GRASP Design Principles

COMP 3700.002
Software Modeling and Design

Shehenaz Shaik

Patterns

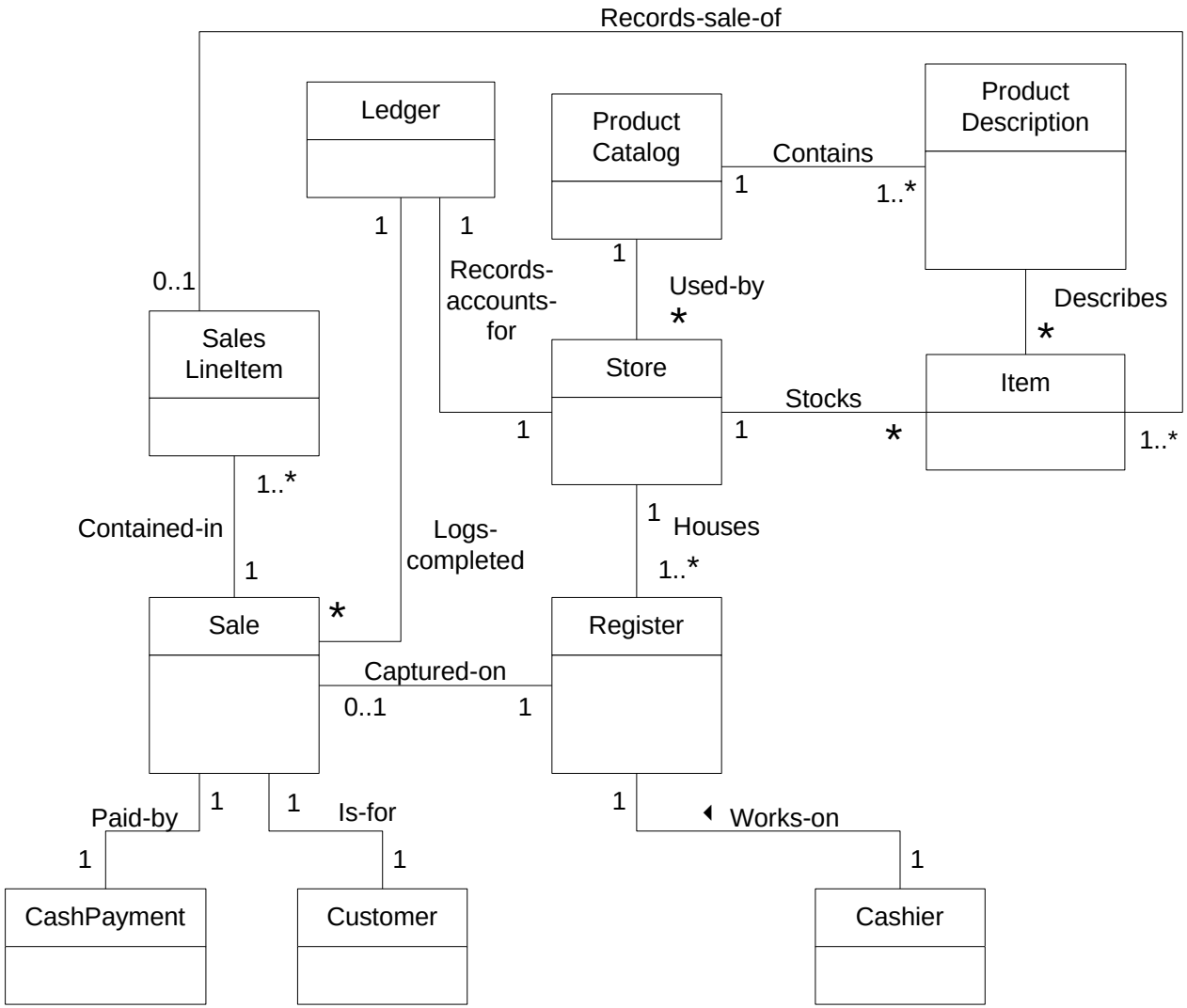
- Pattern
  - Named description of a well-known problem and solution that can be applied to new contexts
    - Guide assignment of responsibility to objects
    - How / when to apply
    - Benefits / Tradeoffs
- Contradictions
- Multiple solutions

GRASP Principles

General Responsibility Assignment Software Patterns

1. Creator
2. Information Expert
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

PoS: Domain Model



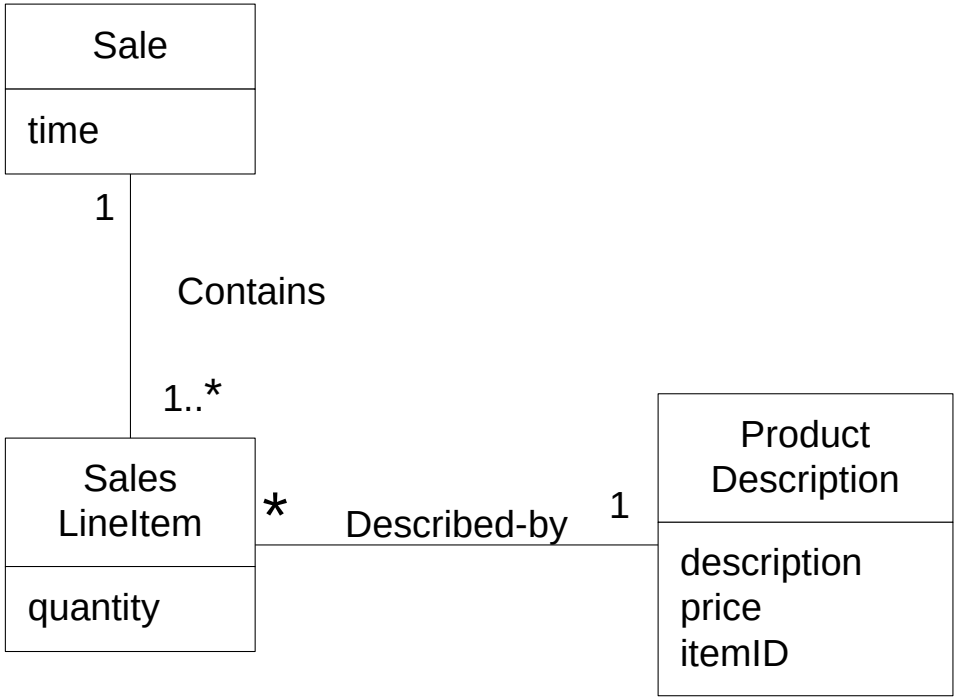
1. Creator

- Problem
  - Who should be responsible for creating a new instance of some class?
- Solution
  - Assign class B to create instance of class A, if
    - B contains or compositely aggregates A
    - B records A
    - B closely uses A
    - B has initializing data for A that will be passed to A when it is created
  - If >1 applicable, prefer class B which aggregates A

1. Creator: Example

- Partial Domain model for PoS system

PoS: Creator for SalesLineItem



1. Creator: Example (Contd.)

PoS: Creator for SalesLineItem

- Assign class B to create instance of class A, if
  - B contains or compositely aggregates A
  - B records A
  - B closely uses A
  - B has initializing data for A that will be passed to A when it is created

1. Creator: Example (Contd.)

PoS: Creator for SalesLineItem

- Assign class B to create instance of class A, if
  - B contains or compositely aggregates A
  - B records A
  - B closely uses A
  - B has initializing data for A that will be passed to A when it is created

1. Creator: Example (Contd.)

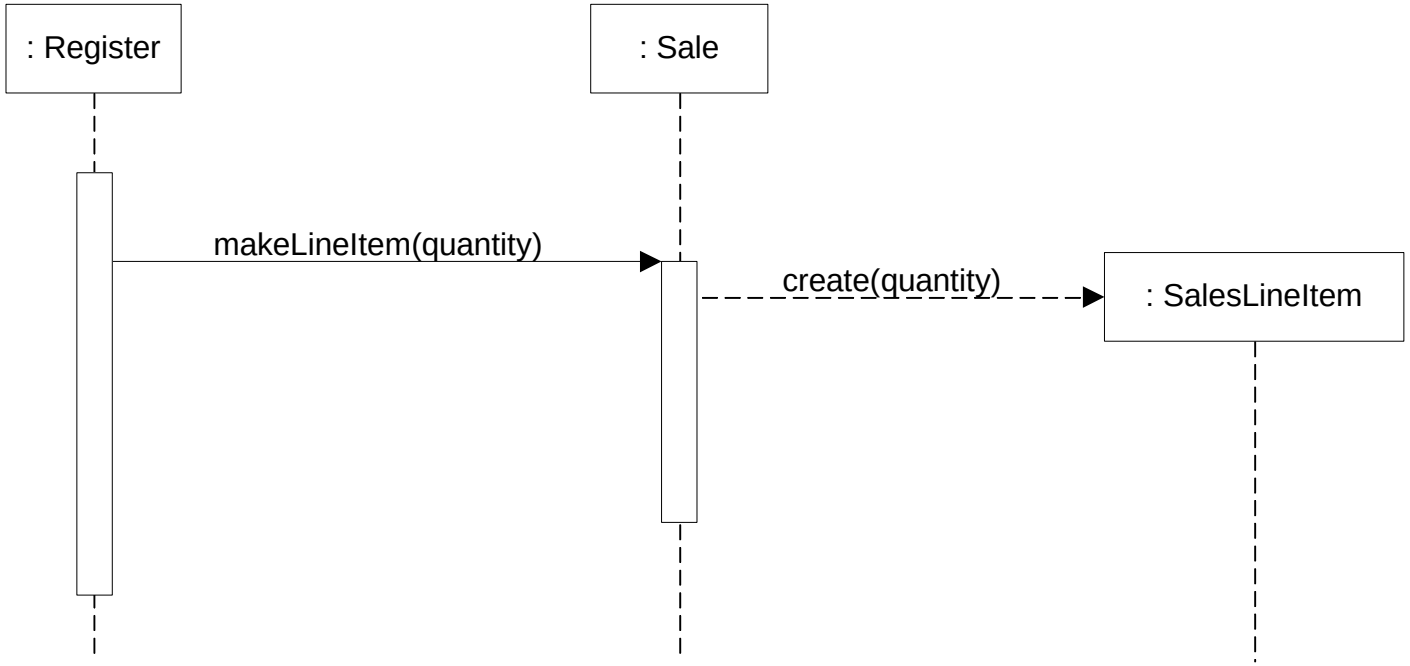
PoS: Creator for SalesLineItem

- Assign class B to create instance of class A, if
  - B contains or compositely aggregates A
  - B records A
  - B closely uses A
  - B has initializing data for A that will be passed to A when it is created

1. Creator: Example (Contd.)

PoS: Creating a SalesLineItem

- Find a Creator that needs to be connected to created object in any event



1. Creator (Contd.)

- Benefits
  - Low coupling is supported
    - Lower maintenance dependencies
    - Higher opportunities for reuse

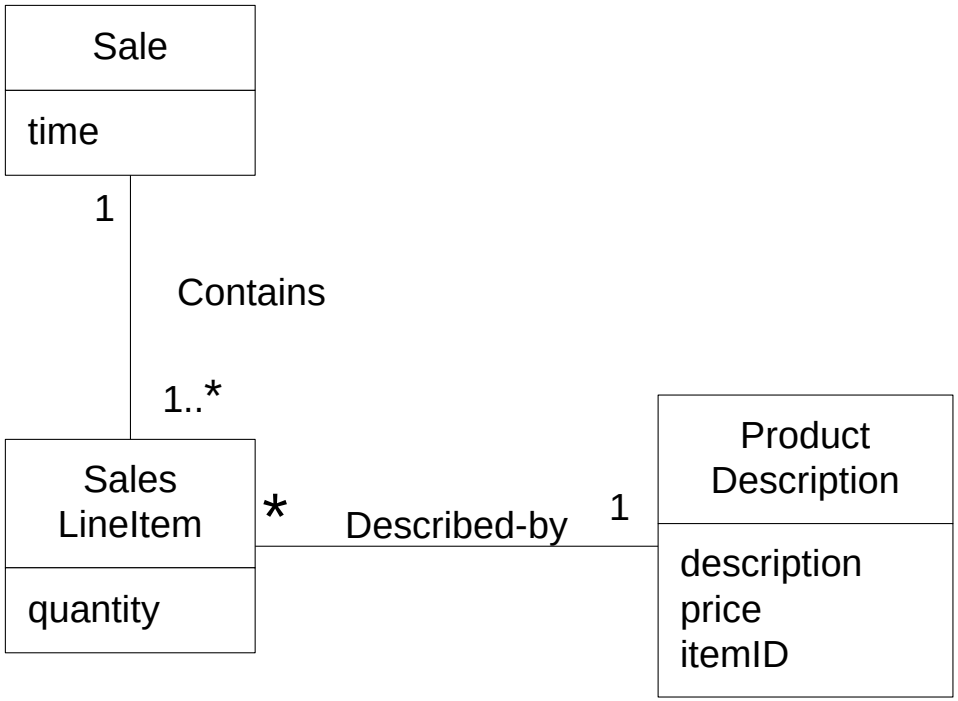
2. Information Expert (Expert)

- Problem
  - What is a general principle of assigning responsibilities to objects?
- Solution
  - Assign a responsibility to the information expert - class that has the information necessary to fulfill the responsibility

2. Information Expert: Example

- Partial Domain model for PoS system

PoS: Responsibility for grand total of a Sale



2. Information Expert: Example (Contd.)

PoS: Responsibility for grand total of a Sale

- Approach
  - Look for class of objects with required information
  - Where to look?
    - Domain Model?
    - Design Model?

2. Information Expert: Example (Contd.)

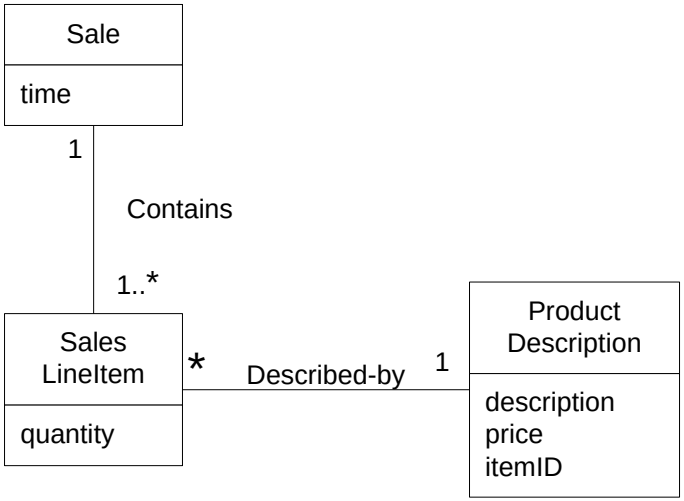
PoS: Responsibility for grand total of a Sale

- Approach
  - Look for class of objects with required information
  - Where to look?
    1. Design Model
    2. Domain Model
       a. Expand Design Model by creating corresponding Design classes

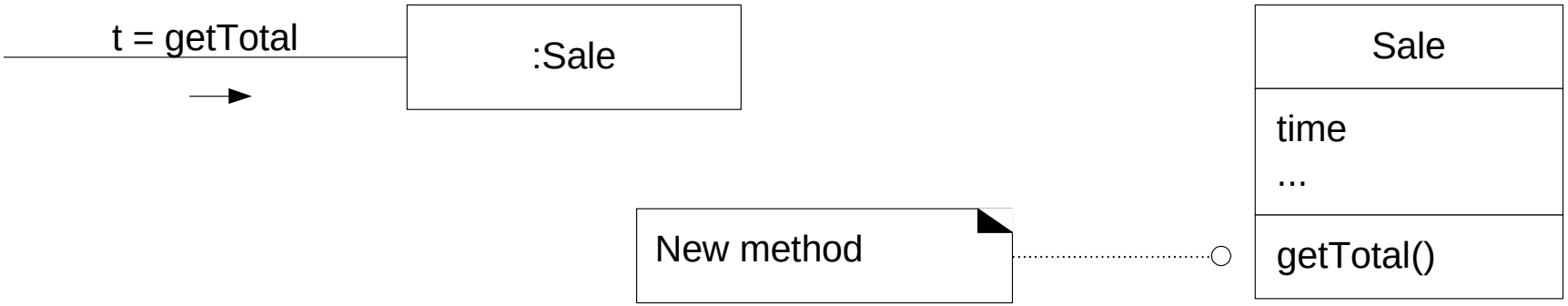
2. Information Expert: Example (Contd.)

Responsibility for grand total of a Sale

- Domain Model



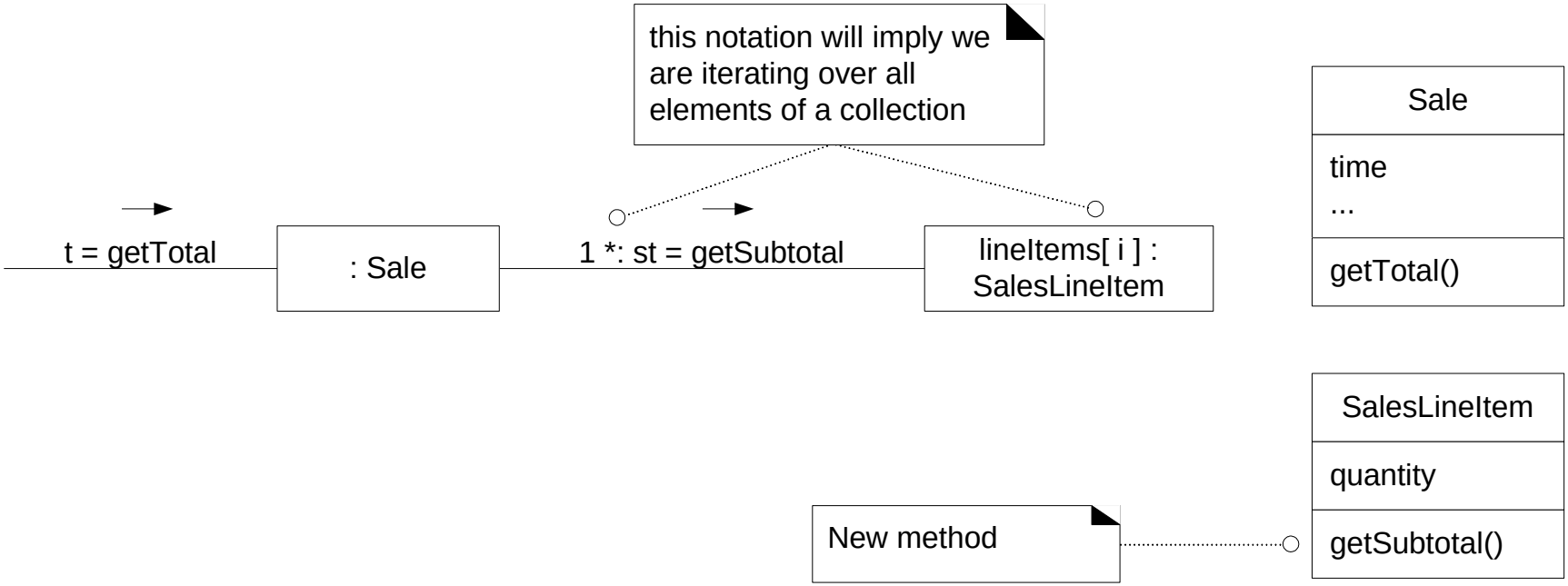
- Design Model



2. Information Expert: Example (Contd.)

Responsibility for grand total of a Sale

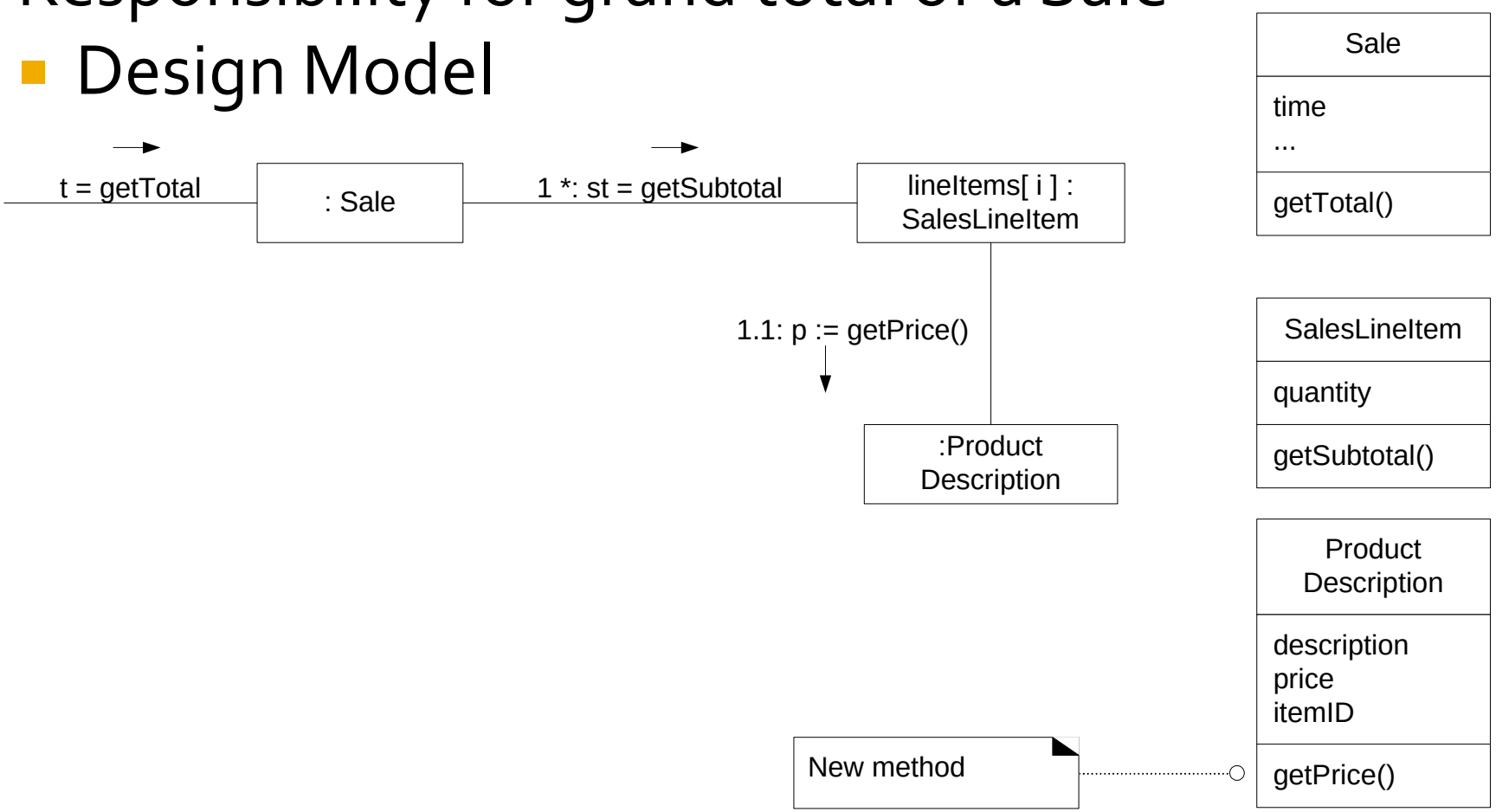
- Design Model



2. Information Expert: Example (Contd.)

Responsibility for grand total of a Sale

- Design Model



2. Information Expert (Contd.)

- Benefits
  - Information encapsulation
    - Objects use their own information
    - Low coupling
    - Lower maintenance dependencies
    - Higher opportunities for reuse
  - Behavior is distributed across classes with information
    - More cohesive lightweight classes
      - Easier to understand and maintain
    - High cohesion

3. Low Coupling

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

3. Low Coupling (Contd.)

- Coupling
  - Measure of how strongly one element is connected to, has knowledge of, or relies on other elements
- Forms of Coupling
  - X has an attribute that refers to a Y instance
  - X has a method that references an instance of Y
    - Parameter / Local / Return variable X calls on services of Y
  - X is a direct or indirect subclass of Y
  - X implements Y, where Y is an interface

3. Low Coupling (Contd.)

- Weak coupling
- Strong coupling
  - Issues
    - Forced local changes because of changes in related classes
    - Harder to understand in isolation
    - Harder to reuse – Requires presence of dependent classes

3. Low Coupling: Example

PoS: Responsibility to create a Payment instance and associate it with the Sale

- Three design classes
  - Payment
  - Register
  - Sale

3. Low Coupling : Example (Contd.)

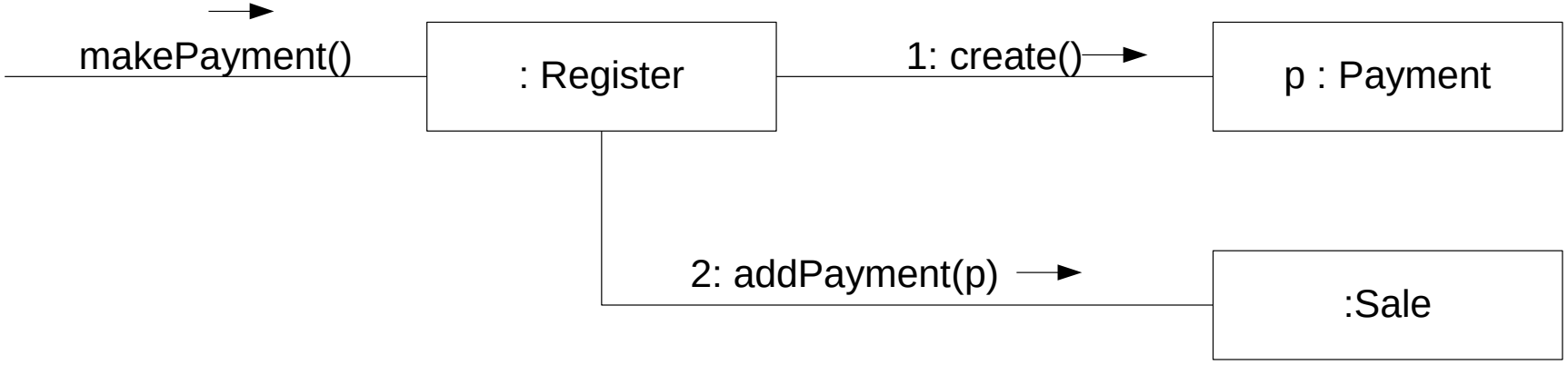
Responsibility to create a Payment instance and associate it with the Sale

- Creator pattern
  - Register 'Records' Payment
- Three design classes
  - Payment, Register, Sale

3. Low Coupling : Example (Contd.)

Responsibility to create a Payment instance and associate it with the Sale

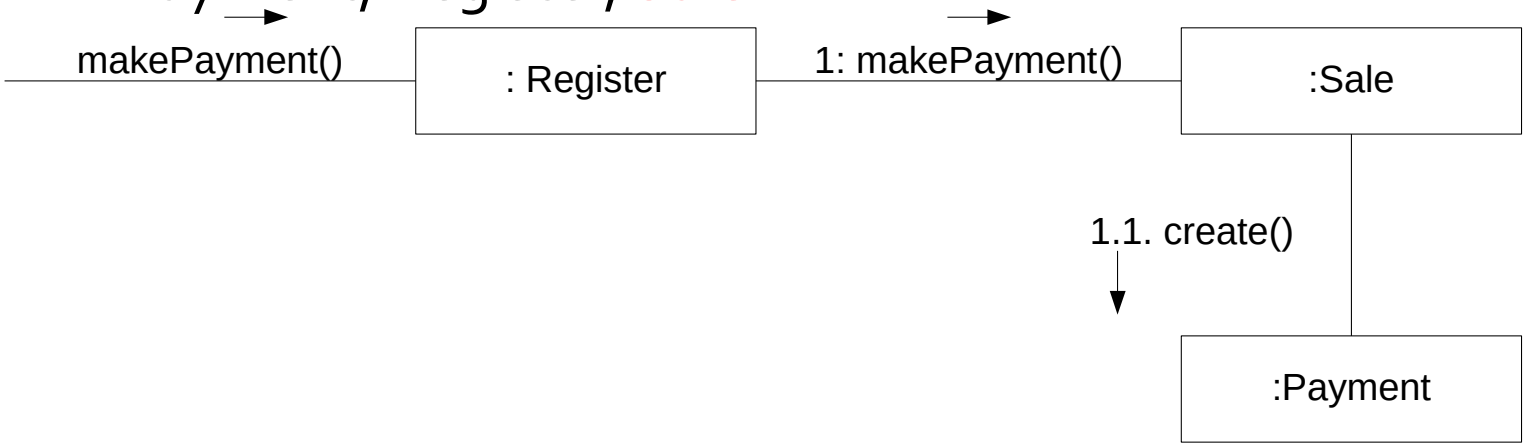
- Creator pattern
  - Register 'Records' Payment
- Three design classes
  - Payment, Register, Sale



3. Low Coupling : Example (Contd.)

Responsibility to create a Payment instance and associate it with the Sale

- Low Coupling pattern
  - Sale creates Payment
- Three design classes
  - Payment, Register, Sale



3. Low Coupling (Contd.)

- Benefits
  - Helps contain the increase in coupling
  - Supports design of more independent classes
    - Reduces impact of change
    - Simple to understand in isolation
    - Convenient to reuse
    - Note: Subclass / Superclass is strongest form of coupling. Avoid, if possible
- Limitations
  - Cannot obtain an absolute measure of when coupling is too high

3. Low Coupling (Contd.)

- Degrees of coupling
  - Low coupling when
    - Classes are inherently generic & high probability of reuse
  - Extreme low coupling
    - Poor design e.g.
      - Few incohesive, bloated, and complex active objects that do all the work and with many passive zero-coupled objects that act as simple data repositories
  - Moderate degree of coupling is normal
  - High coupling to stable elements and to pervasive elements is acceptable
    - Problem: High coupling to unstable elements

4. Controller

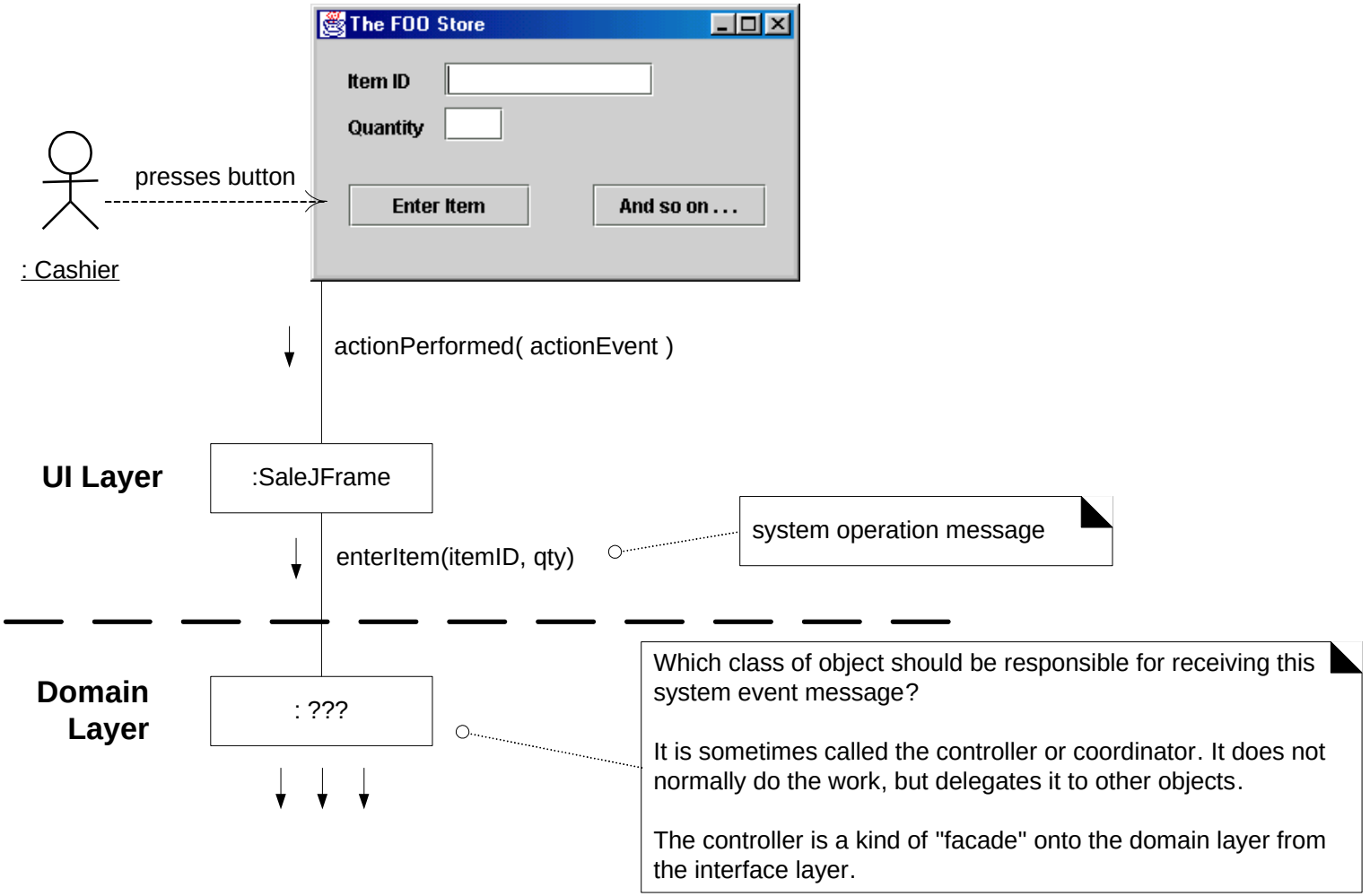
- Problem
  - What first object beyond the UI layer receives and coordinates (controls) a system operation?
- Solution
  - Assign a responsibility to a class representing:
    - Overall ‘system’, ‘root object’, device that software is running within, or a major subsystem
    - Use case scenario within which the system event occurs

4. Controller (Contd.)

- Solution
  - Assign a responsibility to a class representing:
    - Overall 'system', 'root object', device that software is running within, or a major subsystem
    - Use case scenario within which the system event occurs
      - Use same controller class for all system events in same use case
      - Referred as,
        - <UseCaseName>Handler
        - <UseCaseName>Controller
        - <UseCaseName>Session

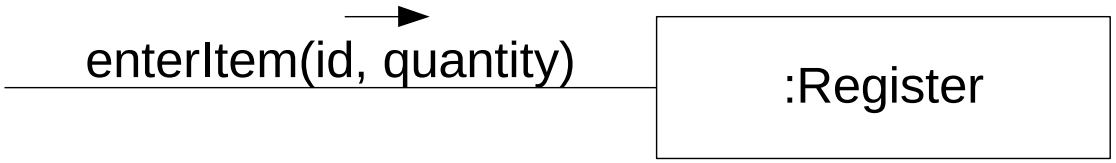
4. Controller: Example

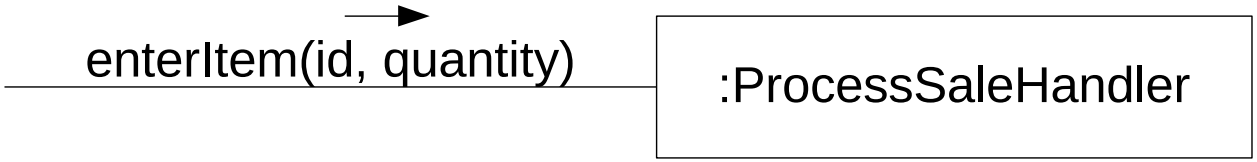
PoS: What object should be the Controller for enterItem?



4. Controller: Example (Contd.)

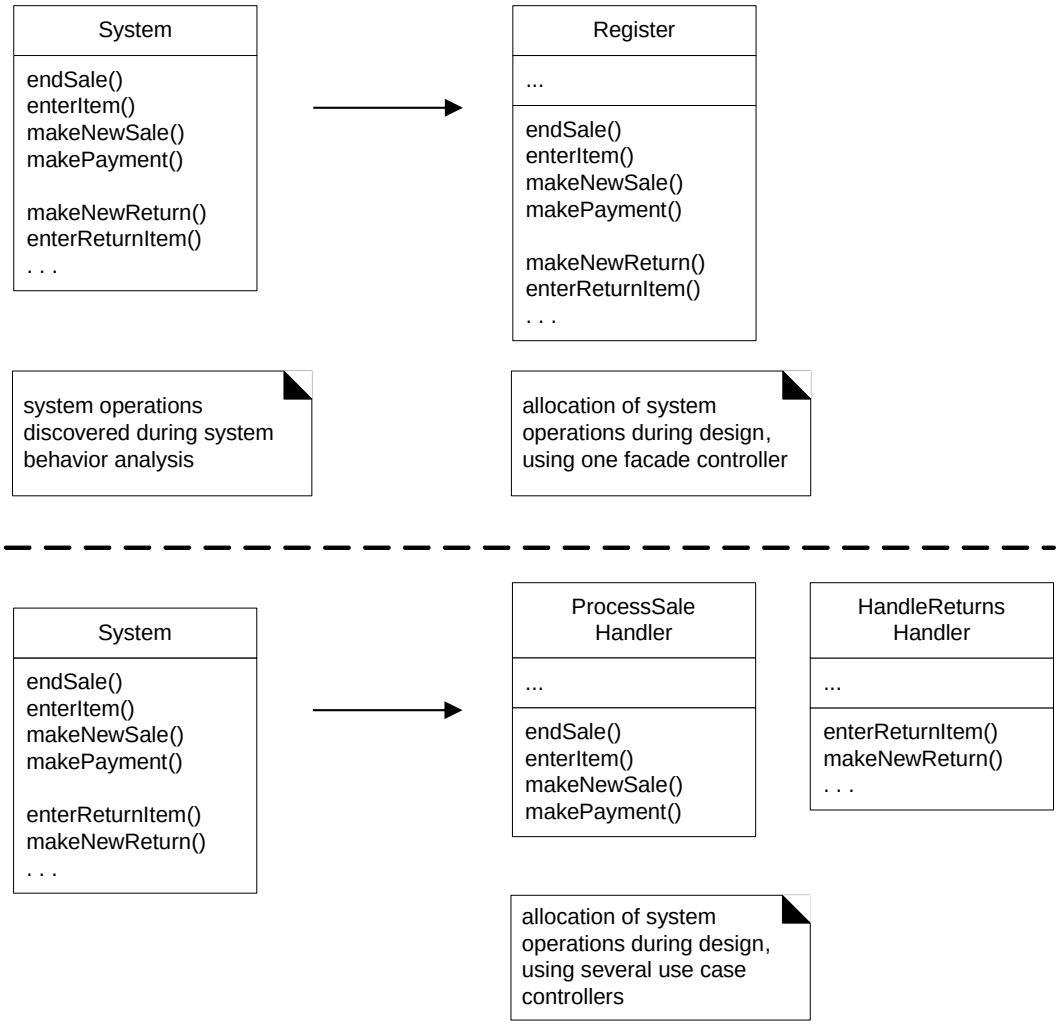
- Controller class choices
  - Represents system / subsystem
    - Register, POSSystem
  - Represents a use case scenario
    - ProcessSaleHandler, ProcessSaleSession





4. Controller: Example (Contd.)

- Allocation of system operations



4. Controller: Example (Contd.)

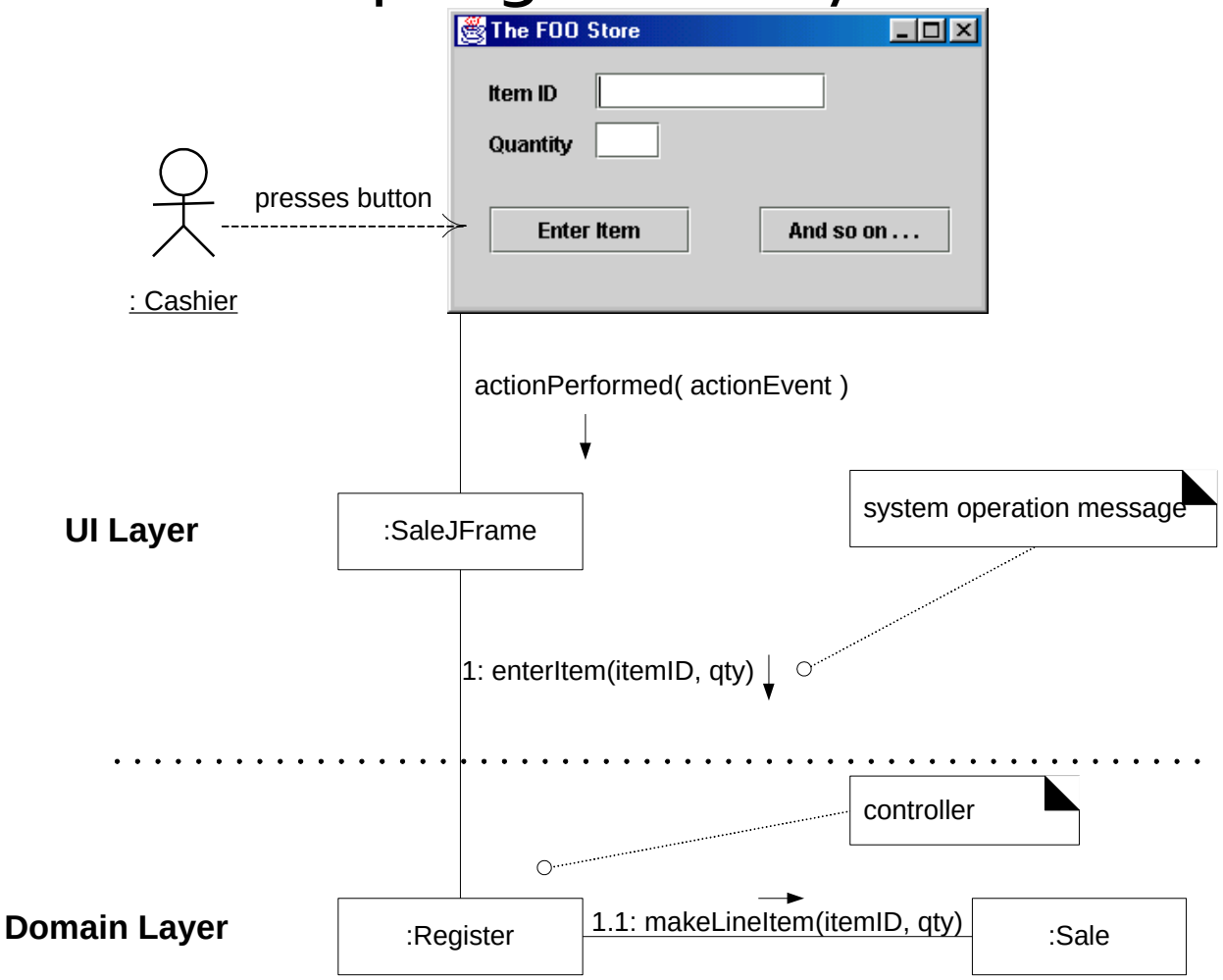
- Facade Controller
  - Suitable when not many system events
  - Issue: Overassignment of responsibilities
    - Low cohesion
- Usecase Controller
  - Same controller for all events in a use case
  - Maintains state information
  - Identifies anomalies
  - Consider when FacadeController leads to Low Cohesion or High Coupling
    - Manageable classes

4. Controller (Contd.)

- Bloated Controller
  - Single controller class receiving all system events
    - Many events
  - Controller itself performs many tasks
    - No delegation
    - Violates High Cohesion and Information Expert
  - Controller has many attributes
    - Maintains significant information about system, which should have been distributed
    - Duplicates information found elsewhere
- Unfocused and handling too many areas of responsibility
  - Solution: Add more controllers

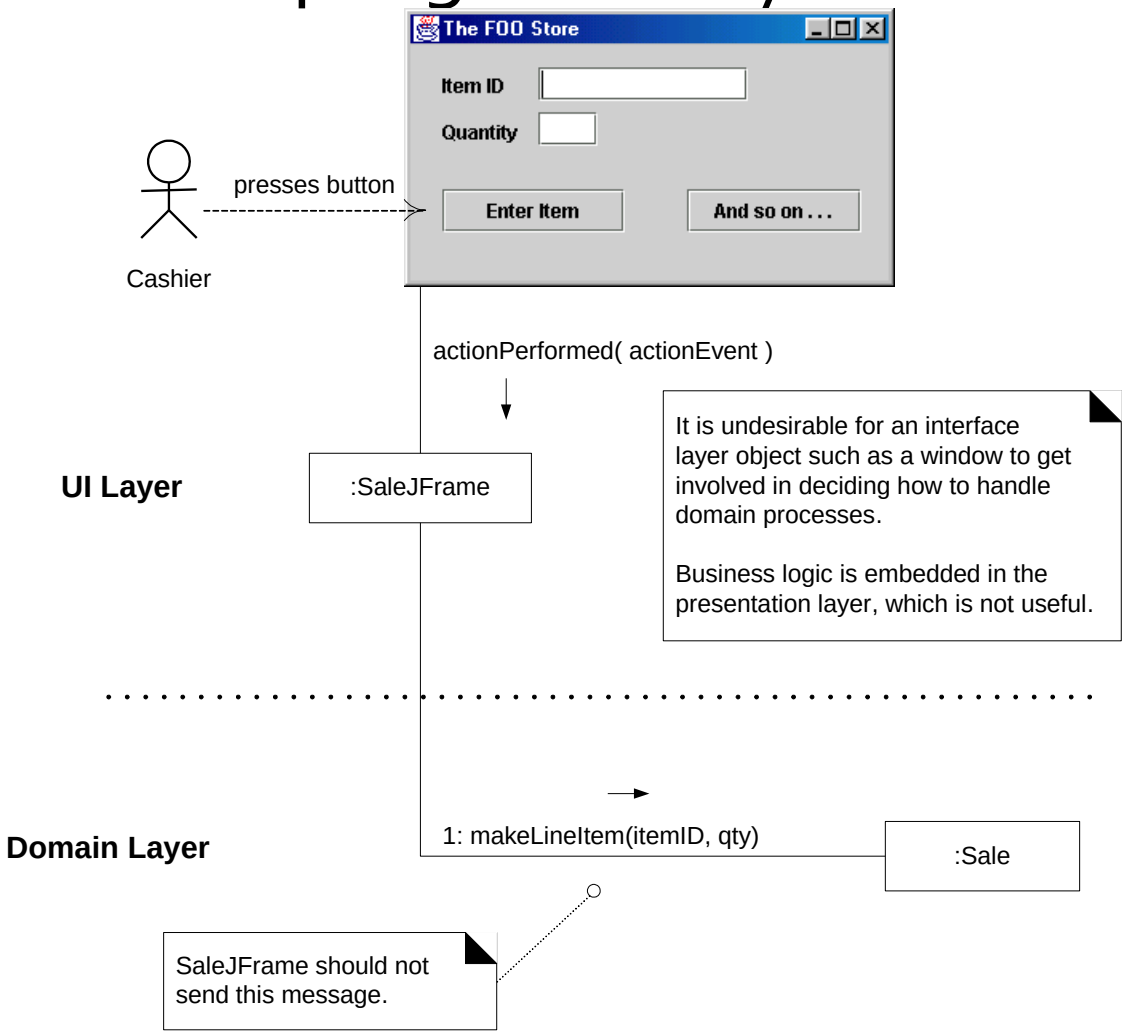
4. Controller (Contd.)

- Desirable coupling of UI Layer to Domain Layer



4. Controller (Contd.)

- Undesirable coupling of UI Layer to Domain Layer



4. Controller (Contd.)

- Benefits
  - Increased potential for reuse and pluggable interfaces
    - Application logic is not handled in interface layer
  - Opportunity to reason about the state of use case

5. High Cohesion

- Problem
  - How to keep objects focused, understandable, and manageable?
- Solution
  - Assign a responsibility so that cohesion remains high
- Use this principle to evaluate alternatives

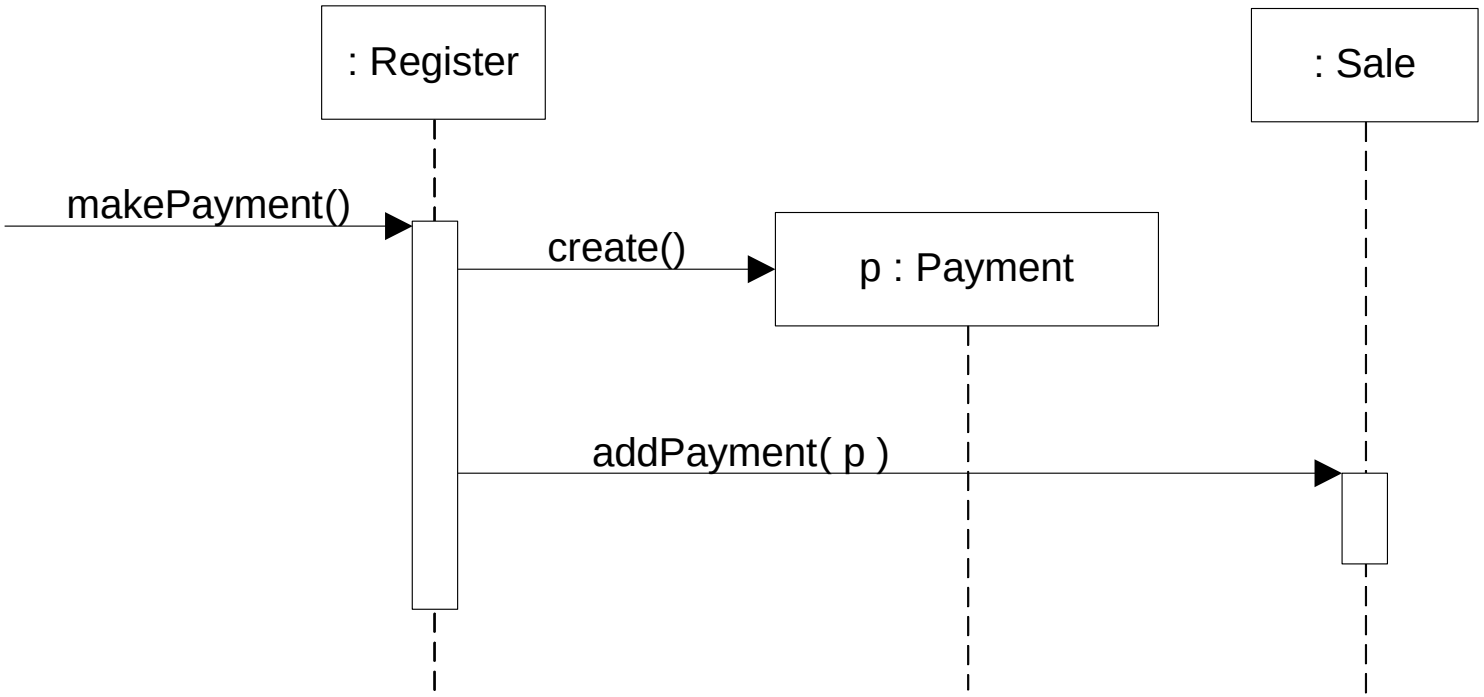
5. High Cohesion (Contd.)

- Cohesion
  - Measure of how strongly related and focused the responsibilities of an element are
- Low Cohesion
  - Issues
    - Hard to comprehend
    - Hard to reuse
    - Hard to maintain
    - Delicate; Constantly affected by change

5. High Cohesion : Example

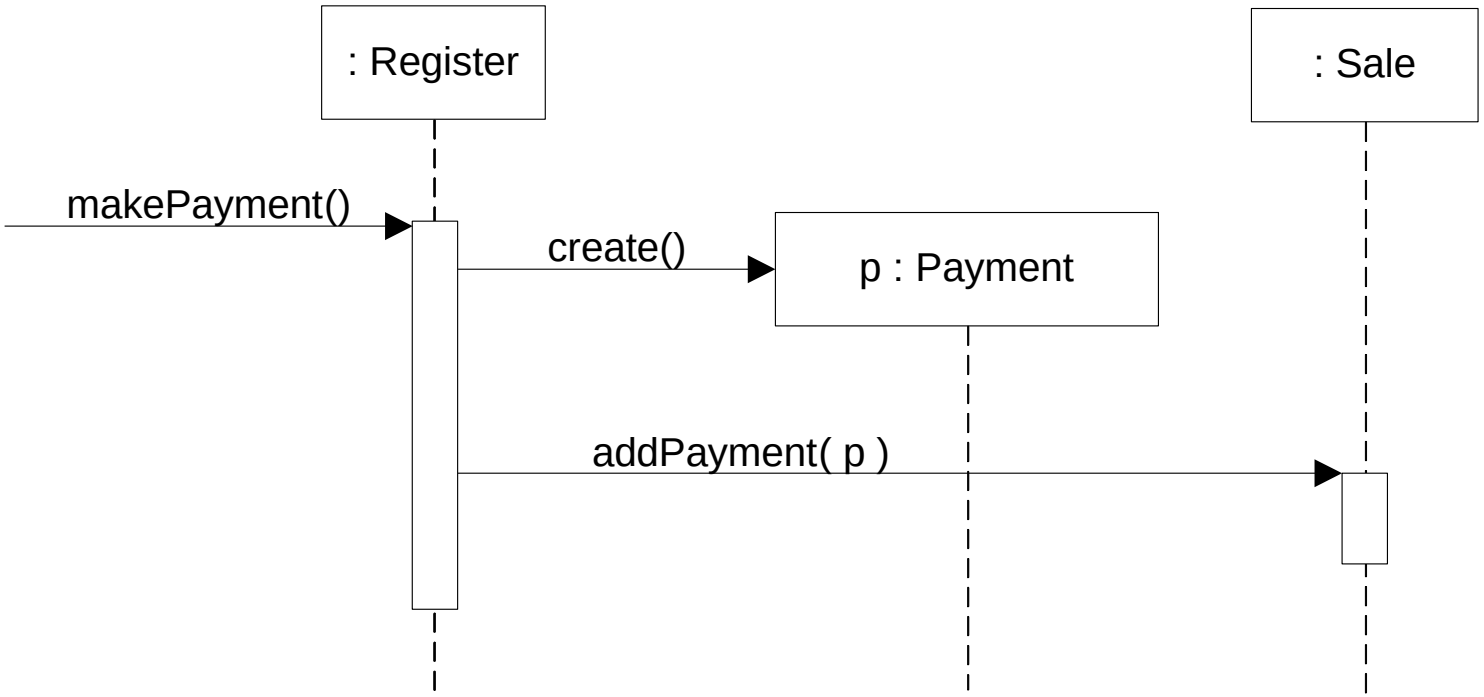
PoS: Create a (cash) payment instance and associate it with the Sale

- Say, Register creates Payment



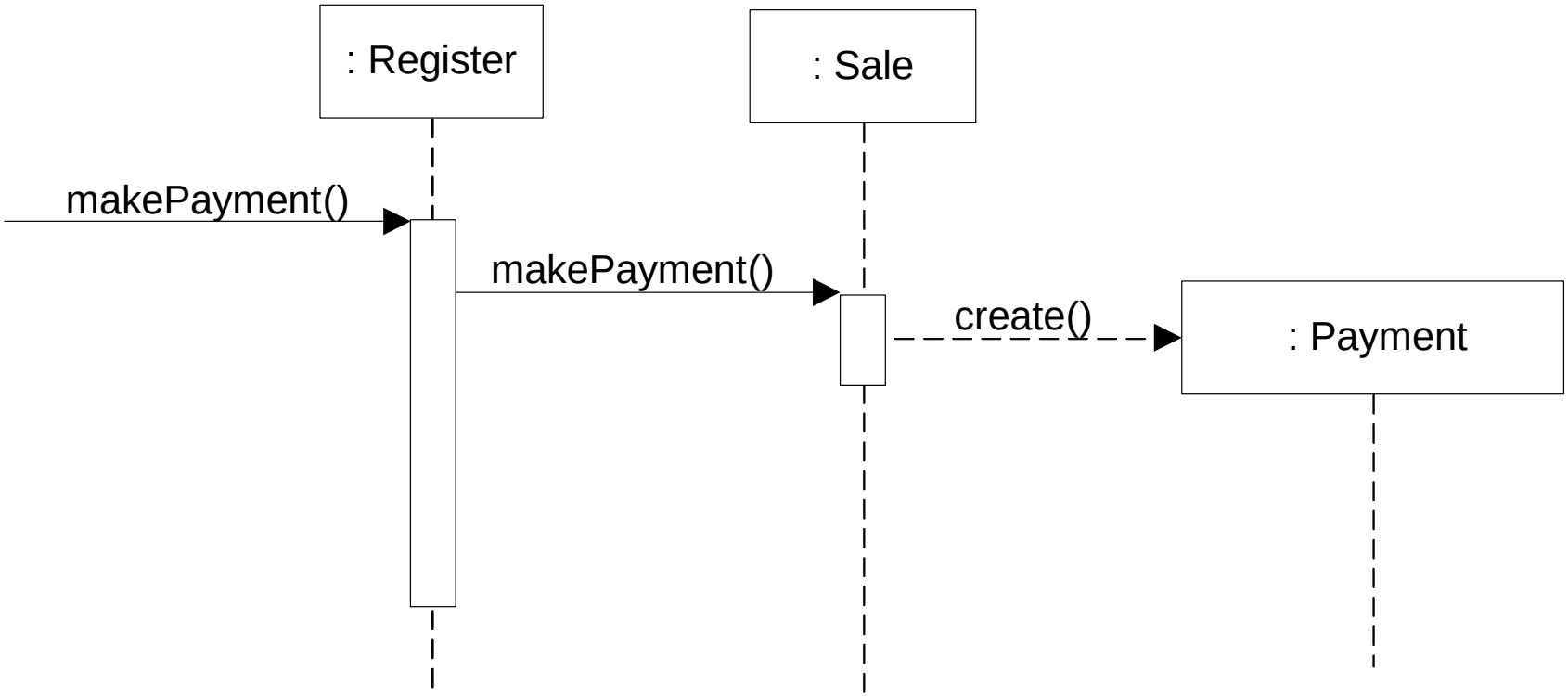
5. High Cohesion : Example

- Say, Register creates Payment
  - Acceptable for one system operation
  - Incohesive, and bloated – if many system operations



5. High Cohesion : Example

- Say, Sale creates Payment
  - Higher cohesion in Register class, from delegation



5. High Cohesion (Contd.)

- Degrees of Cohesion
  - Very low cohesion
    - Complete system functionality in one class
  - Low cohesion
    - Related classes, but many
  - High cohesion
    - Moderate responsibilities in one functional area
    - Collaborates with other classes
    - Partially responsible for a task
    - High coupling
  - Moderate cohesion
    - Lightweight
    - Sole responsibilities for a few logically related, but different, areas

5. High Cohesion (Contd.)

- Benefits
  - High degree of related functionality
    - Increased reuse
  - Small number of operations
    - Relatively easy to understand and maintain

6. Polymorphism

- Problem
  - How to handle alternatives based on type?
  - How to create pluggable software components?
- Solution
  - When related alternatives / behaviors vary by type(class), assign responsibility for the behavior to the types for which the behavior varies
    - Using polymorphic operations

6. Polymorphism: Example

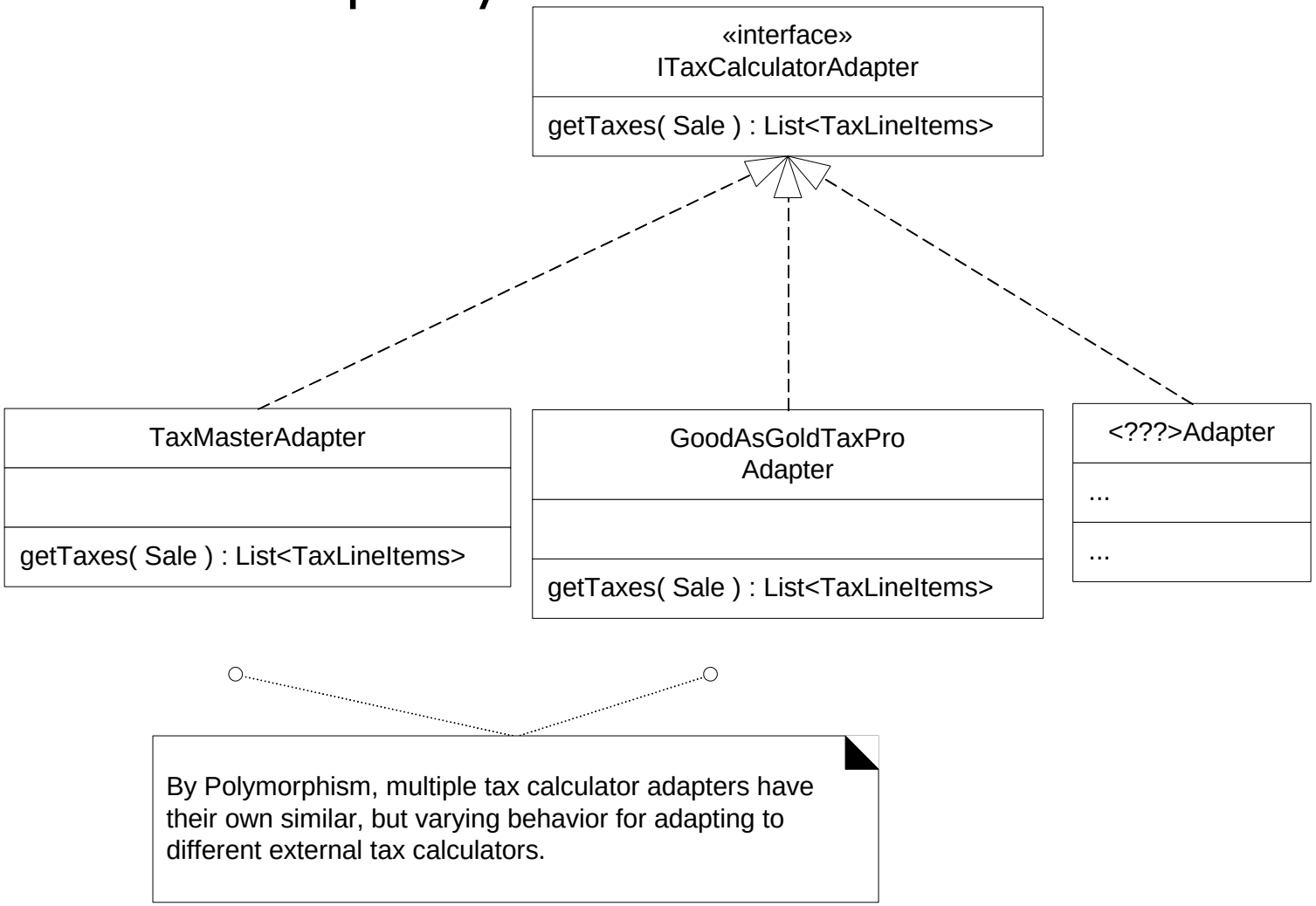
PoS: Third-party tax calculators

- Challenge
  - Multiple external third-party tax calculators
  - Different interfaces (APIs)

- What objects should own the responsibility?
  - Calculator adapter objects
    - Polymorphic methods

6. Polymorphism: Example

- PoS: Third-party tax calculators



6. Polymorphism

- Benefits
  - Extensions required for new variations are easy to add
  - New implementations can be introduced without affecting clients

7. Pure Fabrication

- Problem
  - What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for e.g.) are not appropriate?
- Solution
  - Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept – to support high cohesion, low coupling, and reuse

7. Pure Fabrication: Example

PoS: Saving a Sale Object in a Database

- Solution: by Expert
  - Assign responsibility to Sale class
  - Concerns:
    - Database-oriented operations → Incohesive
    - Coupled to database interface → Increased coupling
    - Duplicated general task → Poor reuse

7. Pure Fabrication: Example

PoS: Saving a Sale Object in a Database

- Solution:
  - Assign as sole responsibility to a new class 'PersistentStorage'
  - Benefits:
    - Sale class remains well-designed
    - PersistentStorage class
      - is cohesive
      - Is generic and reusable

7. Pure Fabrication

- Benefits
  - High Cohesion is supported
  - Reuse potential may increase
- Limitations
  - If overused
    - Too many behavior objects
    - Responsibilities not co-located with data
    - Affects coupling

8. Indirection

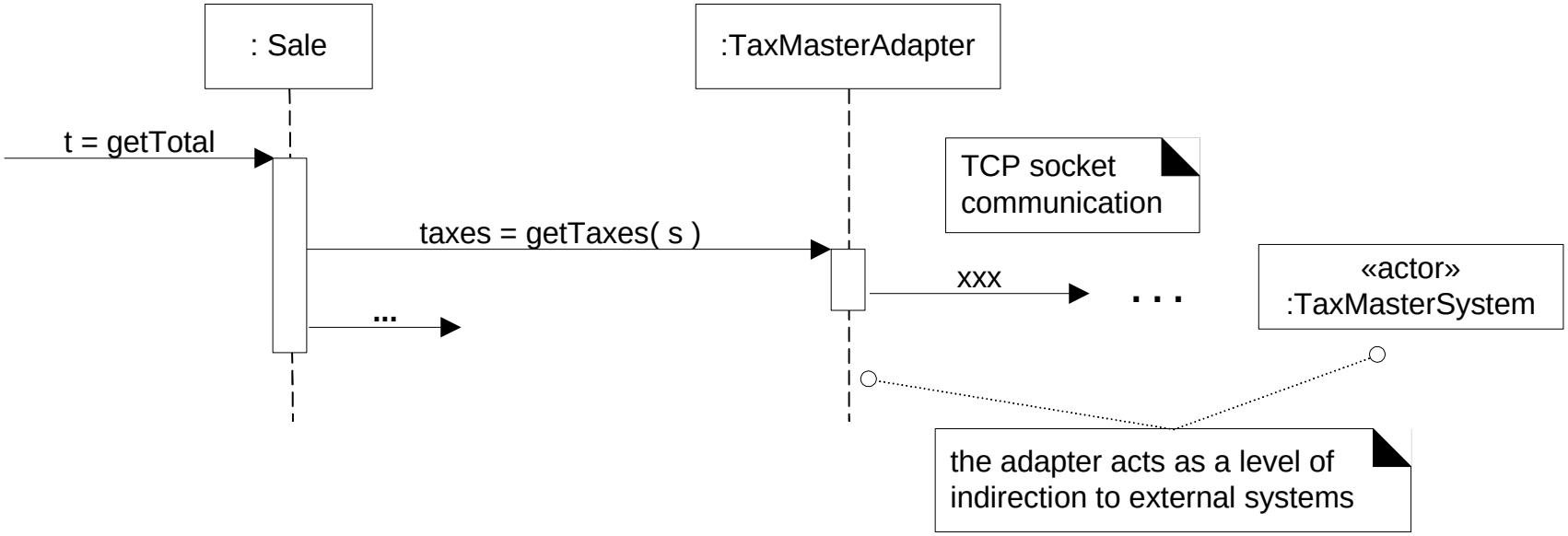
- Problem
  - Where to assign a responsibility, to avoid direct coupling between two (or more) things?
  - How to decouple objects so that low coupling is supported, and reuse potential remains higher?
- Solution
  - Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.
  - The intermediary creates an indirection between the other components

8. Indirection: Example

- Tax calculator Adapter
  - Act as Intermediary
  - Provide a consistent interface to inner objects
    - By Polymorphism
  - Hide variations of external APIs

8. Indirection

- Benefit
  - Lower coupling between components



9. Protected Variations

- Problem
  - How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
- Solution
  - Identify points of predicted variation or instability
  - Assign responsibilities to create a stable interface around them

9. Protected Variations: Example

PoS: External tax calculator

- Protection from variations in external APIs
- Internal objects collaborate with stable interface
- Adapter implementations hide variations to external systems

9. Protected Variations

- Benefits
  - Extensions required for new variations are easy to add
  - New implementations can be introduced without affecting clients
  - Coupling is lowered
  - Impact or cost of changes can be lowered

9. Protected Variations: Don't Talk to Strangers: Example

- Direct objects are 'familiars'
  - sale.getPayment()
- Indirect objects are 'strangers'
  - sale.getPayment().getTenderedAmount()
  - foo.getA().getB() .getC() .getD() .getE()
  - Fragile code
  - Depends on instability of object structure
    - Acceptable for standard libraries
    - But, not in new systems

9. Protected Variations: Don't Talk to Strangers

- Objects allowed to send messages to 'familiars', listed below
  - The 'this' object
  - A parameter of the method
  - An attribute of 'this'
  - An element of a collection which is an attribute of 'this'
  - An object created within the method

GRASP Principles: Summary

General Responsibility Assignment Software Patterns

1. Creator
2. Information Expert
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

Visibility

- Ability to see, or have a reference to another object
- Class A has visibility to Class B, if
  - **Attribute visibility**–when B is an attribute of A
  - **Parameter visibility**–when B is passed as a parameter to a method of A
  - **Local visibility**–when B is declared as a local object within a method of A
  - **Global visibility**–when B is global to A

Next sessions…

- Interaction and Class Design

Reading assignment

- Reference Book
  - Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition, Craig Larman, 2004
    - Chapter 17: GRASP: Designing Objects with Responsibilities: Pages 291-318.
    - Chapter 19: Designing For Visibility: Pages 363-368.
    - Chapter 25: GRASP: More Objects with Responsibilities: Pages 413-434.