Principles of Software Construction: Objects, Design and Concurrency

Design: GRASP and Refinement

15-214

toad

Fall 2013

Jonathan Aldrich Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

With slides from Klaus Ostermann

Object Design

- "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."
- But how?
  - What method belongs where?
  - How should the objects interact?
  - This is a critical, important, and non-trivial task

GRASP Patterns / Principles

- The GRASP patterns are a *learning aid* to
  - help one understand essential object design
  - apply design reasoning in a methodical, rational, explainable way.
- This approach to understanding and using design principles is based on patterns of assigning **responsibilities**

GRASP - Responsibilities

- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - knowing
  - doing
- Doing responsibilities of an object include:
  - 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 responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

GRASP

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software
- Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Describe fundamental principles of object design and responsibility
- General principles, may be overruled by others

Patterns/Principles aid Communication

Fred: "Where do you think we should place the responsibility for creating a SalesLineItem? I think a Factory."

Wilma: "By Creator, I think Sale will be suitable."

Fred: "Oh, right - I agree."

Nine GRASP Principles:

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

Low Coupling Principle

Problem:

How to increase reuse and decrease the impact of change.

Solution:

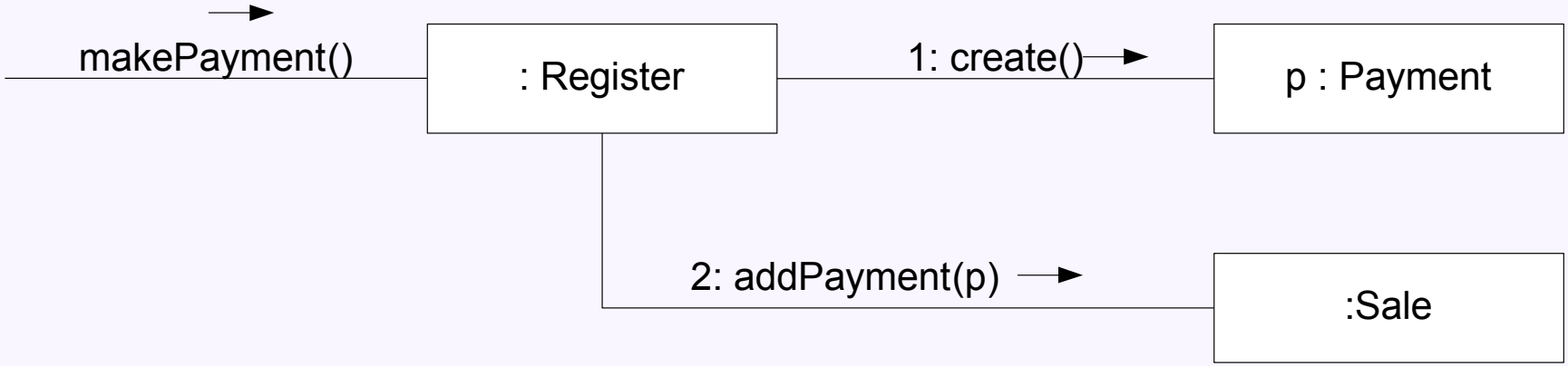
Assign responsibilities to minimize coupling.

Use this principle when evaluating alternatives

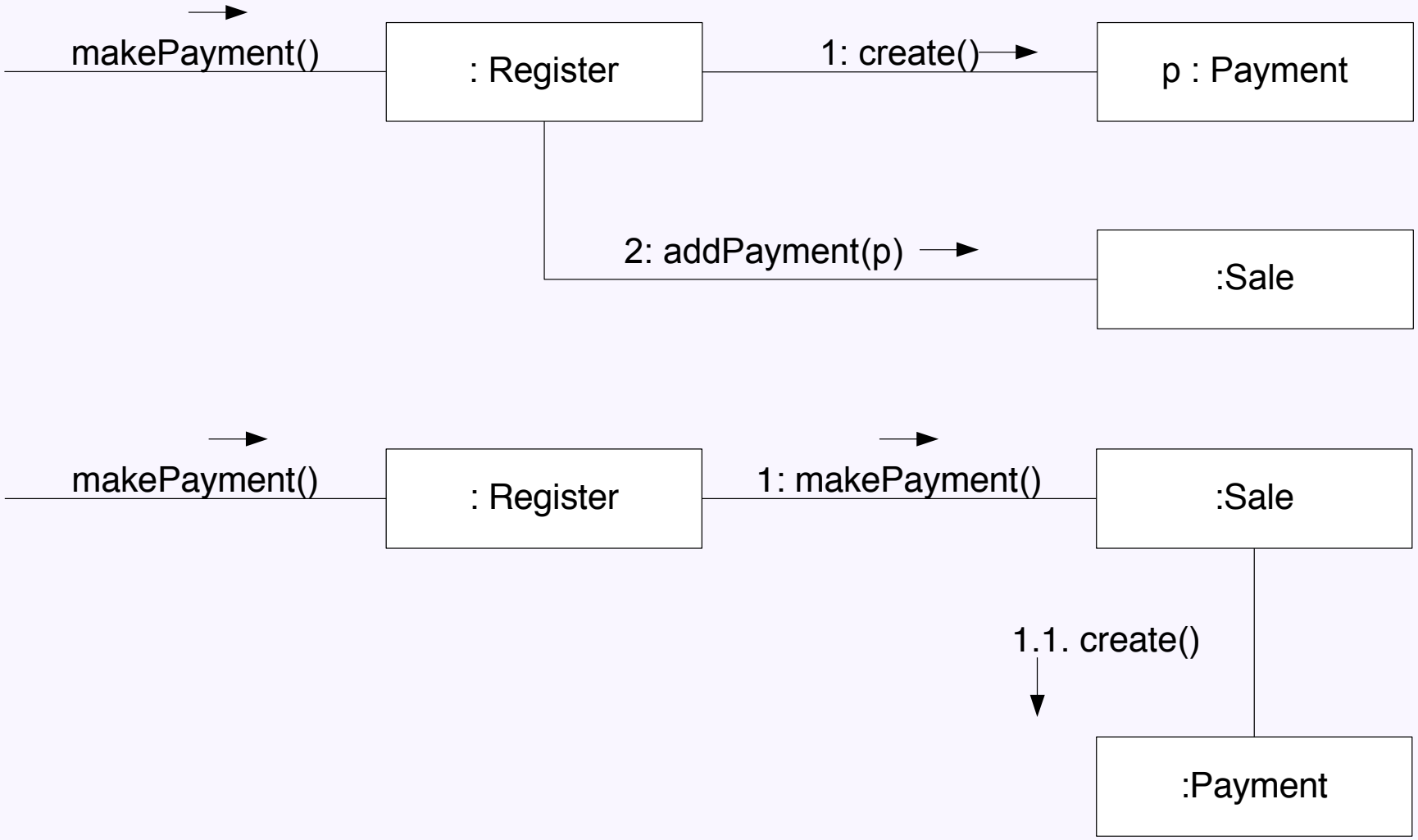
Example

- Create a Payment and associate it with the Sale.

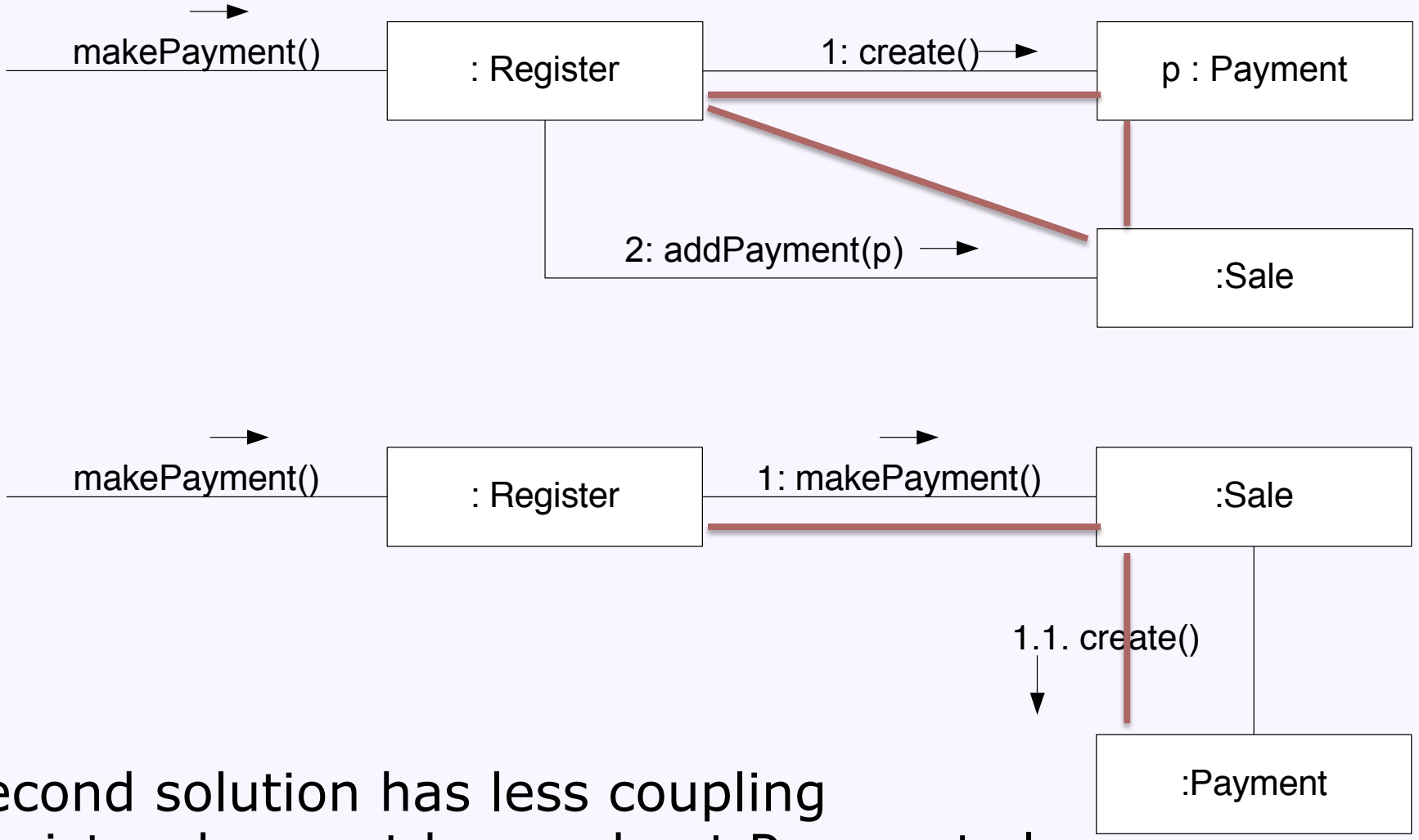
Example


makePayment()
: Register
1: create()
p : Payment
2: addPayment(p)
:Sale

Example


makePayment()
: Register
1: create()
p : Payment
2: addPayment(p)
:Sale
makePayment()
: Register
1: makePayment()
:Sale
1.1. create()
:Payment

Coupling



Second solution has less coupling
Register does not know about Payment class

Why High Coupling is undesirable

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements (classes, subsystems, …)
  - "too many" is context-dependent
- A class with high (or strong) coupling relies on many other classes.
  - Changes in related classes force local changes.
  - Such classes are harder to understand in isolation.
  - They are harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Low Coupling

- Benefits of making classes independent of other classes
  - changes are localised
  - easier to understand code
  - easier to reuse code

Common Forms of Coupling in OO Languages

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- TypeX has a method which references an instance of TypeY, or TypeY itself, by any means.
  - Typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.

Low Coupling: Discussion

- Low Coupling is a principle to keep in mind during all design decisions
- It is an underlying goal to continually consider.
- It is an evaluative principle that a designer applies while evaluating all design decisions.
- Low Coupling supports the design of classes that are more independent
  - reduces the impact of change.
- Can't be considered in isolation from other patterns such as Expert and High Cohesion
- Needs to be included as one of several design principles that influence a choice in assigning a responsibility.

Low Coupling: Discussion

- Subclassing produces a particularly problematic form of high coupling
  - Dependence on implementation details of superclass
  - -> Prefer composition over inheritance
- Extremely low coupling may lead to a poor design
  - Few incohesive, bloated classes do all the work; all other classes are just data containers
- Contraindications: High coupling to very stable elements is usually not problematic

High Cohesion Principle

Problem:

How to keep complexity manageable.

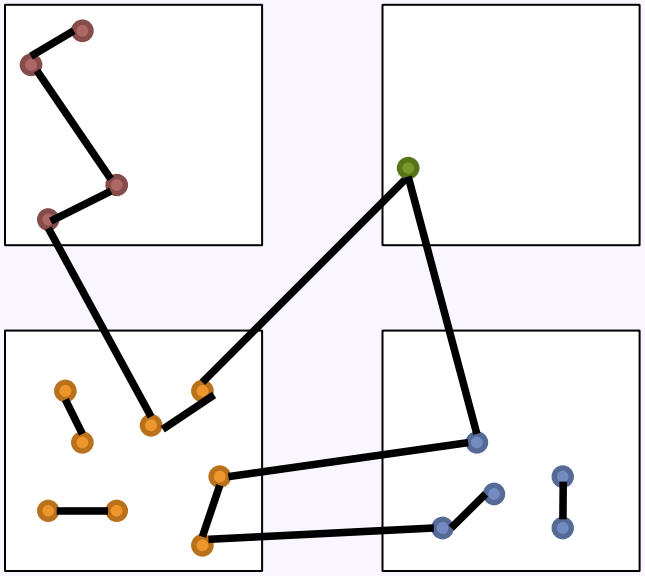
Solution:

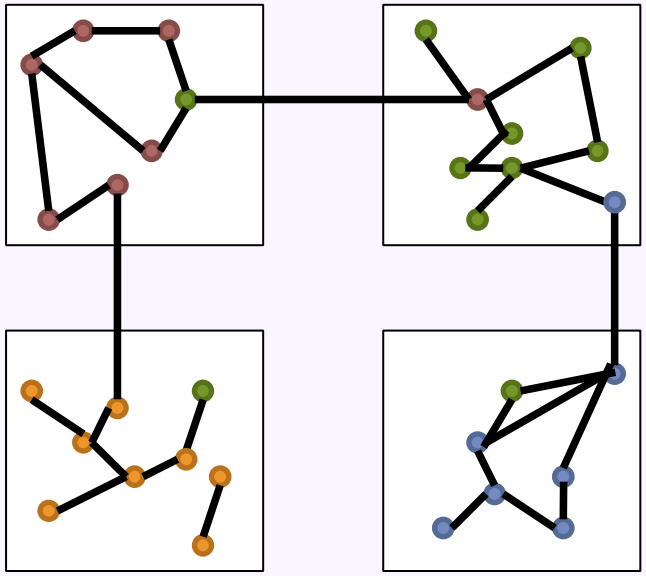
Assign responsibilities so that cohesion remains high.

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion

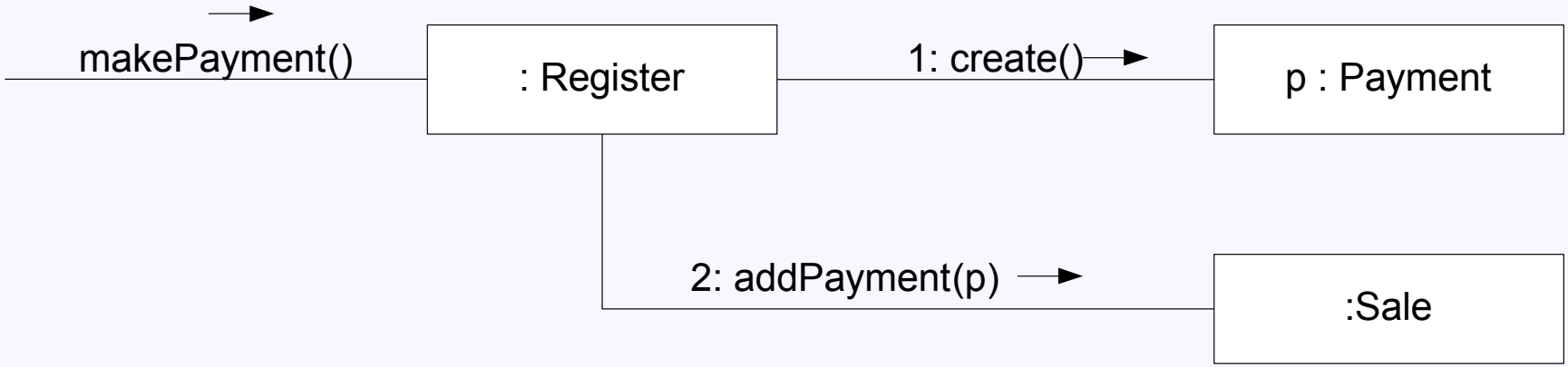
High cohesion





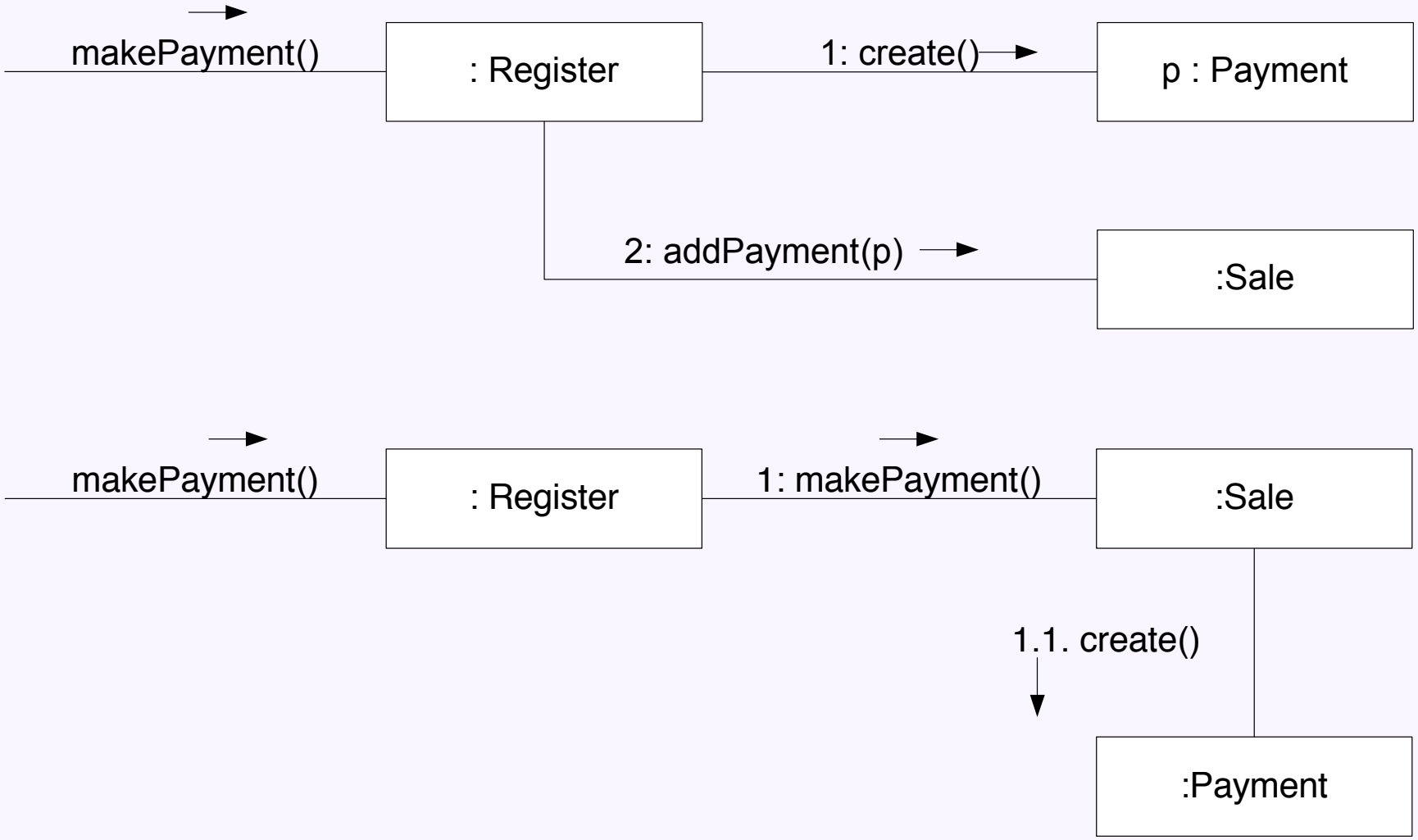
- Classes are easier to maintain
- Easier to understand
- Often support low coupling
- Supports reuse because of fine grained responsibility

Example



(except for cohension), looks OK if *makePayment* considered in isolation, but adding more system operations, *Register* would take on more and more responsibilities and become less cohesive.

Example


makePayment()
: Register
1: create()
p : Payment
2: addPayment(p)
:Sale
makePayment()
: Register
1: makePayment()
:Sale
1.1. create()
:Payment

Extra: isVisited

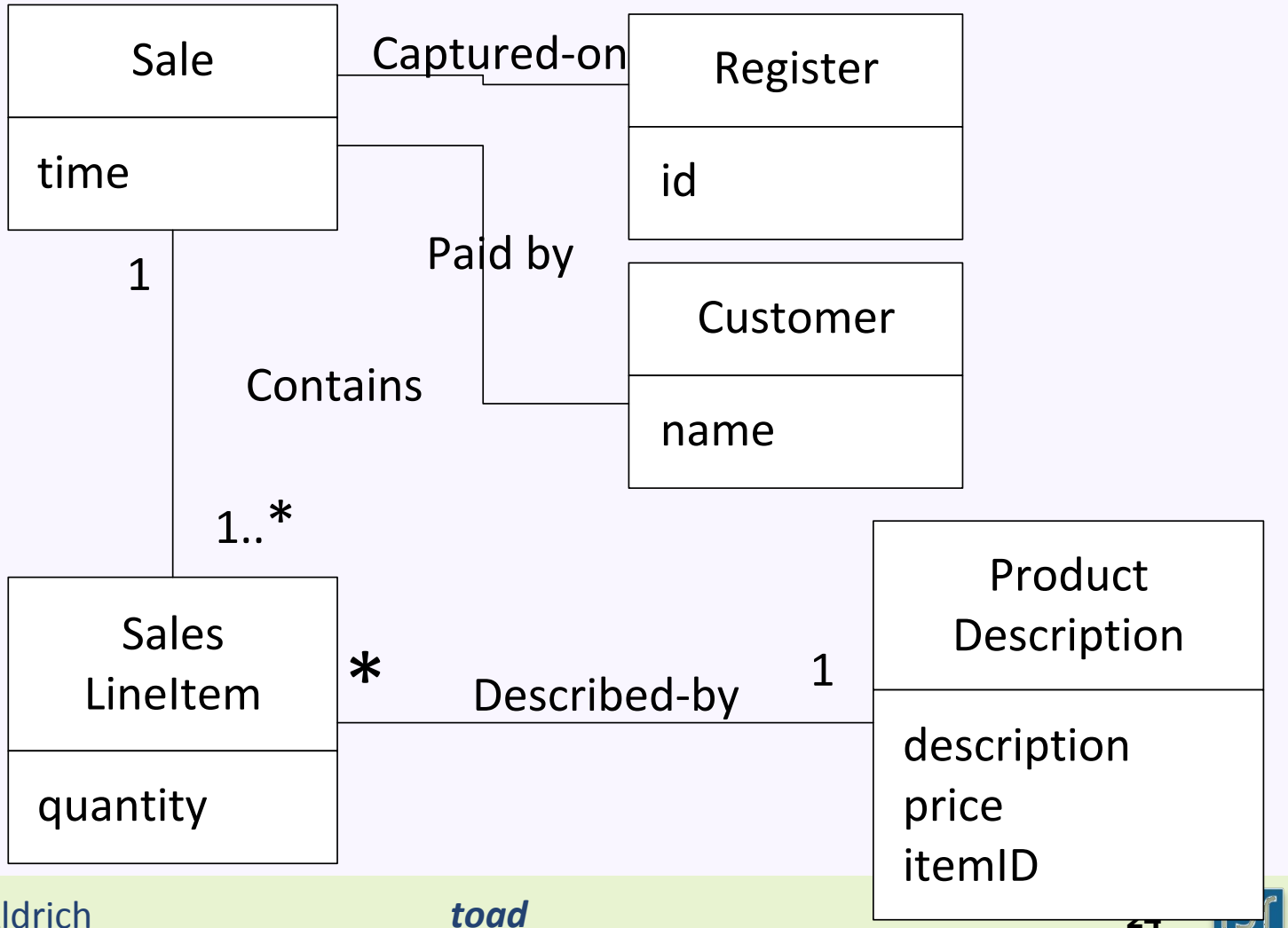
```
class Graph {
        Node[] nodes;
        boolean[] isVisited;
}
class Algorithm {
        int shortestPath(Graph g, Node n, Node m) {
                for (int i; …)
                        if (!g.isVisited[i]) {
                                …
                                g.isVisited[i] = true;
                        }
                }
                return v;
        }
}
```

High Cohesion: Discussion

- Scenarios:
  - Very Low Cohesion: A Class is solely responsible for many things in very different functional areas
  - Low Cohesion: A class has sole responsibility for a complex task in one functional area.
  - High Cohesion. A class has moderate responsibilities in one functional area and collaborates with other classes to fulfil tasks.
- Advantages:
  - Classes are easier to maintain
  - Easier to understand
  - Often support low coupling
  - Supports reuse because of fine grained responsibility
- Rule of thumb: a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work.

Information Expert Principle

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

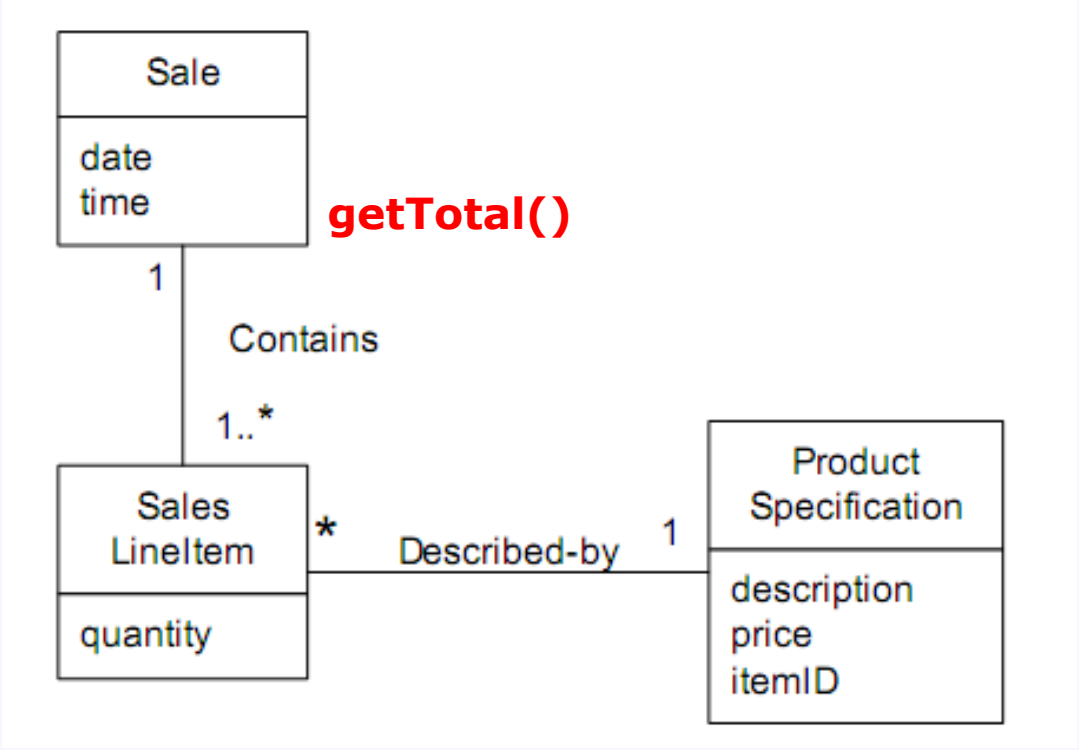


Information Expert

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert, **the class that has the information necessary to fulfill the responsibility**
- Start assigning responsibilities by clearly stating responsibilities!
- Typically follows common intuition
- Design Classes (Software Classes) instead of Conceptual Classes
  - If Design Classes do not yet exist, look in Domain Model for fitting abstractions (-> low representational gap)

Information Expert

- What information is needed to determine the grand total?
  - Line items and the sum of their subtotals
- *Sale* is the information expert for this responsibility.

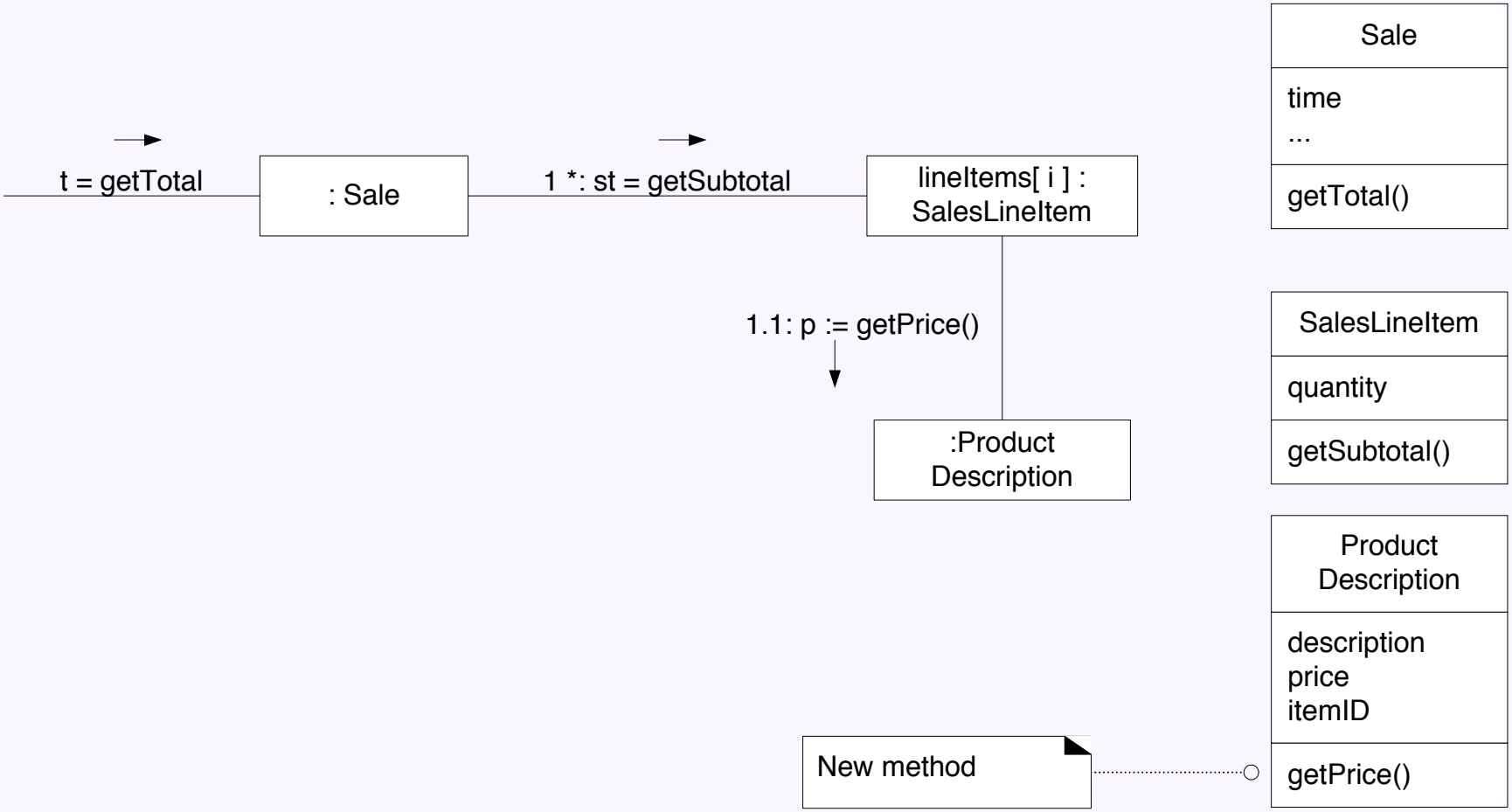


Information Expert

- To fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

Information Expert



Information Expert -> "Do It Myself Strategy"

- Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents
  - a sale does not tell you its total; it is an inanimate thing
- In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.
- They do things related to the information they know.

Information Expert: Discussion

- Contraindication: Conflict with separation of concerns
  - Example: Who is responsible for saving a sale in the database?
  - Adding this responsibility to Sale would distribute database logic over many classes → low cohesion
- Contraindication: Conflict with late binding
  - Late binding is available only for the receiver object
  - But maybe the variability of late binding is needed in some method argument instead
    - So make the argument the receiver instead
    - Example: use a strategy pattern to compute the total. Different strategies may capture special discounts, for example.

Creator Principle: Problem

- Who creates Nodes in a Graph?
- Who creates instances of SalesLineItem?
- Who creates Rabbit-Actors in a Game?
- Who creates Tiles in a Monopoly game?
  - AI? Player? Main class? Board? Meeple (Dog)?

Creator: Problem

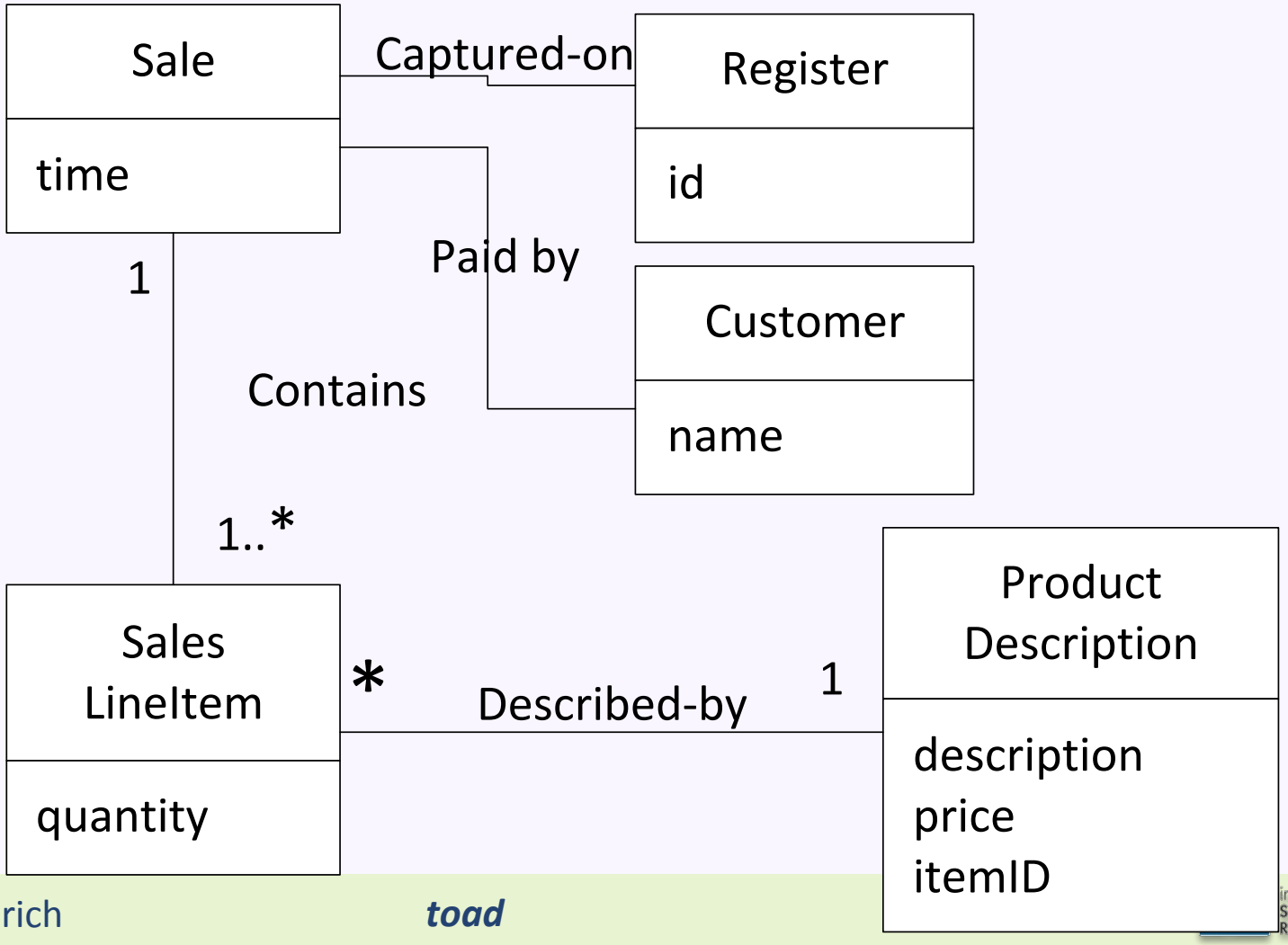
- Who creates Tiles in a Monopoly game?
  - Typical Answer: The board
  - Container creates things contained

Creator Principle

- Assign class B responsibility of creating instance of class A if
  - B aggregates A objects
  - B contains A objects
  - B records instances of A objects
  - B closely uses A objects
  - B has the initializing data for creating A objects
- where there is a choice, prefer
  - B aggregates or contains A objects
- Key idea: Creator needs to keep reference anyway and will frequently use the created object

Creator : Example

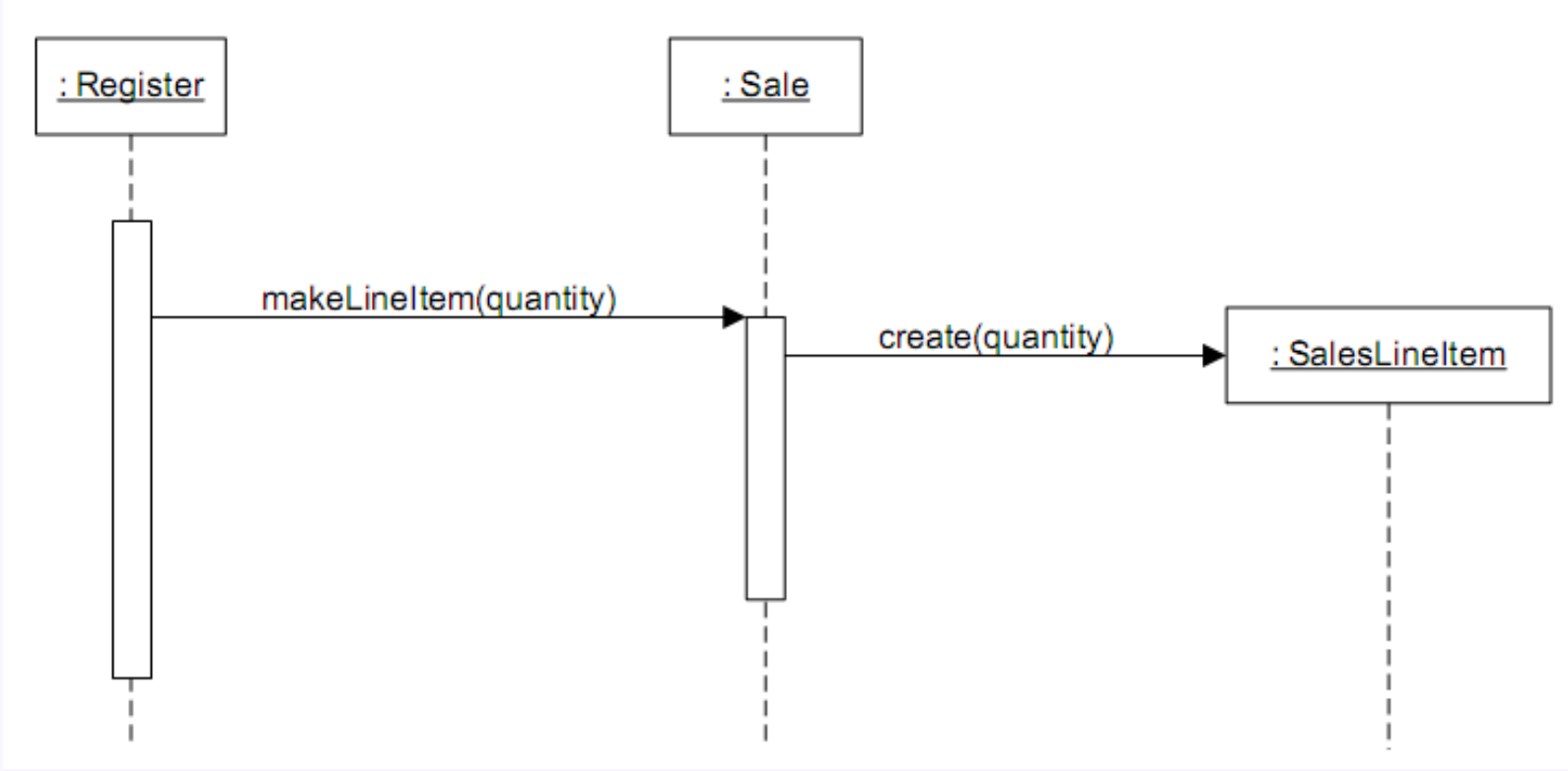
- Who is responsible for creating SalesLineItem objects?



Creator : Example

- Creator pattern suggests Sale.

- Sequence diagram is



Creator: Discussion

- Promotes low coupling by making instances of a class responsible for creating objects they need to reference
- By creating the objects themselves, they avoid being dependent on another class to create the object for them
- Contraindications:
  - creation may require significant complexity, such as
    - using recycled instances for performance reasons
    - conditionally creating an instance from one of a family of similar classes based upon some external property value
    - Sometimes desired to outsource object wiring ("dependency injection")

Controller Principle

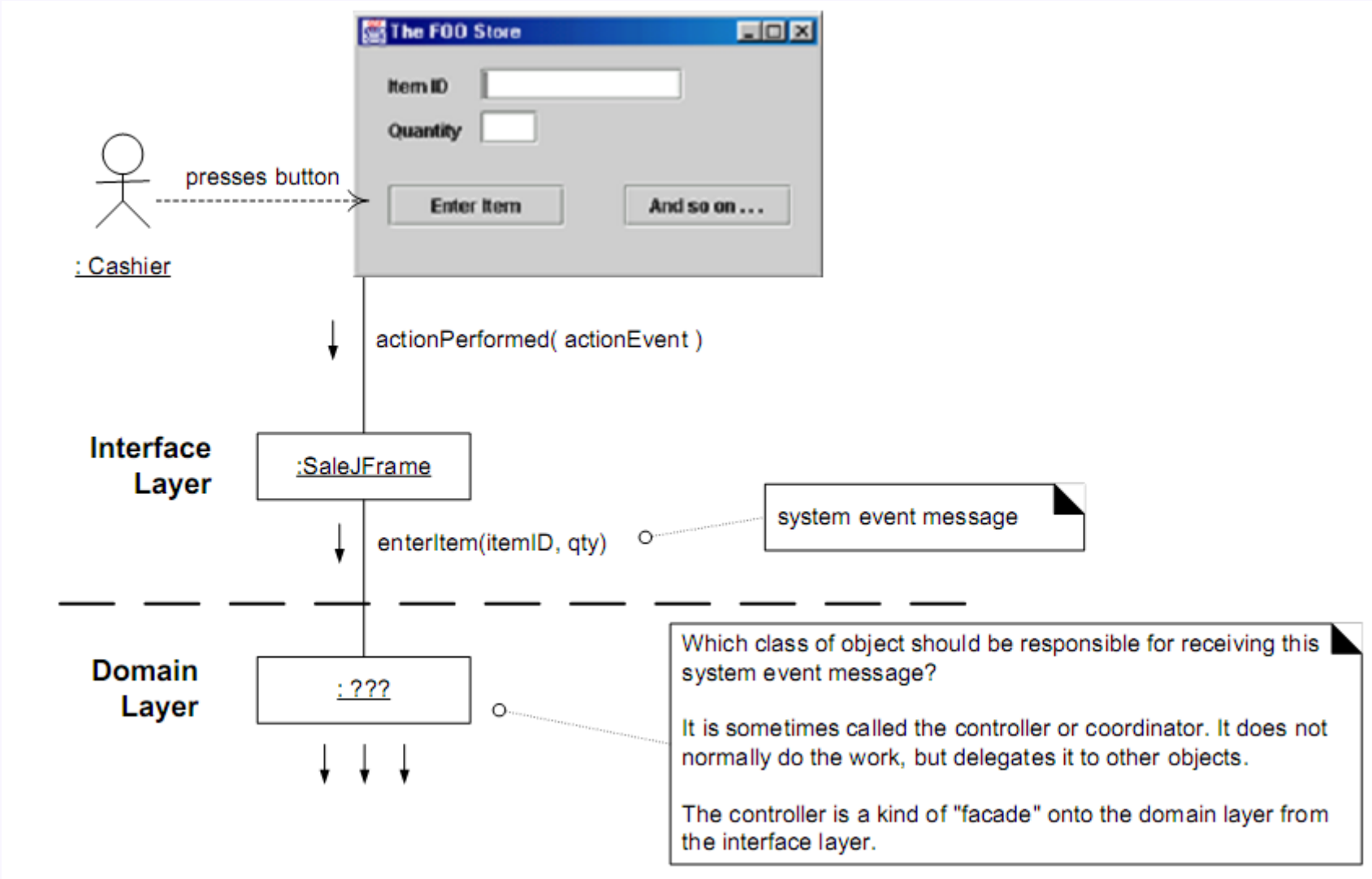
Problem:

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 the overall system, device, or subsystem (facade controller) or a use case scenario within which the system event occurs (use case controller)

Controller: Example



Controller: Example

- By the Controller pattern, here are some choices:
- *Register, POSSystem*: represents the overall "system," device, or subsystem
- *ProcessSaleSession, ProcessSaleHandler*: represents a receiver or handler of all system events of a use case scenario

Controller: Discussion

- Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.
- Facade controllers are suitable when there are not "too many" system events
- A use case controller is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling
  - typically when the facade controller is becoming "bloated" with excessive responsibilities.

Controller: Discussion

- Benefits
  - Increased potential for reuse, and pluggable interfaces
    - No application logic in the GUI
  - Dedicated place to place state that belongs to some use case
    - E.g. operations must be performed in a specific order
- Avoid bloated controllers!
  - E.g. single controller for the whole system, low cohesion, lots of state in controller
  - Split into use case controllers, if applicable
- Interface layer does not handle system events

Resulting Design Model (example, excerpt)