Principles of Software Construction: Objects, Design, and Concurrency

Assigning Responsibilities to Objects

toad

Fall 2014

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Key concepts from Thursday

Requirements and Design Overview

- Requirements Engineering
  - Requirements Elicitation *(see 15-313)*
    - Functional Requirements (often as *Use Cases*)
    - Quality Attributes (often as *Quality Attribute Scenarios*)
  - (Object-Oriented) Requirements Analysis
    - Domain Modeling
  - System Specification
    - System Sequence Diagrams
    - Behavioral Contracts
- (Object-Oriented) Software Design
  - Architectural Design *(mostly in 15-313)*
  - Responsibility Assignment
    - Object sequence diagrams
    - Object model (class diagrams)
    - GRASP heuristics for assigning responsibilities
  - Method specifications / code contracts

Today's Lecture: Learning Goals

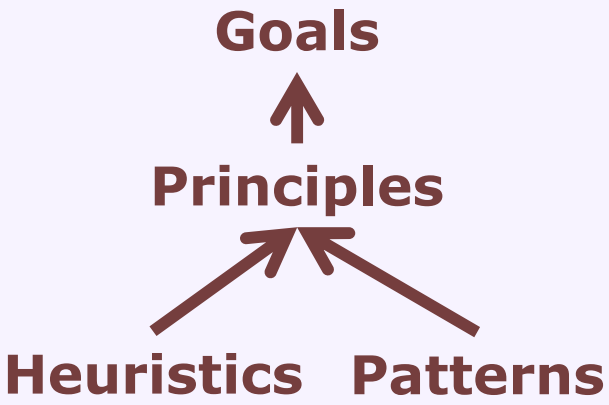
- Review high-level design goals
- Understand design principles such as coupling, cohesion, and correspondence, and how these support design goals
- Understand how to apply GRASP guidelines such as Creator, Expert, and Controller to promote these design principles
- Introduce the idea of design patterns by examining the Decorator pattern

Software Design Goals

- For any program specification, multiple programs fulfill it
  - What are the differences between the programs?
  - Which program should we choose?

- Of course, we usually synthesize a program, not choose it
  - How can we design a program with the right properties?

Goals, Principles, Guidelines



- Design Goals
  - Desired quality attributes of software
  - Driven by cost/benefit economics
  - Examples: Evolvability, separate development, reuse, performance, robustness, …
- Design Principles
  - Guidelines for designing software
  - Support one or more design goals
  - Examples: Low coupling, high cohesion, high correspondence, …
- Design Heuristics
  - Rules of thumb for low-level design decisions
  - Promote design principles, and ultimately design goals
  - Example: Creator, Expert, Controller
- Design Patterns
  - General solutions to recurring design problems
  - Promote design goals, but may add complexity or involve tradeoffs
  - Examples: Composite, Decorator, Strategy
- Goals, principles, heuristics, patterns may conflict
  - Use high-level goals of project to resolve

Principles for Assigning Responsibilities: Coupling

- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

Design Principle: Coupling

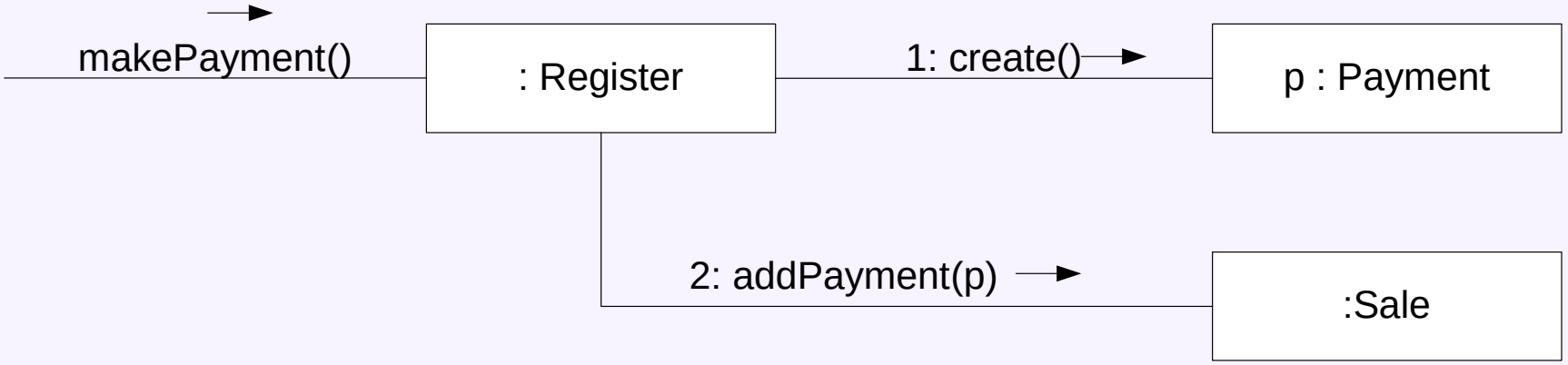
A module should depend on as few other modules as possible

- Enhances understandability → evolvability
  - Little context necessary to make changes
- Reduces the cost of change → evolvability
  - When a module interface changes, few modules are affected
- Enhances reuse
  - Fewer dependencies, easier to adapt to a new context

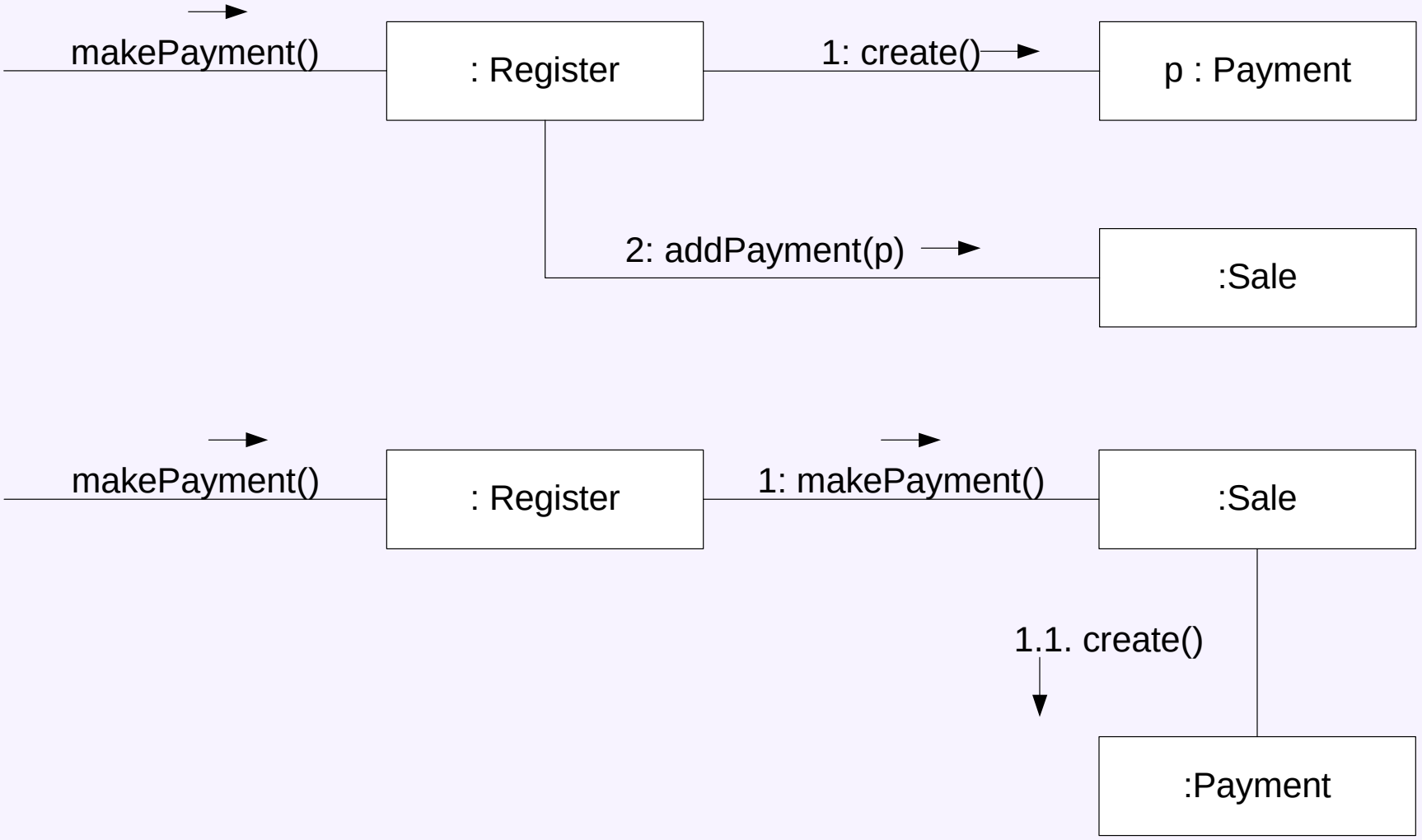
Coupling Example

- Create a Payment and associate it with the Sale.

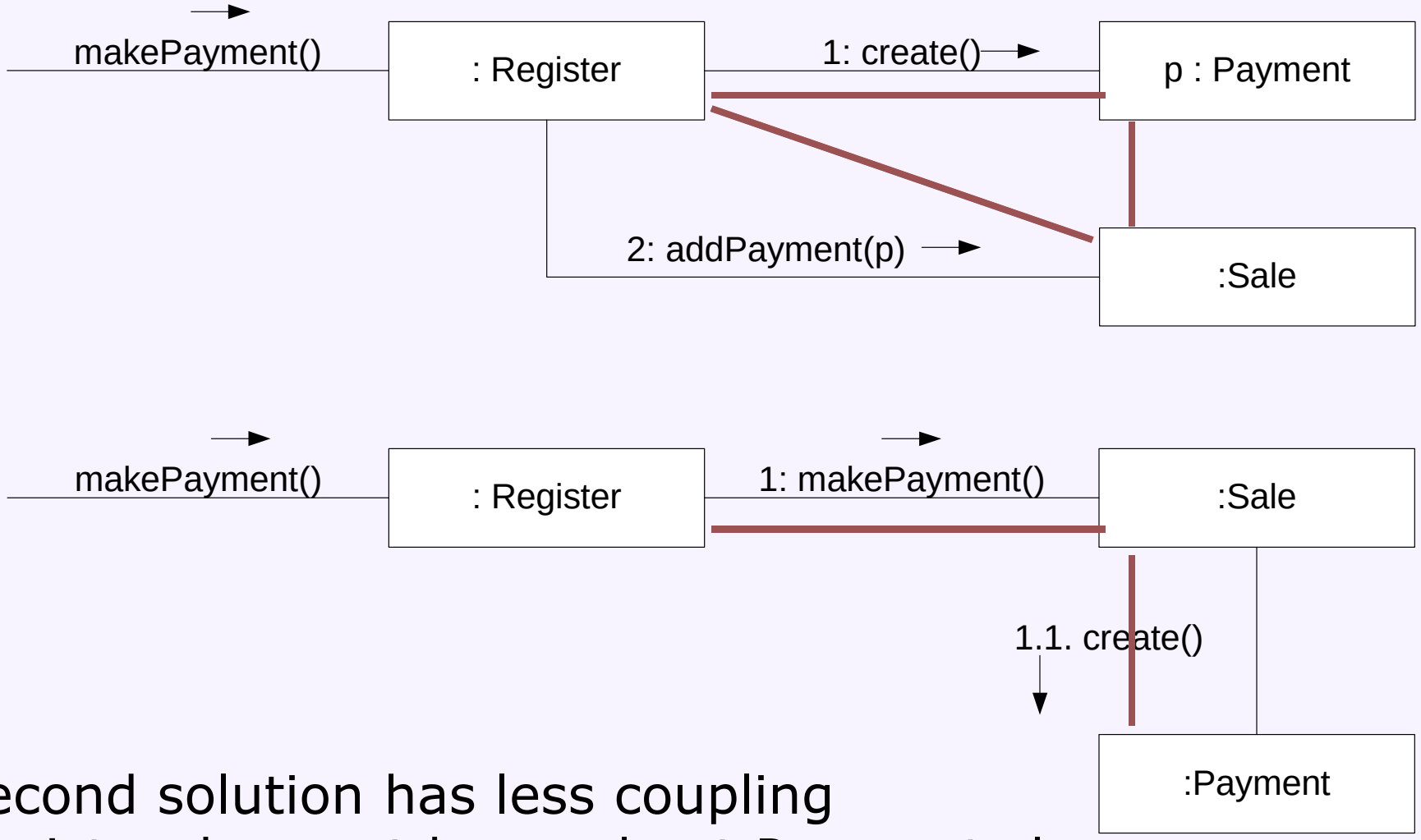
Coupling Example



Coupling Example



Coupling Example



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

Common Forms of Coupling in OO Languages

- Type X has a field of type Y
- Method m in type X refers to type Y
  - e.g. a method argument, return value, local variable, or static method call
- Type X is a direct or indirect subclass of Type Y
- Type Y is an interface, and Type X implements that interface

Coupling: Discussion

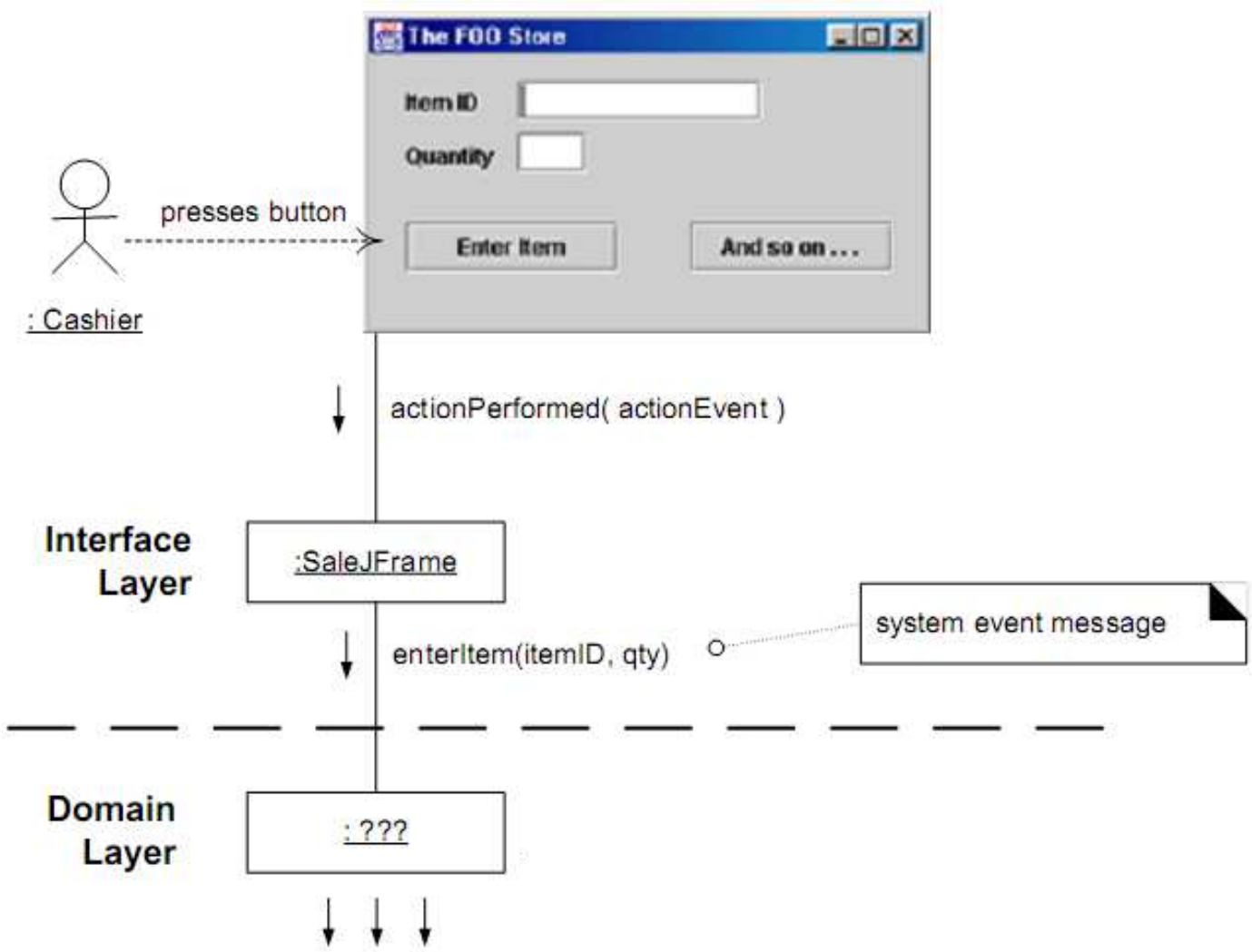
- Subclass/superclass coupling is particularly strong
  - protected fields and methods are visible
  - subclass is fragile to many superclass changes
    - e.g. change in method signatures, added abstract methods
  - Guideline: prefer composition to inheritance, to reduce coupling
- Not all coupling is equal
  - Are you coupled to a stable interface?
    - A stable interface is unlikely to change, and likely well-understood
    - Therefore this coupling carries little evolveability cost
- Coupling is one principle among many
  - Consider cohesion, correspondence, and other principles
  - Extreme low coupling → one class does everything → poor cohesion (discussed later!)

Design Heuristics: Controller

- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

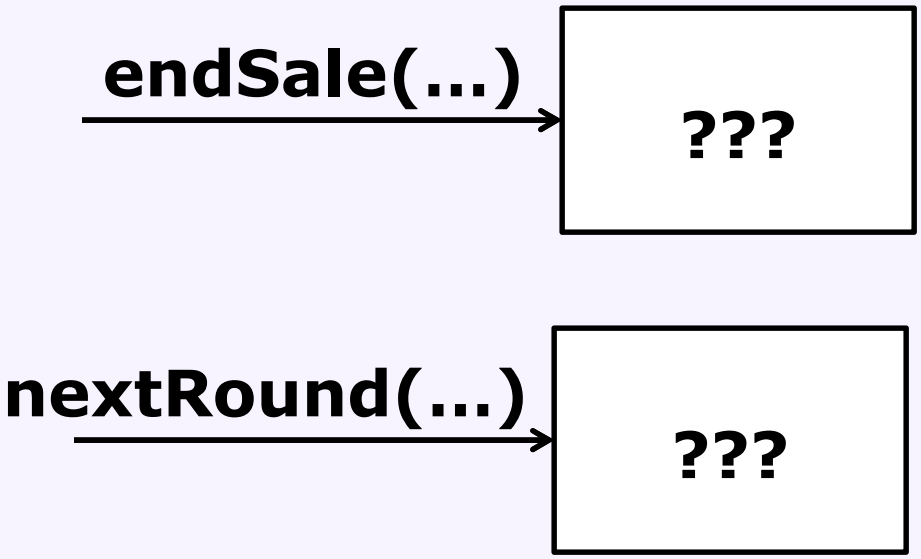
Controller (GRASP heuristic 1)

- What first object receives and coordinates a system operation (event)?



Controller (GRASP heuristic 1)

- What first object receives and coordinates a system operation (event)?
  - a user clicking on a button
  - a network request arriving
  - a database connection dropped

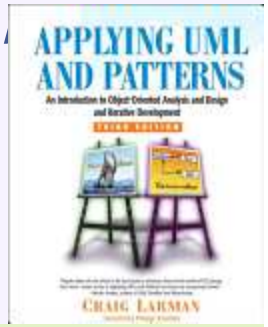


Controller (GRASP heuristic 1)

- Problem: What object receives and coordinates a system operation (event)?
- Solution: Assign the responsibility to an object representing
  - the **overall system**, device, or subsystem (*façade controller*), or
  - a use case **scenario** within which the system event occurs (*use case controller*)

- Controller is a GRASP heuristic
  - General Responsibility Assignment Software Patterns
    - "pattern" is a misnomer here – they are more heuristics than patterns

- Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  - Chapter 16+17+22 introduce GRASP



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 operations in a use case scenario

Controller: Discussion

- A Controller is a coordinator
  - does not do much work itself
  - delegates to other objects
- Façade controllers suitable when not "too many" system events
  - -> one overall controller for the system
- Use case controller suitable when façade controller "bloated" with excessive responsibilities (low cohesion, high coupling)
  - -> several smaller controllers for specific tasks
- Closely related to Façade design pattern (*future lecture*)

Controller: Discussion of Design Goals/Strategies

- Decrease coupling
  - User interface and domain logic are decoupled from each other
    - Understandability: can understand these in isolation, leading to:
    - Evolvability: both the UI and domain logic are easier to change
  - Both are coupled to the controller, which serves as a *mediator*
    - This coupling is less harmful
      - The controller is a smaller and more stable interface
      - Changes to the domain logic affect the controller, not the UI
      - The UI can be changed without knowing the domain logic design
- Support reuse
  - Controller serves as an interface to the domain logic
  - Smaller, explicit interfaces support evolvability
- But, bloated controllers increase **coupling** and decrease **cohesion**; split if applicable

Principles for Assigning Responsibilities: Cohesion

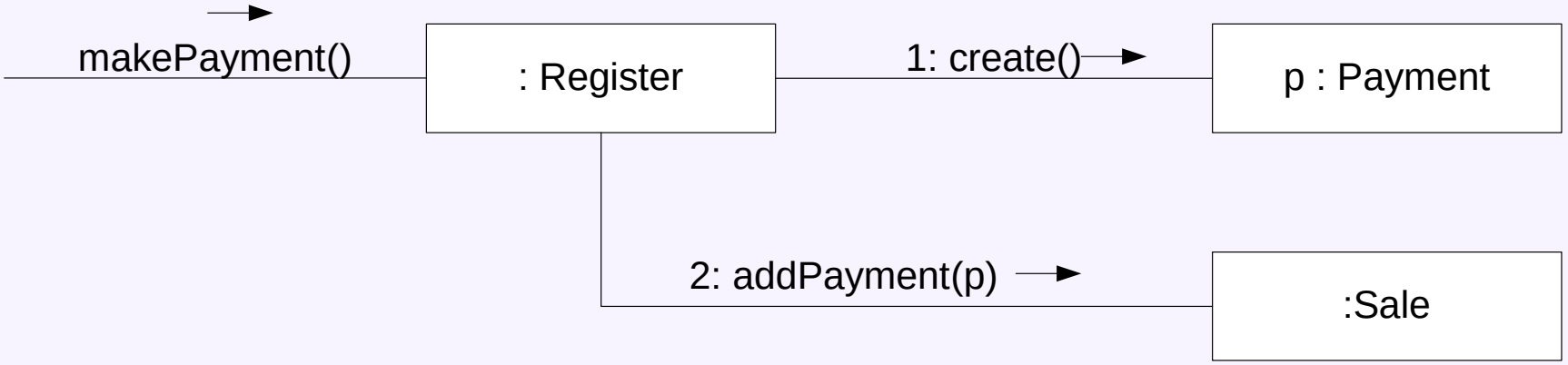
- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

Design Principle: Cohesion

A module should have a small set of related responsibilities

- Enhances understandability → evolvability
  - A small set of responsibilities is easier to understand
- Enhances reuse
  - A cohesive set of responsibilities is more likely to recur in another application

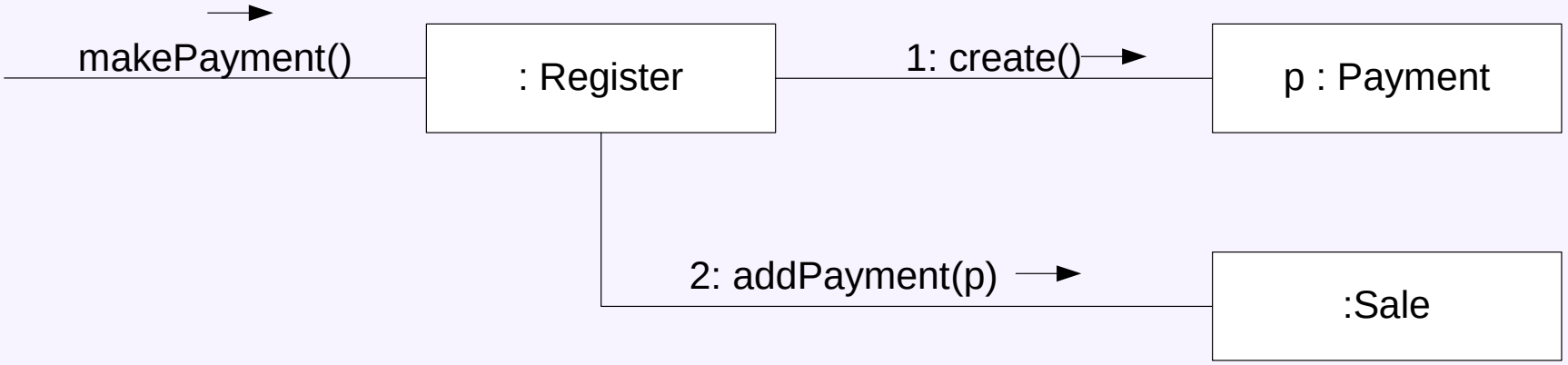
Cohesion Example



Register responsibilities

- Accept makePayment event from UI
- Coordinate payment among domain objects

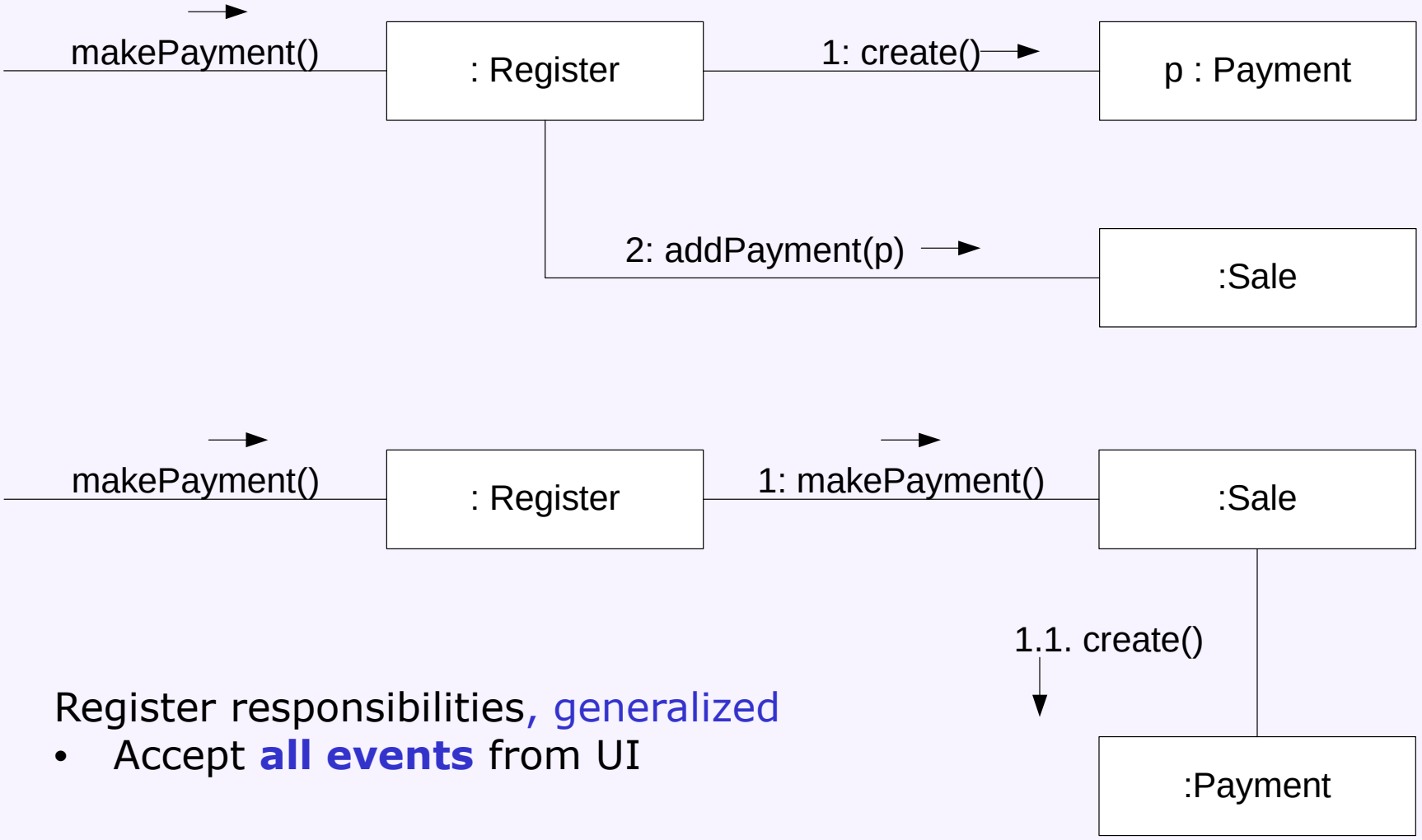
Cohesion Example



Register responsibilities, generalized

- Accept **all events** from UI
- ~~Coordinate **all interactions** among domain objects~~

Cohesion Example



Register responsibilities, generalized

- Accept **all events** from UI

Cohesion in Graph Implementations

```
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;
        }
}
```

Graph is tasked with not just data, but also algorithmic responsibilities

Monopoly Example

```
class Player {
        Board board;
        /* in code somewhere… */ getSquare(n);
        Square getSquare(String name) {
                for (Square s: board.getSquares())
                        if (s.getName().equals(name))
                                return s;
                return null;
}}
```

Which design has higher cohesion?

```
class Player {
        Board board;
        /* in code somewhere… */ board.getSquare(n);
}
class Board{
        List<Square> squares;
        Square getSquare(String name) {
                for (Square s: squares)
                        if (s.getName().equals(name))
                                return s;
                return null;
}}
```

Principles for Assigning Responsibilities: Correspondence

- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

Design Principle: Correspondence

Correspondence: align object model with domain model

- Class names, attributes, associations
- Also called **low representational gap**

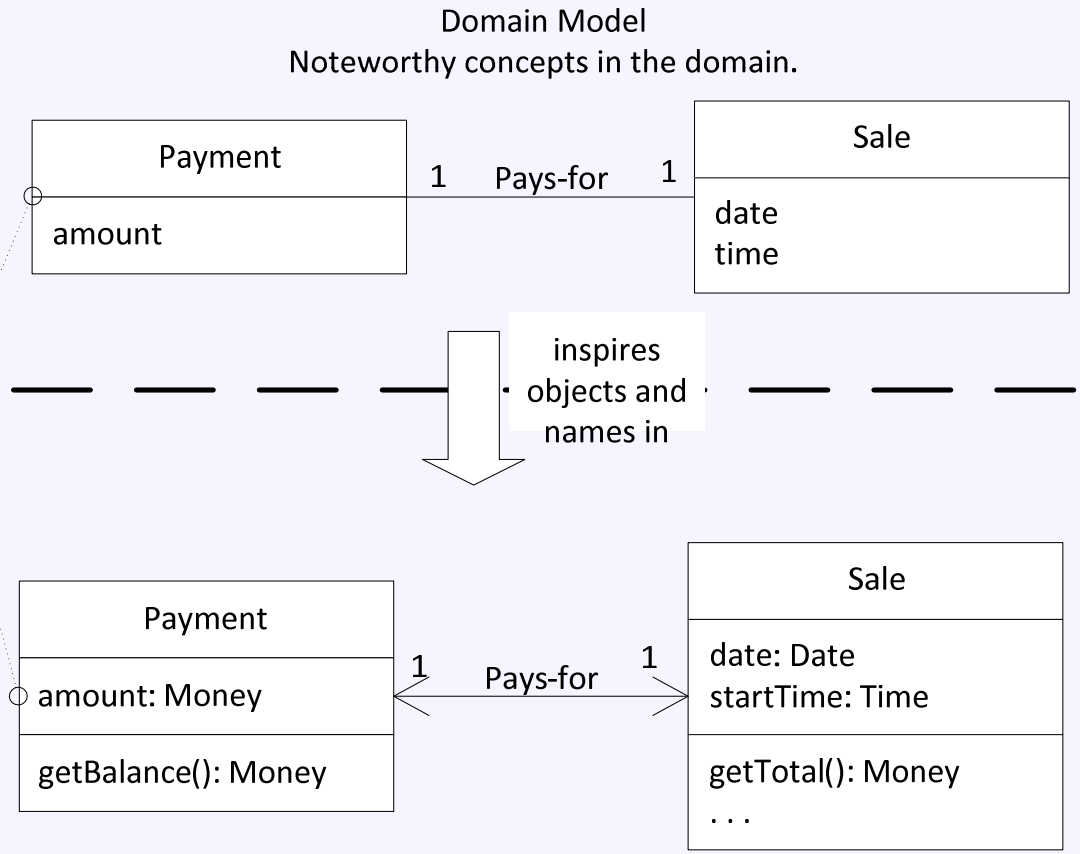
- Enhances evolvability
  - Small changes in domain model yield small changes in code
    - The best we could hope to do!
  - Code is more understandable: knowledge of domain carries over
- Enhances separate development
  - Experts in different domain concepts can work on different code

Correspondence/Low Representational Gap

A Payment in the Domain Model is a concept, but a Payment in the Object Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



The object-oriented developer has taken inspiration from the real world domain in creating software classes.

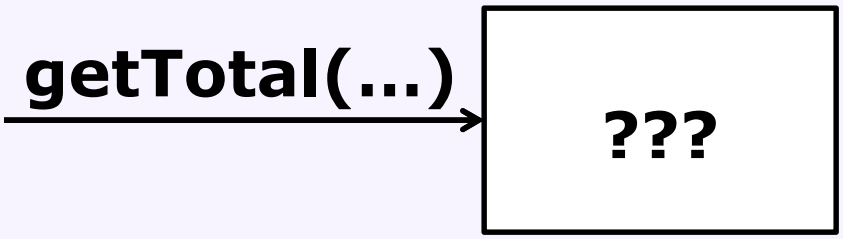
Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Design Heuristics: Expert

- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

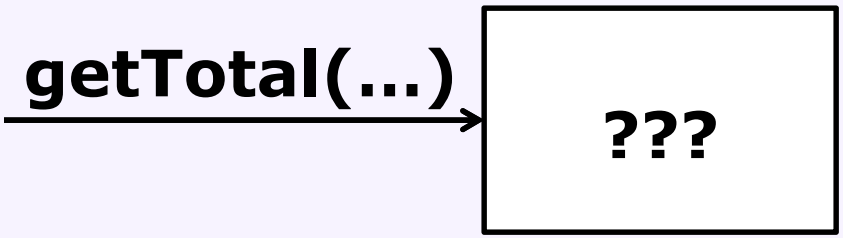
Information Expert (GRASP heuristic 2)

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



Information Expert (GRASP heuristic 2)

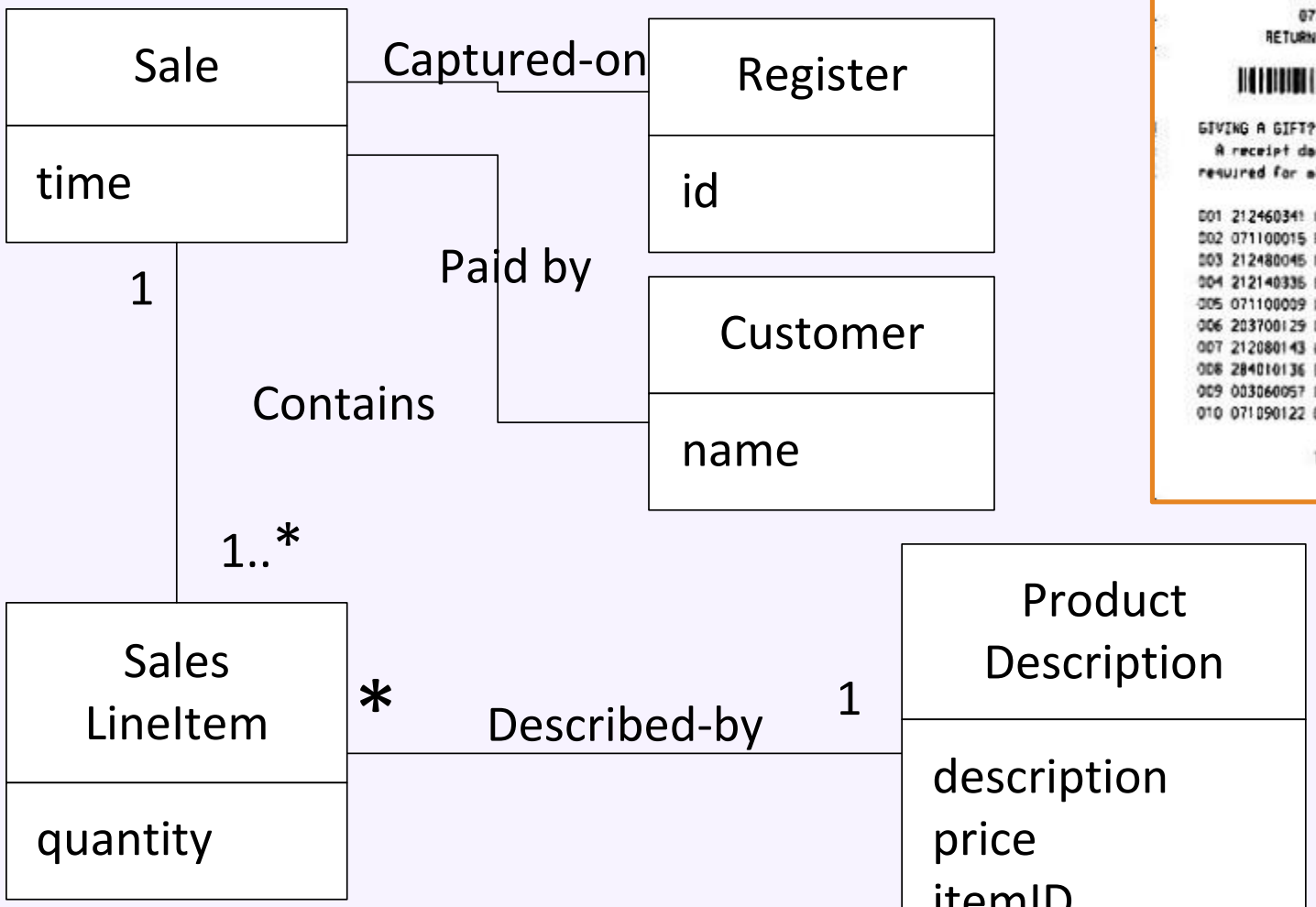
- Who should be responsible for **knowing** the grand total of a sale?



Register
Sale
LineItem
Product Decr.

Information Expert (GRASP heuristic 2)

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

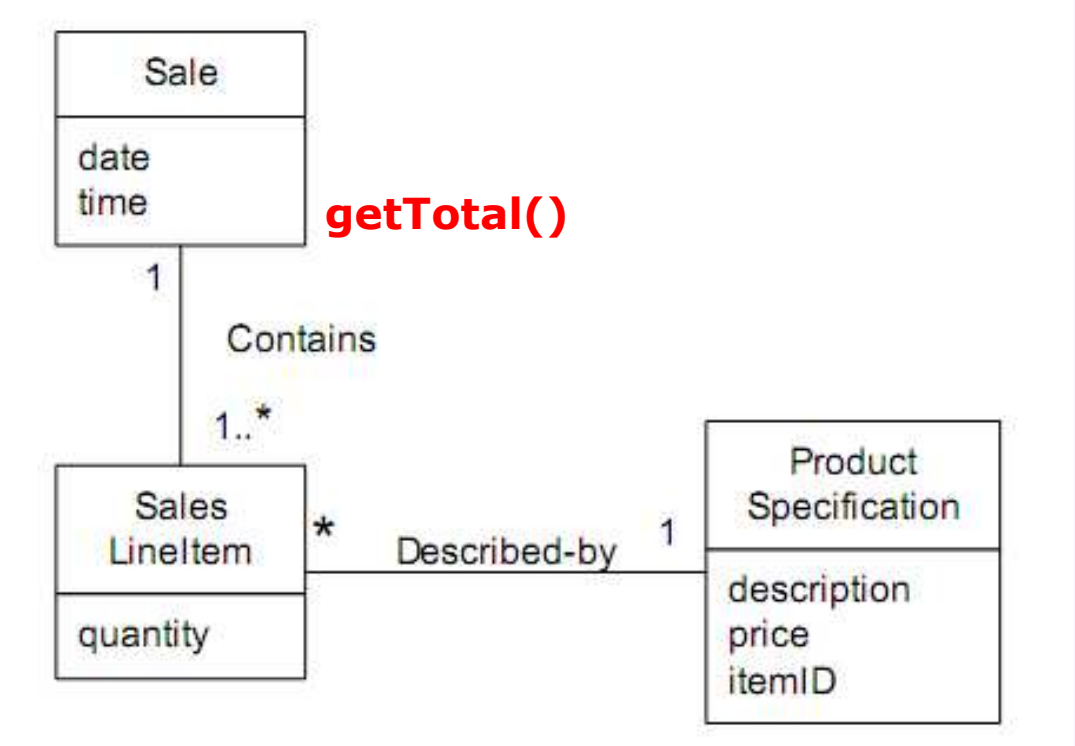


Information Expert

- Heuristic**: Assign a responsibility to the class that has the information necessary to fulfill the responsibility**
- Start assigning responsibilities by clearly stating responsibilities!
- Typically follows common intuition
- Software classes instead of Domain Model classes
  - If software classes do not yet exist, look in Domain Model for fitting abstractions (-> correspondence)

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.

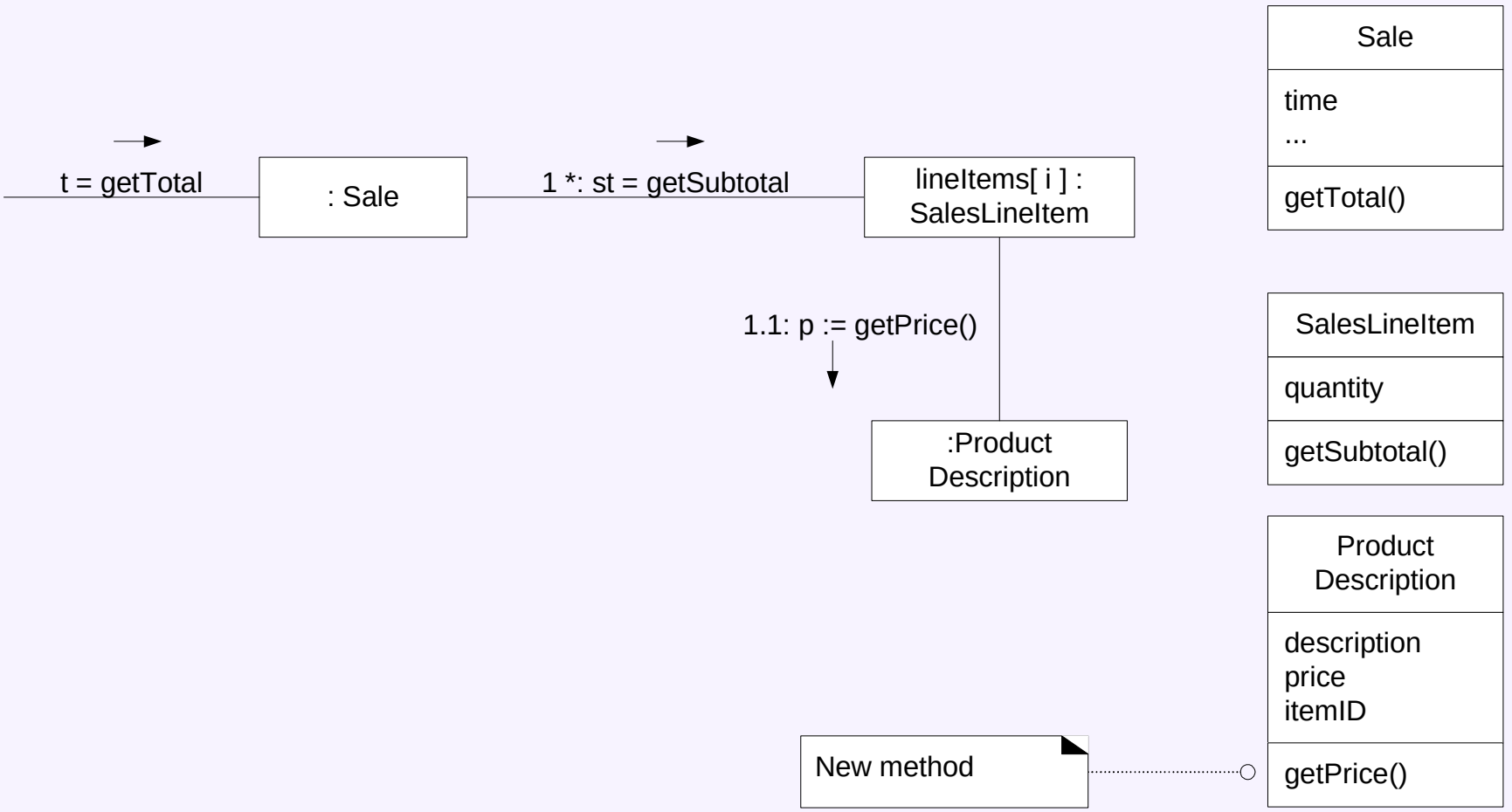


Information Expert

- 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 of Design Goals/Strategies

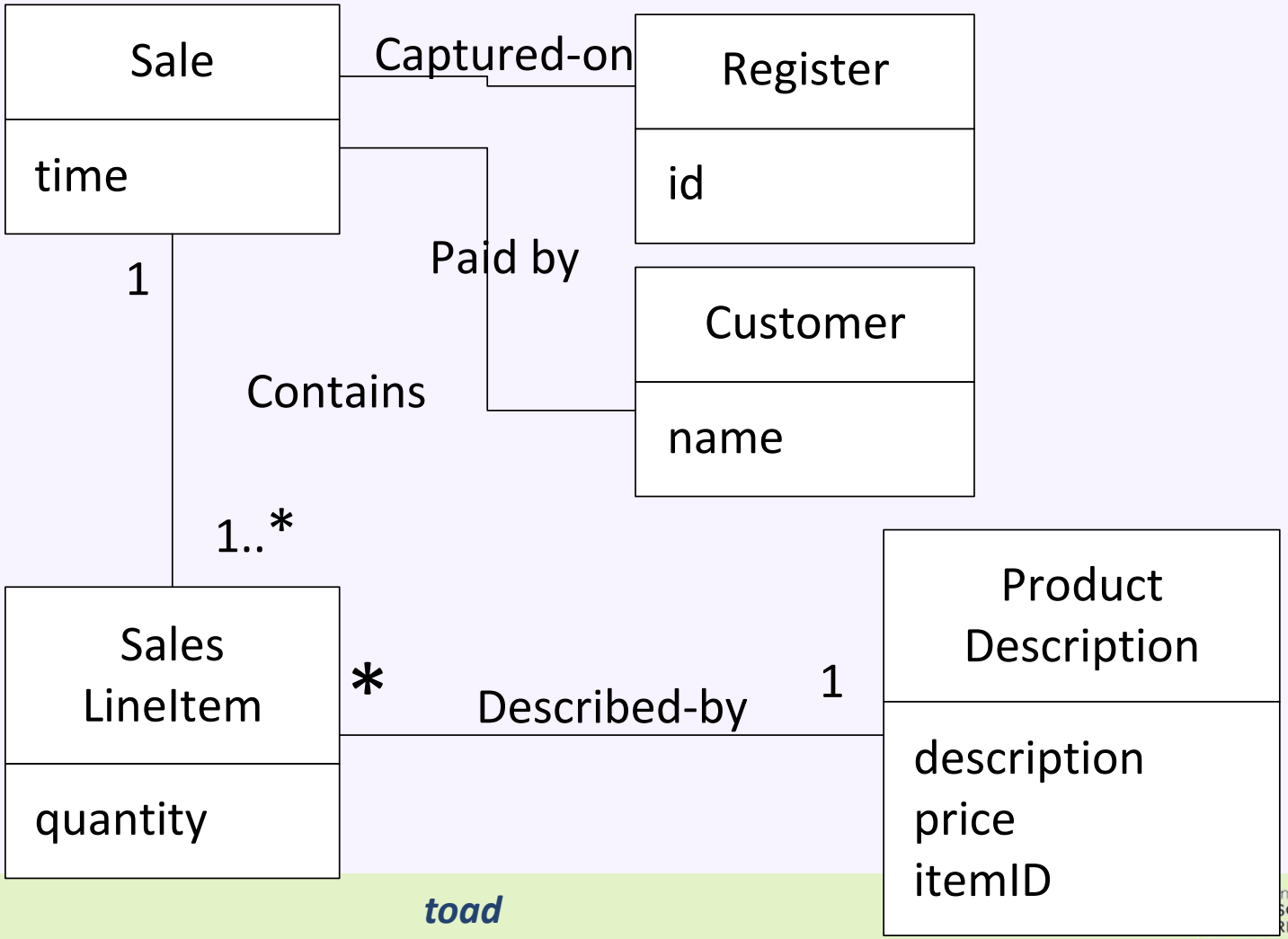
- Cohesion → understandability → evolvability
  - Code lives with the data it manipulates
- Low coupling → evolvability
  - Client does not need to know about LineItem and ProductDescription → they can change independently
  - Client does not know how total is computed → that can change
- Correspondence → evolvability
  - Total is associated with a sale in the real world → coordinated changes to the sale and total computation are easily managed
- May conflict with cohesion
  - 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

Design Heuristics: Creator

- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

Creator (GRASP heuristic 3)

- Who is responsible for **creating** SalesLineItem objects?



Creator Pattern (GRASP heuristic 3)

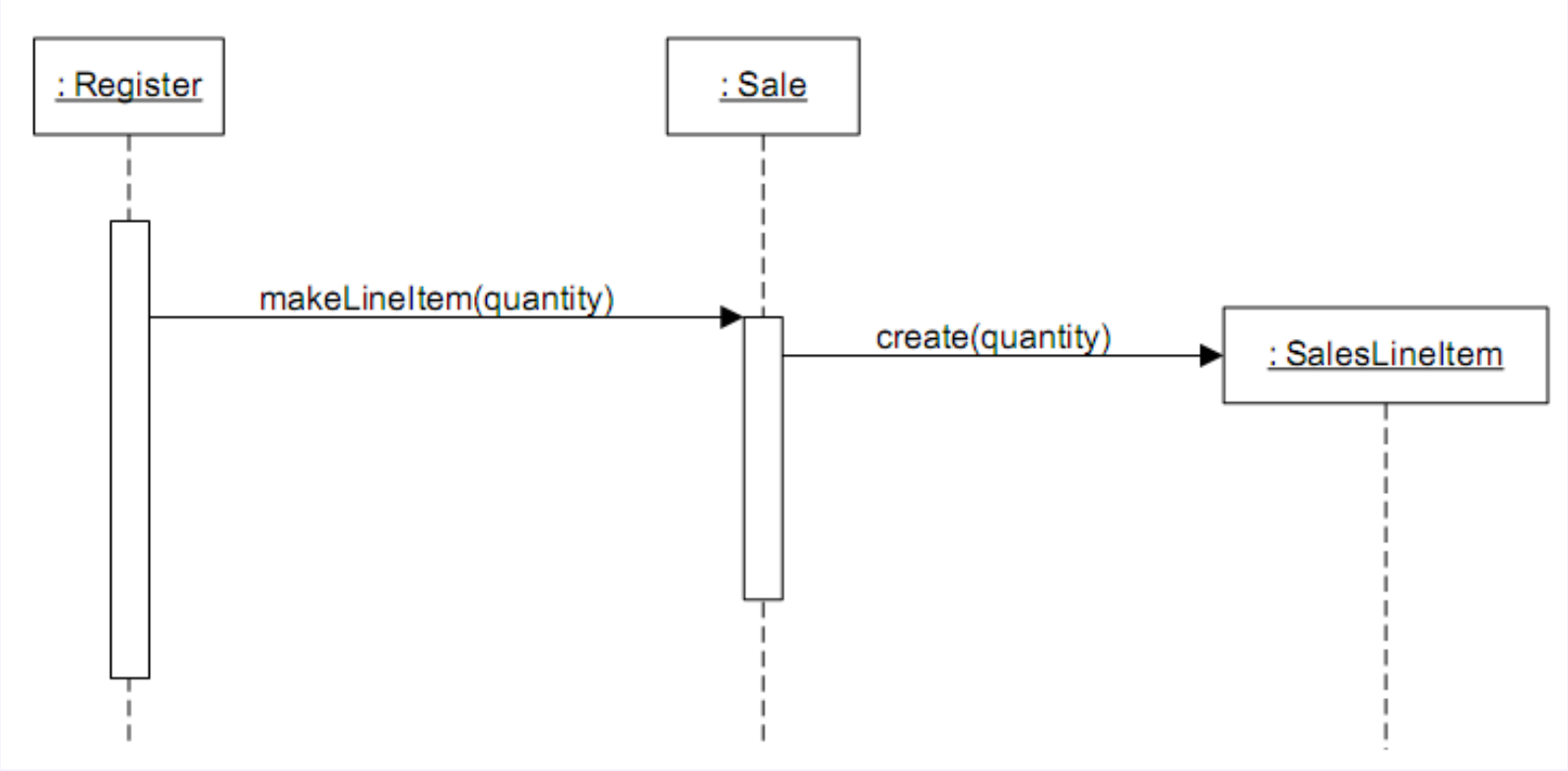
- Problem: Assigning responsibilities for creating objects
  - Who creates Nodes in a Graph?
  - Who creates instances of SalesItem?
  - Who creates Children in a simulation?
  - Who creates Tiles in a Monopoly game?
    - AI? Player? Main class? Board? Meeple (Dog)?

Creator Pattern

- Problem: Who creates an A?
- Solution: **Assign class responsibility of creating instance of class A to B 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
- the more the better; 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 pattern suggests Sale
- Interaction diagram:



Creator: Discussion of Design Goals/Strategy

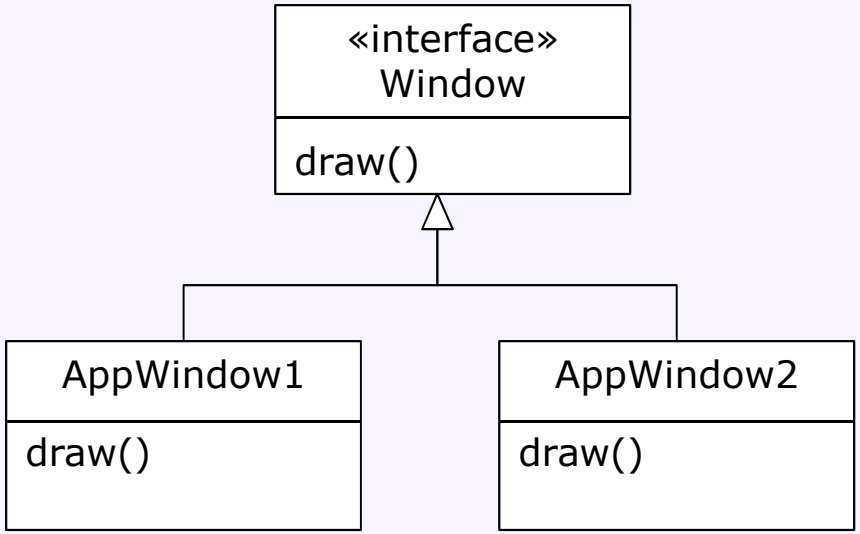
- Promotes **low coupling**, **high cohesion**
  - class responsible for creating objects it needs to reference
  - creating the objects themselves avoids depending on another class to create the object
- Promotes **evolvability**
  - Object creation is hidden, can be replaced locally
- Contra: sometimes objects must be created in special ways
  - complex initialization
  - instantiate different classes in different circumstances
  - *then **cohesion** suggests putting creation in a different object*
    - see *design patterns* such as builder, factory method

Design Heuristics: Creator

- Design Principles: guidelines for software design
  - Coupling (low)
  - Cohesion (high)
  - Correspondence (high)
- Design Heuristics: rules of thumb
  - Controller
  - Expert
  - Creator
- Design Patterns: solutions to recurring problems
  - Decorator

Problem: Scrollable Windows

- Imagine you are building a GUI application
  - Various windows show different views
  - Some views are too big to fit on the screen – we need scrolling
- Design considerations
  - Cohesion: put scrolling functionality in its own class
  - Coupling: don't treat scrolling windows specially
    - Clients of Window generally shouldn't know which windows scroll
  - Reuse: reuse scrolling across multiple windows
- Preliminary design:

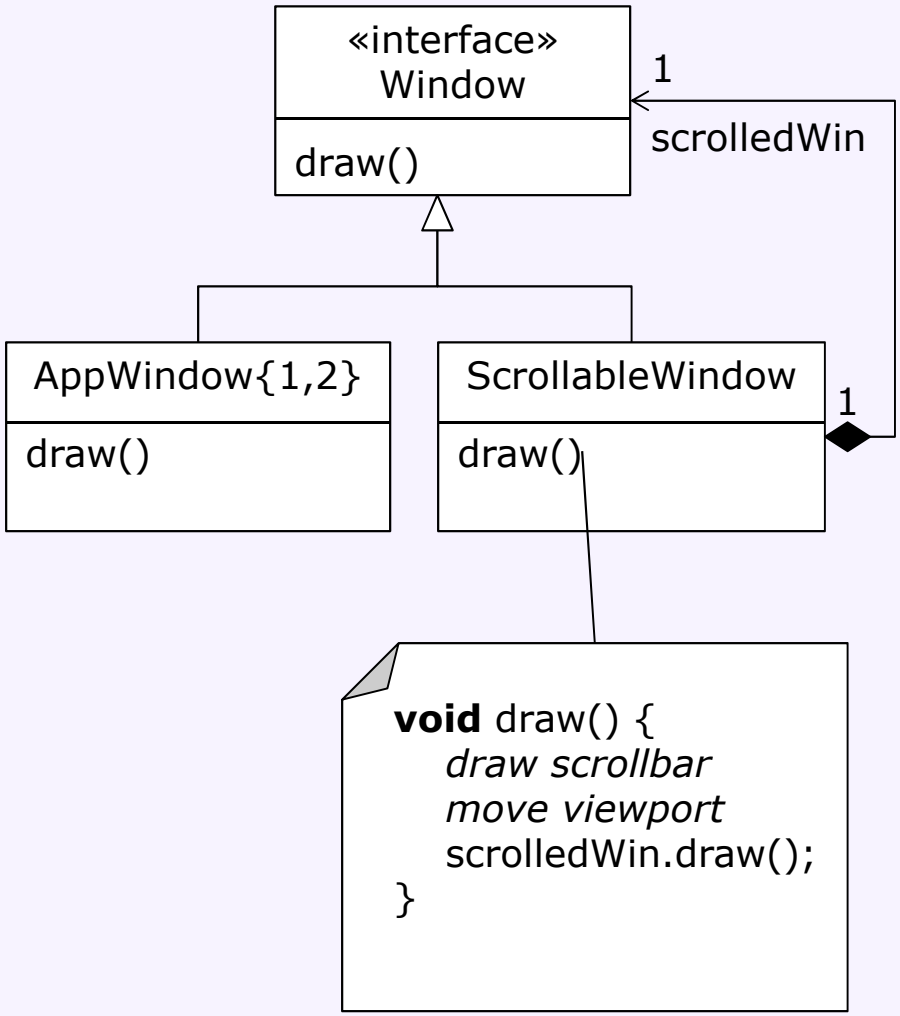


How do we add scrolling?
- Similar: window chrome

The ScrollableWindow Decorator

ScrollableWindow class

- A separate class
  - Avoids making Window incohesive
- Implements Window
  - Clients are not coupled to scrolling
- Wraps/decorates another window
  - Any window can be (re)used
- Implements draw() to add a scrollbar, then asks the wrapped window to draw itself

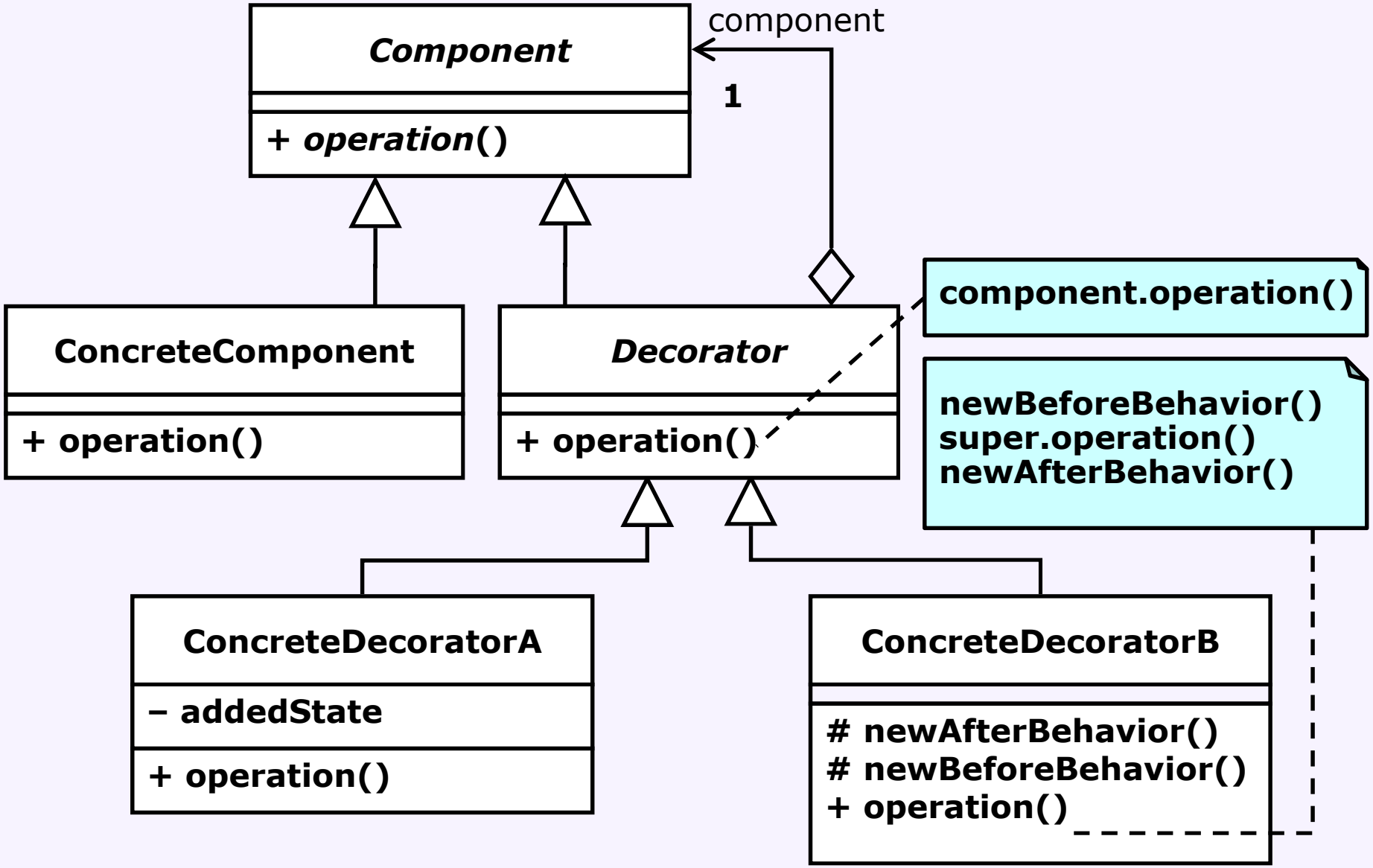


Decorator is a Design Pattern

- A general design problem
  - Need to add new functionality
  - Need to add it dynamically to different instances of an abstraction
  - Want to treat enhanced object just like the original object

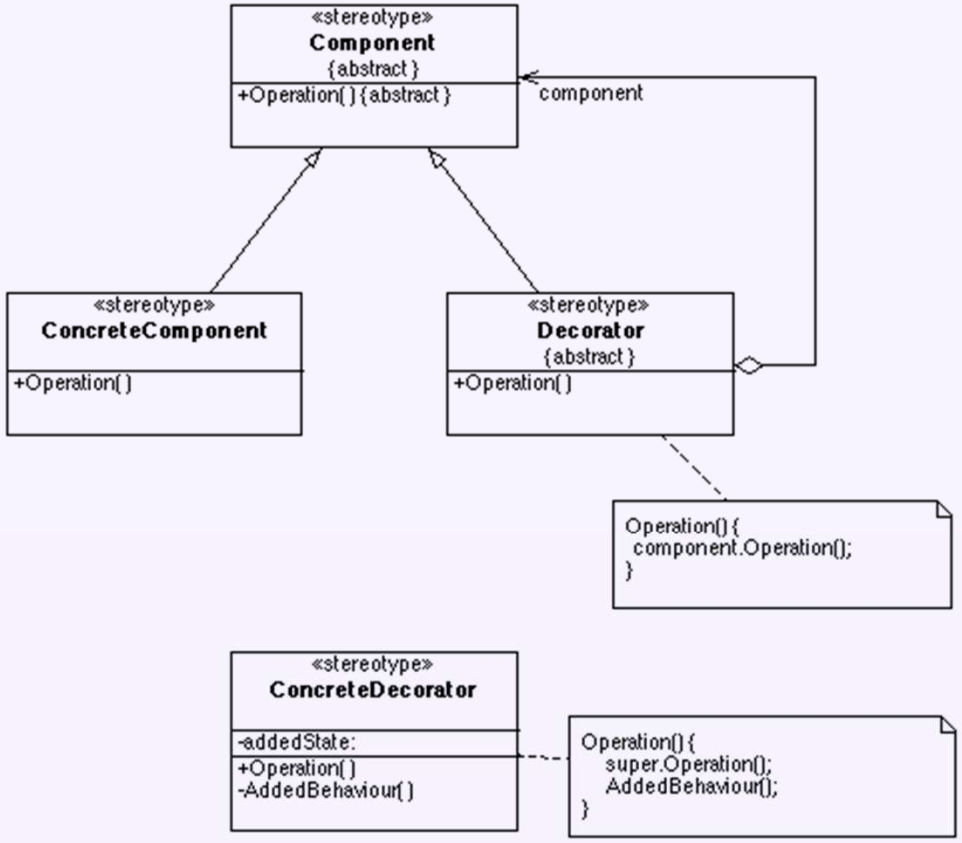
- We can generalize the solution as well
  - Create a new class that implements the abstraction's interface
  - Have the new class refer to a wrapped instance of the abstraction
  - Implement new behavior, requesting behavior from the wrapped object where appropriate

The Decorator Pattern



Structural: Decorator

- Applicability
  - To add responsibilities to individual objects dynamically and transparently
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical
- Consequences
  - More flexible than static inheritance
  - Avoids monolithic classes
  - Breaks object identity
  - Lots of little objects



Toad's Take-Home Messages

- Design is driven by quality attributes
  - Evolvability, separate development, reuse, performance, …
- Design principles provide guidance on achieving qualities
  - Low coupling, high cohesion, high correspondence, …
- GRASP *design heuristics* promote these principles
  - Creator, Expert, Controller, …
- Applying principles to common problems yields *design patterns*
  - Decorator, …