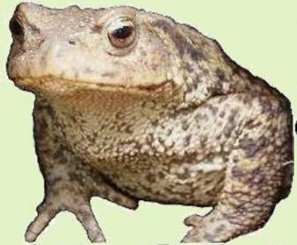


Objects Analysis

Threads



Design

15-214

**15-214**

***toad***

Spring 2014



# Principles of Software Construction: Objects, Design and Concurrency

## Design Patterns

**Jonathan Aldrich**

**Charlie Garrod**

## Exercise (paper): System Sequence Diagram for Scrabble

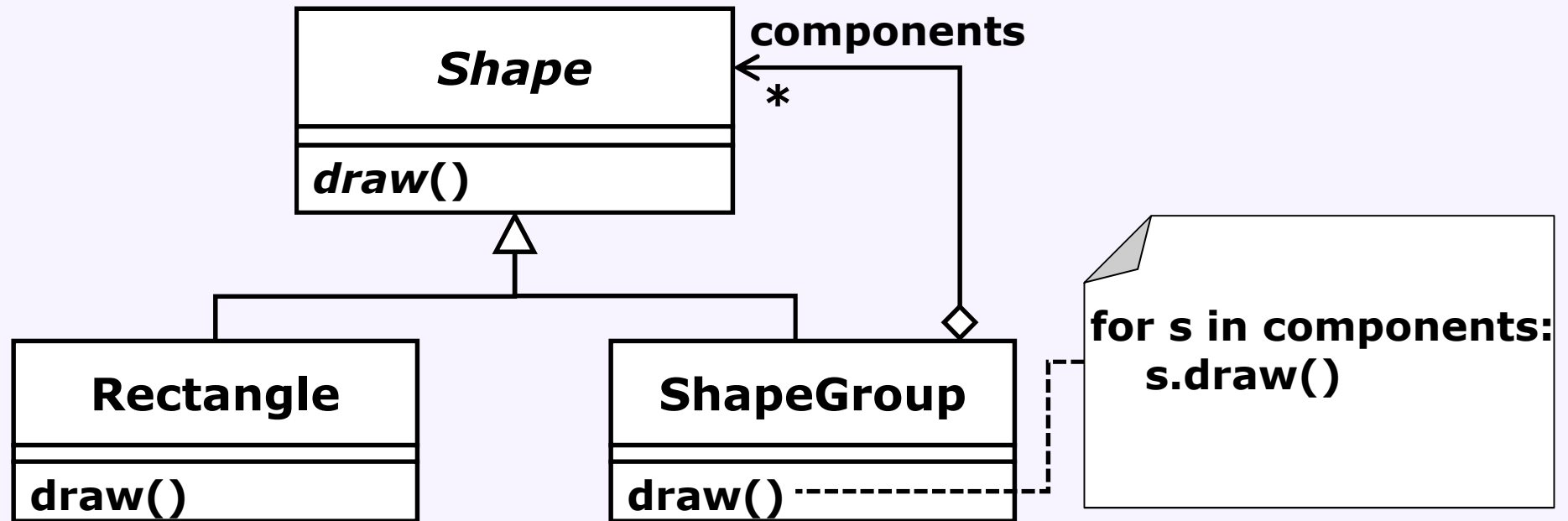


photo credit: Antrikshy (at Wikipedia)

## Learning Goals

- Understand the nature of design patterns
  - Parts of a design pattern
  - Applicability and benefits of design patterns
  - Limitations and pitfalls of design patterns
- Apply several design patterns:
  - Composite, Template Method, Decorator, ...

## Example: Composite Windows



- Problem
  - Express a part-whole hierarchy of shapes
  - Allow treating a group of shapes just like shapes
- Consequences
  - Makes clients simple; they can ignore the difference
  - Easy to add new kinds of shapes

## We have seen this before!

```
interface IntSet {  
    boolean contains(int element);  
    boolean isSubsetOf(IntSet otherSet);  
    IntSet union(IntSet otherSet);  
}  
class UnionSet implements IntSet {  
    private IntSet set1;  
    private IntSet set2;  
    public UnionSet(IntSet s1, IntSet s2) {  
        this.set1 = s1; this.set2 = s2; }  
    public boolean contains(int elem) {  
        return set1.contains(elem) || this.set2.contains(elem); }  
    public boolean isSubsetOf(IntSet otherSet) {  
        return set1.isSubsetOf(elem) && set2.isSubsetOf(elem);}  
    public IntSet union(IntSet otherSet) {  
        return new UnionSet(this, otherSet); }  
}
```

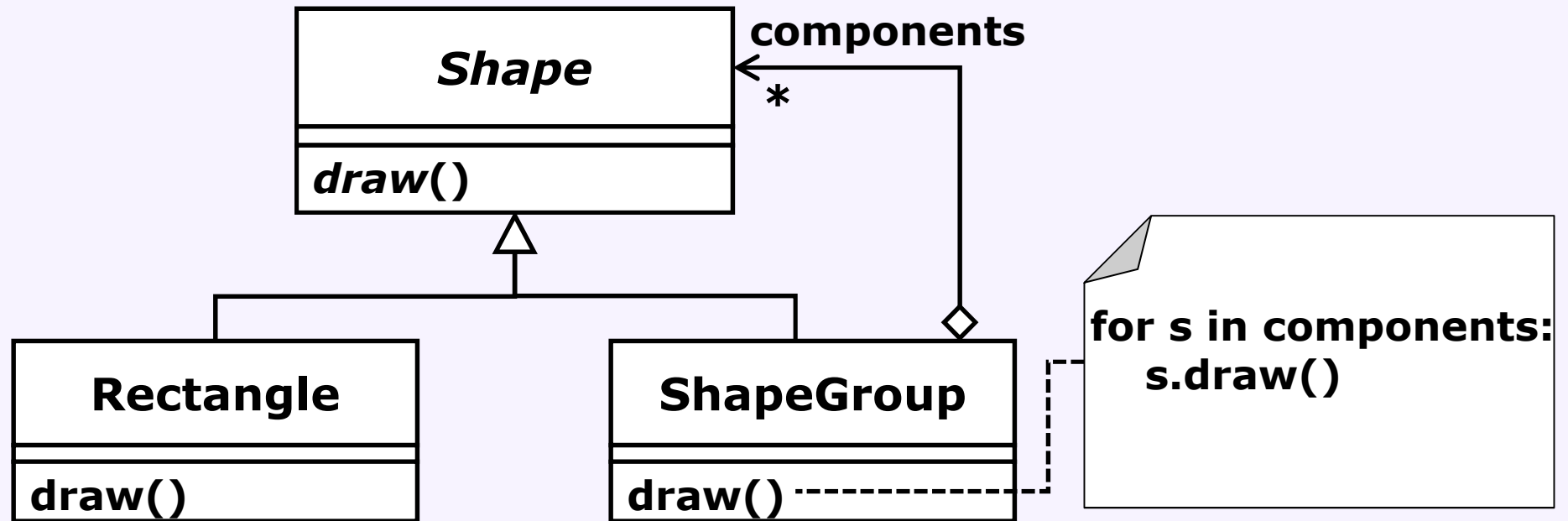
- Our designs for composite figures, grouped packages, and union sets solve similar problems in similar ways
- We call this problem-solution pair a **design pattern**

## Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
  - Christopher Alexander
- Every Composite has its own domain-specific interface
  - But they share a common problem and solution

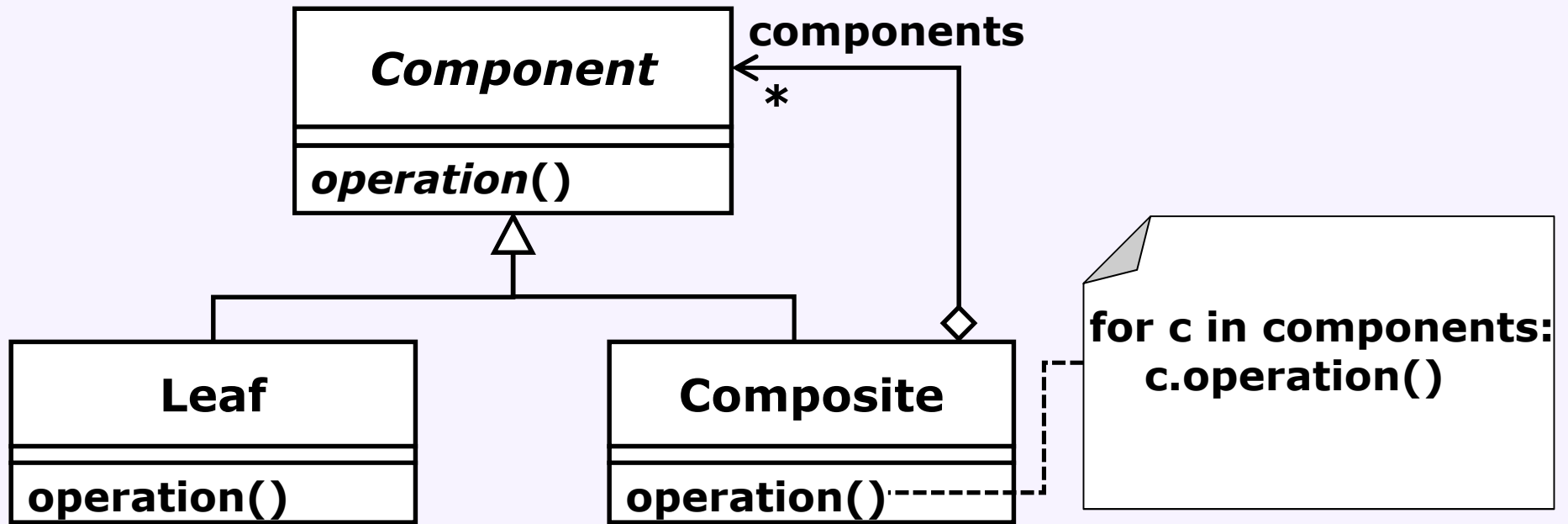


## Example: Composite Windows



- Problem
  - Express a part-whole hierarchy of shapes
  - Allow treating a group of shapes just like shapes
- Consequences
  - Makes clients simple; they can ignore the difference
  - Easy to add new kinds of shapes

# Composite Pattern



- Problem (generic)
  - Express a part-whole hierarchy of components
  - Allow treating a composite just like a component
- Consequences (generic)
  - Makes clients simple; they can ignore the difference
  - Easy to add new kinds of components
  - Can be overly general – uniformity is not always good



## History

- Christopher Alexander, *The Timeless Way of Building* (and other books)
  - Proposes patterns as a way of capturing design knowledge in architecture
  - Each pattern represents a tried-and-true solution to a design problem
  - Typically an engineering compromise that resolves conflicting forces in an advantageous way
    - Composite: you have a part-whole relationship, but want to treat individual objects and object compositions uniformly

## Patterns in Physical Architecture

- When a room has a window with a view, the window becomes a focal point: people are attracted to the window and want to look through it. The furniture in the room creates a second focal point: everyone is attracted toward whatever point the furniture aims them at (usually the center of the room or a TV). This makes people feel uncomfortable. They want to look out the window, and toward the other focus at the same time. If you rearrange the furniture, so that its focal point becomes the window, then everyone will suddenly notice that the room is much more “comfortable”.
  - Leonard Budney, Amazon.com review of *The Timeless Way of Building*

## Benefits of Patterns

- Shared language of design
  - Increases communication bandwidth
  - Decreases misunderstandings
- Learn from experience
  - Becoming a good designer is hard
    - Understanding good designs is a first step
  - Tested solutions to common problems
    - Where is the solution applicable?
    - What are the tradeoffs?

## Illustration [Shalloway and Trott]

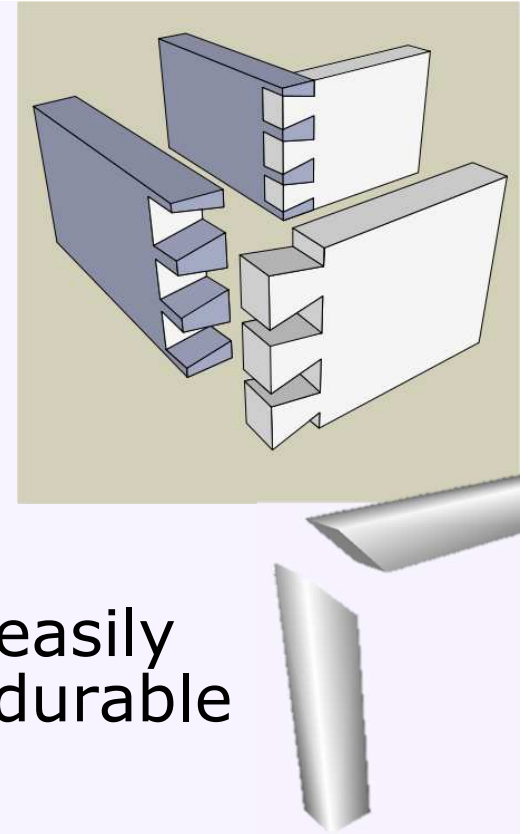
- Carpenter 1: How do you think we should build these drawers?
- Carpenter 2: Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...

## Illustration [Shalloway and Trott]

- Carpenter 1: How do you think we should build these drawers?
- Carpenter 2: Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...
- SE example: "I wrote this if statement to handle ... followed by a while loop ... with a break statement so that..."

## A Better Way

- Carpenter 1: Should we use a dovetail joint or a miter joint?
- Subtext:
  - miter joint: cheap, invisible, breaks easily
  - dovetail joint: expensive, beautiful, durable
- Shared terminology and knowledge of consequences raises level of abstraction
  - CS: Should we use a Composite?
  - Subtext
    - Is there a part-whole relationship here?
    - Might there be advantages to treating compositions and individuals uniformly?

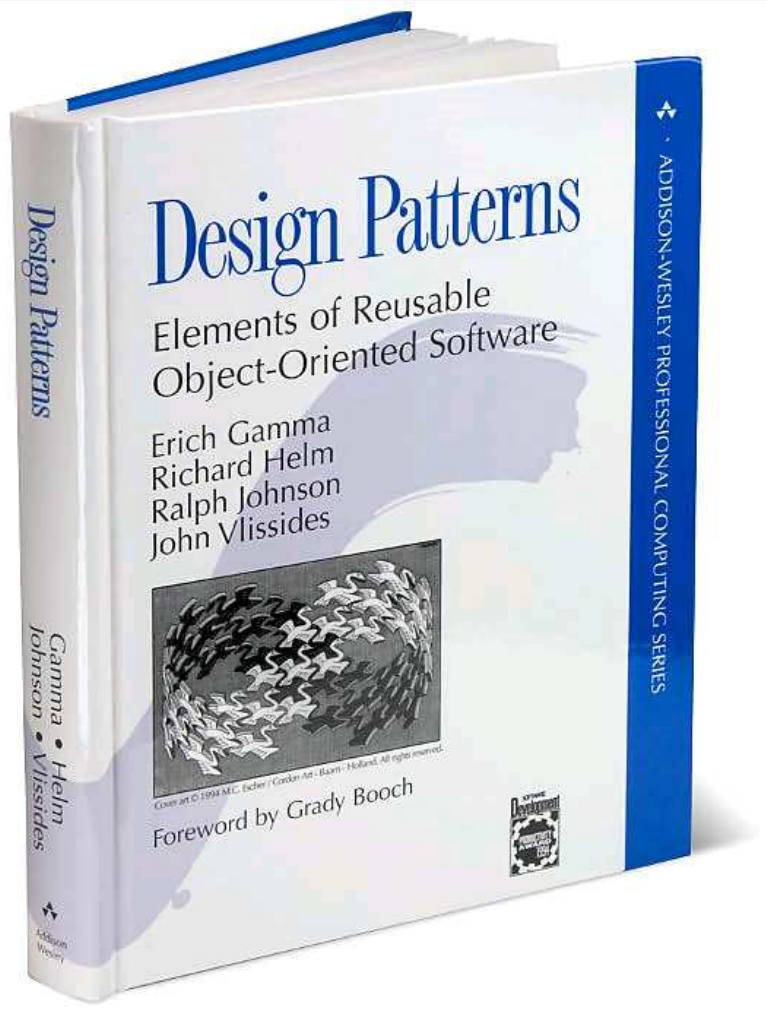


## Elements of a Pattern

- Name
  - Important because it becomes part of a design vocabulary
  - Raises level of communication
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract: must be specialized
- Consequences
  - Tradeoffs of applying the pattern
    - Each pattern has costs as well as benefits
    - Issues include flexibility, extensibility, etc.
    - There may be variations in the pattern with different consequences

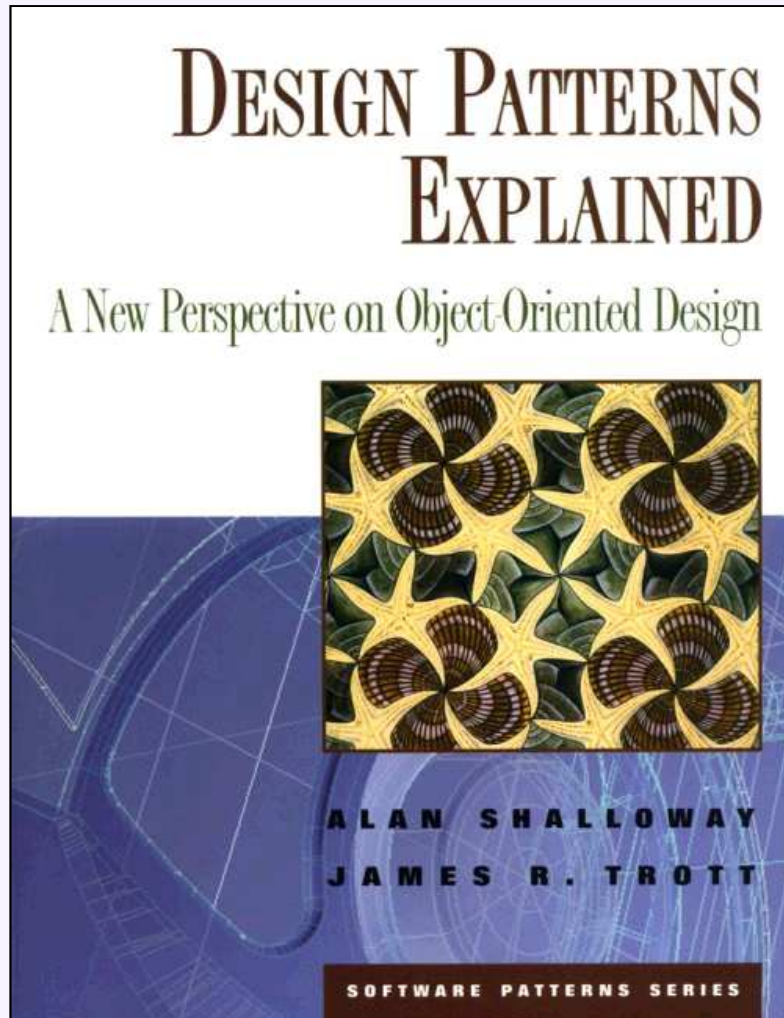


## History: Design Patterns Book



- Brought Design Patterns into the mainstream
- Authors known as the Gang of Four (GoF)
- Focuses on *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*
- Great as a reference text
- Uses C++, Smalltalk

## A More Recent Patterns Text



- Uses Java
  - The GoF text was written before Java went mainstream
- Good pedagogically
  - General design information
  - Lots of examples and explanation
  - GoF is really more a reference text
- Mandatory reading
- Helpful for HW4 and 5

# Fundamental OO Design Principles

- Patterns emerge from fundamental principles applied to recurring problems

**Composite:** uses a common interface for leaf and composite objects

- Design to **interfaces**
- Favor **composition** over inheritance
- Find what **varies** and hide (encapsulate) it

**Composite:** uses composition to represent a group of objects

- Patterns are discovered, not invented

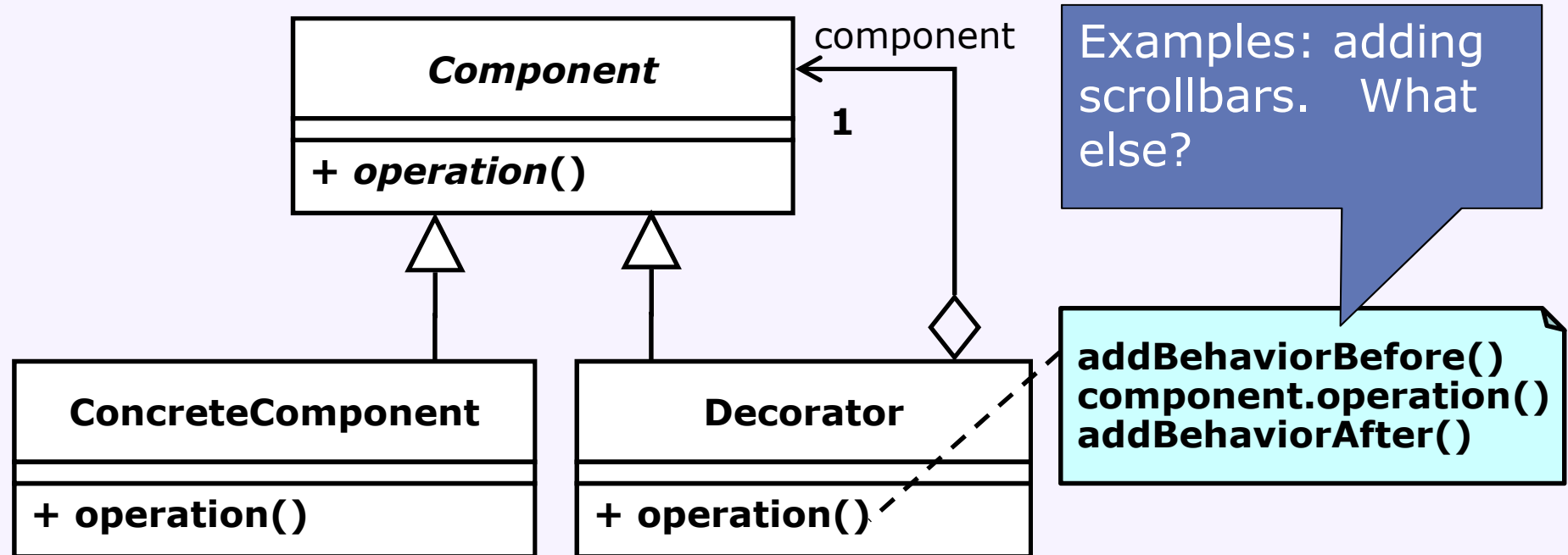
- Best practices by experienced developers
- Shared experience

**Composite:** hides the difference between an object and a group

# Introduction to Patterns

- Categories
  - Structural – vary object structure
  - Behavioral – vary the behavior you want
  - Creational – vary object creation
- Some UML diagrams by Pekka Nikander
  - <http://www.tml.tkk.fi/~pnr/GoF-models/html/>

## Recall a Structural pattern: 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

# Fundamental OO Design Principles

- Patterns emerge from fundamental principles applied to recurring problems

**Decorator:** uses a common interface for basic and decorated objects

- Design to **interfaces**
- Favor **composition** over inheritance
- Find what **varies** and hide (encapsulate) it

**Decorator:** uses composition to modify an object's behavior

- Patterns are discovered, not invented

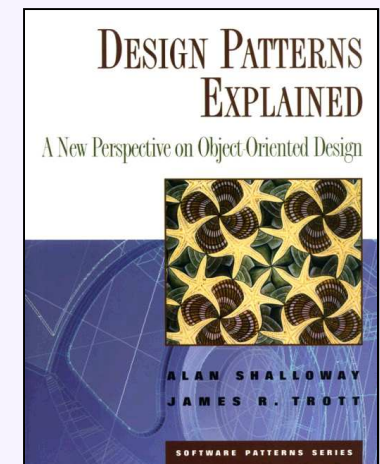
- Best practices by experienced developers
- Shared experience

**Decorator:** hides the difference between the base object and decorator

## Patterns to Know

- Façade\*, Adapter\*, Composite, Iterator, Strategy\*, Abstract Factory\*, Factory Method\*, Decorator\*, Observer\*, Template Method\*, Singleton\*, Command, Proxy, and Model-View-Controller
- Know pattern name, problem, solution, and consequences
- Know when to use them and when not

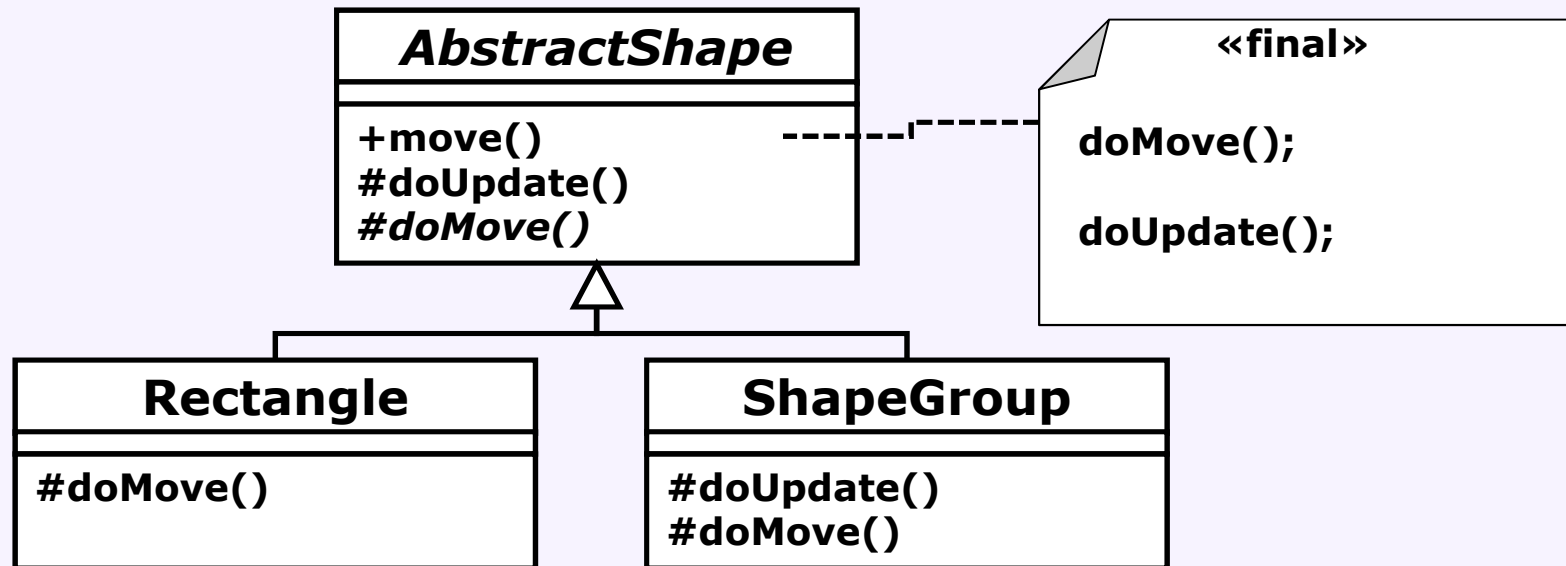
\* explained in:





## Back to the Motivating Problem...

## Example: Shape Change Notification



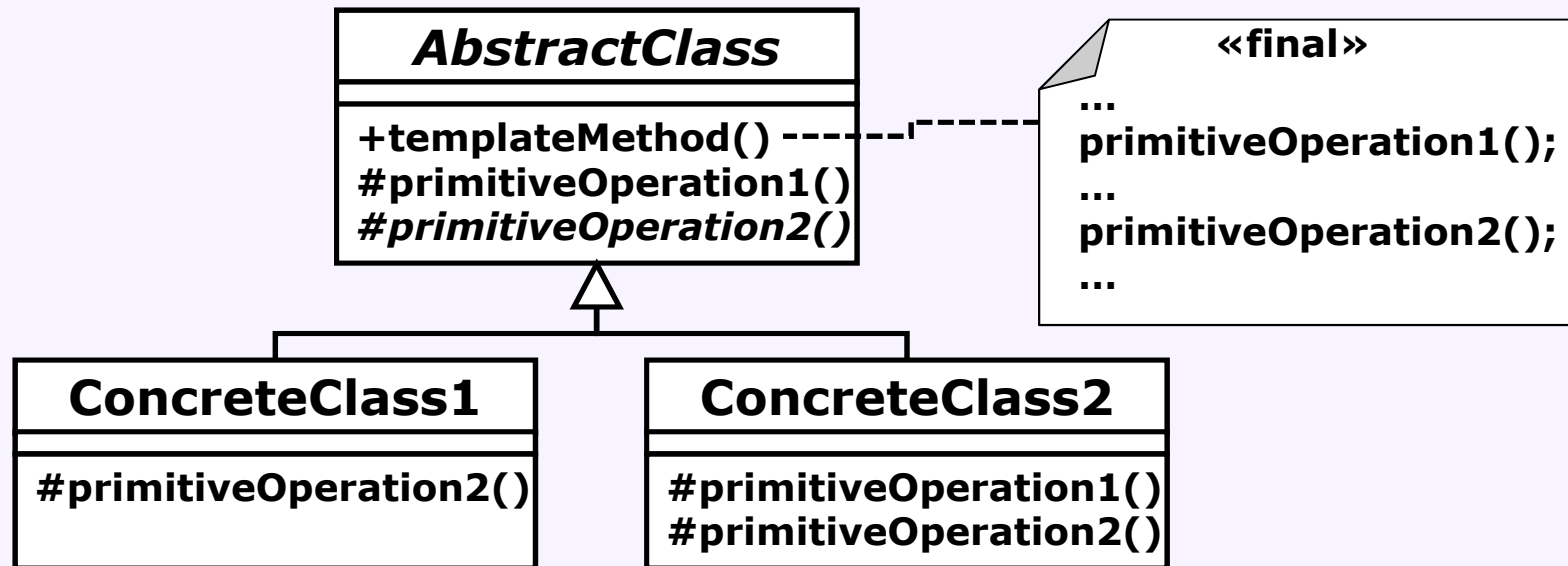
- Problem

- Moving changes from shape to shape, but updating doesn't - want to reuse updating code
- Future shape implementations should not forget to update

- Consequences

- Code reuse
- Authors of subclasses will not unintentionally forget to do the update

# Template Method Pattern

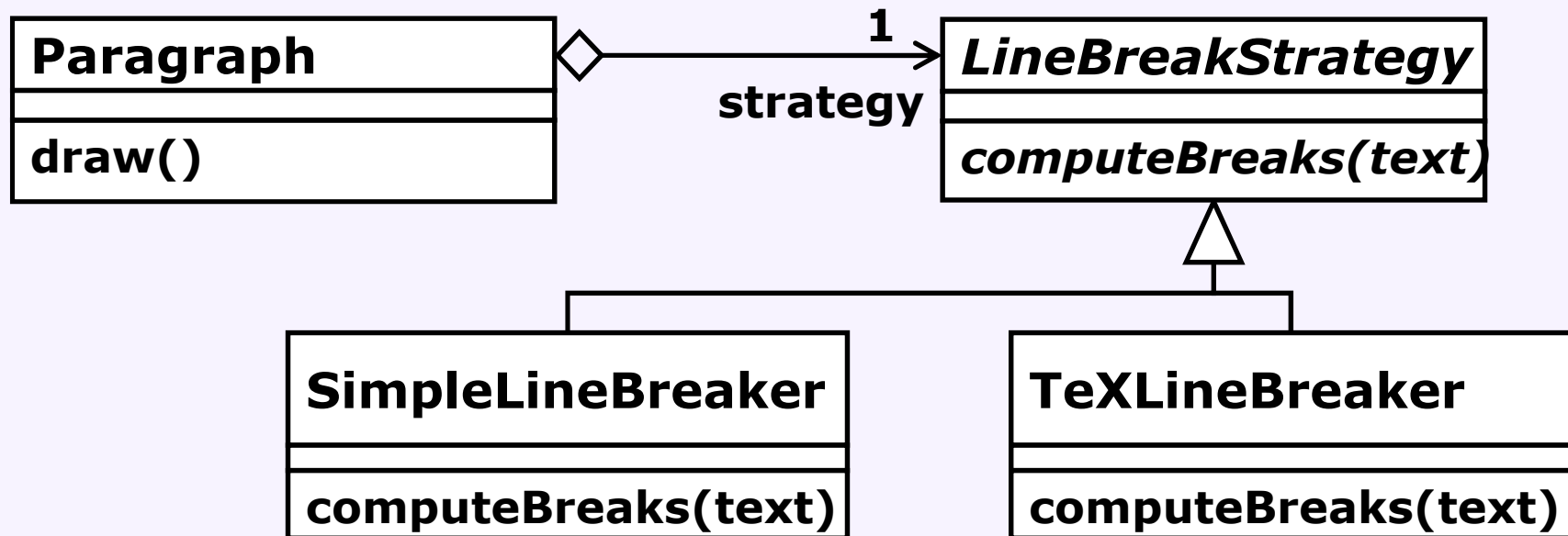


- Problem (generic)
  - Express an algorithm with varying and invariant parts
  - When common behavior should be factored and localized
  - When subclass extensions should be limited
- Consequences (generic)
  - Code reuse
  - Inverted "Hollywood" control: don't call us, we'll call you
  - Invariant algorithm parts are not changed by subclasses

## Problem: Line Breaking Implementations

- Context: document editor
- Many ways to break a paragraph into lines
  - Blind: just cut off at 80 columns
  - Greedy: fit as many words in this line, then wrap
  - Global (e.g. TeX): minimize badness in entire paragraph
    - Might move a small word to next line if it reduces extra spaces there
- Option 1: We could put this in class Paragraph
  - But this is not Paragraph's main function
  - Putting many algorithms into Paragraph makes it too big
  - Other classes might need line breaking, too
  - Adding new line breaking algorithms is difficult
- Option 2?

## Option 2: Encapsulate the Line Breaking Strategy



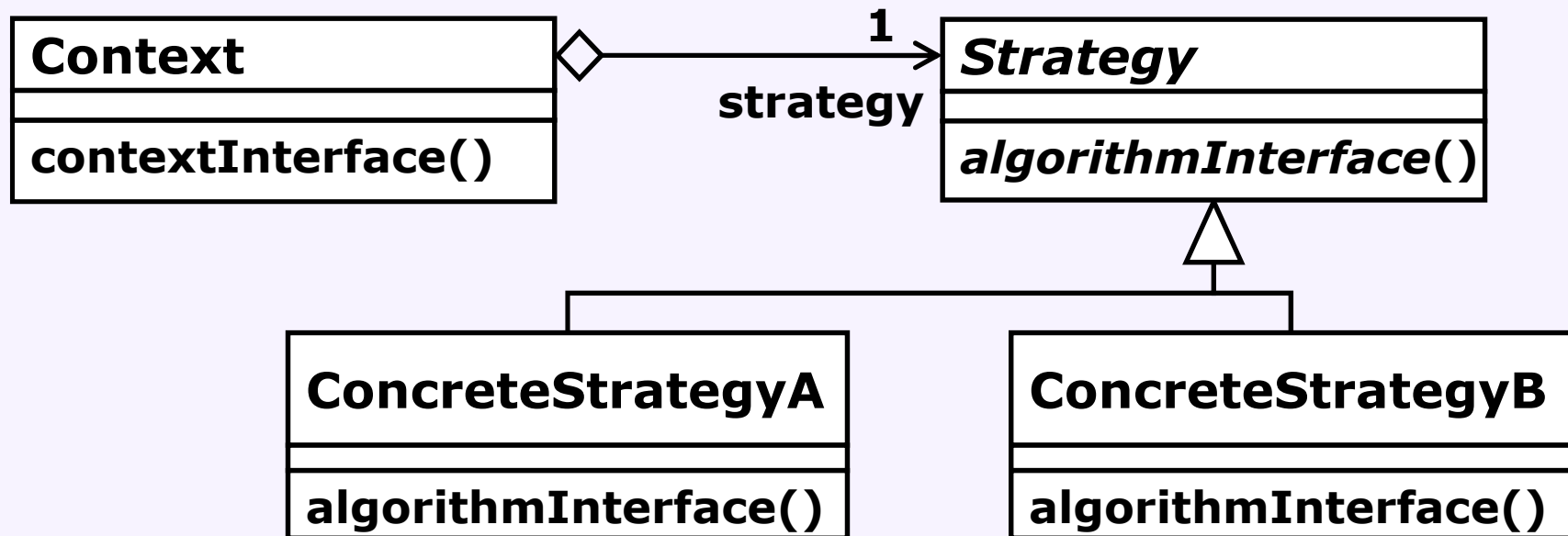
- **Problem**

- Paragraphs need to break lines in different ways
- Want to easily change or extend line breaking algorithm
- Want to reuse algorithm in new places

- **Consequences**

- Easy to add new line breaking strategies
- Separates strategy → vary strategy, paragraph independently
- Adds objects and dynamism → code harder to understand

# Strategy Pattern



- Problem (generic)
  - Behavior varies among instances of an abstraction
  - An abstraction needs different variants of an algorithm
- Consequences (generic)
  - Easy to add new strategies (e.g. compared to conditionals)
  - Separates algorithm → vary algorithm, context independently
  - Adds objects and dynamism → code harder to understand
  - Fixed strategy interface → high overhead for some impls.

# Specific Patterns

- Facade
- Adapter
- Strategy
- Template Method
- Factory Method
- Abstract Factory
- Decorator
- Observer
- *Command*
- *State*
- *Composite*
- Bridge
- *Singleton*
- *Proxy*
- *Visitor*



# Scenario

- You need to load and print pictures in your application
- You found a library that provides far more than you need
  - Many classes
  - Different representations
  - Sophisticated image manipulation routines
- You may want to switch to a different library later
- What's the right design?

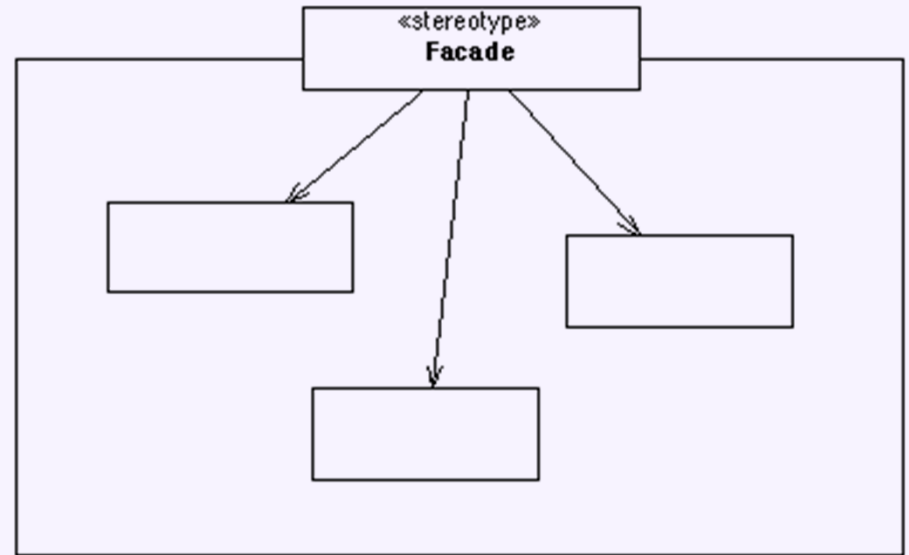
# Façade (Structural)

- Applicability

- You want to provide a simple interface to a complex subsystem
- You want to decouple clients from the implementation of a subsystem
- You want to layer your subsystems

- Consequences

- It shields clients from the complexity of the subsystem, making it easier to use
- Decouples the subsystem and its clients, making each easier to change
- Clients that need to can still access subsystem classes



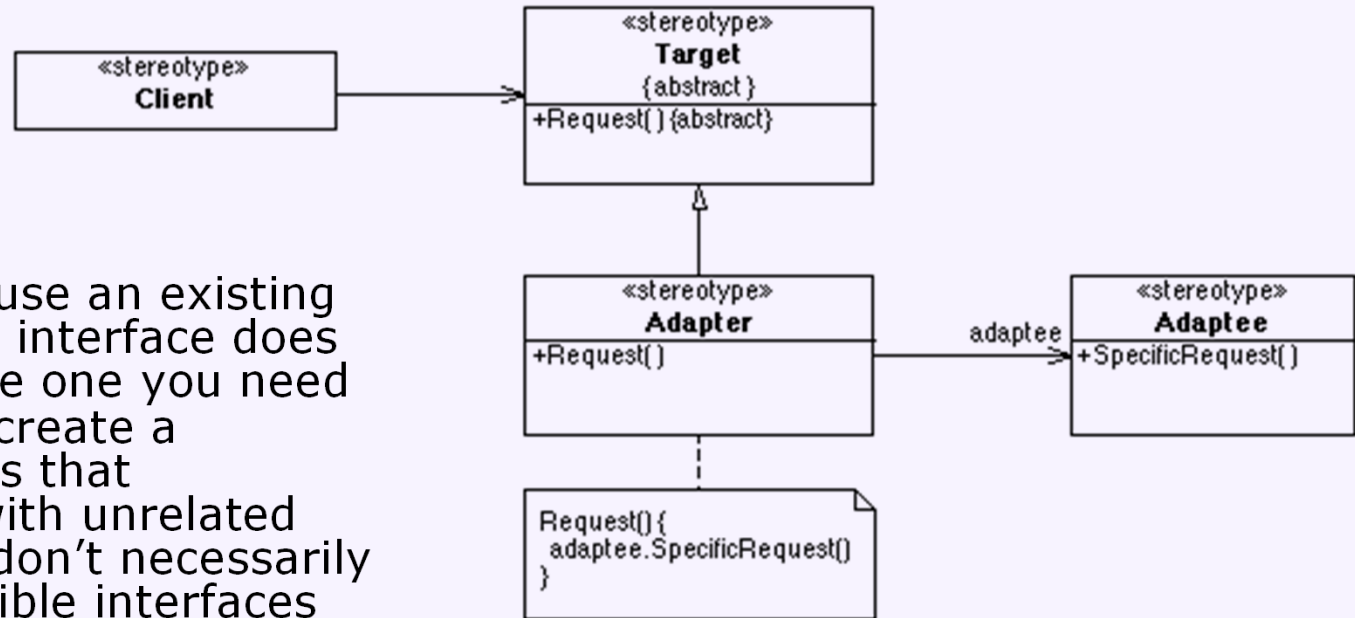
## Scenario

- You have an application that processes data with an Iterator. Methods are:
  - **boolean** hasNext();
  - Object next();
- You need to read that data from a database using JDBC. Methods are:
  - **boolean** next();
  - Object getObject(int column);
- You might have to get the information from other sources in the future.

# Structural: Adapter

- Applicability

- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
- You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one



- Consequences

- Exposes the functionality of an object in another form
- Unifies the interfaces of multiple incompatible adaptee objects
- Lets a single adapter work with multiple adaptees in a hierarchy

## Back to Fundamental Principles

- Design to interfaces
  - Façade – a new interface for a library
  - Adapter – design application to a common interface, adapt other libraries to that
- Favor composition over inheritance
  - Façade – library is composed within Façade
  - Adapter – adapter object interposed between client and implementation
- Find what varies and encapsulate it
  - Both Façade and Adapter – shields variations in the implementation from the client

## Façade vs. Adapter

- Motivation
  - Façade: simplify the interface
  - Adapter: match an existing interface
- Adapter: interface is given
  - Not typically true in Façade
- Adapter: polymorphic
  - Dispatch dynamically to multiple implementations
  - Façade: typically choose the implementation statically

# Scenario

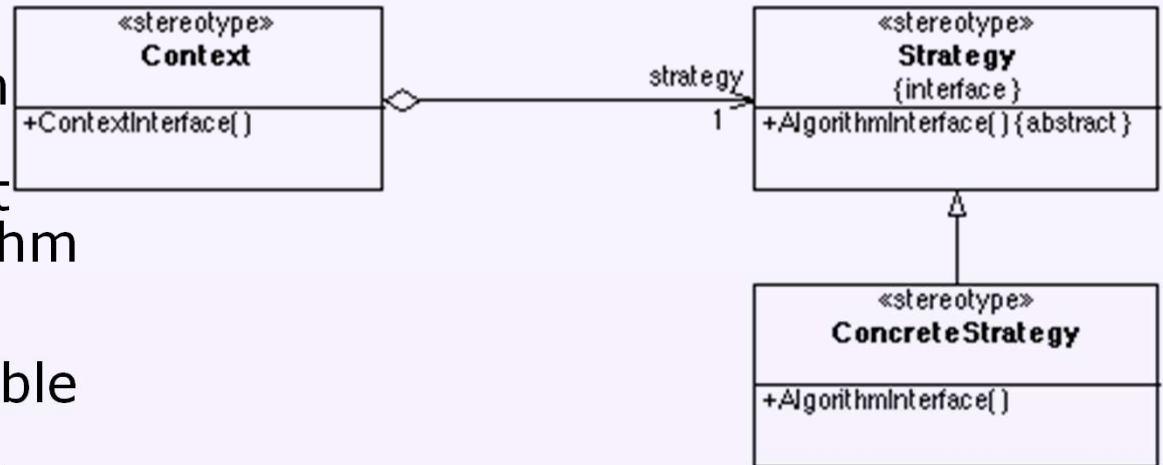
- Context: eCommerce application
  - Cart object holds Items
- Problem: how to compute taxes?
  - State sales tax
  - Local sales tax
  - Differing exemptions
  - ...
- How can we make the taxation algorithm easy to change?



# Behavioral: Strategy

- **Applicability**

- Many classes differ in only their behavior
- Client needs different variants of an algorithm



- **Consequences**

- Code is more extensible with new strategies
  - Compare to conditionals
- Separates algorithm from context
  - each can vary independently
- Adds objects and dynamism
  - code harder to understand
- Common strategy interface
  - may not be needed for all Strategy implementations – may be extra overhead

# Tradeoffs

```
void sort(int[] list, String order) {  
    ...  
    boolean mustswap;  
    if (order.equals("up")) {  
        mustswap = list[i] < list[j];  
    } else if (order.equals("down")) {  
        mustswap = list[i] > list[j];  
    }  
    ...  
}
```

```
void sort(int[] list, Comparator cmp) {  
    ...  
    boolean mustswap;  
    mustswap = cmp.compare(list[i], list[j]);  
    ...  
}  
interface Comparator {  
    boolean compare(int i, int j);  
}  
class UpComparator implements Comparator {  
    boolean compare(int I, int j) { return i<j; }}  
  
class DownComparator implements Comparator {  
    boolean compare(int I, int j) { return i>j; }}
```

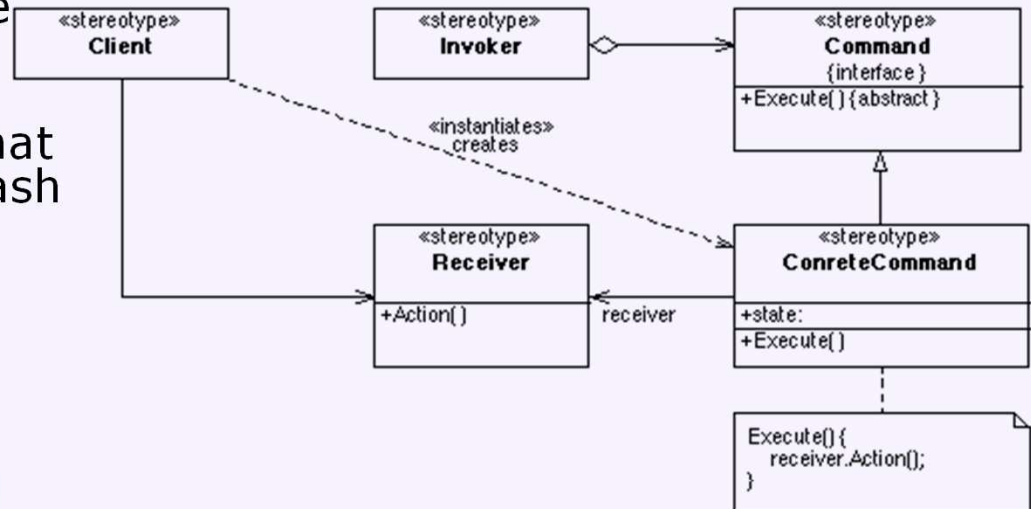
## Back to Fundamental Principles

- Design to interfaces
  - Strategy: the algorithm interface
- Favor composition over inheritance
  - Strategy could be implemented with inheritance
    - Multiple subclasses of Context, each with an algorithm
    - Drawback: couples Context to algorithm, both become harder to change
    - Drawback: can't change algorithm dynamically
- Find what varies and encapsulate it
  - Strategy: the algorithm used
- Side note: how do you implement the Strategy pattern in functional languages?

# Behavioral: Command

- Applicability

- Parameterize objects by an action to perform
- Specify, queue and execute requests at different times
- Support undo
- Support logging changes that can be reapplied after a crash
- Structure a system around high-level operations built out of primitives



- Consequences

- Decouples the object that invokes the operation from the one that performs it
- Since commands are objects they can be explicitly manipulated
- Can group commands into composite commands
- Easy to add new commands without changing existing code

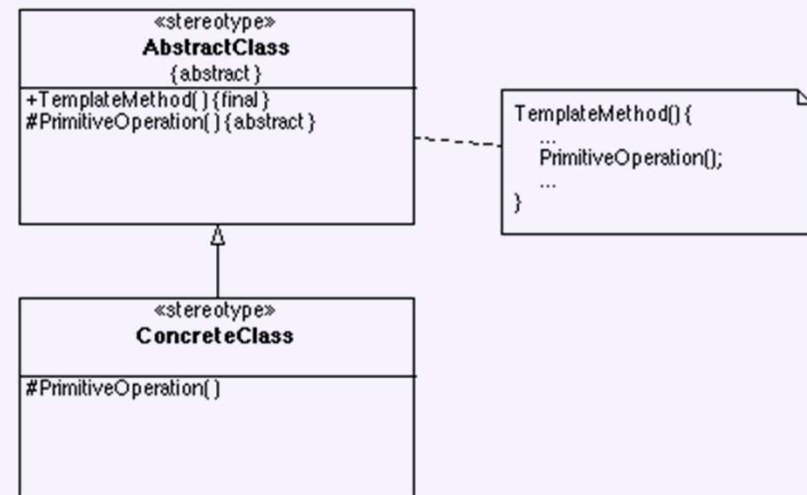
# Behavioral: Template Method

- **Applicability**

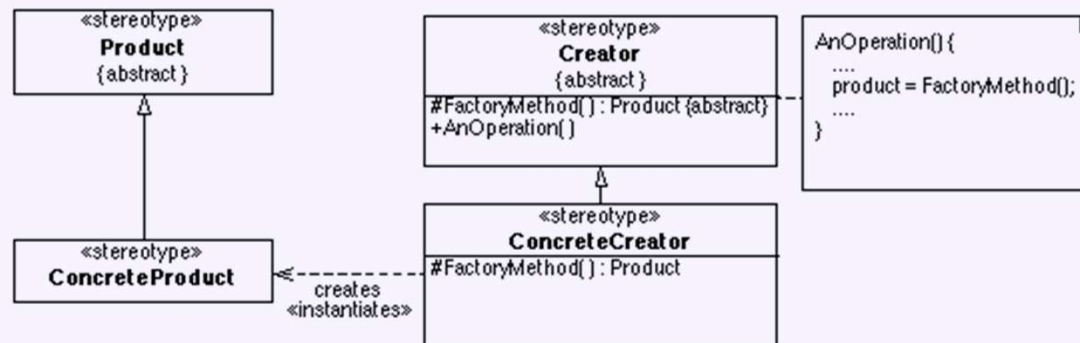
- When an algorithm consists of varying and invariant parts that must be customized
- When common behavior in subclasses should be factored and localized to avoid code duplication
- To control subclass extensions to specific operations

- **Consequences**

- Code reuse
- Inverted “Hollywood” control: don’t call us, we’ll call you
- Ensures the invariant parts of the algorithm are not changed by subclasses



# Creational: Factory Method



- **Applicability**

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates

- **Consequences**

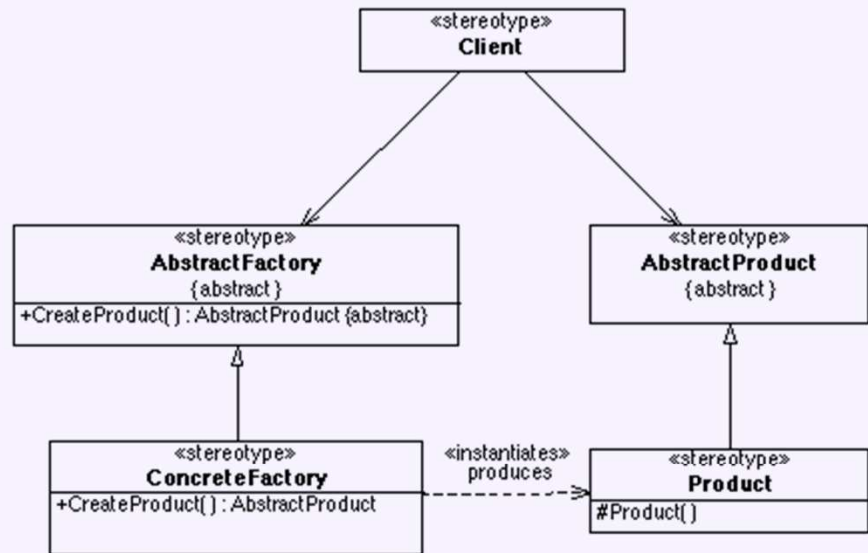
- Provides hooks for subclasses to customize creation behavior
- Connects parallel class hierarchies

# Scenario

- Context: Window library
  - Multiple kinds of windows
  - Multiple implementation families (by library, OS, etc.)
  - Bridge pattern
- Problem: how to create the implementation objects?
  - Avoid tying window interface to particular back ends
  - Back ends must work together (no mixing Swing and SWT components)

# Creational: Abstract factory

- **Applicability**
  - System should be independent of product creation
  - Want to configure with multiple families of products
  - Want to ensure that a product family is used together
- **Consequences**
  - Isolates concrete classes
  - Makes it easy to change product families
  - Helps ensure consistent use of family
  - Hard to support new kinds of products





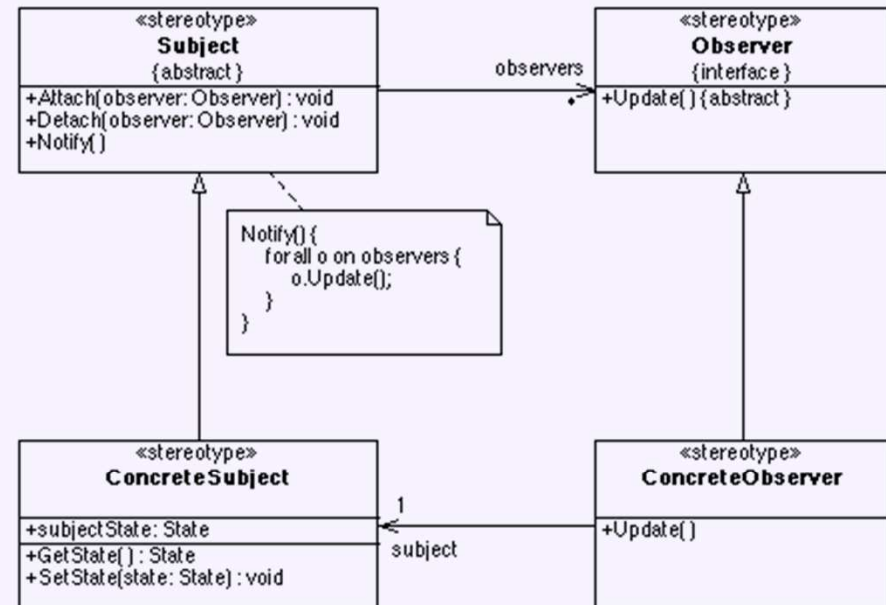
# Behavioral: Observer

- **Applicability**

- When an abstraction has two aspects, one dependent on the other, and you want to reuse each
- When change to one object requires changing others, and you don't know how many objects need to be changed
- When an object should be able to notify others without knowing who they are

- **Consequences**

- Loose coupling between subject and observer, enhancing reuse
- Support for broadcast communication
- Notification can lead to further updates, causing a cascade effect

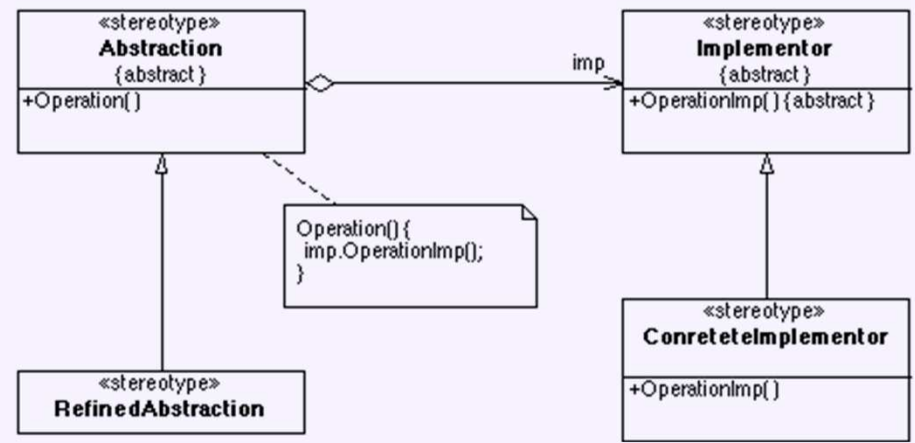


# Scenario

- Shape graphics library
  - rectangles, circles, squares
  - extensible to more shapes
- Need flexible implementation
  - Java Swing
  - Eclipse SWT
  - Printing libraries, etc.
- How can we allow both:
  - extension with new shapes
  - adaptation to new back ends?

# Structural: Bridge

- Applicability
  - Want to define multiple abstractions
  - Need to implement in multiple ways
- Consequences
  - Avoid blow-up in number of classes
  - Decouples abstraction from implementation
    - Choose each separately, even at run time
    - Extend each independently
  - Hide implementation from clients
  - Requires fixed implementation interface



# Scenario

- You have global data & operations
  - Must be used consistently within the app
  - Might be changed later
  - Don't want to pass around explicitly
- No good existing place to create and store the object

# Creational: Singleton

- Applicability

- There must be exactly one instance of a class
- When it must be accessible to clients from a well-known place
- When the sole instance should be extensible by subclassing, with unmodified clients using the subclass

- Consequences

- Controlled access to sole instance
- Reduced name space (vs. global variables)
- Can be refined in subclass or changed to allow multiple instances
- More flexible than class operations
  - Can change later if you need to



- Implementation

- Constructor is protected
- Instance variable is private
- Public operation returns singleton
  - May lazily create singleton

- Subclassing

- Instance() method can look up subclass to create in environment

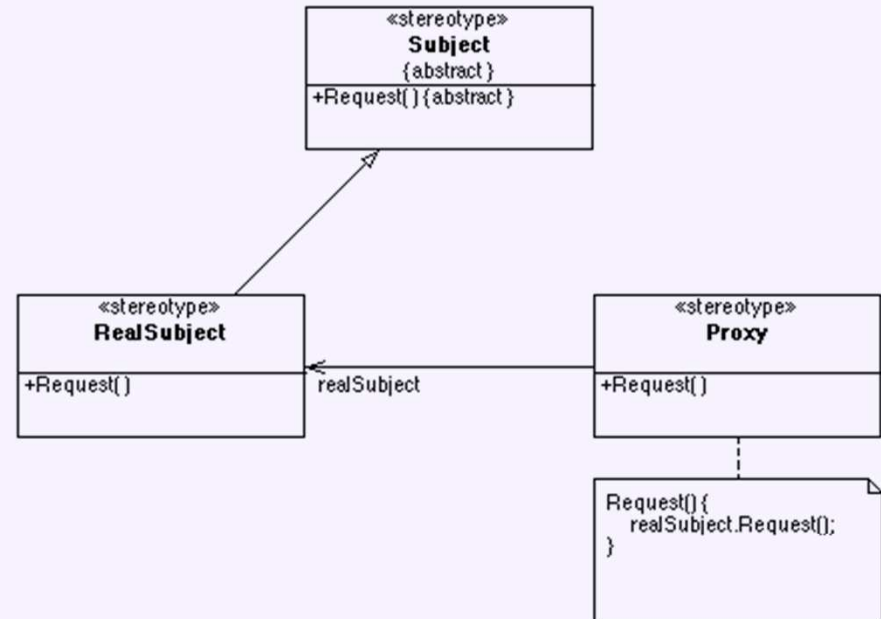
# Structural: Proxy

- Applicability

- Whenever you need a more sophisticated object reference than a simple pointer
  - Local representative for a remote object
  - Create or load expensive object on demand
  - Control access to an object
  - Reference count an object

- Consequences

- Introduces a level of indirection
  - Hides distribution from client
  - Hides optimizations from client
  - Adds housekeeping tasks



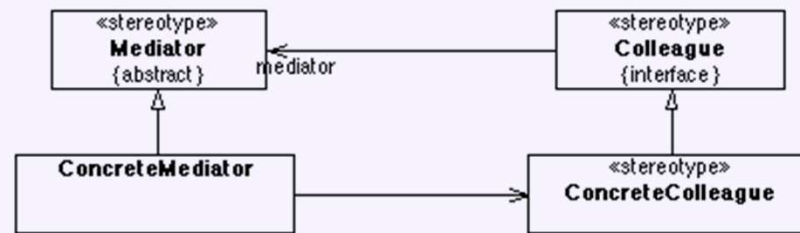
# Behavioral: Mediator

- **Applicability**

- A set of objects that communicate in well-defined but complex ways
- Reusing an object is difficult because it communicates with others
- A behavior distributed between several classes should be customizable without a lot of subclassing

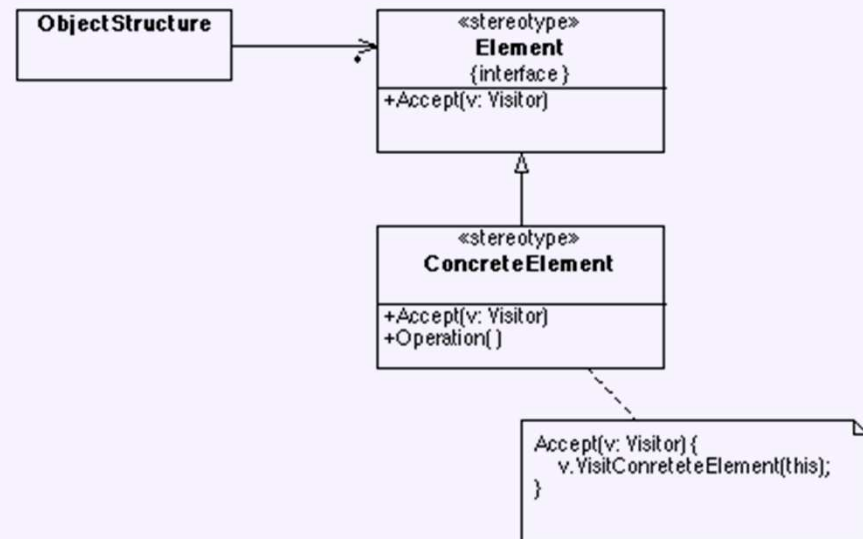
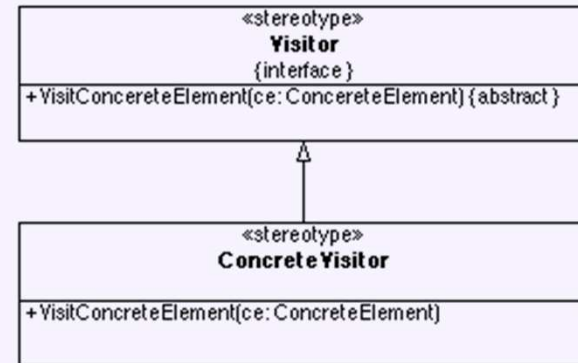
- **Consequences**

- Avoids excessive subclassing to customize behavior
- Decouples colleagues, enhancing reuse
- Simplifies object protocols: many-to-many to one-to-many
- Abstracts how objects cooperate into the mediator
- Centralizes control
  - Danger of mediator monolith



# Behavioral: Visitor

- Applicability
  - Structure with many classes
  - Want to perform operations that depend on classes
  - Set of classes is stable
  - Want to define new operations
- Consequences
  - Easy to add new operations
  - Groups related behavior in Visitor
  - Adding new elements is hard
  - Visitor can store state
  - Elements must expose interface





## Other GoF Patterns

- **Creational**
  - Builder – separate creation from representation
  - Prototype – create objects by copying
- **Structural**
  - Flyweight – use sharing for fine-grained objects
- **Behavioral**
  - Chain of Responsibility – sequence of objects can respond to a request
  - Interpreter – canonical implementation technique
  - Memento – externalize/restore an object's state
  - Mediator – a set of objects that communicate in well-defined but complex ways
  - State – allow object to alter behavior when state changes