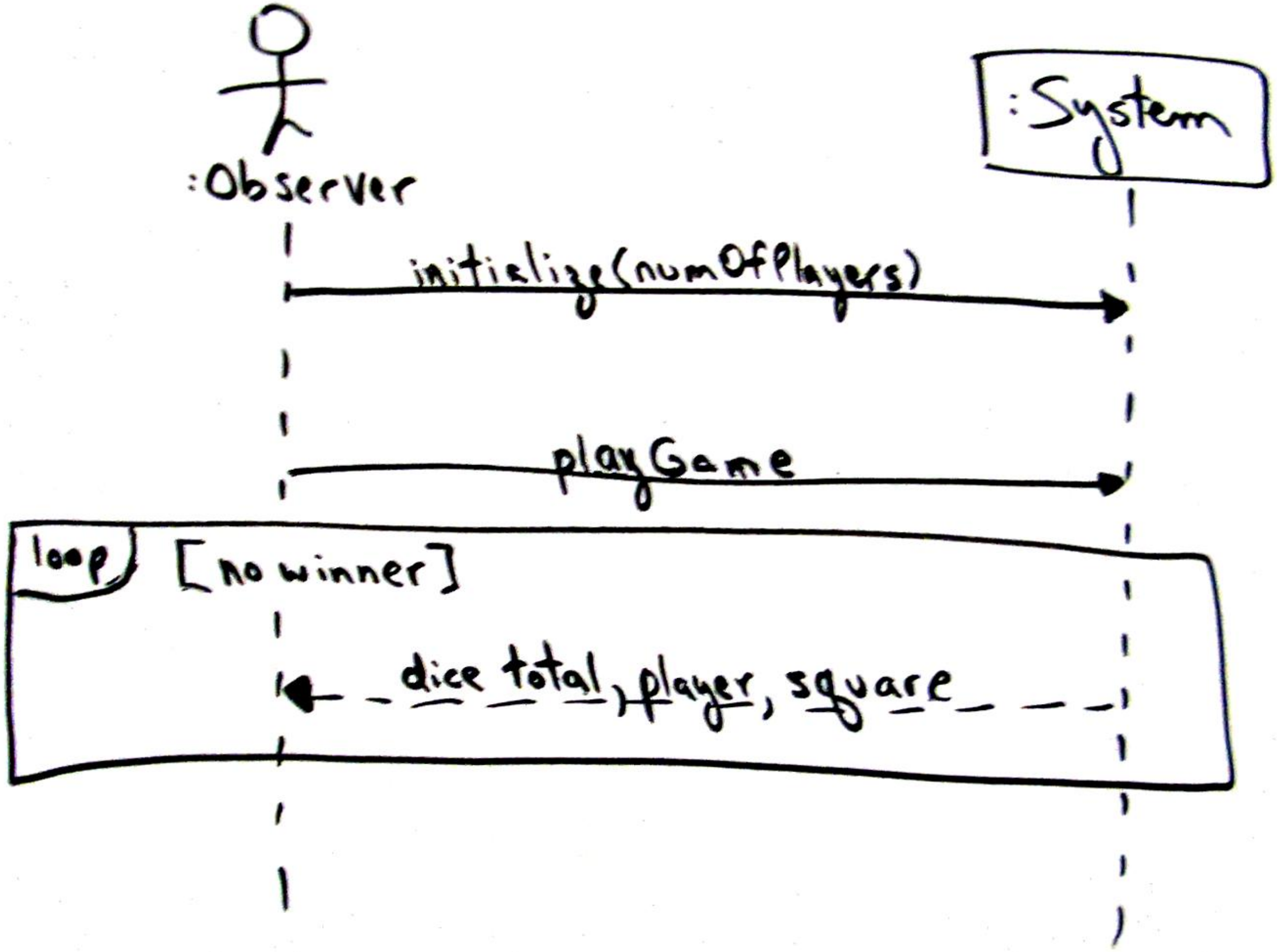
- Solution (advice): Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

Why is low coupling good?

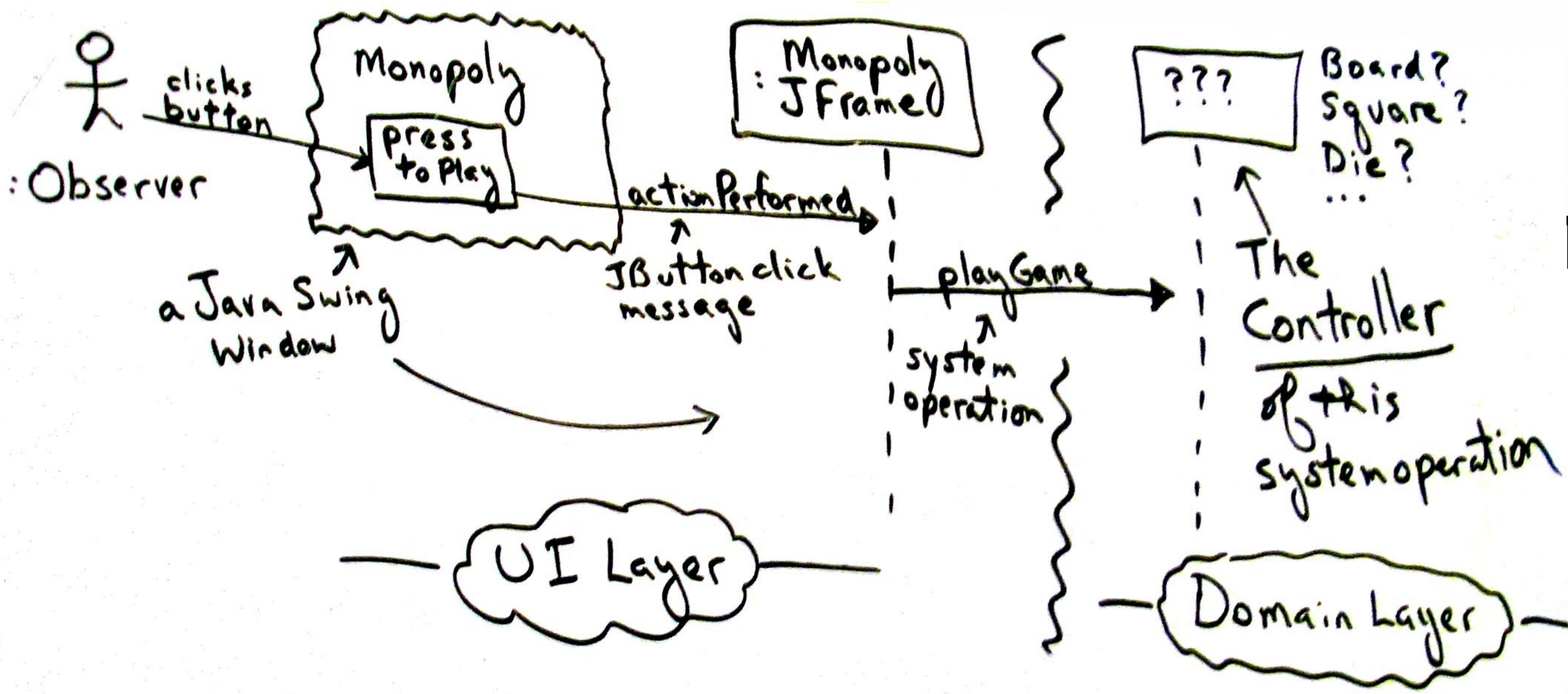
- It reduces time, effort and defects involved in modifying software
- The “Expert” pattern supports low coupling

The "Controller Pattern"

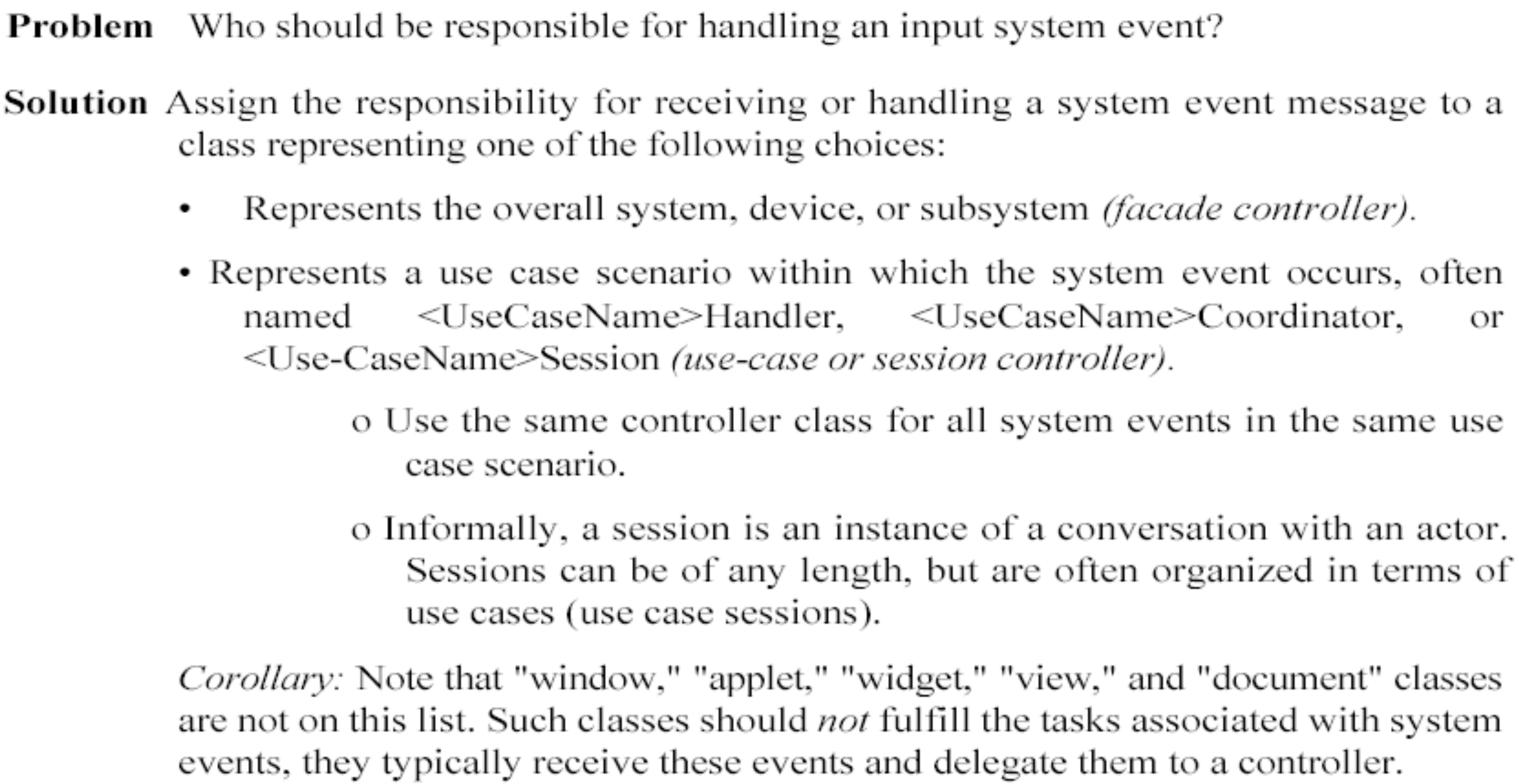
- Question: What first object after or beyond the UI layer should receive the message from the UI layer?



A Finer-Grained Look



The “Controller” Pattern



Problem Who should be responsible for handling an input system event?

Solution Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

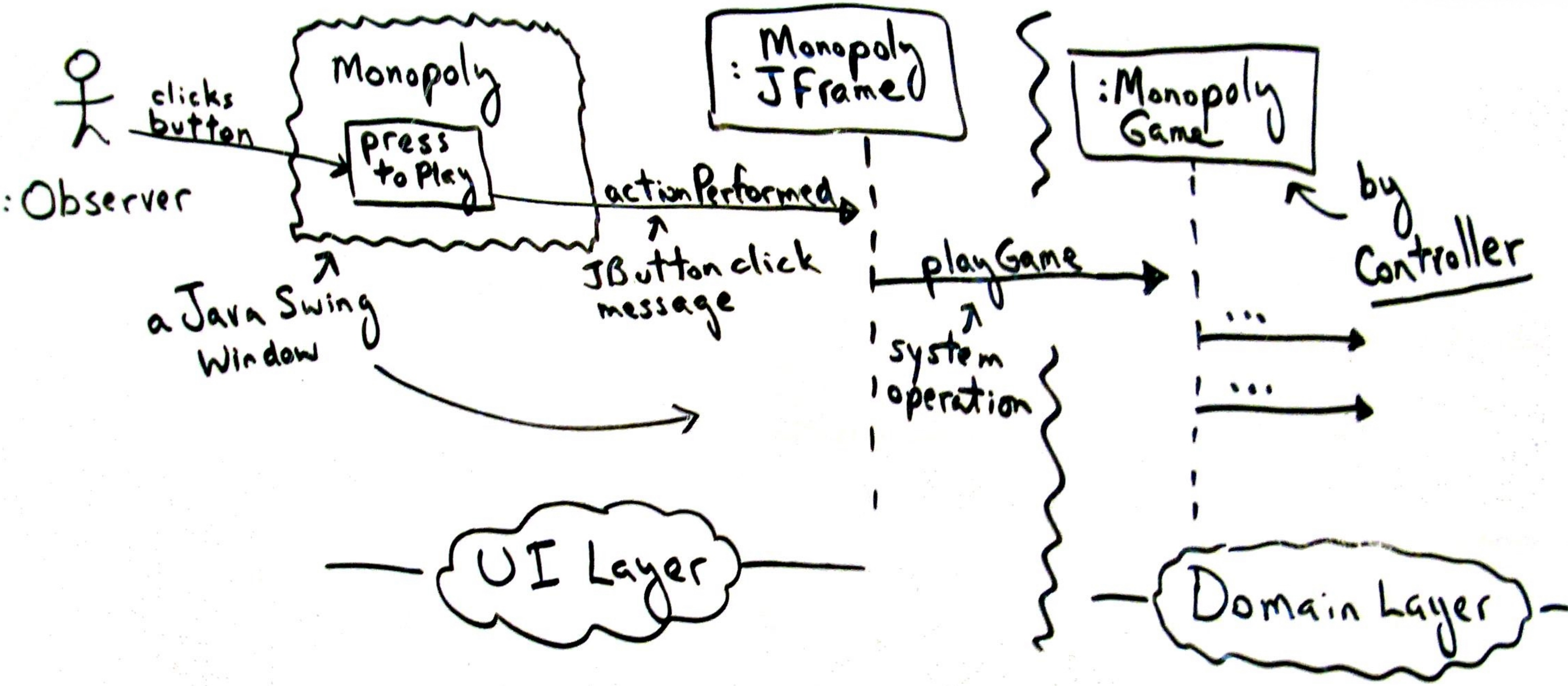
- Represents the overall system, device, or subsystem *(facade controller).*
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session *(use-case or session controller).*
  - o Use the same controller class for all system events in the same use case scenario.
  - o Informally, a session is an instance of a conversation with an actor. Sessions can be of any length, but are often organized in terms of use cases (use case sessions).

Corollary: Note that "window," "applet," "widget," "view," and "document" classes are not on this list. Such classes should *not* fulfill the tasks associated with system events, they typically receive these events and delegate them to a controller.

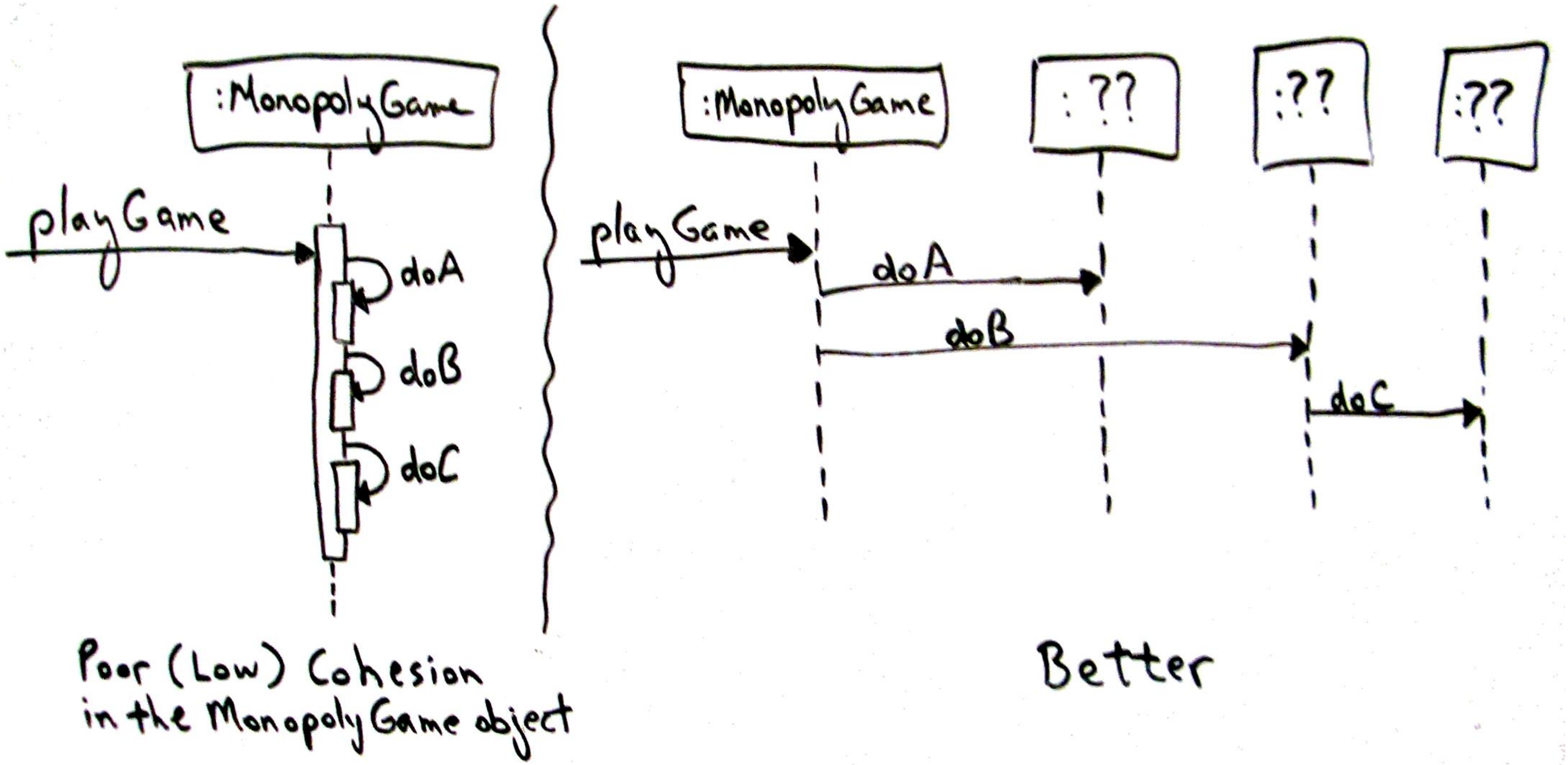
Monopoly Controller Alternatives

- Option 1: Represents the overall “system” or a “root object”
  - An object called MonopolyGame
- Option 1: Represents a device that the software is running within
  - It doesn’t really apply here.
  - It applies to designs with specialized hardware devices: Phone, BankCashMachine, …
- Option 2: Represents the use case or session.
  - The use case that the playGame system operation occurs in is called Play Monopoly Game
  - A class such as PlayMonopolyGameHandler (or ...Session) might be used
- The very first option looks good if there are only a few system operations

Monopoly Design Based on the "Controller" Pattern



Considering Alternatives Using the “High Cohesion” Principle



High Cohesion

- 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.

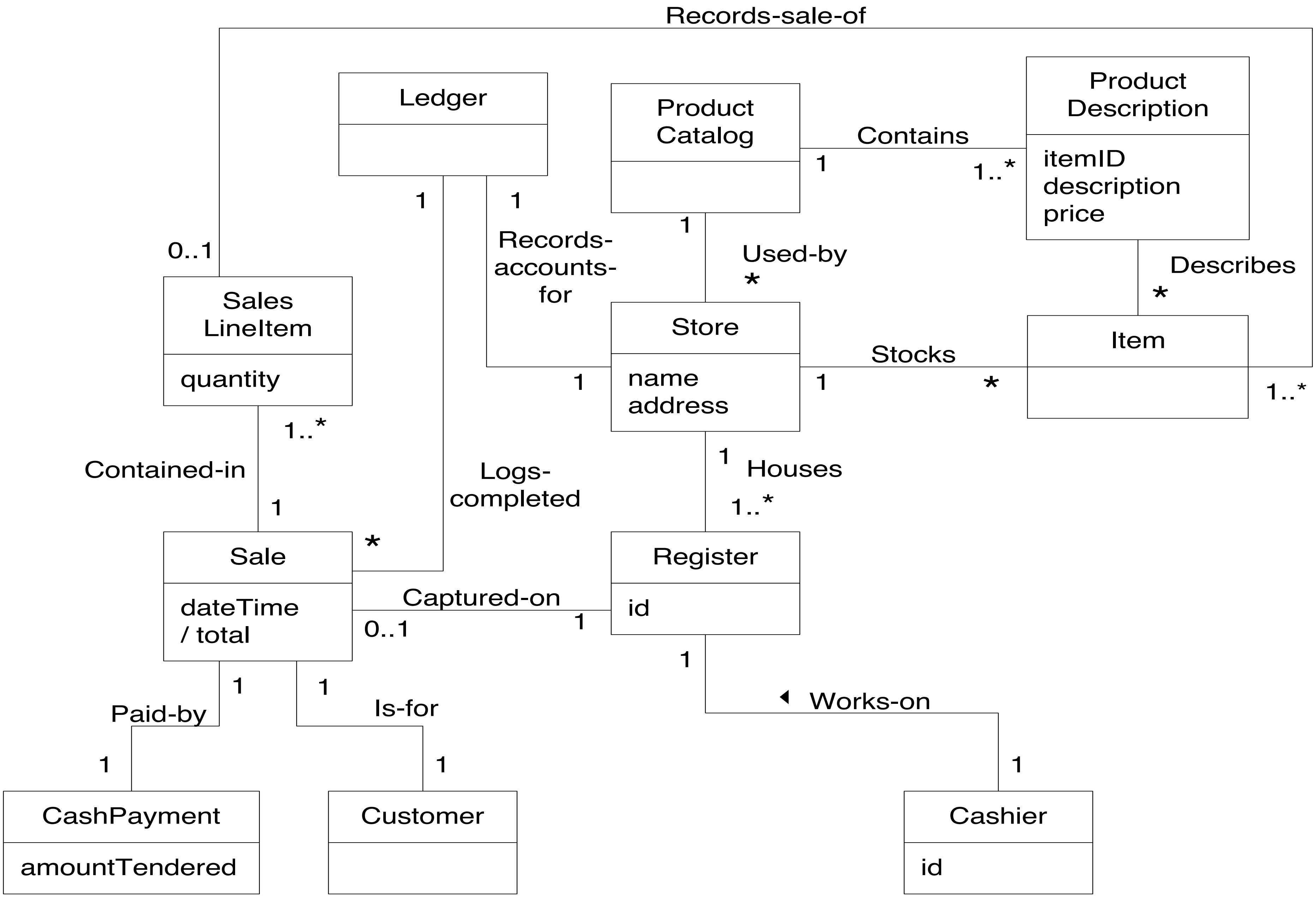
The “High Cohesion” Principle

- Name: High Cohesion
- Problem: How to keep objects focused, understandable, and manageable, and as as side effect, support low coupling?
- Solution (advice): Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

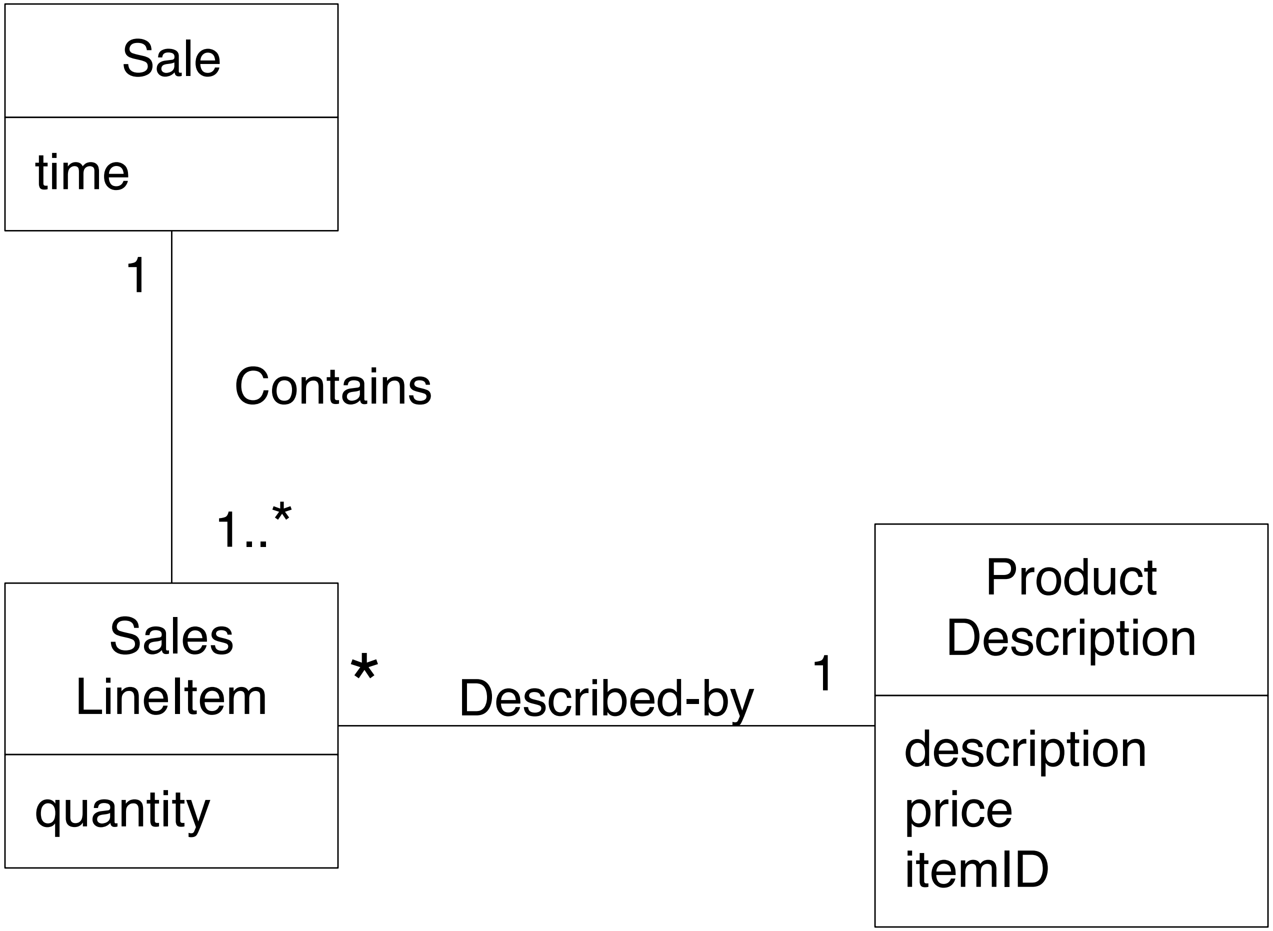
A more detailed look at “Creator”

- 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 compositely aggregates A
  - B records A
  - B closely uses A
  - B has the initializing data for A that will be passed to A when it is created. Thus, B is an Expert with respect to creating A.
- If more than one of the above applies, prefer a class B which aggregates or contains A.

POS Domain Model Example:

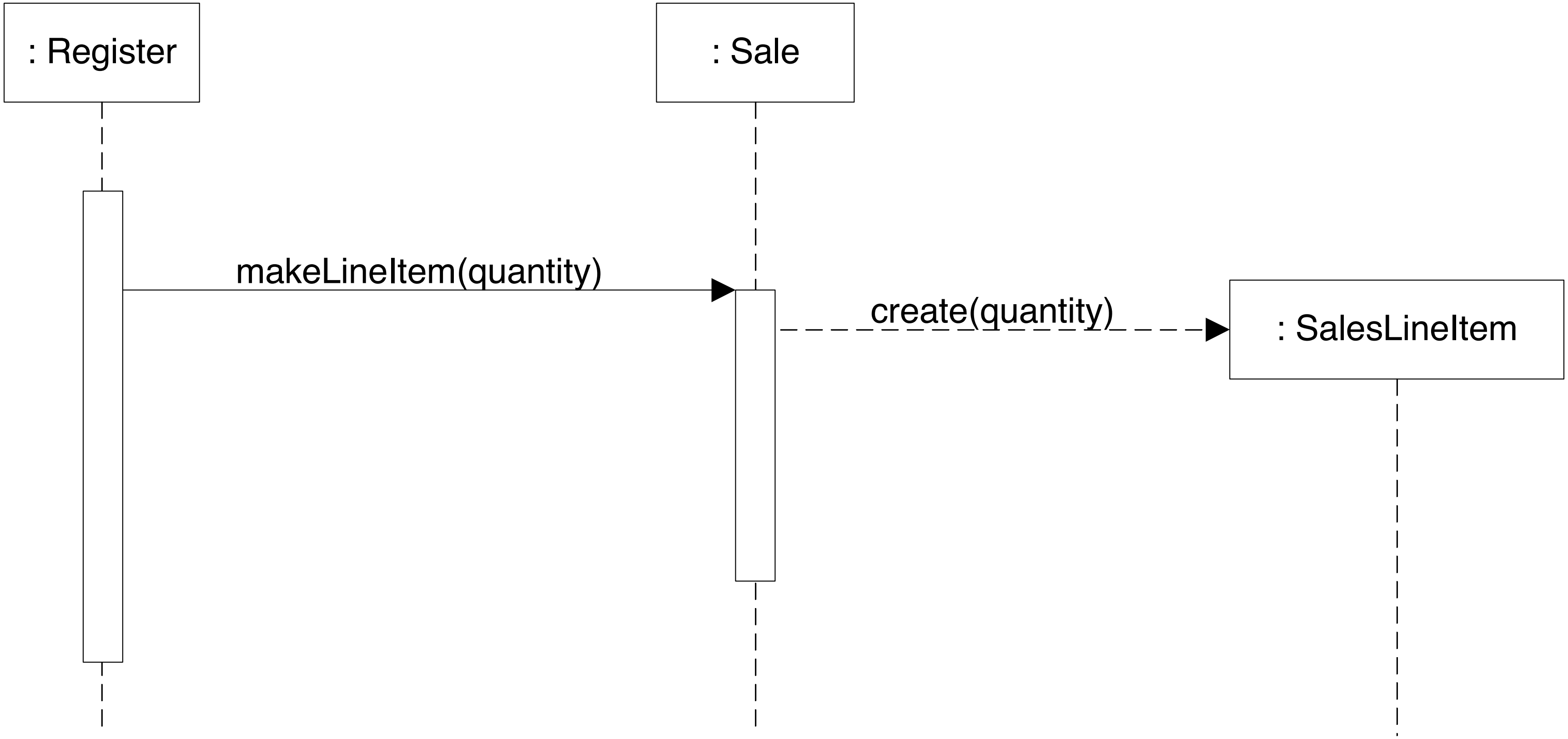


Partial Domain Model



- Basic intent of the "Creator" pattern:
  Find a creator that needs to be connected to (dependent on or associated with) the created object anyway. Supports low coupling.

Creating a SalesLineItem object



The “Creator” pattern

- Another use: Identify a creator by looking for a class that has the initialization data that will be passed in during creation.
  - Actually an example of the “Expert” pattern
- Initialization data passed in to an “initialization method”
  - Often a constructor with parameters
  - Or a factory method
- Example:
  - A Payment instance, when created needs to be initialized with the Sale total.
  - Sale knows Sale total. Good candidate for creating Payment.

Information Expert

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert
  - The information expert: The class that has the information to fulfill the responsibility
- Guideline: Start by clearly stating the responsibility
  - Example: Who should be responsible for finding out what checker is located at a given coordinate on the board?

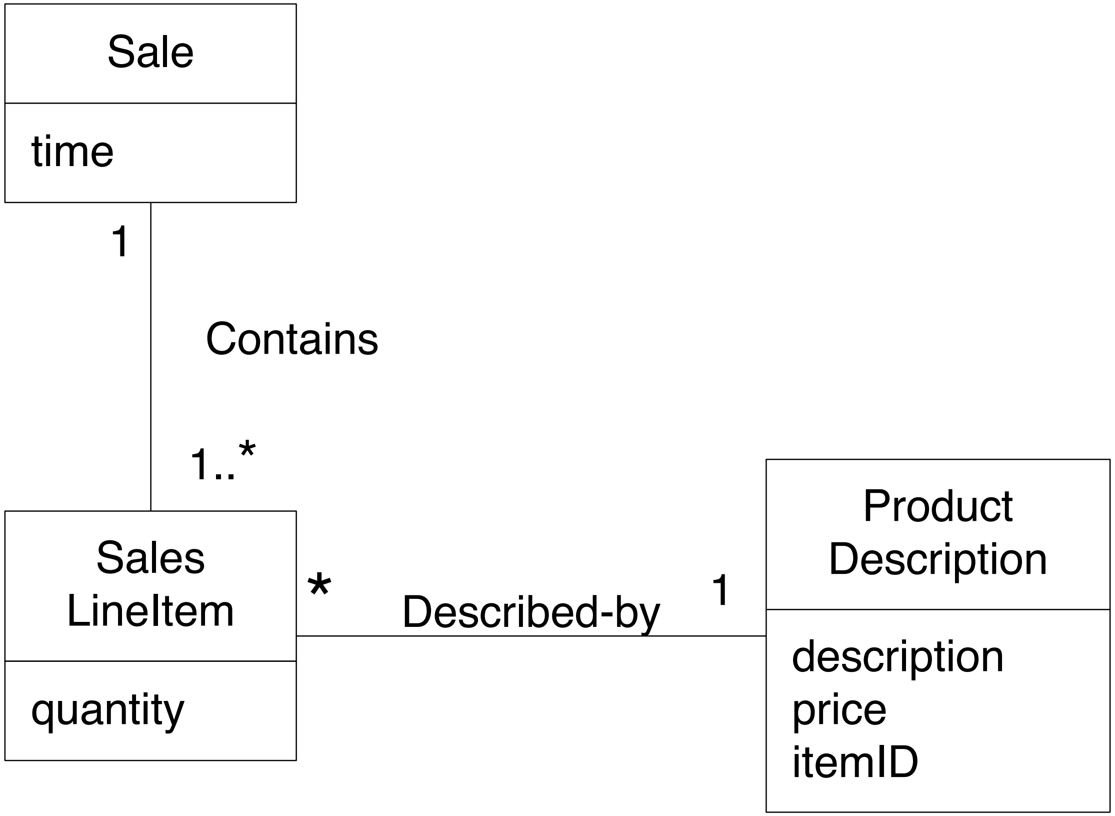
Information Expert

- Question: Where do we look to analyze the classes that have the info needed?
  - The Domain Model? Or,
  - The Design Model (class and interaction diagrams)

Information Expert

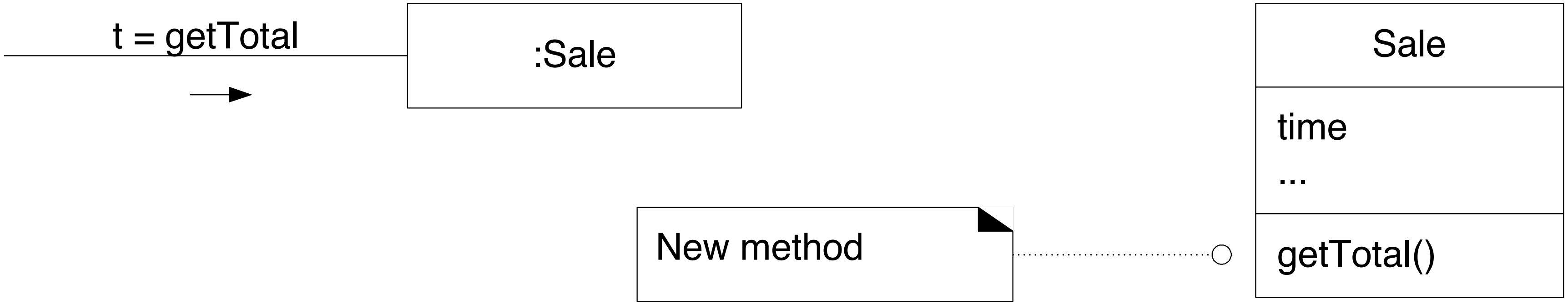
- Question: Where do we look to analyze the classes that have the info needed?
  - The Domain Model? Or,
  - The Design Model (class and interaction diagrams)
- Answer:
  - If there are relevant classes in the Design Model, look there first
  - Otherwise, look in the domain model. Use it to inspire the definition of new Design classes.

Applying the "Information Expert" Pattern



- 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

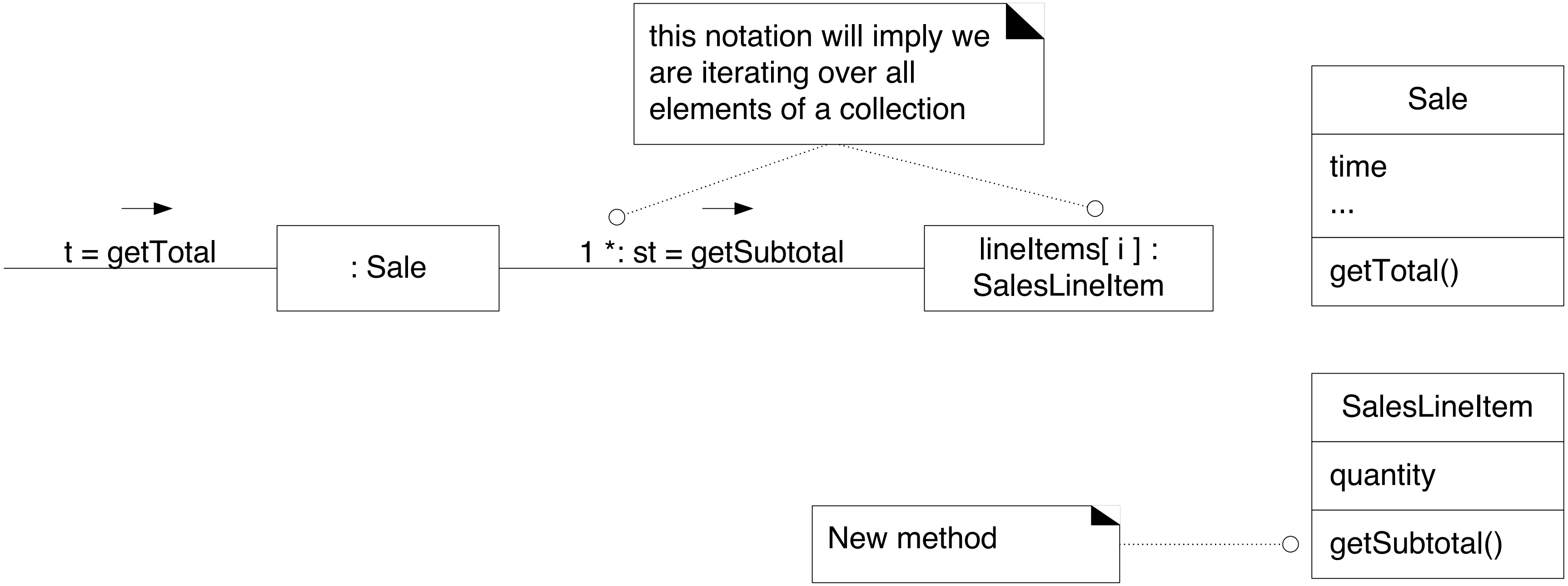
Applying Information Expert



Applying Information Expert (cont'd)

- Next question: What information do we need to know to determine the line item subtotal?
  - Answer: SalesLineItem.quantity and SalesLineItem.price
- Question: Who knows these?
  - Answer: SalesLineItem
  - By "information expert", SalesLineItem should compute the subtotal

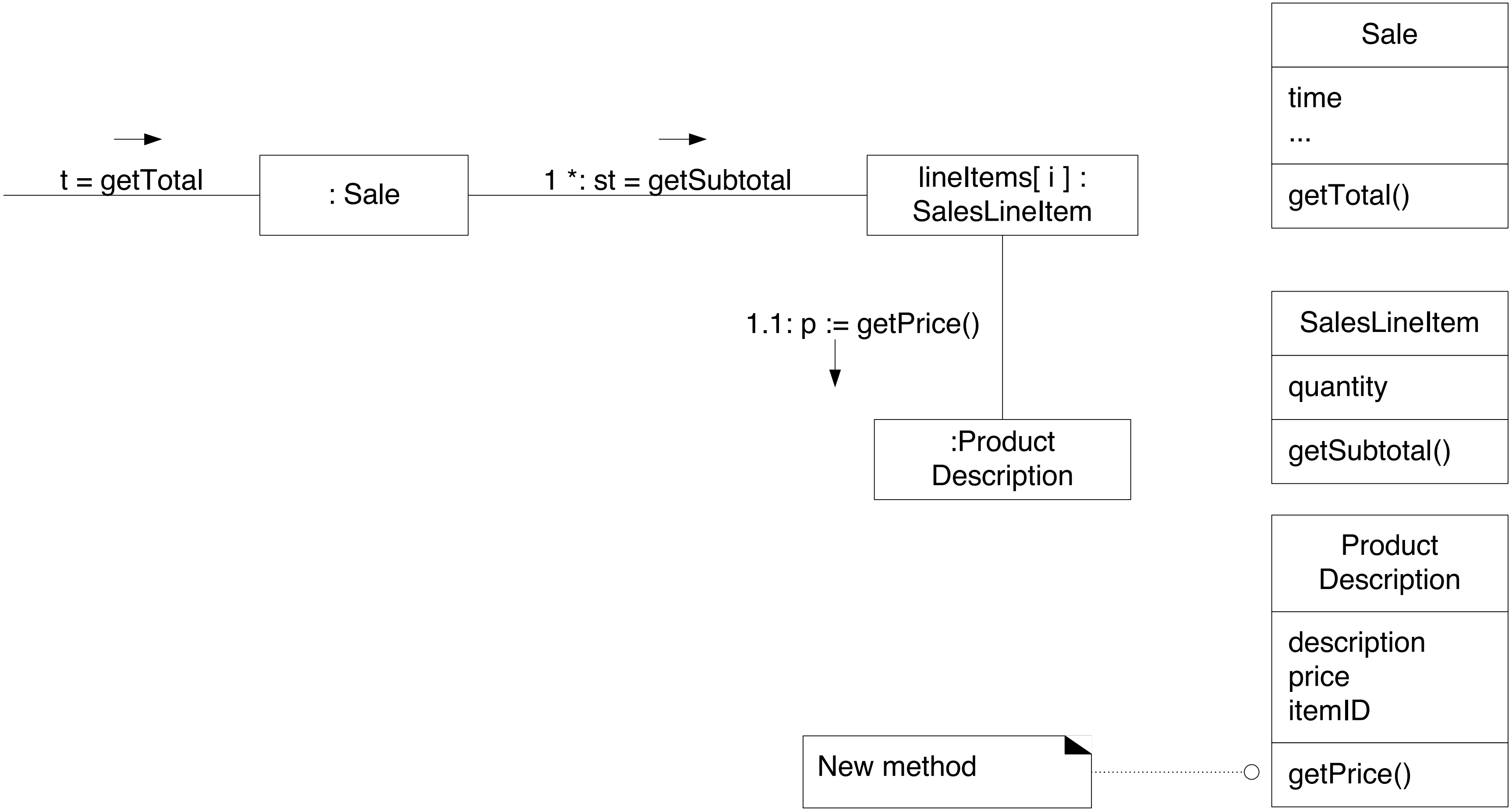
Applying Information Expert (cont’d)


this notation will imply we are iterating over all elements of a collection
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i] : SalesLineItem
Sale
time
...
getTotal()
SalesLineItem
quantity
getSubtotal()
New method

Applying Information Expert (cont’d)

- Next issue: Who should return the price of an item?
  - Who knows the price of an item?
    - ProductDescription
  - By “information expert”, SalesLineItem asks ProductDescription to return the price.

Applying Information Expert (cont'd)



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

Warning: Don’t blindly apply “Information Expert”

- Example:
  - Who should be responsible for saving a Sale in a database?
    - Sale has all the information that needs to be saved
  - But acting on this alone leads to problems in cohesion, coupling, and duplication
  - Sale class must now contain logic related to database handling
    - SQL, JDBC, ...
  - Many other objects that need to be stored will duplicate the “saving” code

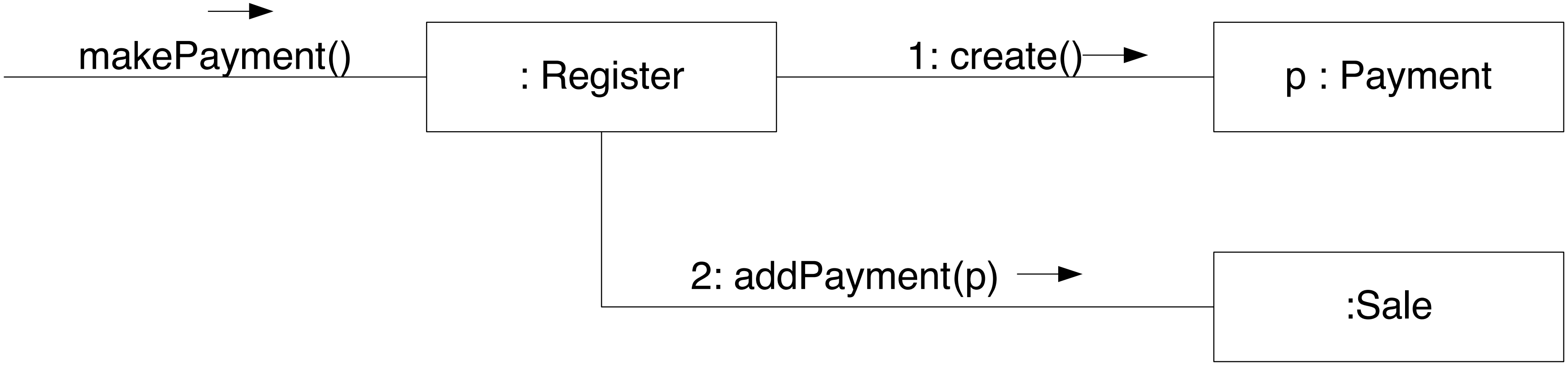
Low Coupling

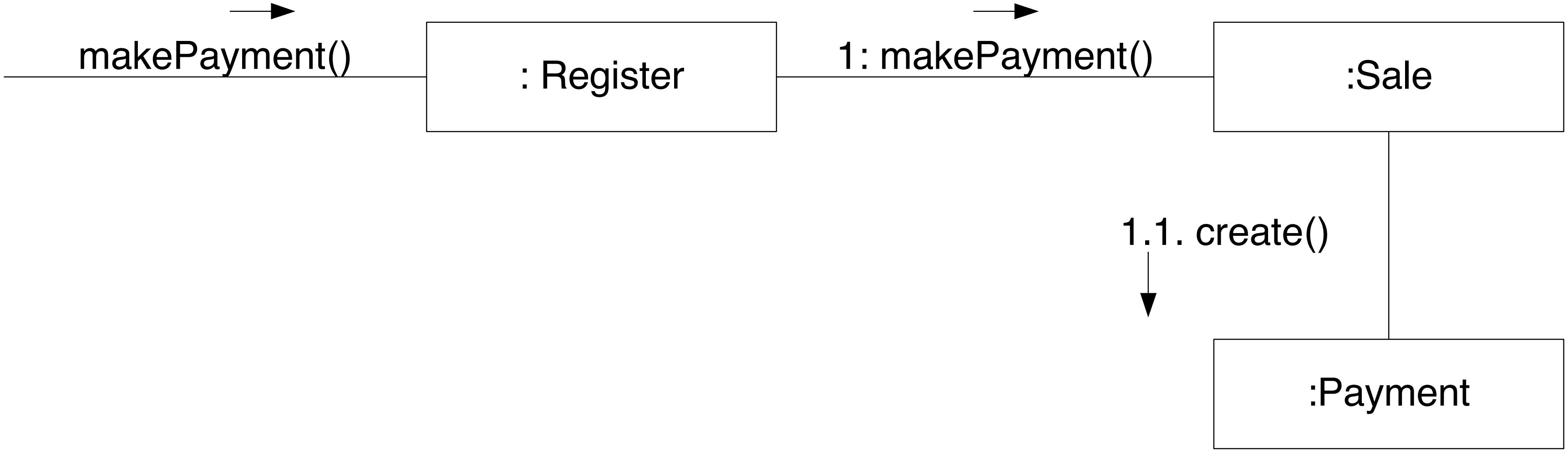
- Problem: How to support low dependency, low change impact, and increased re-use?

- Why is a class with high (or strong) coupling bad?
  - Forced local changes because of changes in related classes
  - Harder to understand in isolation
  - Harder to re-use
    - Because it requires the presence of classes that it depends on
- Solution:
  Assign responsibilities so that coupling remains low.

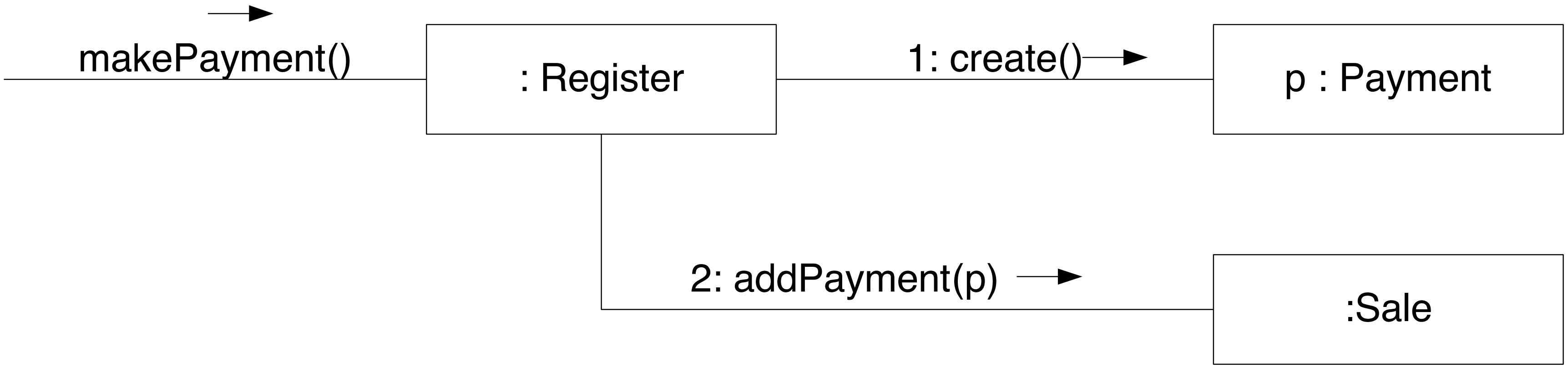
- Important: Use this principle to evaluate alternatives

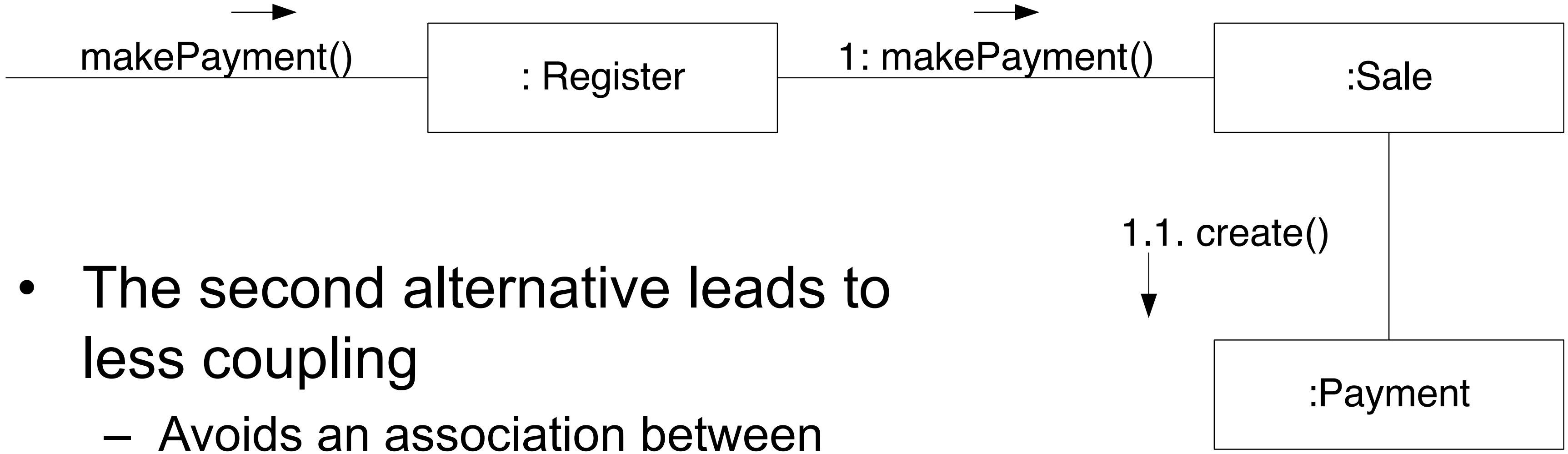
Two alternative responses to “Who creates Payment?”





Two alternative responses to “Who creates Payment?”





- The second alternative leads to less coupling
  - Avoids an association between Register and Payment
  - Sale and Payment already related

Common Forms of Coupling from Type X to Type Y

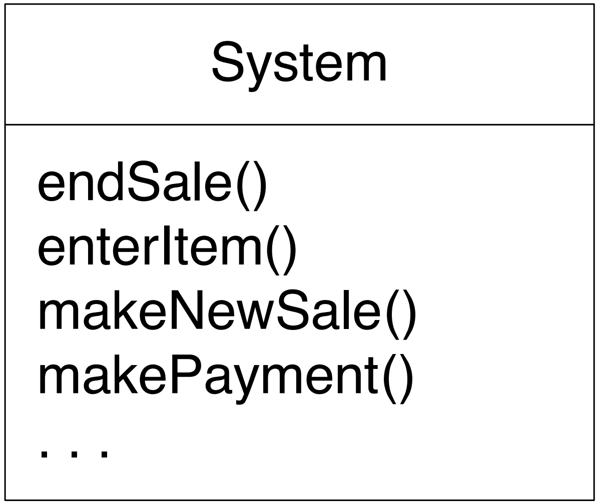
- TypeX has an attribute that refers to a TypeY instance, or TypeY itself
- A TypeX object calls on services of a TypeY object
- TypeX has a method that references
  - an instance of TypeY, or
  - TypeY itself (static method call, etc.)
  - Typically an argument, return value or a local variable of TypeX has type TypeY
- TypeX is a direct or indirect subclass of TypeY
- TypeY is an interface, and TypeX implements that interface

Guidelines

- A subclass is VERY strongly coupled to its superclass
  - Think carefully before using inheritance
- Some moderate degree of coupling between classes is normal and necessary for collaboration
- High coupling to stable or pervasive elements is NOT a problem
  - Examples: Java libraries
- High coupling is a problem only in areas where change is likely
  - Example: Your design, as it evolves

Controller pattern

“System” class: Represents system-level operations

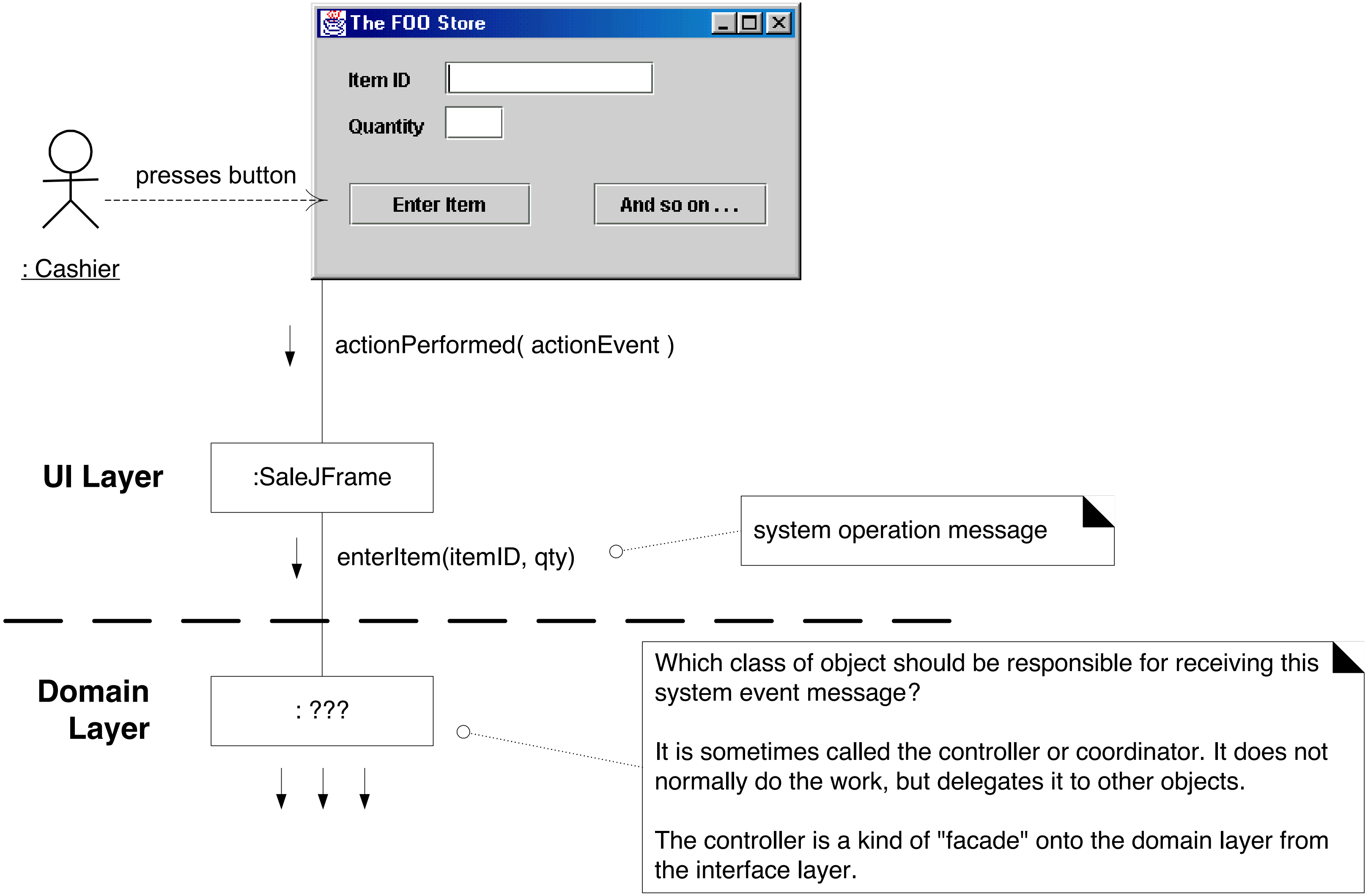


System

endSale()
enterItem()
makeNewSale()
makePayment()
. . .

- No class called system really
- The controller class is assigned responsibility for system-level operations

What object should be the Controller for enterItem?



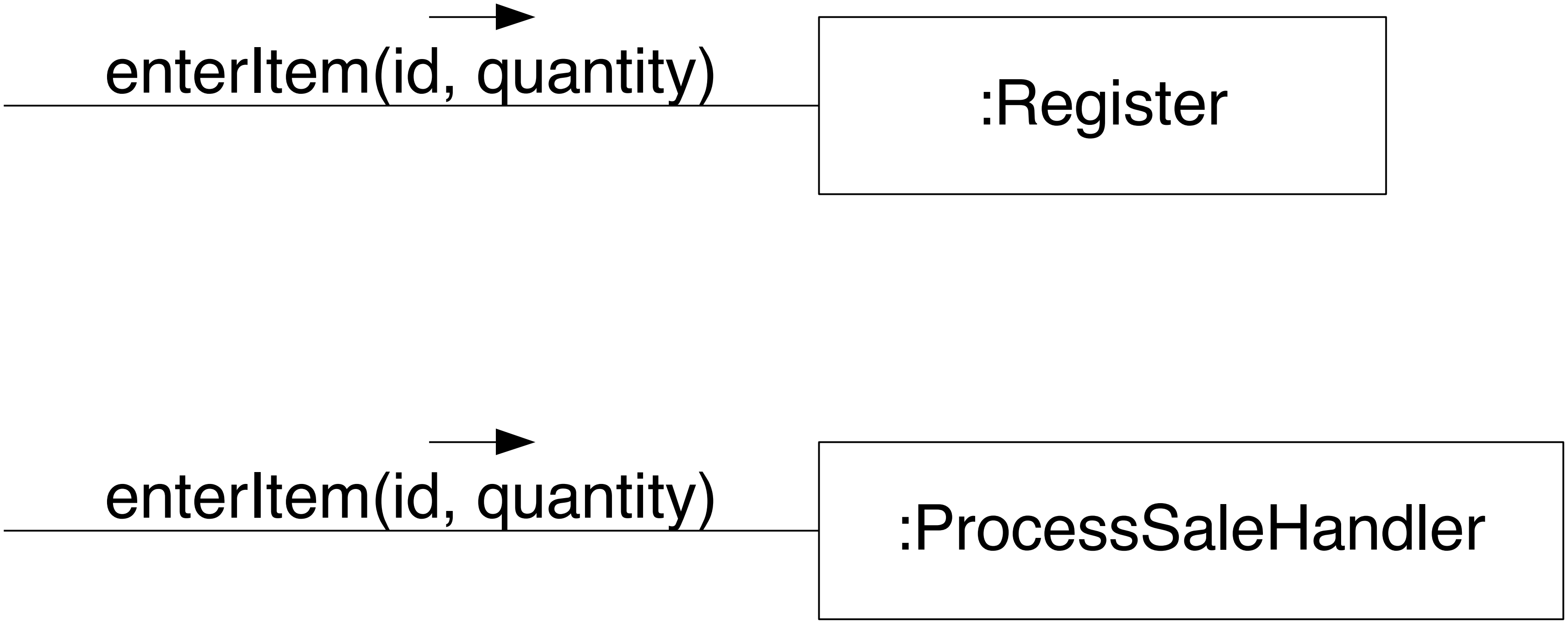
system operation message

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.

Two controller alternatives



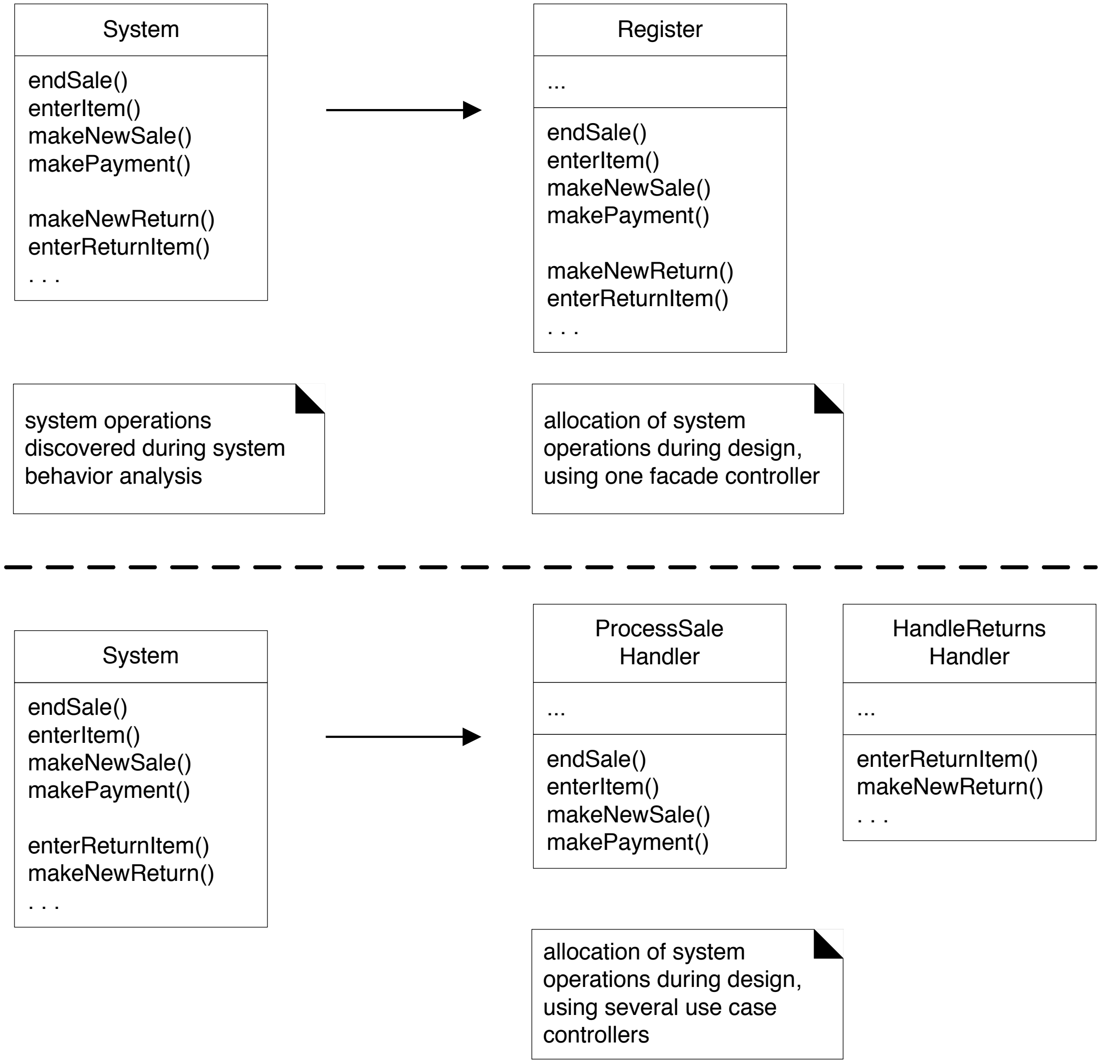
enterItem(id, quantity)

:Register

enterItem(id, quantity)

:ProcessSaleHandler

Allocation of system operations to two kinds of controllers



System

endSale()
enterItem()
makeNewSale()
makePayment()

makeNewReturn()
enterReturnItem()
. . .

Register

...

endSale()
enterItem()
makeNewSale()
makePayment()

makeNewReturn()
enterReturnItem()
. . .

system operations discovered during system behavior analysis

allocation of system operations during design, using one facade controller

System

endSale()
enterItem()
makeNewSale()
makePayment()

enterReturnItem()
makeNewReturn()
. . .

ProcessSale Handler

...

endSale()
enterItem()
makeNewSale()
makePayment()

HandleReturns Handler

...

enterReturnItem()
makeNewReturn()
. . .

allocation of system operations during design, using several use case controllers

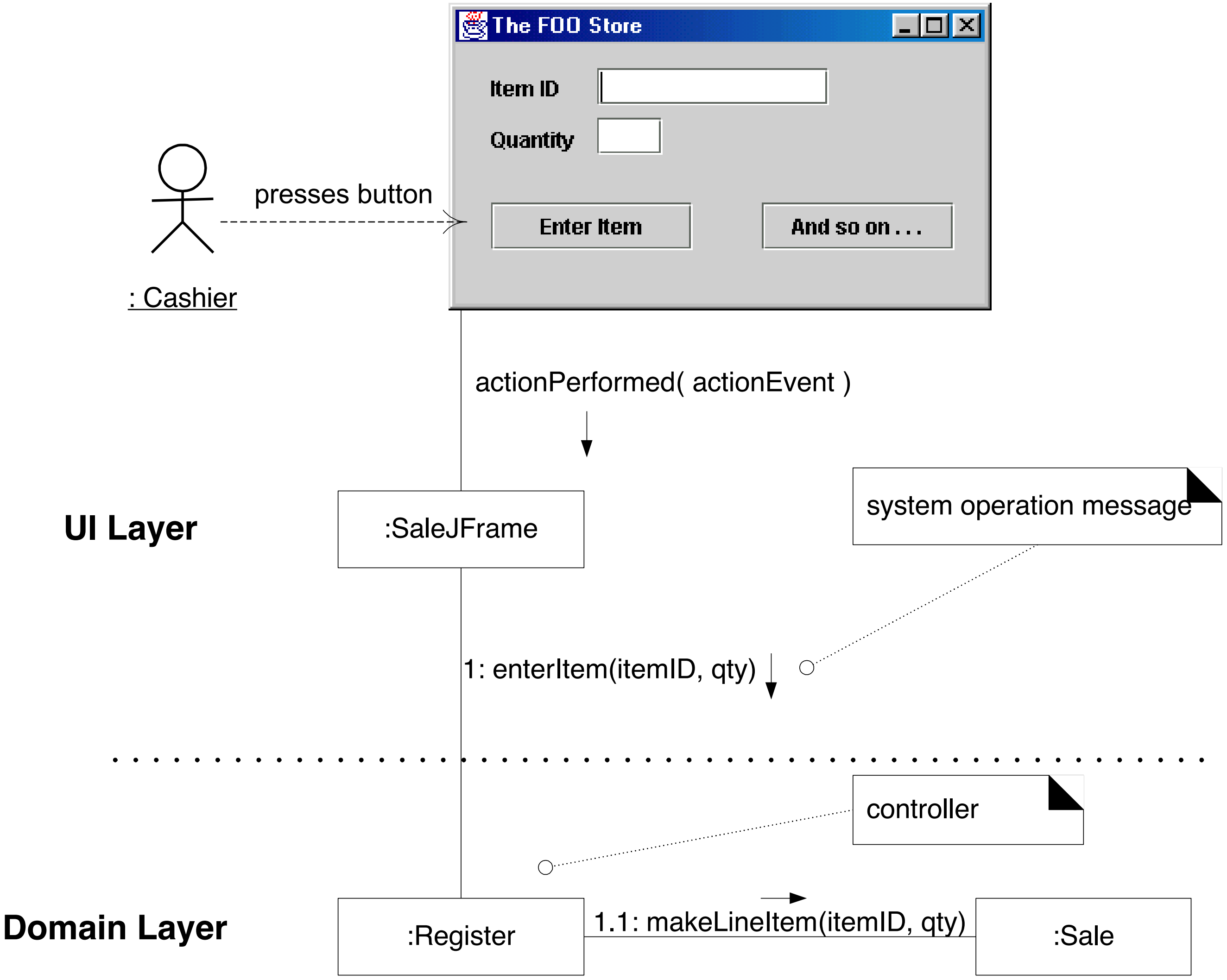
The controller pattern

- A delegation pattern
- Use same controller class for all system events of one use case
  - Controller maintains information about the state of the use case
  - Helps identify out-of-sequence operations
    - Example: makePayment before endSale
- Common defect in design of controllers: Overassignment of responsibility
  - Controller may suffer from low cohesion
- Benefits of controller pattern:
  - Increased potential for reuse and pluggable interfaces
  - Opportunity to reason about the state of the use case

Bloated controllers

- Signs of a bloated controller
  - There is a single controller receiving many system events
  - Controller itself performs the tasks instead of delegating them
  - Controller has many attributes, maintains a lot of info about the system or domain
    - Better to distribute these to other objects
- Cures
  - Add more controllers
  - Design the controller to delegate the work

Domain Layer Controller



: Cashier

presses button

actionPerformed(actionEvent)

UI Layer

:SaleJFrame

system operation message

1: enterItem(itemID, qty)

controller

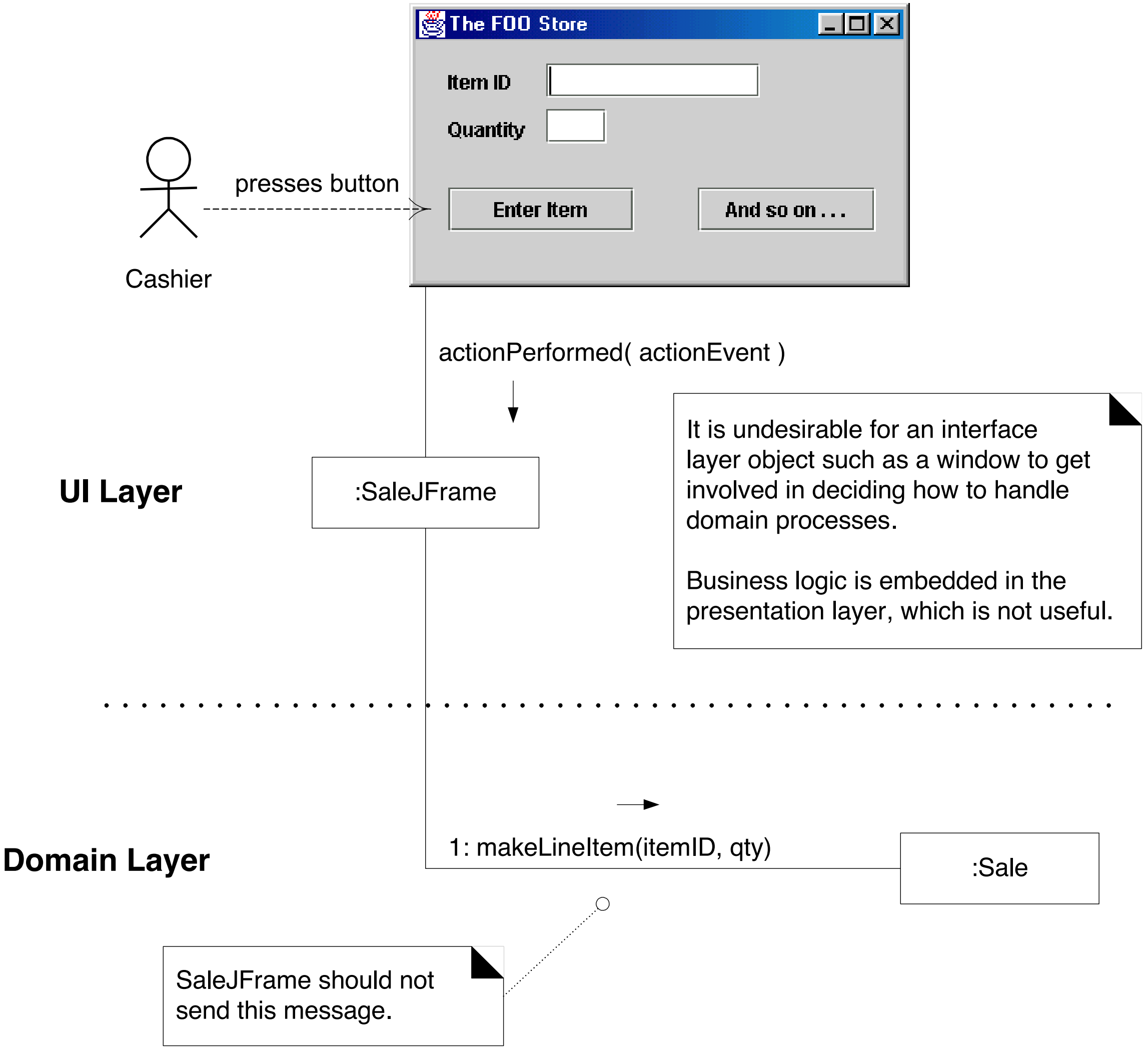
Domain Layer

:Register

1.1: makeLineItem(itemID, qty)

:Sale

Undesirable: UI Layer object is controller



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.

SaleJFrame should not send this message.

Swing Example

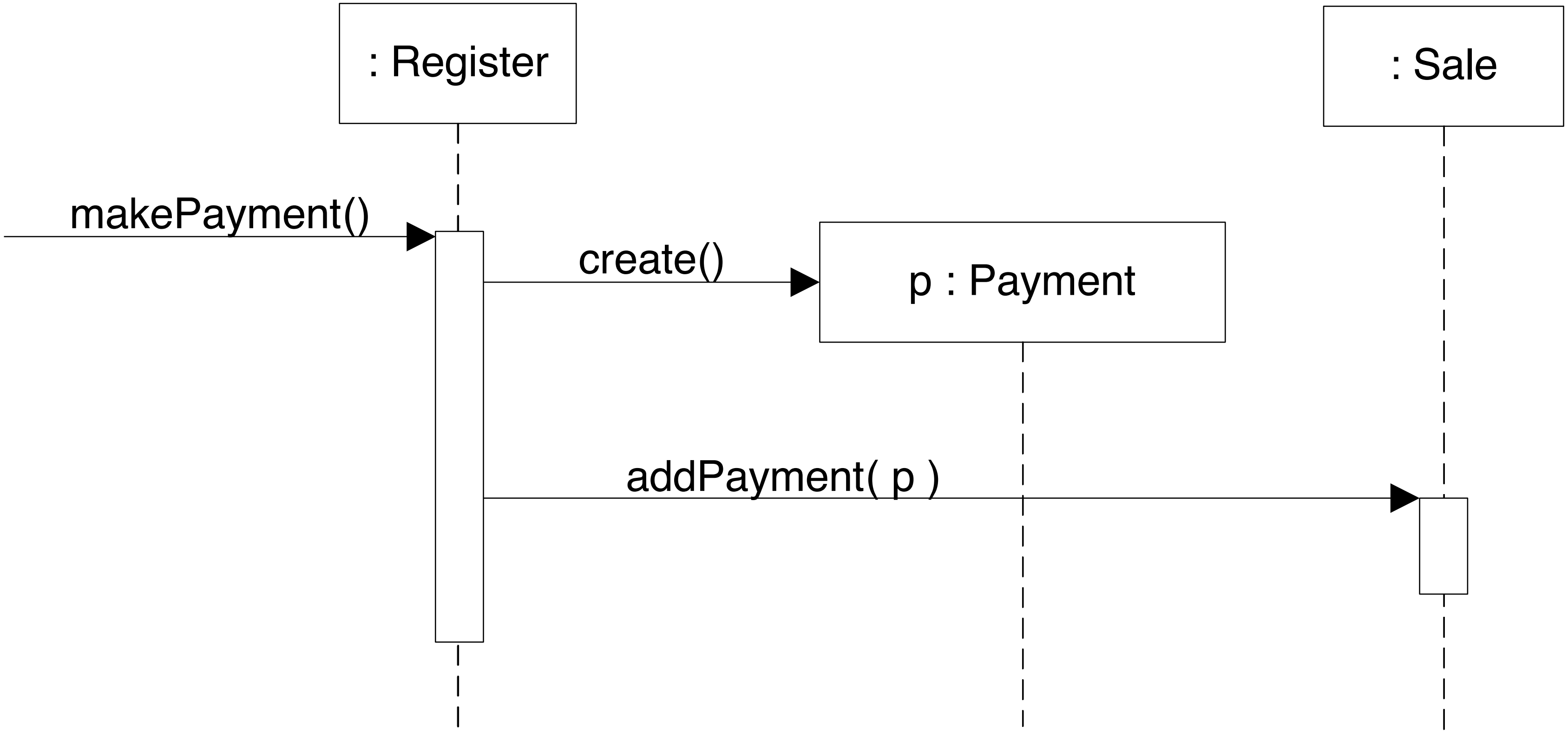
- Code where controller is called

```
….
public void actionPerformed(ActionEvent e) {

     // read itemID and quantity from Swing
    // GUI components
     ...
     register.enterItem(itemID, quantity);

}
```

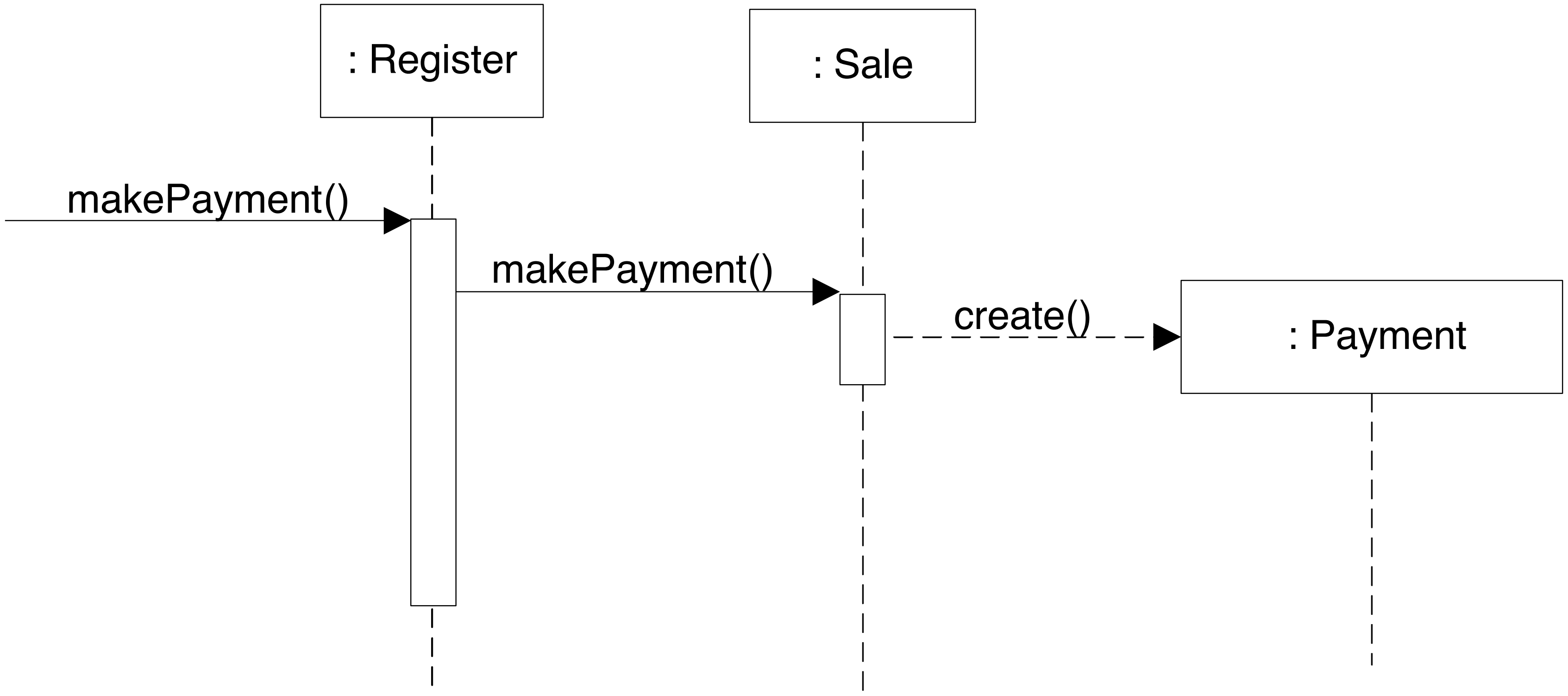
High Cohesion

- Problems with a class with low cohesion:
  - Hard to comprehend
  - Hard to reuse
  - Hard to maintain
  - Constantly affected by change

Danger: Register has potential to do too much work



Better design: Register delegates, has high cohesion



Modular Design

- Modularity: The property of a system that has been decomposed into a set of cohesive and loosely coupled modules
- Exceptions to high cohesion and modularity
  - Example: To simplify maintenance by one SQL expert, group all SQL-related functionality of all classes into one separate class
  - Distributed server objects: Because of cost of communication, might make sense to have few, large server objects
    - Example: Instead of three fine-grained operations, setName, setSalary, setHireDate, have one coarse-grained operation, setData