SWEN30006
Software Modelling and Design

GRASP: Designing Objects with Responsibilities

Larman Chapter 17

Understanding responsibilities is key to good object-oriented design.

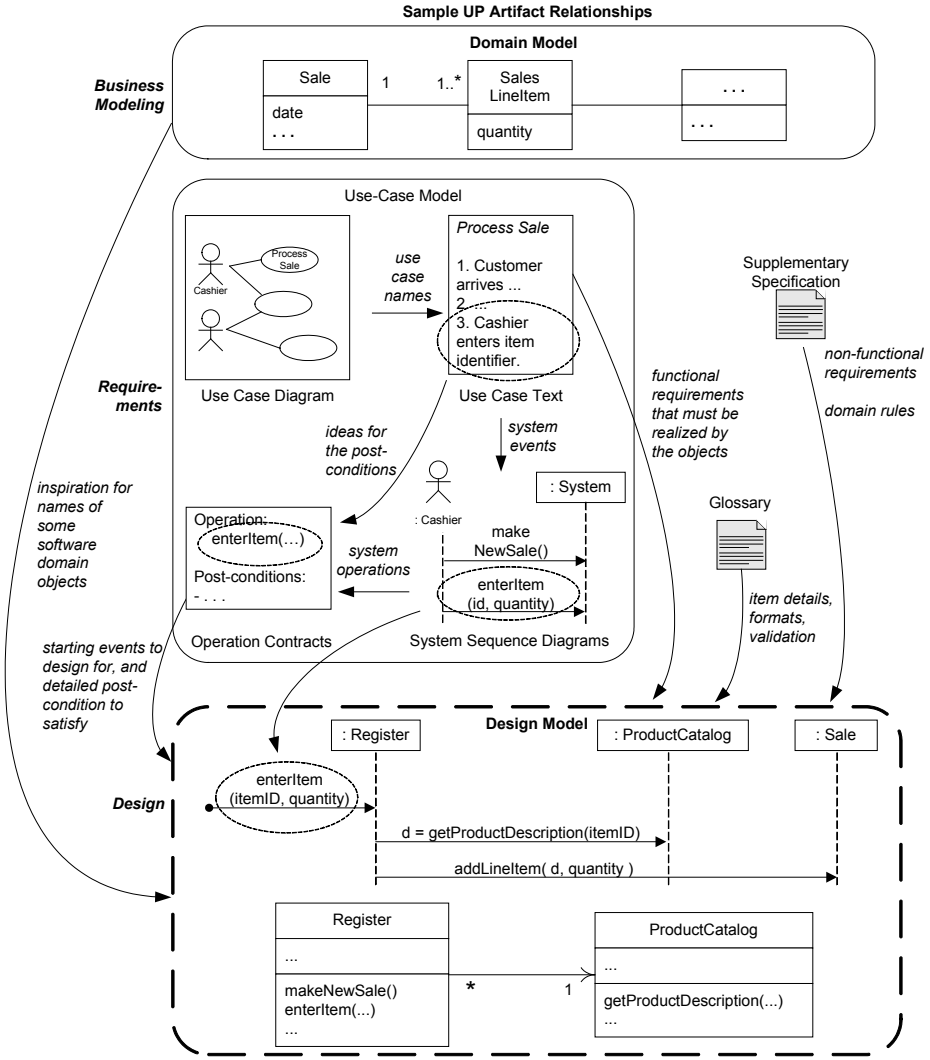
—Martin Fowler

Objectives

On completion of this topic you should be able to:

- Understand the concepts of responsibility and responsibility-driven design
- Apply five of the GRASP principles or patterns for Object-Oriented Design.

Artifact Relationships: Influence on OO Design



Influences on Design:

- Use Cases
- Domain Model
- System Sequence Diagrams
- Glossary
- Non-functional requirements
- …

Responsibility

Definition: *A contract or obligation of a classifier*

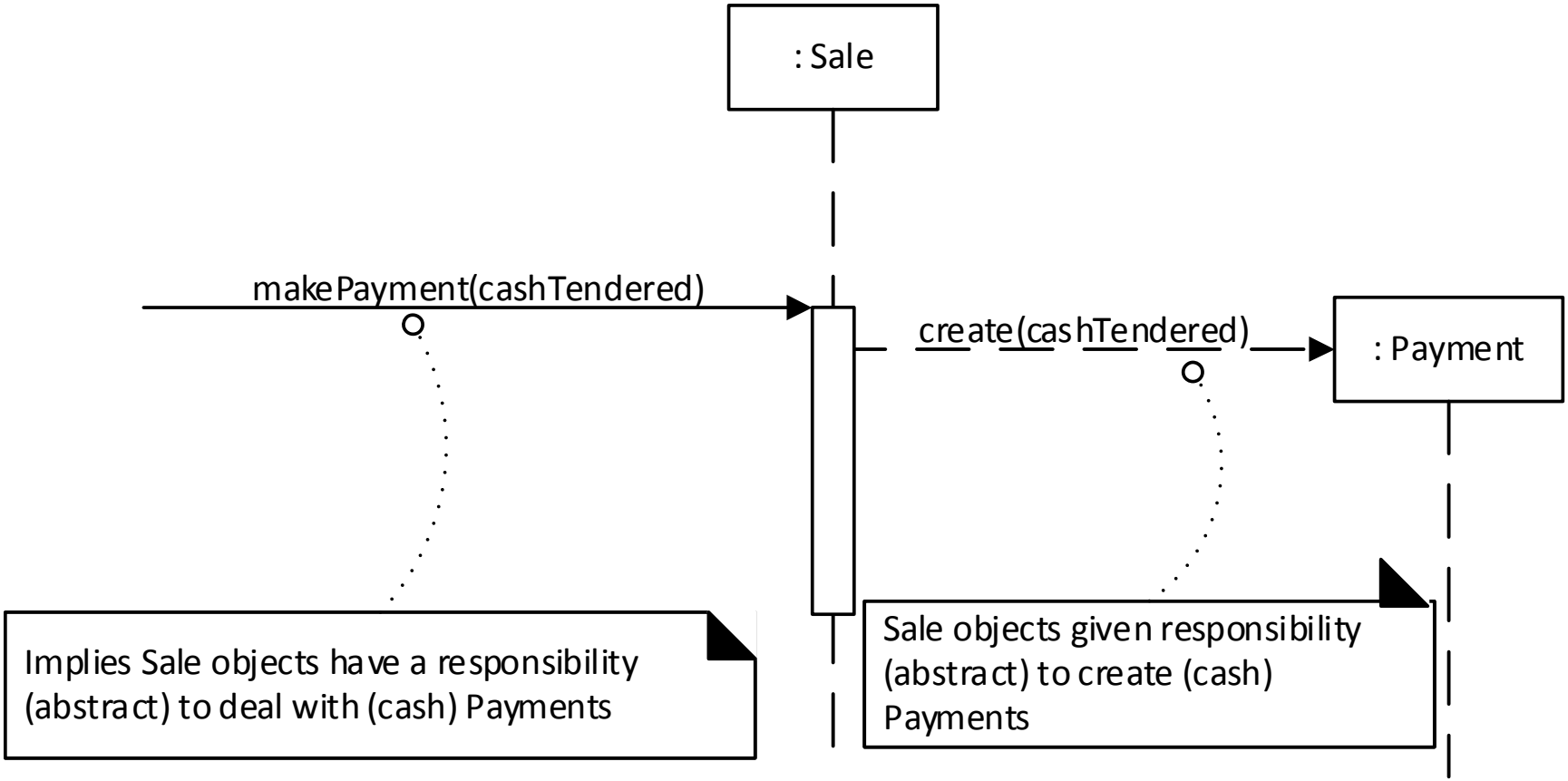
Doing responsibilities include:

- directly—e.g. create object, perform calculation
- initiate action in other objects
- control and coordinate activities in other objects

Knowing responsibilities include knowledge of:

- Private encapsulated data
- Related objects
- Derivable or calculable items

Responsibilities and Methods are Related



Responsibility-Driven Design

RDD sees an OO Design as a

- community of collaborating responsible objects.

RDD involves

- assigning responsibilities to classes
- which should be based on proven principles.

E.g. Domain *Sale* has a *time* attribute:

low rep gap → Design *Sale* should know its *time.*

GRASP: Learning Aid for RDD

General **R**esponsibility **A**ssignment **S**oftware **P**atterns (or **P**rinciples):

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Creator

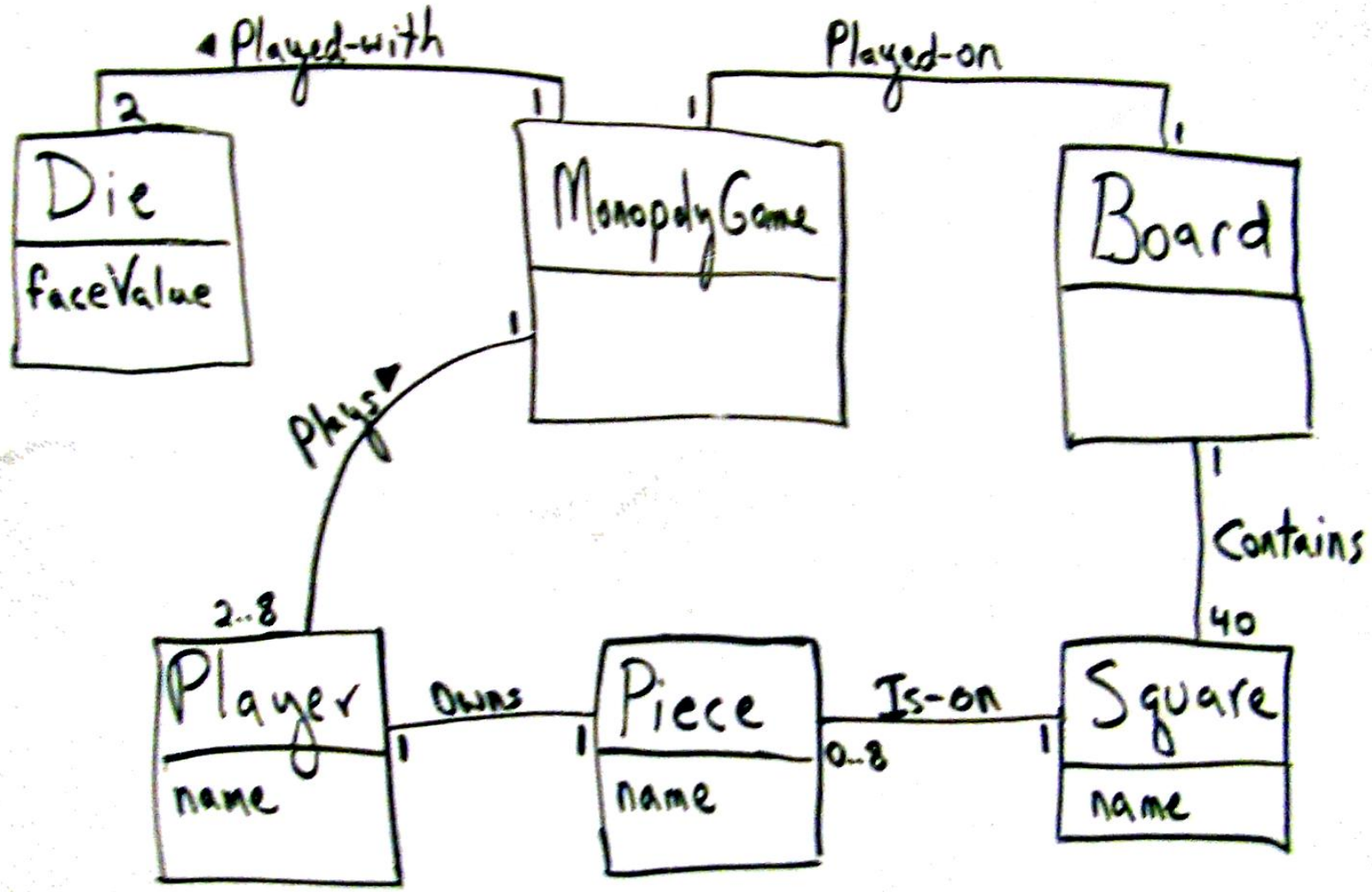
Problem

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

Creation of objects:

- One of the most common OO activities
- Useful to have a general principle to deal with this
- Want outcomes with low coupling, increased clarity, encapsulation, and reusability

Monopoly Iteration-1 Domain Model



Creator

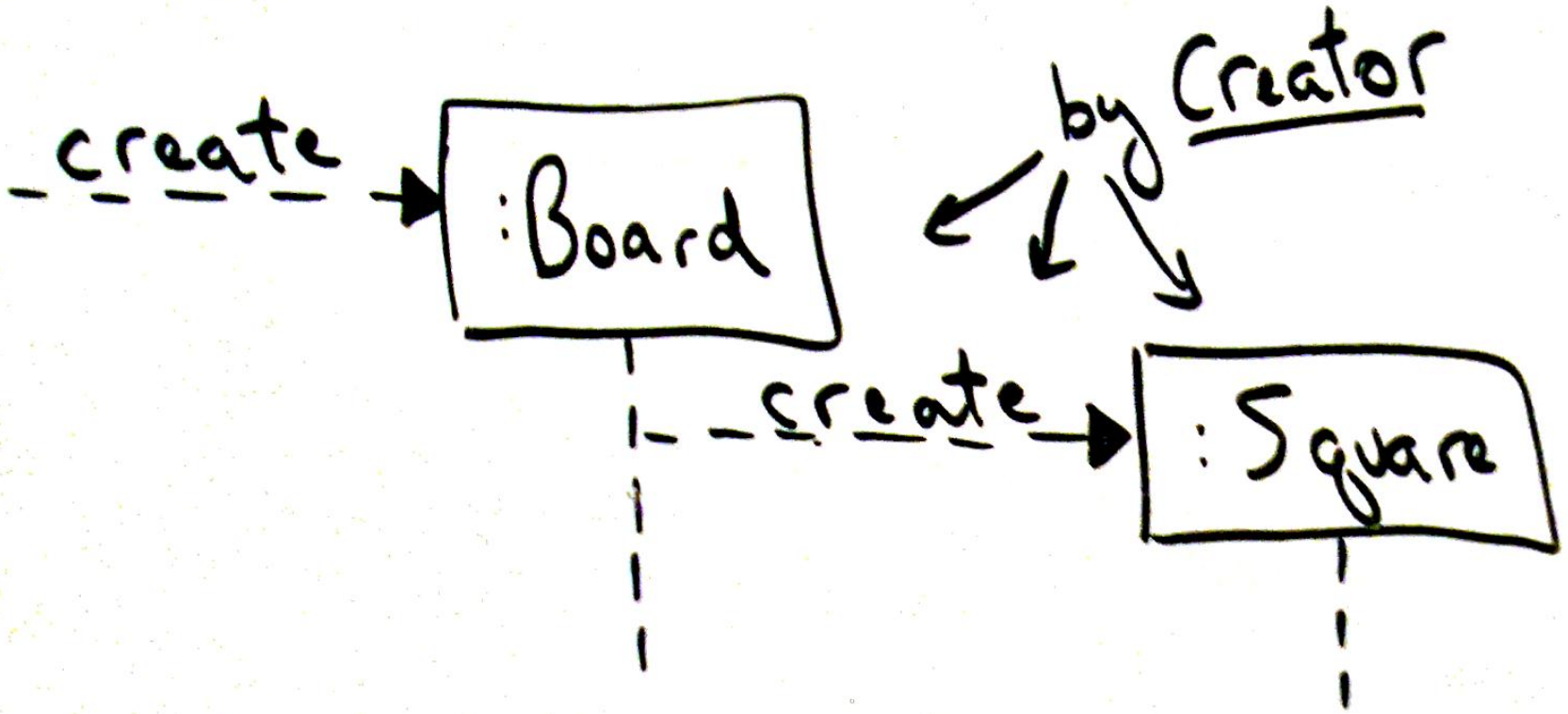
Solution

Assign class B responsibility to create instances of class A if one of these is true (the more the better):

- 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 >1 applies, choose B which *aggregates*/*contains* A.

Applying Creator Pattern: Dynamic Model



Applying Creator Pattern: Static Model

Creator

Contraindications

- Creation of significant complexity, e.g.
  - recycling instances for performance.
  - instances from A1, A2, …, based on property

In these cases, delegate to helper class in the form of a *Concrete Factory* or *Abstract Factory*.

Information Expert (or Expert)

Problem

What is a general principle of assigning responsibilities to objects?

Design model:

- May have 100s or 1000s of software classes
- May have 100s or 1000s of responsibilities
- Useful to have a general principle to guide choice of assignment
- Want outcomes that are easier to understand, maintain, extend, and reuse

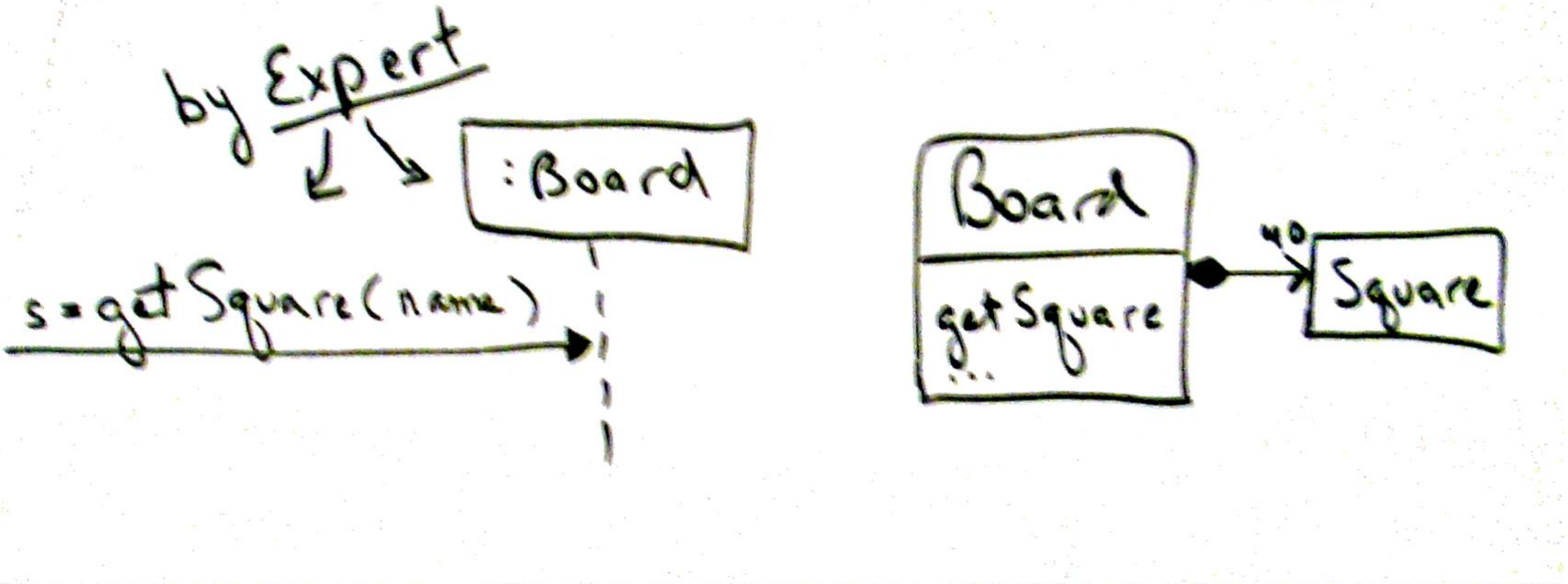
Information Expert (or Expert)

Solution

Assign responsibility to the information expert—the class that has the *information* necessary to fulfil the responsibility.

- Start assigning responsibilities by clearly stating the responsibility.
- Look in Domain Model or Design Model?
  1. If relevant classes in Design, look there first.
  2. Otherwise look in Domain, and use or expand to inspire creation of design classes.

Applying Expert



Information Expert (or Expert)

Contraindications

- Suggested solution can result in poor coupling and cohesion

Example:

- Need to store objects in a database (DB)
- Each class is an expert on what needs to be stored
- Add DB handling to each class?
- No: DB elements and other class elements not cohesive; couples all classes with DB services
- Separate application logic and DB logic (DB Expert)

Low Coupling

Problem

How to support low dependency, low change impact, and increased reuse?

Coupling:

- Measure of how strongly one element is connected to, has knowledge of or relies on others.

Problems for a class with high coupling:

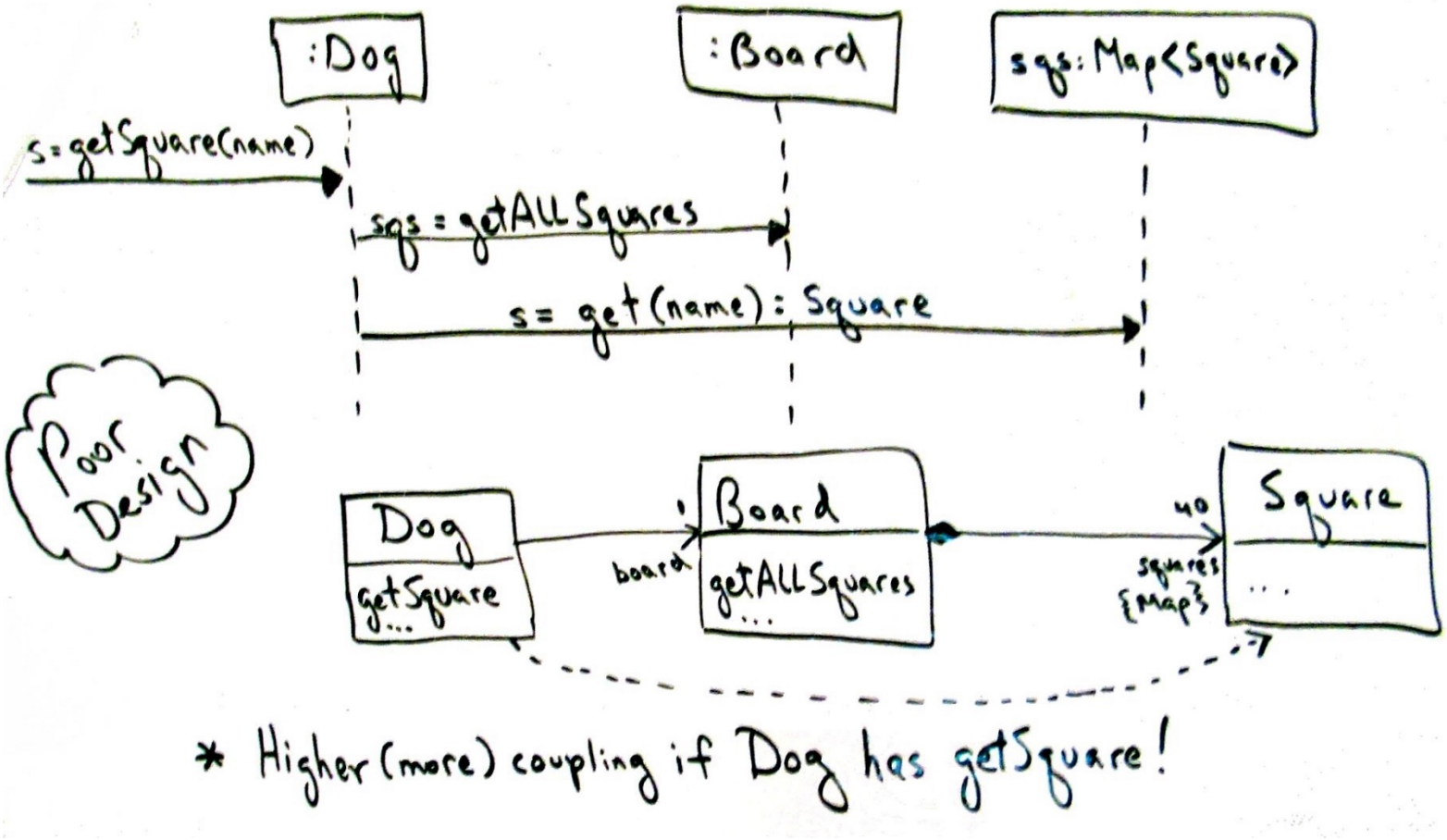
- Forced changes: result of changes in related classes
- Harder to understand in isolation
- Harder to reuse, as harder to isolate

Low Coupling

Solution

Assign responsibility so that coupling remains low.
Use this principle to evaluate alternatives.

Evaluating the Effect of Coupling



Low Coupling

Contraindications

- High coupling to stable elements is rarely a problem

Example:

- Standard Java Libraries can be used extensively without concern as their use does not generate concerns relating to stability, understanding or reuse.

Note: Coupling is essential. We are choosing where it can be kept low without compromising other design aspects.

Controller

Problem

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

System operations:

- major input events
- appear on System Sequence Diagrams (SSDs)

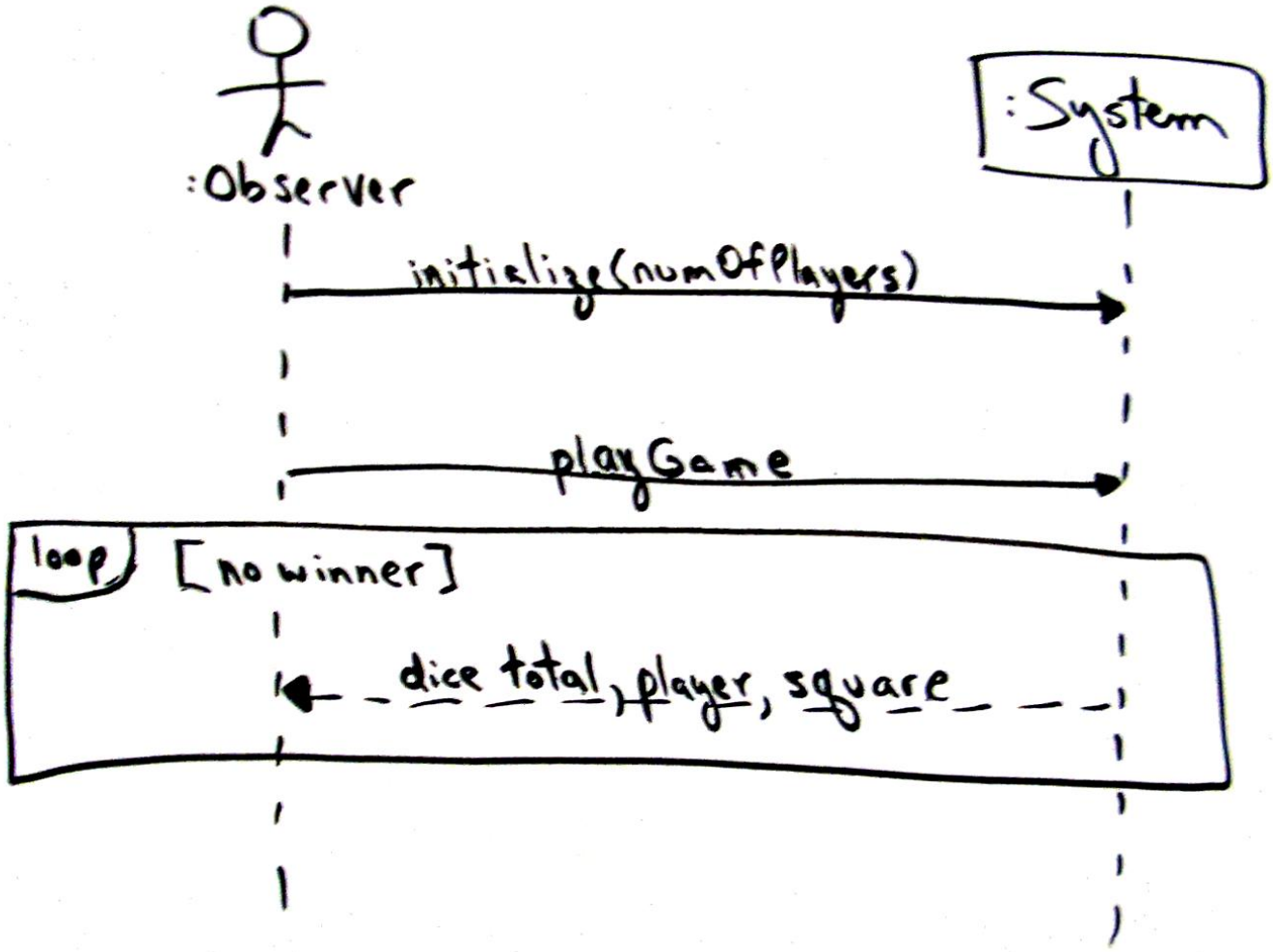
Controller

Solution

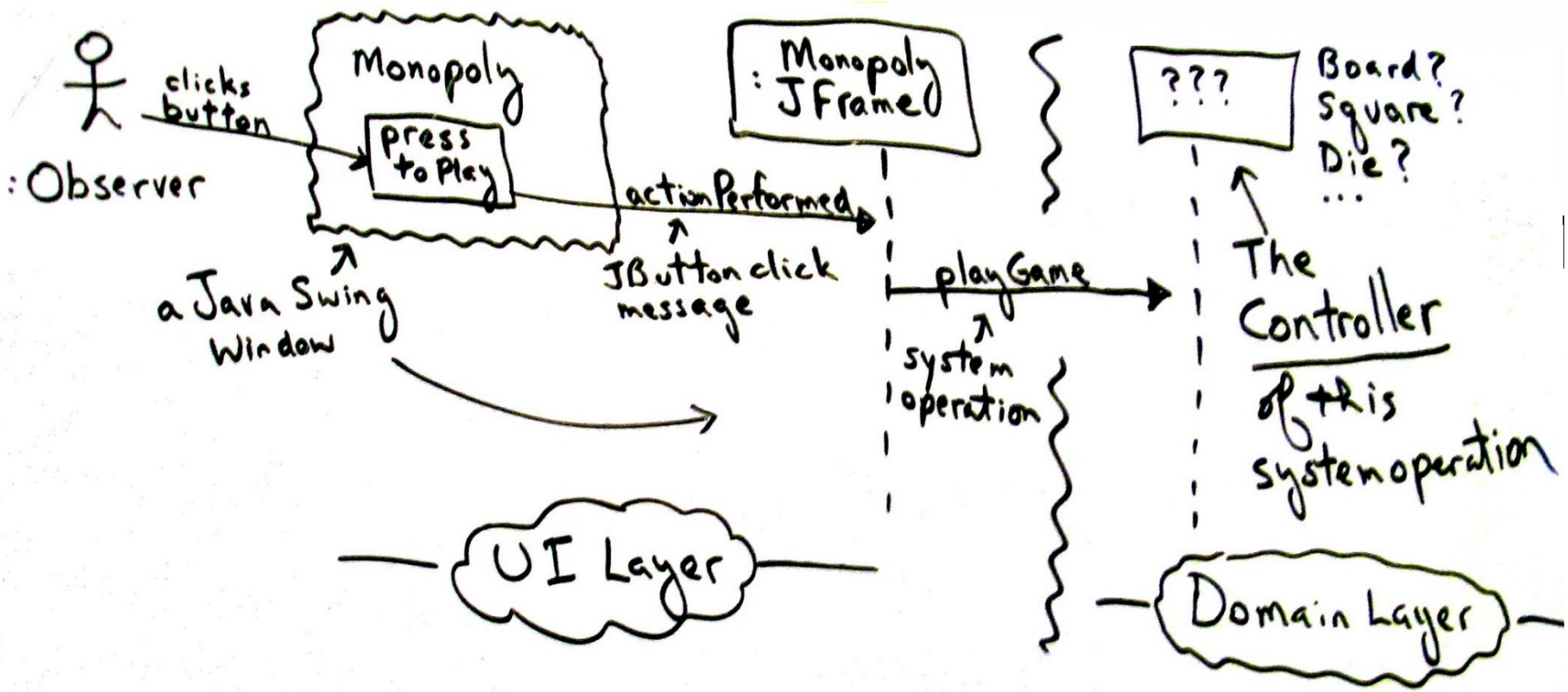
Assign responsibility to a class representing one of:

- the overall “system”, a “root” object, a device the software is running within, or a major subsystem
  - *façade controller*
- a use case scenario that deals with the event, e.g. <UseCaseName>(Handler|Coordinator|Session)
  - *use case* or *session controller*

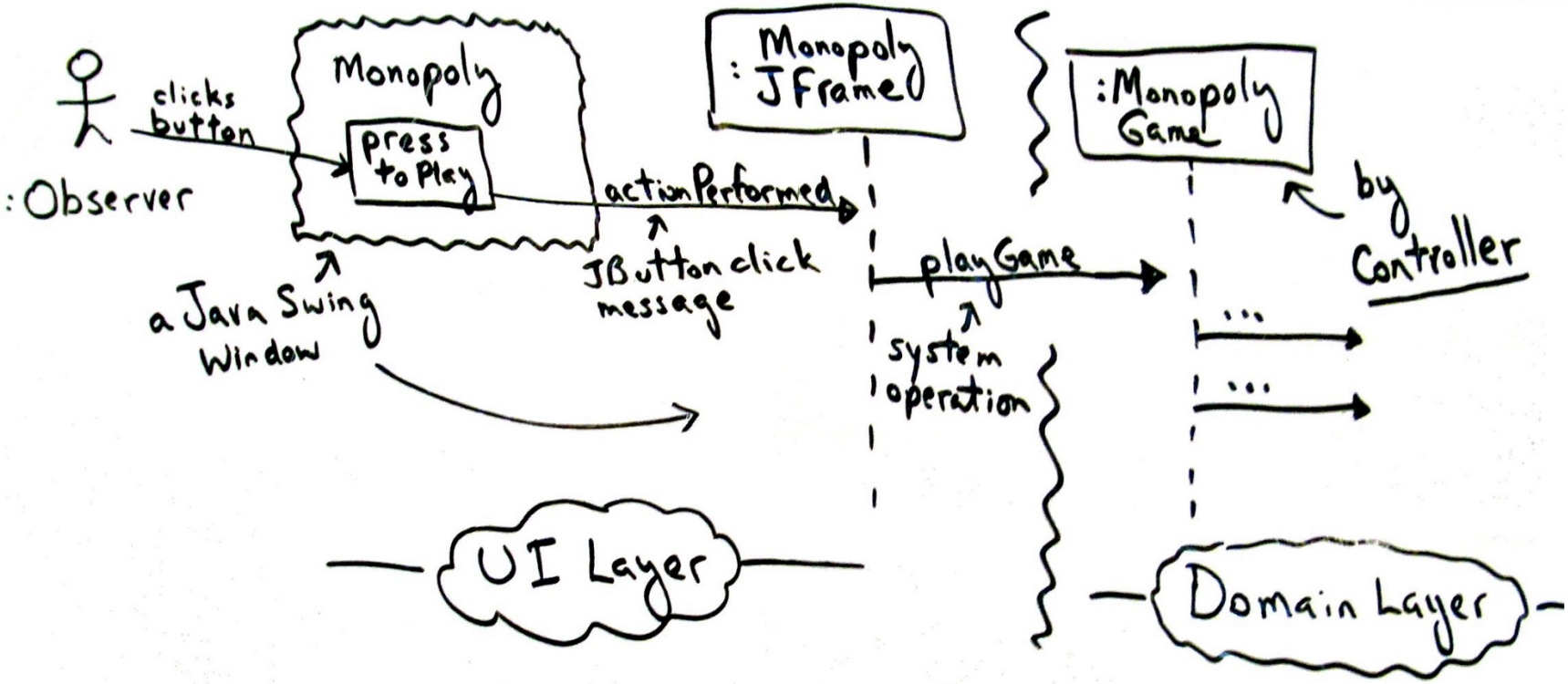
SSD for Monopoly



Who is the Controller for playGame?



Applying the Controller Pattern



Controller

Contraindications

- **Bloated controller**: too many responsibilities, low cohesion, unfocussed
  - Single controller class for all system events
    - Soln: add more controllers (façade → use case)
  - Controller too complicated, contains too much or duplicated information: violates Information Expert and High Cohesion
    - Soln: delegate!

High Cohesion

Problem

How to keep objects focussed, understandable, and manageable, and as a side effect, support Low Coupling?

(functional) cohesion:

- A measure of how strongly (functionally) related and focussed the responsibilities of an element are

High Cohesion

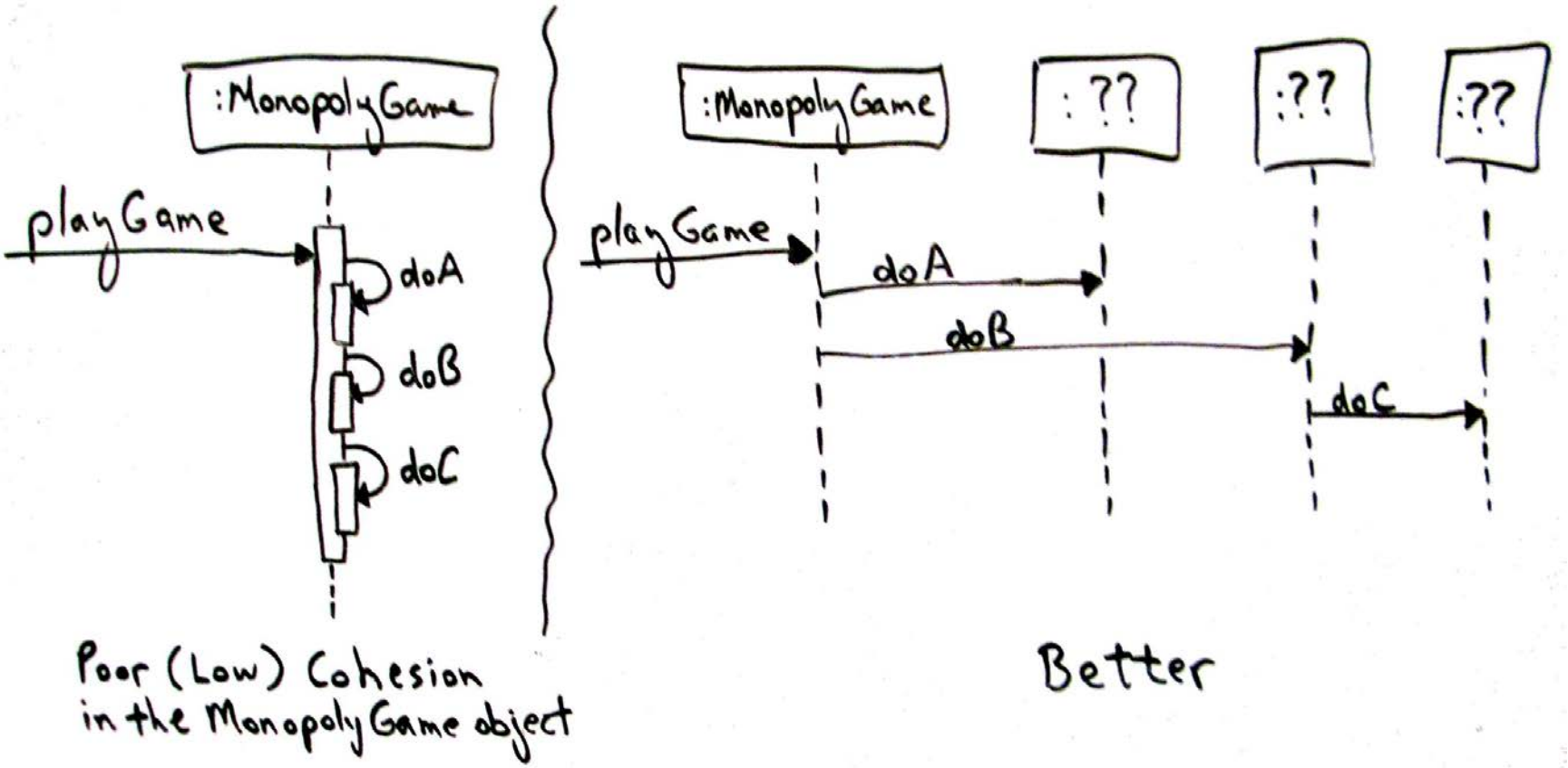
Solution

Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

Class with low cohesion:

- Hard to comprehend
- Hard to reuse
- Hard to maintain
- Delicate; constantly affected by change

Contrasting Levels of Cohesion



High Cohesion

Contraindications

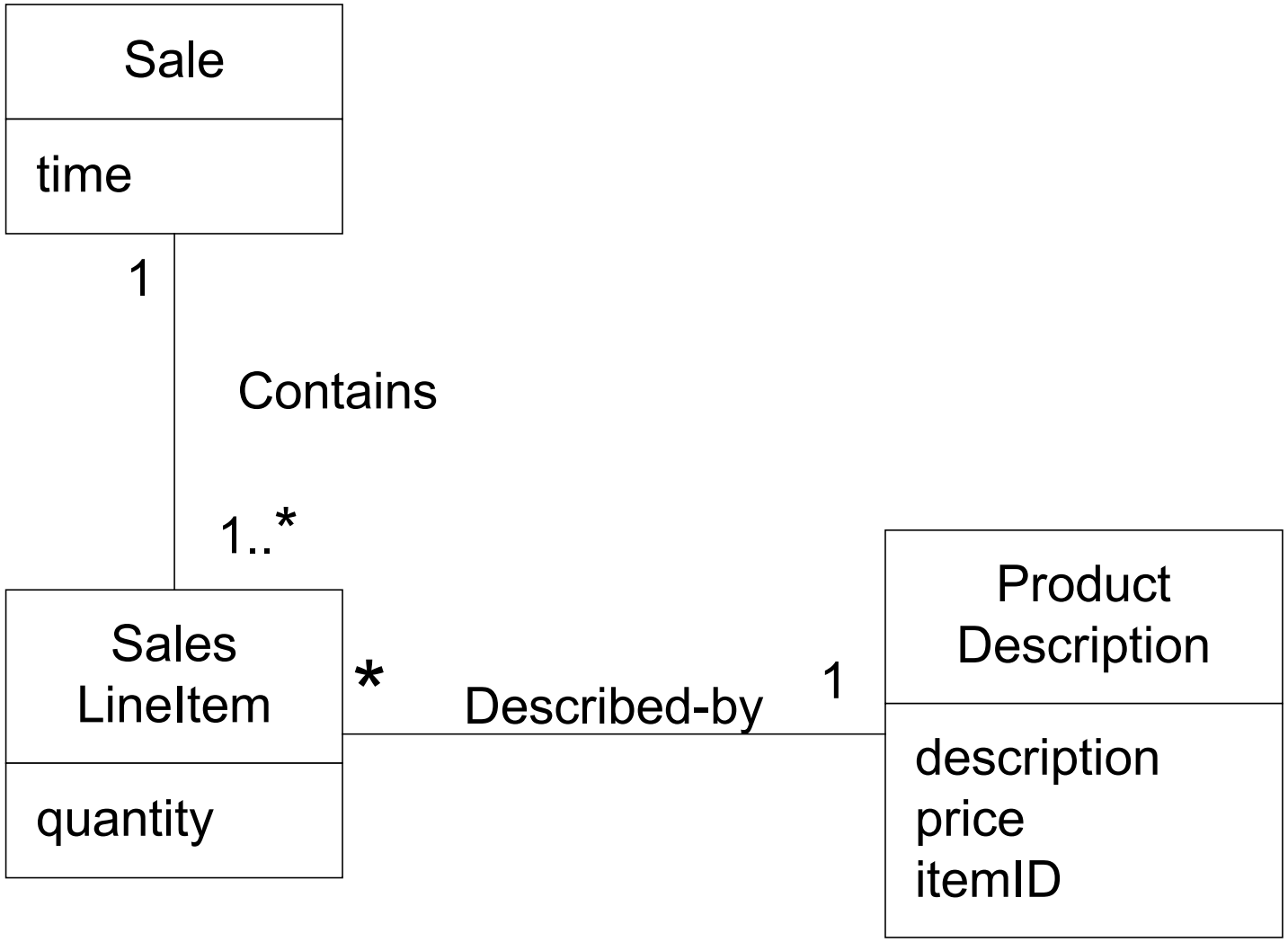
- Lower cohesion is sometimes justified to meet non-functional requirements

Example:

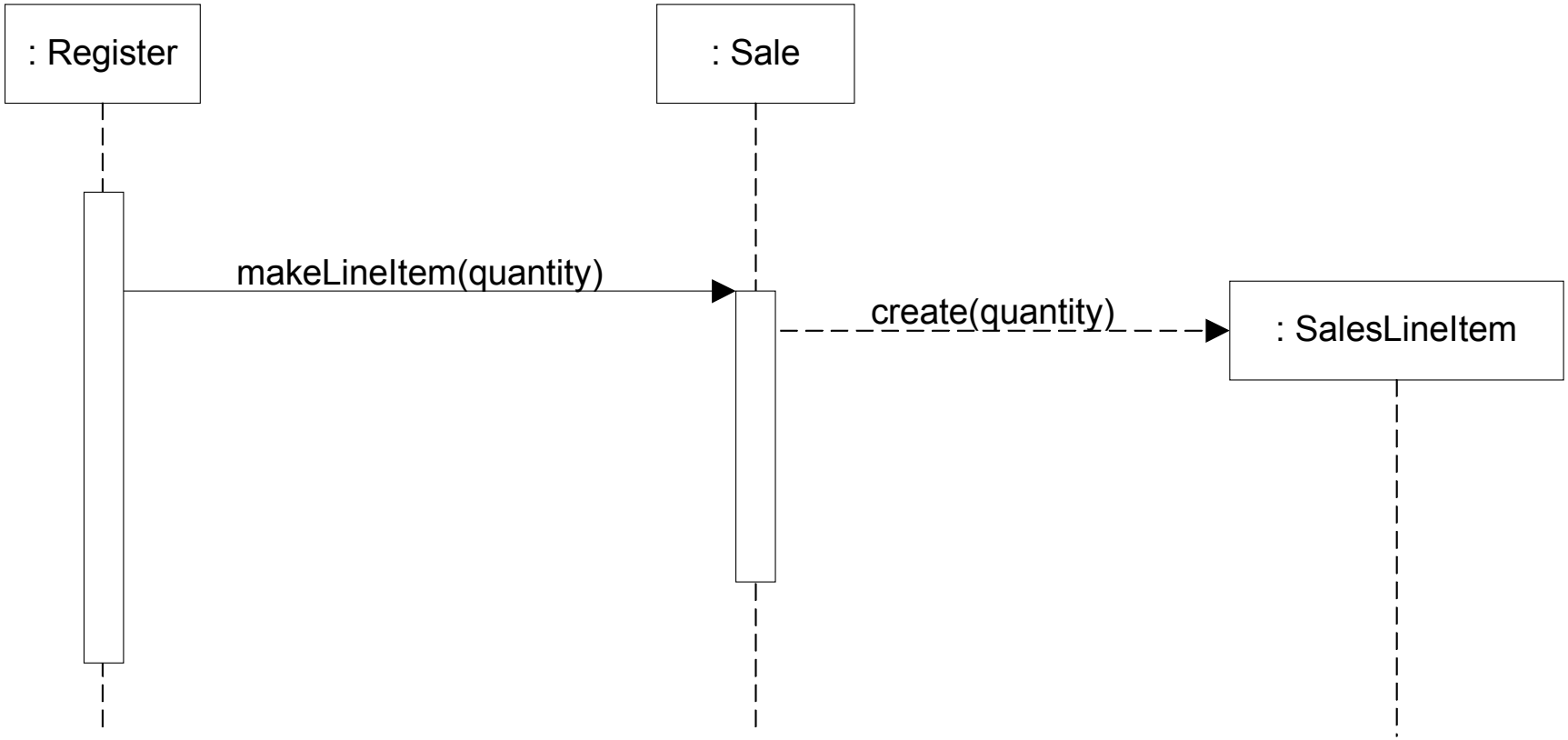
- To meet performance requirements, larger, less cohesive classes are used to reduce processing overheads (e.g. transmission, storage, retrieval)

Note: Coupling and cohesion are fundamental design properties which are interdependent: both must be considered together.

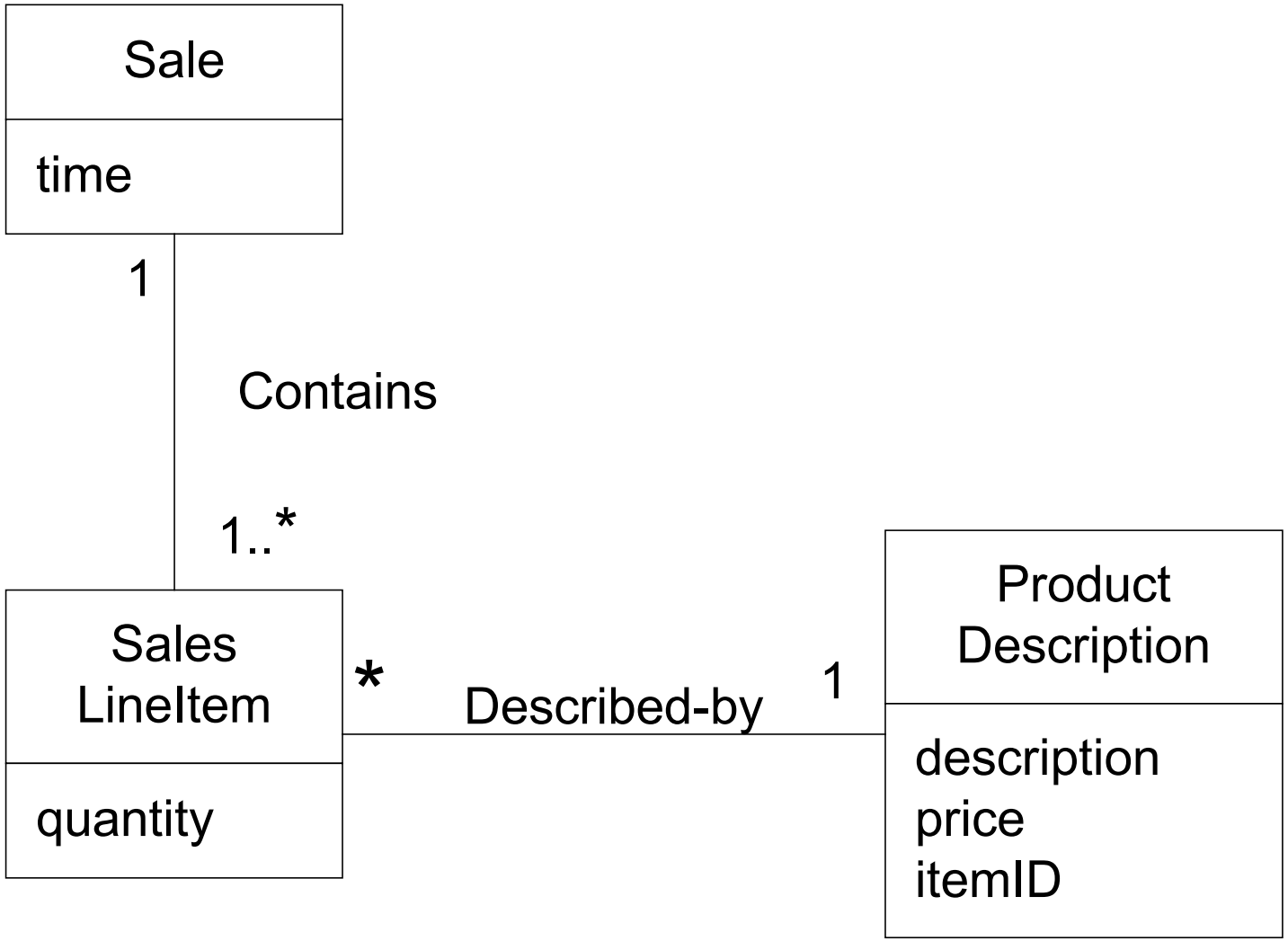
NextGen POS: Partial Domain Model



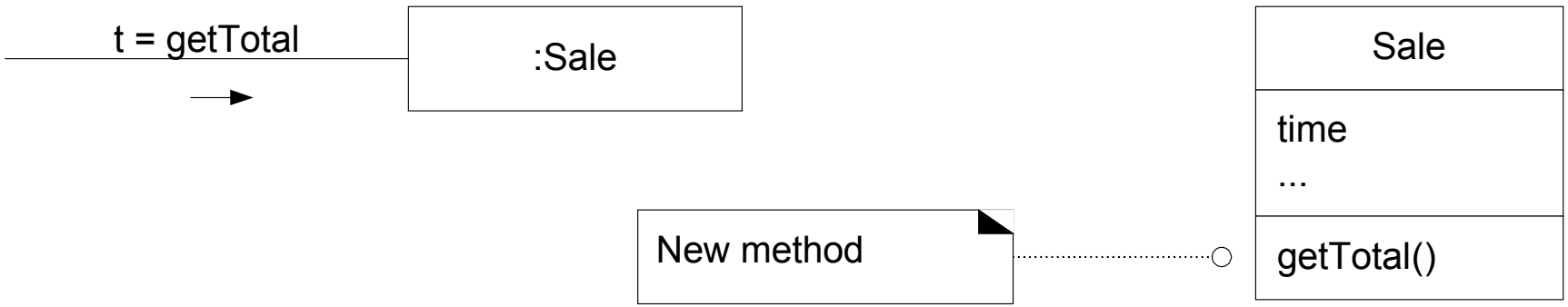
Creating a *SalesLineItem*



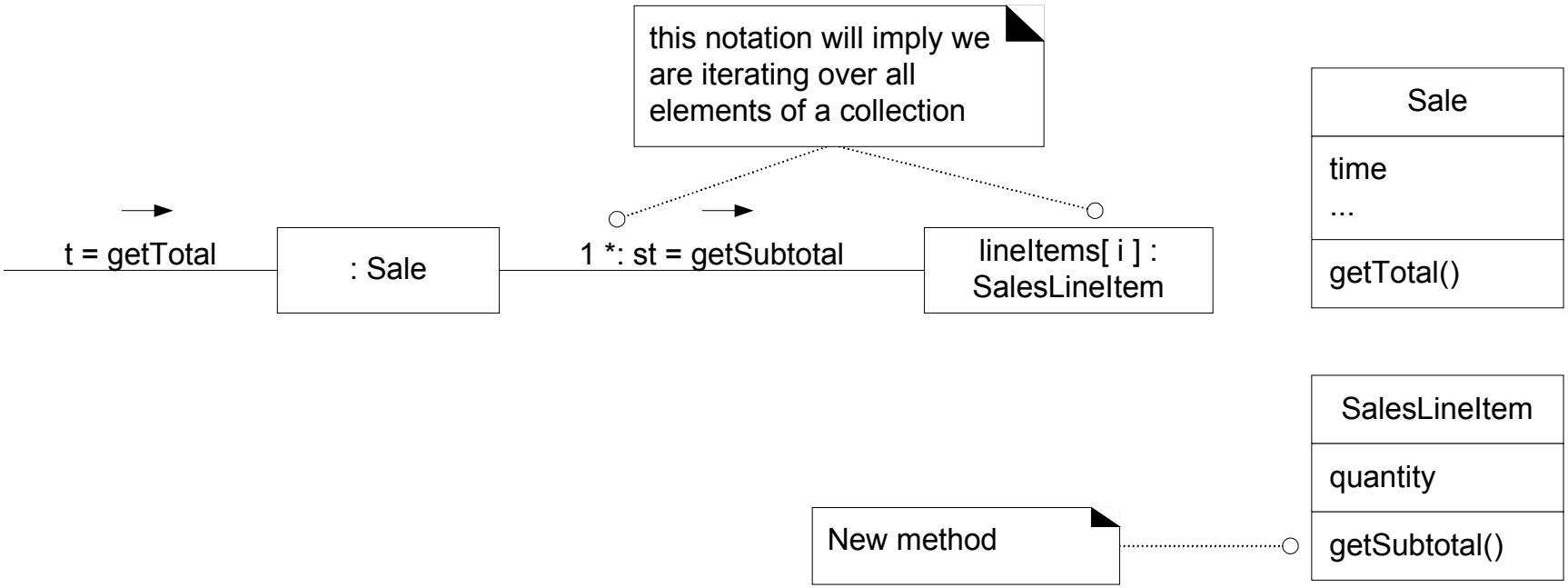
Associations of *Sale*



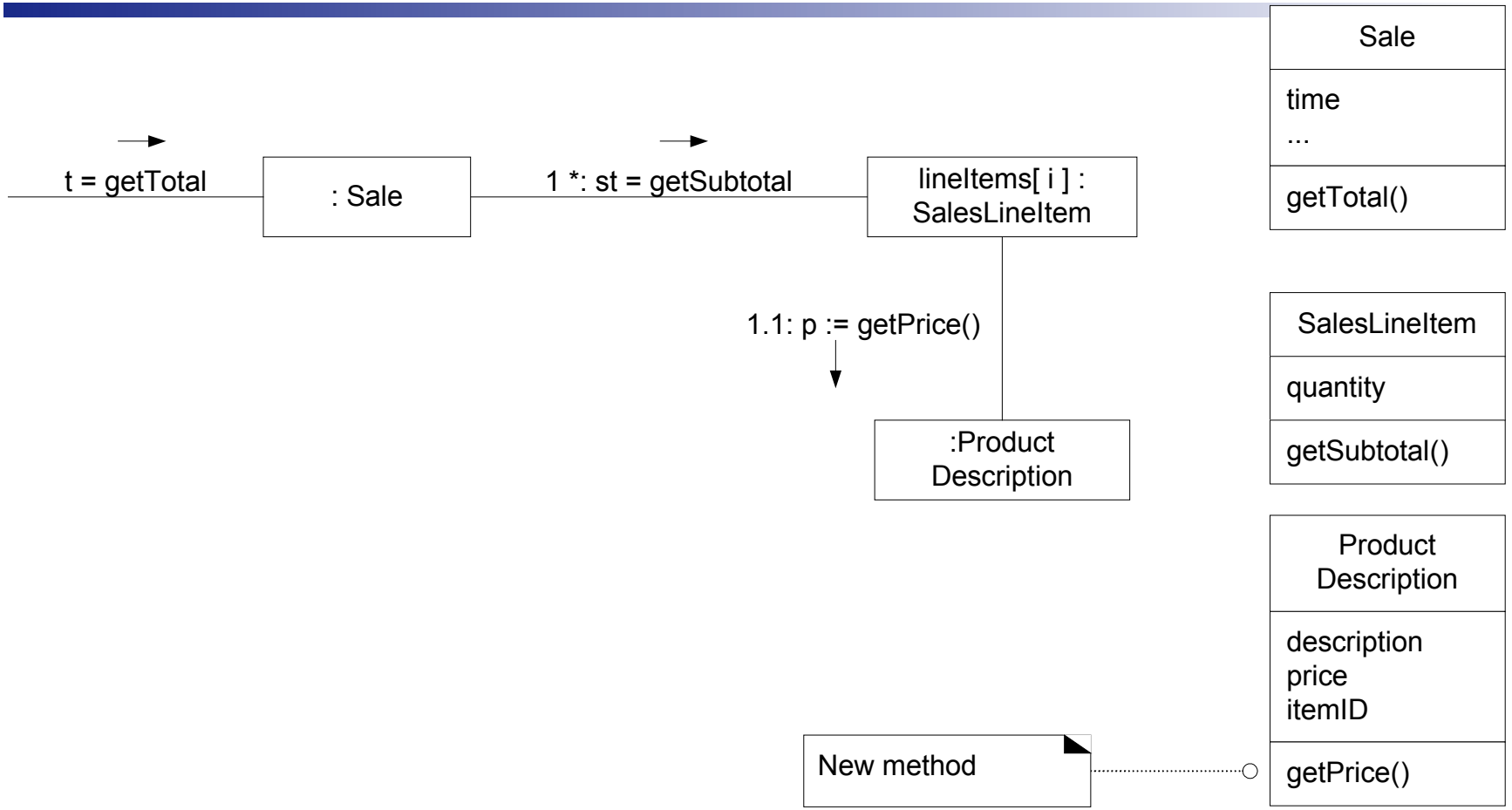
Partial Interaction and Class Diagrams



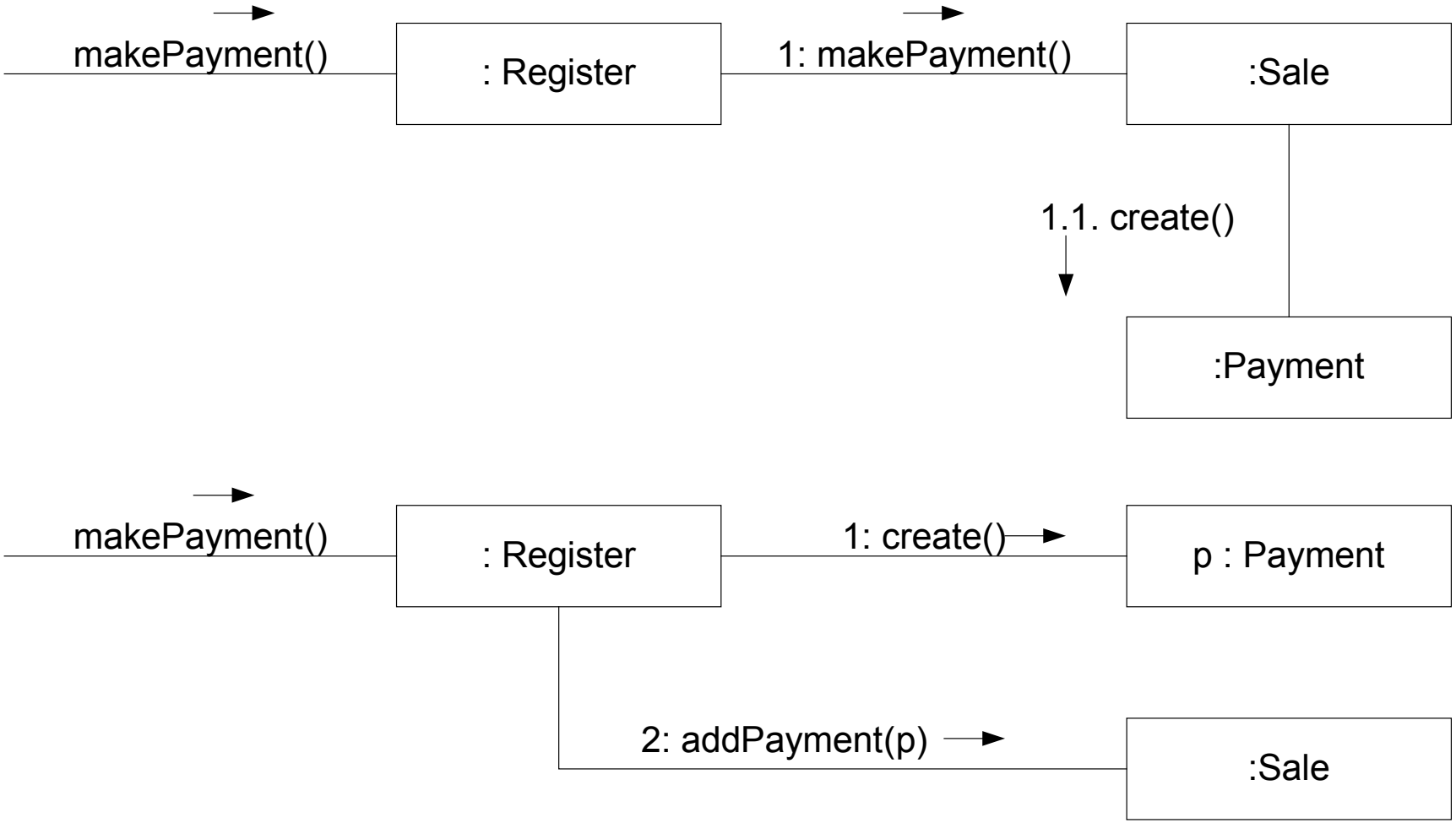
Calculating the *Sale* Total (1)



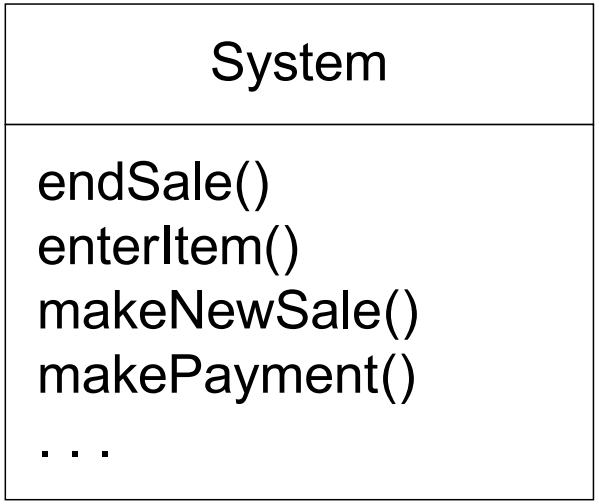
Calculating the *Sale* Total (2)



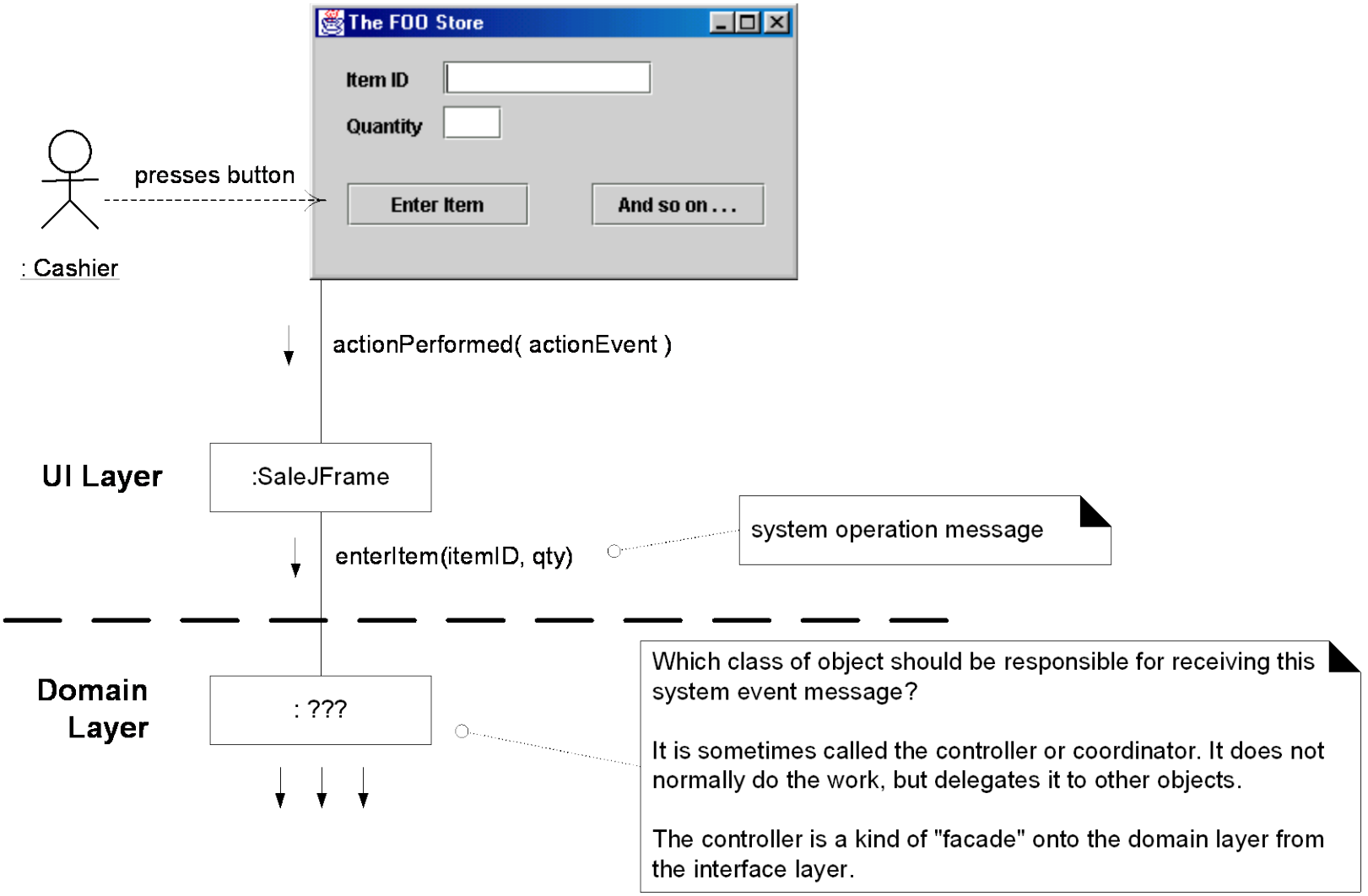
Create *Payment*: *Sale* v. *Register*



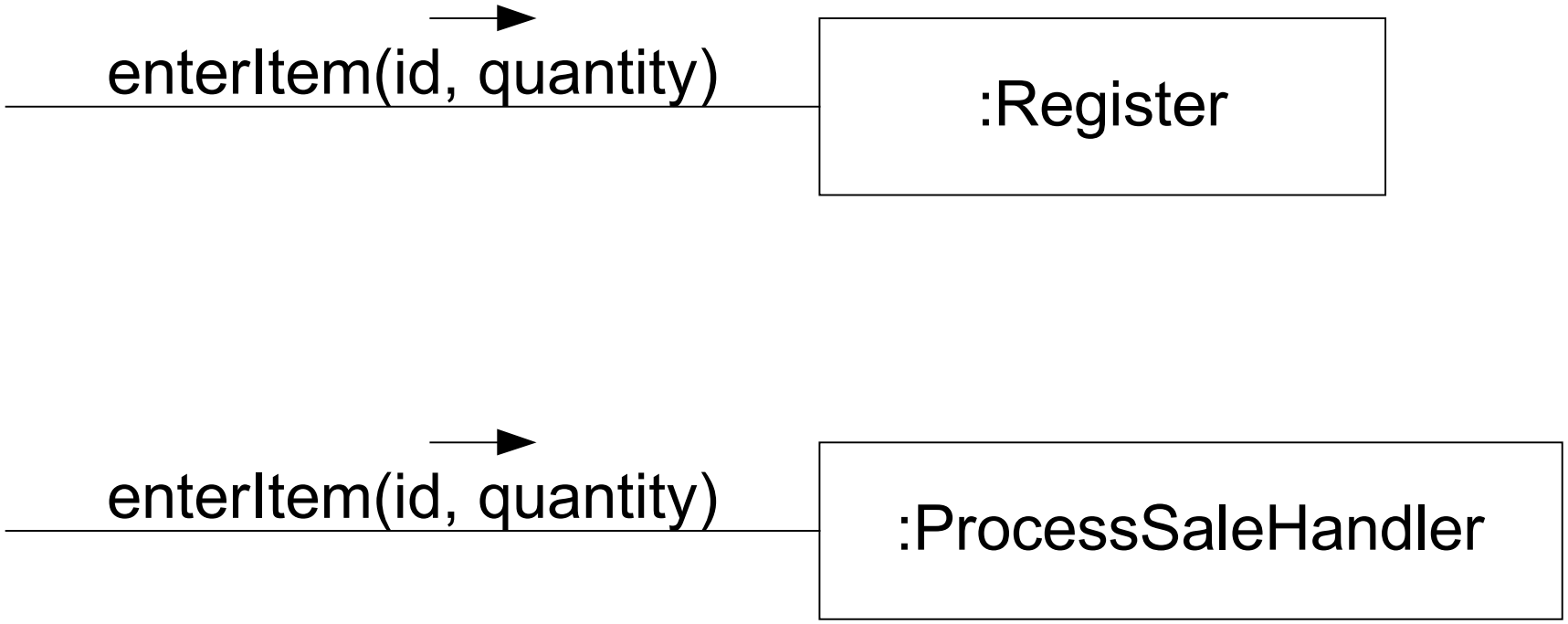
NextGen POS: Some System Ops



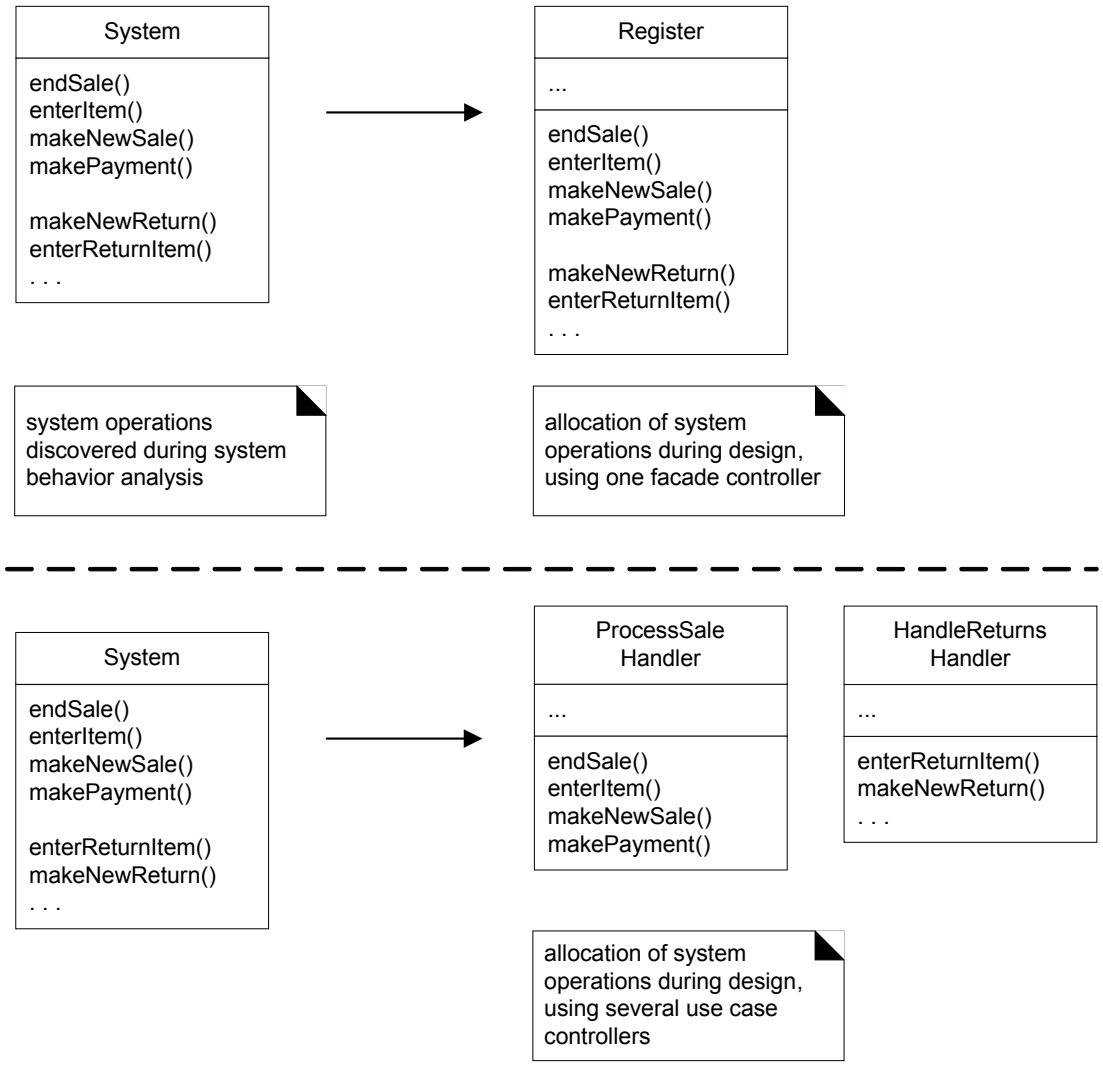
Controller for *enterItem*?



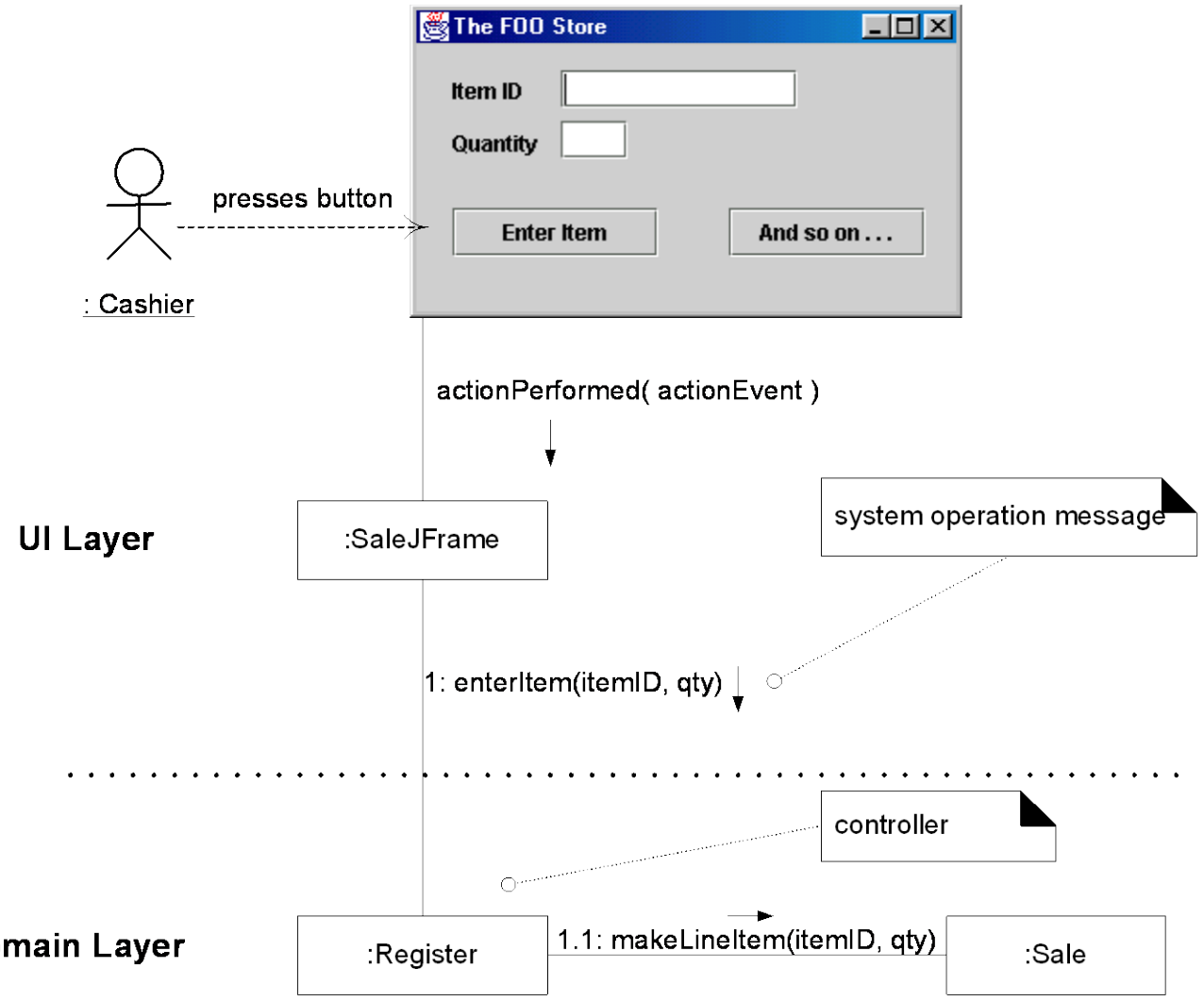
Controller Choices



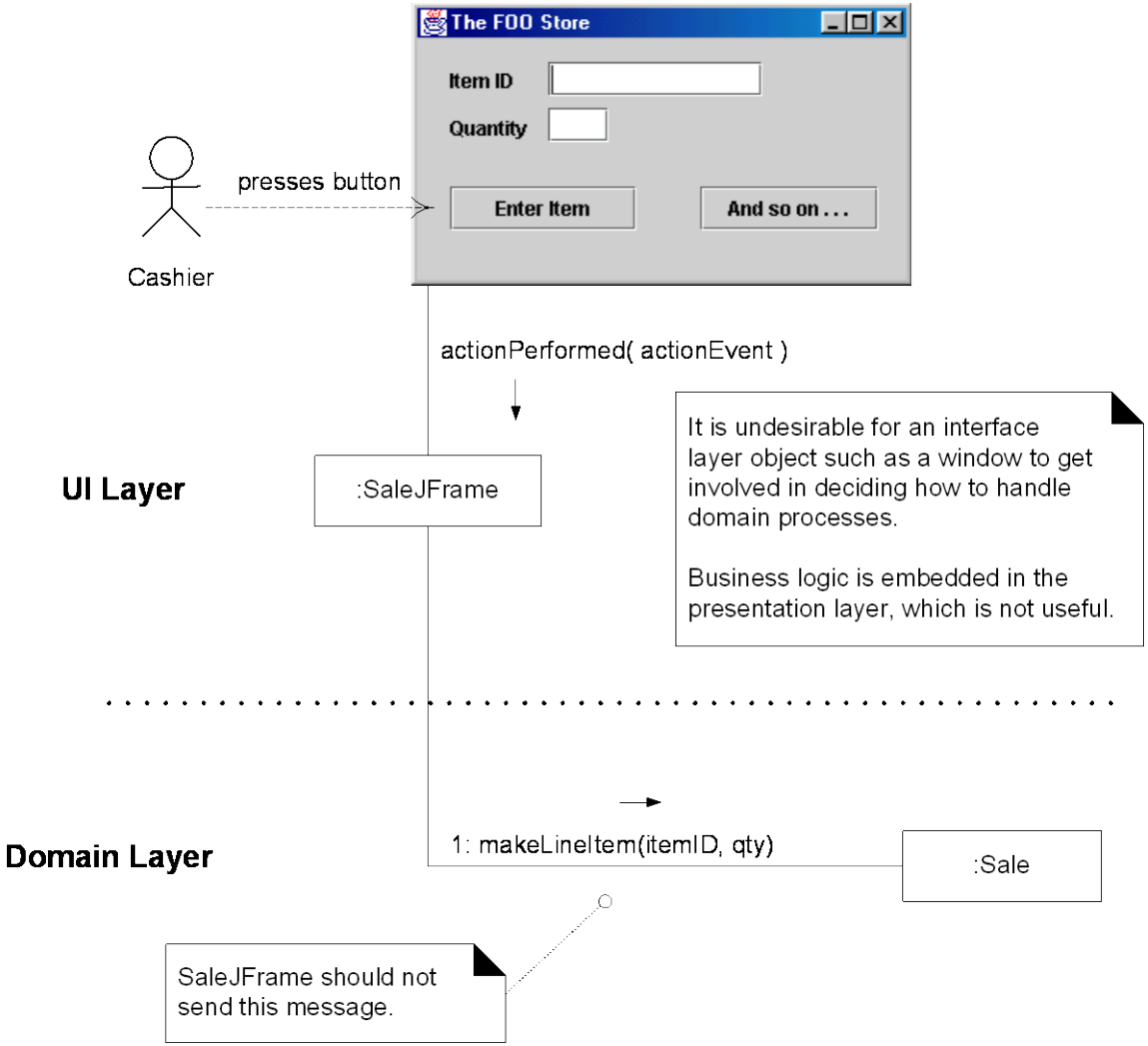
Allocation of System Operations



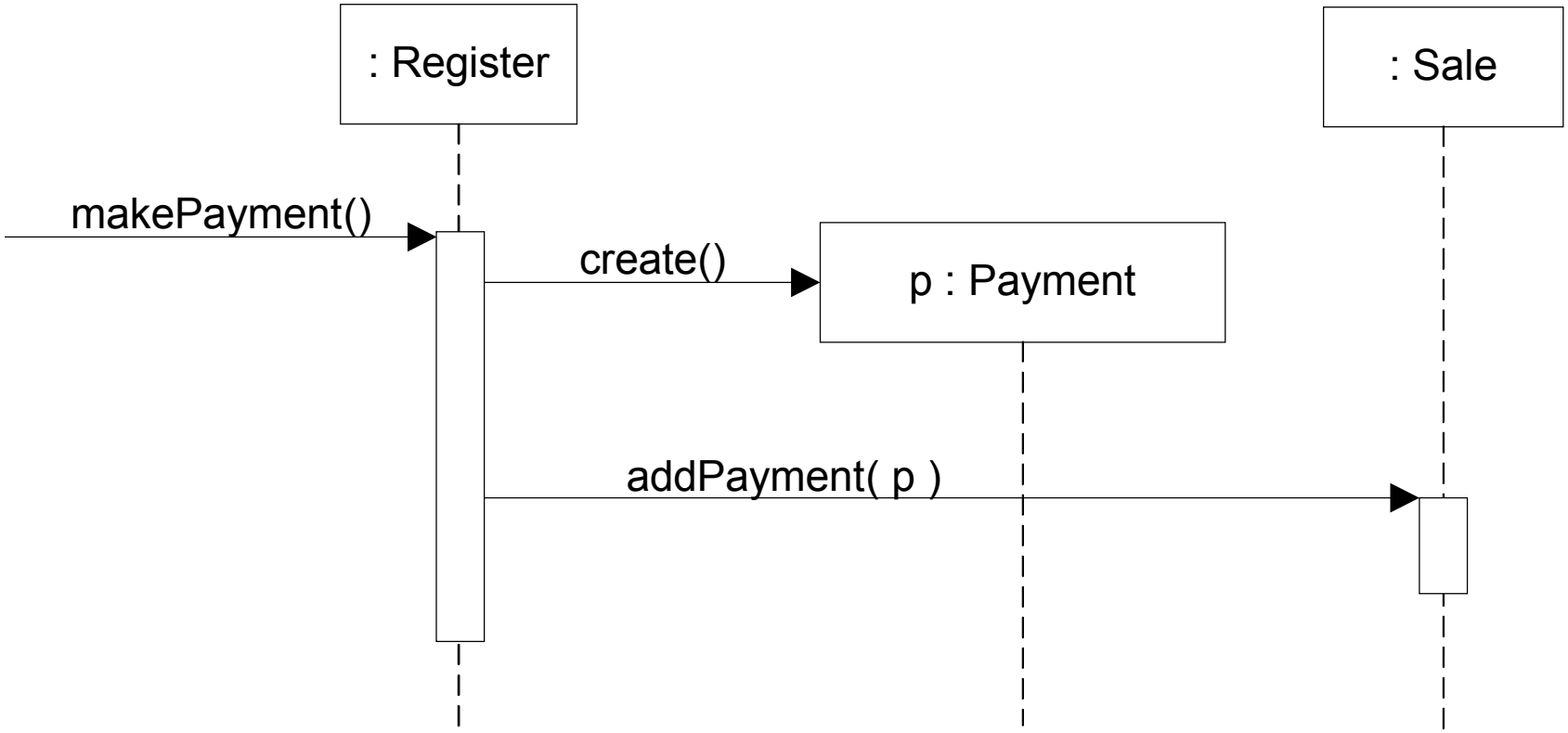
Desirable Coupling: UI to Domain Layer



Less Desirable Coupling: UI to Domain Layer



Register Creates *Payment*



Sale Creates *Payment*