Principles of Software Construction: Objects, Design, and Concurrency

A Quick Tour of all 23 GoF Design Patterns

Christian Kästner Vincent Hellendoorn

Carnegie Mellon University
School of Computer Science

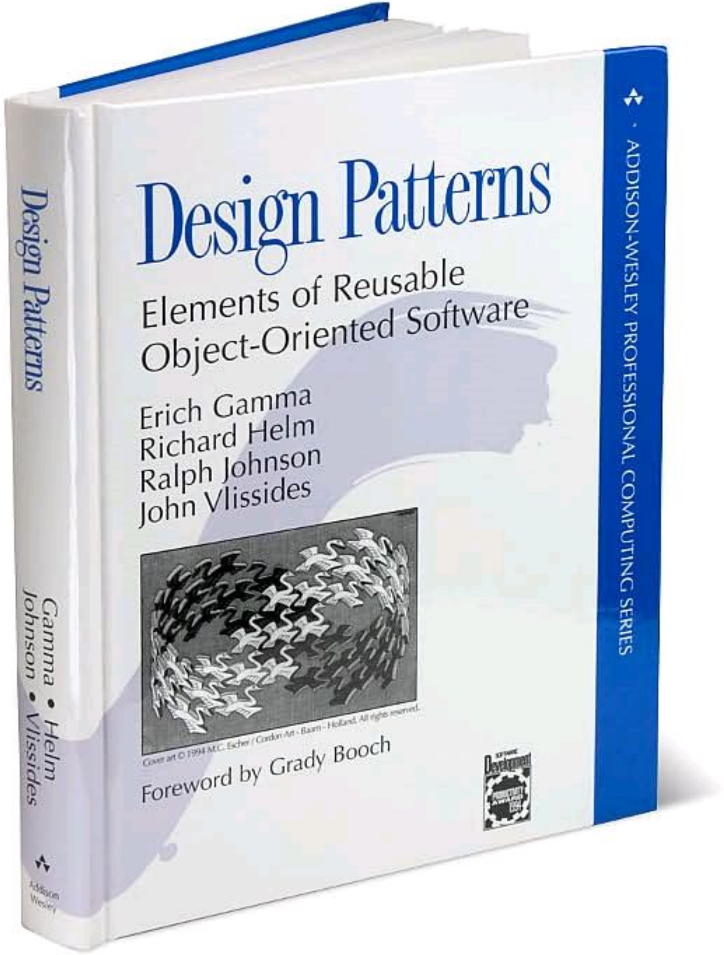
institute for SOFTWARE RESEARCH

Where we are

	Small scale: One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
Design for understanding change/ext. reuse robustness ...	Subtype Polymorphism ✓ Information Hiding, Contracts ✓ Immutability ✓ Types Unit Testing ✓	Domain Analysis ✓ Inheritance & Del. ✓ Responsibility Assignment, **Design Patterns**, Antipattern ✓ Promises/ Reactive P. ✓ Integration Testing ✓	GUI vs Core ✓ Frameworks and Libraries ✓, APIs ✓ Module systems, microservices ✓ Testing for Robustness ✓ CI ✓, DevOps, Teams

institute for SOFTWARE RESEARCH

- Published 1994
- 23 Patterns
- Widely known



Warmup: Scenario

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 1: You want to create monthly reports. However different cities want this report slightly differently, with different text on top and sorted in different ways. You want to vary text and sorting in different ways.

Our course so far

- Composite
- Strategy
- Template Method
- Iterator
- Proxy
- Adapter
- Decorator
- Observer
- Factory Method

Not in the book:

- Model view controller
- Promise
- Generator
- Module (JS)

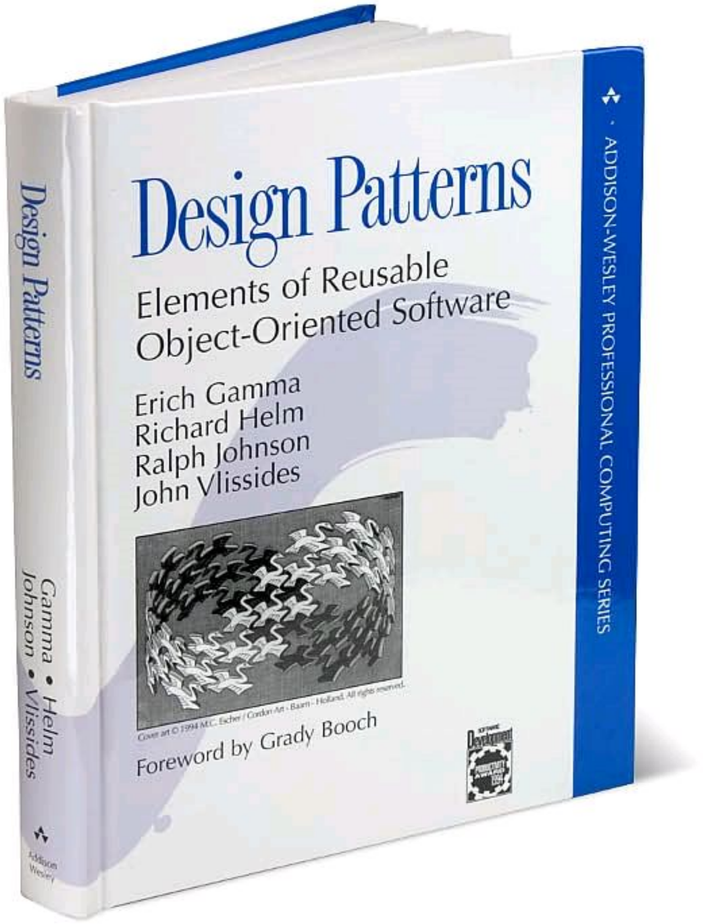
Why?

- Seminal and canonical list of well-known patterns
- Not all patterns are commonly used
- Does not cover all popular patterns

- At least know where to look up when somebody mentions the “Bridge pattern”

Grouping Patterns

I. Creational Patterns
II. Structural Patterns
III. Behavioral Patterns



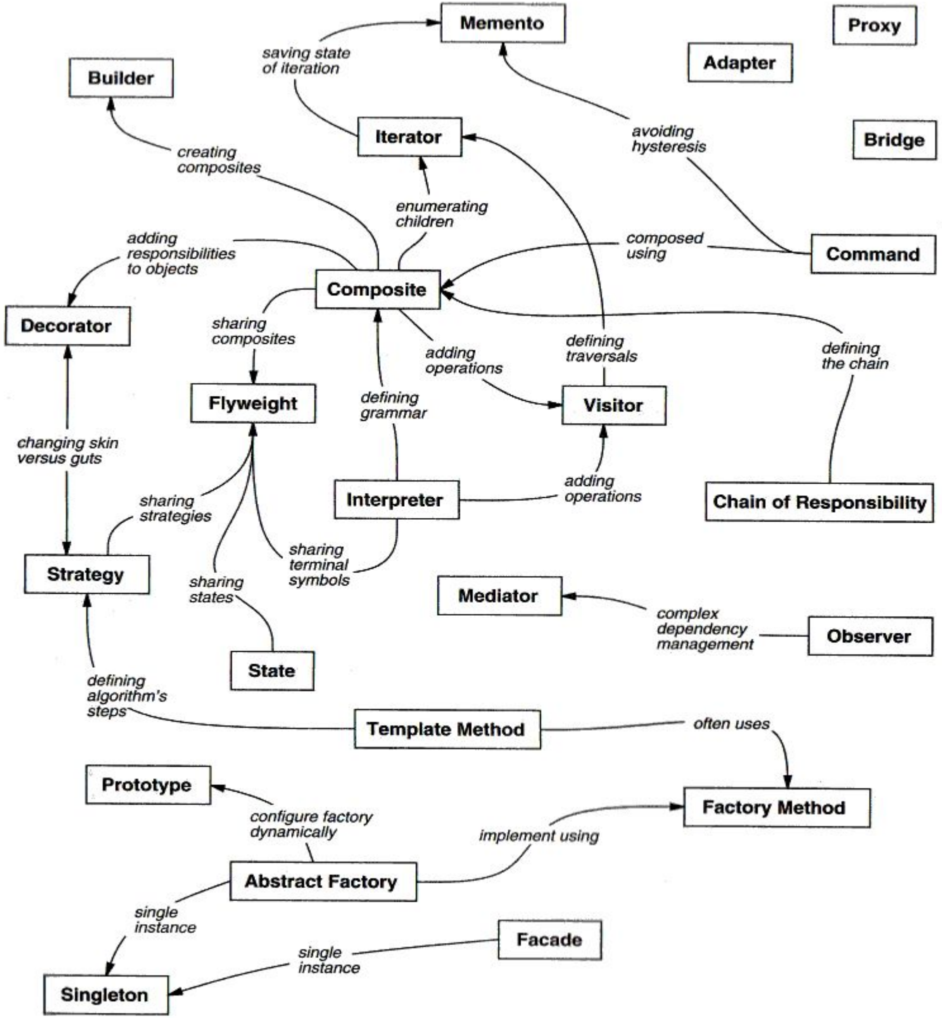


Figure 1.1: Design pattern relationships

Pattern Name

- **Intent** – the aim of this pattern
- **Use case** – a motivating example
- **Key types** – the types that define pattern
  - Italic type name indicates abstract class; typically this is an interface when the pattern is used in Java
- **Examples**

Illustration

- **Code sample, diagram, or drawing**
  - Time constraints make it impossible to include illustrations from some patterns

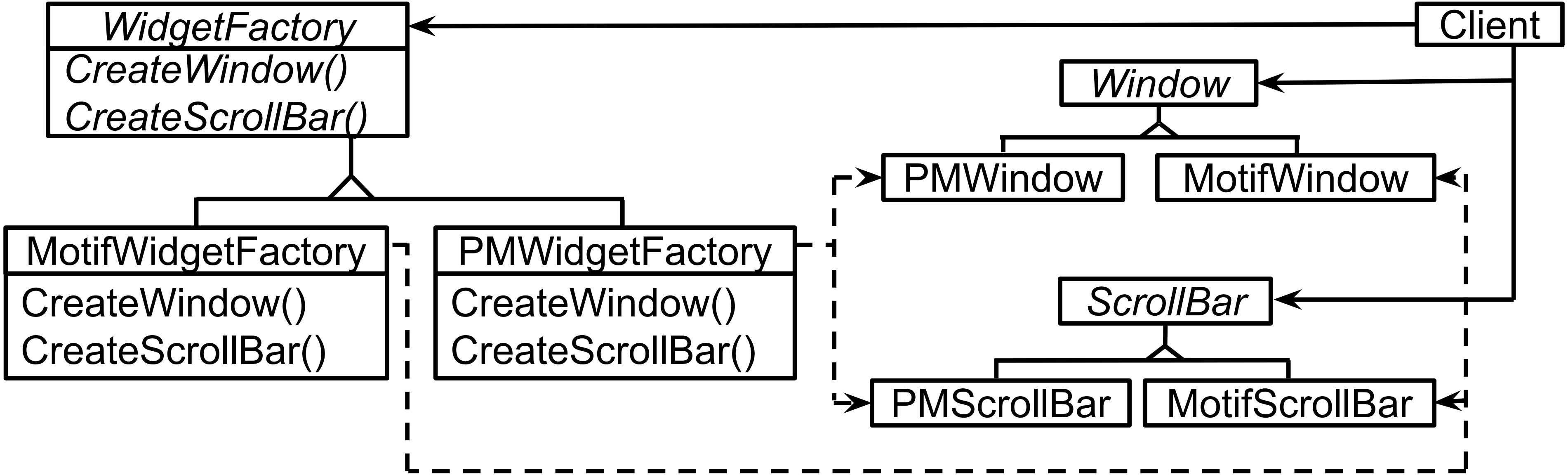
I. Creational Patterns

1. Abstract factory
2. Builder
3. Factory method
4. Prototype
5. Singleton

1. Abstract Factory

- Intent – allow creation of families of related objects independent of implementation
- Use case – look-and-feel in a GUI toolkit
  - Each L&F has its own windows, scrollbars, etc.
- Key types – *Factory* with methods to create each family member, *Products*
- Not common in JDK / JavaScript

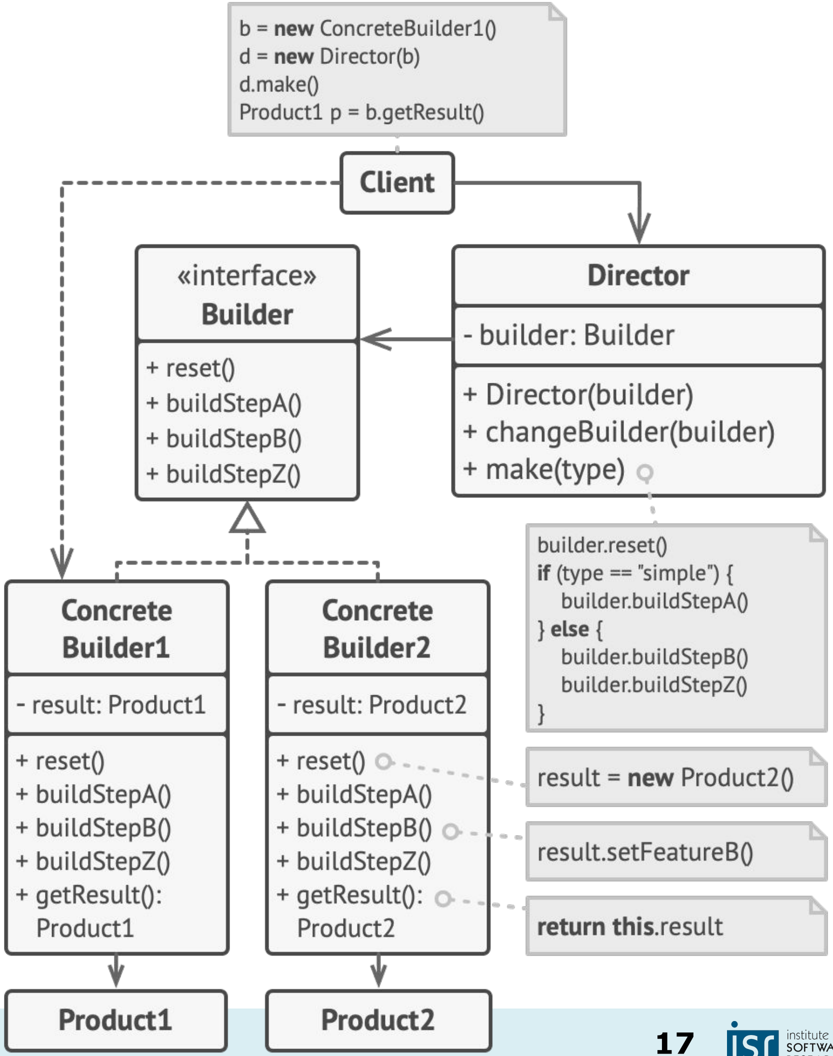
Abstract Factory Illustration



Builder Pattern

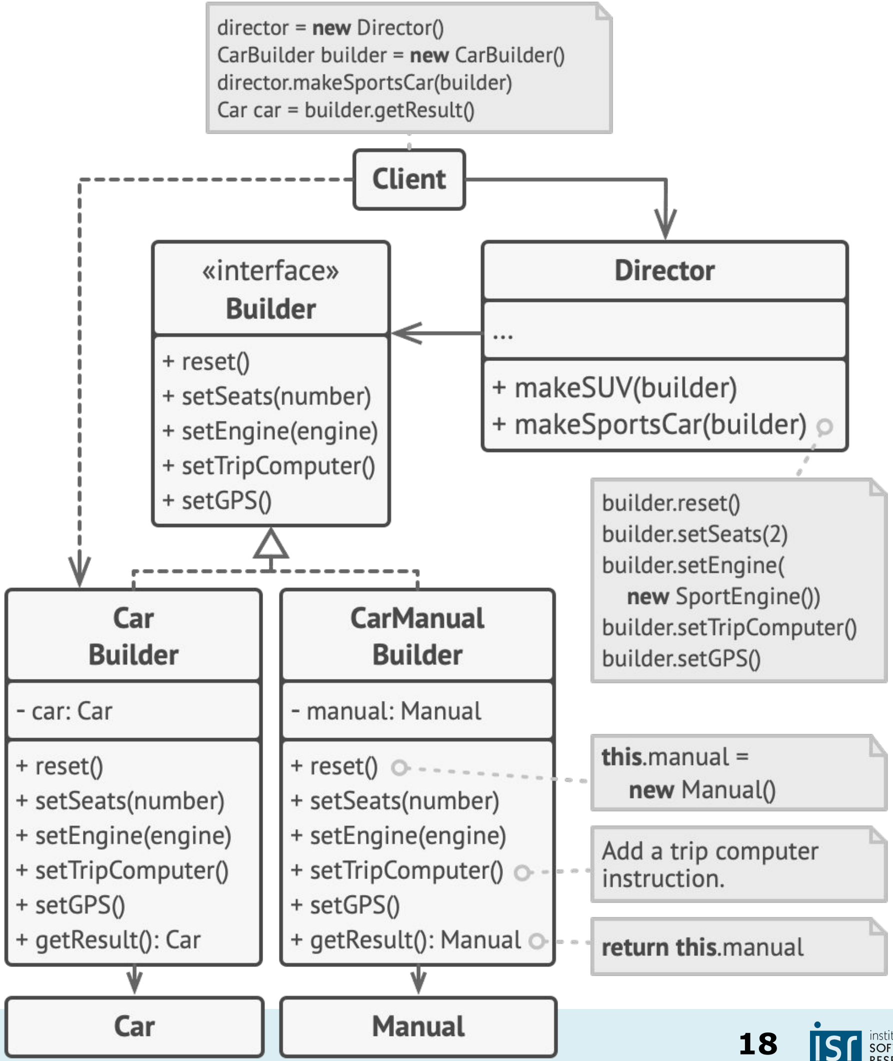
- Intent – separate construction of complex object from representation so same creation process can create different representations
- use case – converting rich text to various formats
- types – *Builder*, ConcreteBuilders, Director, Products
- `StringBuilder (Java), DirectoryBuilder (HW2)`

Gof4 Builder Illustration



https://refactoring.guru/design-patterns/builder

Builder Example



Builder Code Example

```java
NutritionFacts twoLiterDietCoke = new NutritionFacts.Builder(
    "Diet Coke", 240, 8).sodium(1).build();
public class NutritioanFacts {
    public static class Builder {
        public Builder(String name, int servingSize,
                int servingsPerContainer) { ... }
        public Builder totalFat(int val)     { totalFat = val; }
        public Builder saturatedFat(int val) { satFat = val; }
        public Builder transFat(int val)     { transFat = val; }
        public Builder cholesterol(int val)  { cholesterol = val; }
        ... // 15 more setters
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder) { ... }
}
```

Builder Discussion

- Emulates named parameters in languages that don't support them
- Emulates 2^n constructors or factories with n builder methods, by allowing them to be combined freely
- Cost is an intermediate (Builder) object

Recall: Factory Method Pattern

- Intent – abstract creational method that lets subclasses decide which class to instantiate
- Use case – creating documents in a framework
- Key types – *Creator*, which contains abstract method to create an instance
- Java: `Iterable.iterator()`
- Related *Static Factory pattern* is very common
  - Technically not a GoF pattern, but close enough, e.g. Integer.valueOf(int)

Factory Method Illustration

```java
public interface Iterable<E> {
    public abstract Iterator<E> iterator();
}

public class ArrayList<E> implements List<E> {
    public Iterator<E> iterator() { ... }
    ...
}

public class HashSet<E> implements Set<E> {
    public Iterator<E> iterator() { ... }
    ...
}
```

Static Factory Method Example

```
public DatabaseConnection {
    private DatabaseConnection(String address) { … }
    public static DatabaseConnection create
                        (String address) {
        //optional caching or checking…
        return new DatabaseConnection(address);
    }
}


~~c = new DatabaseConnection("localhost");~~
c = DatabaseConnection.create("localhost");
```

Prototype Pattern

- Intent – create an object by cloning another and tweaking as necessary
- Use case – writing a music score editor in a graphical editor framework
- Key types – *Prototype*
- Java: `Cloneable`, but avoid (except on arrays)
- JavaScript: Builtin language feature

Singleton Pattern

- Intent – ensuring a class has only one instance
- Use case – GoF say print queue, file system, company in an accounting system
  - **Compelling uses are rare** but they do exist
- Key types – Singleton
- Java: `java.lang.Runtime.getRuntime()`, `java.util.Collections.emptyList()`

Singleton Illustration

```java
public class Elvis {
    private static final Elvis ELVIS = new Elvis();
    public static Elvis getInstance() { return ELVIS; }
    private Elvis() { }
    ...
}
```

```javascript
const elvis = { … }
function getElvis() {

export { getElvis }
```

Singleton Discussion

Singleton = global variable

No flexibility for change or extension

Tends to be overused

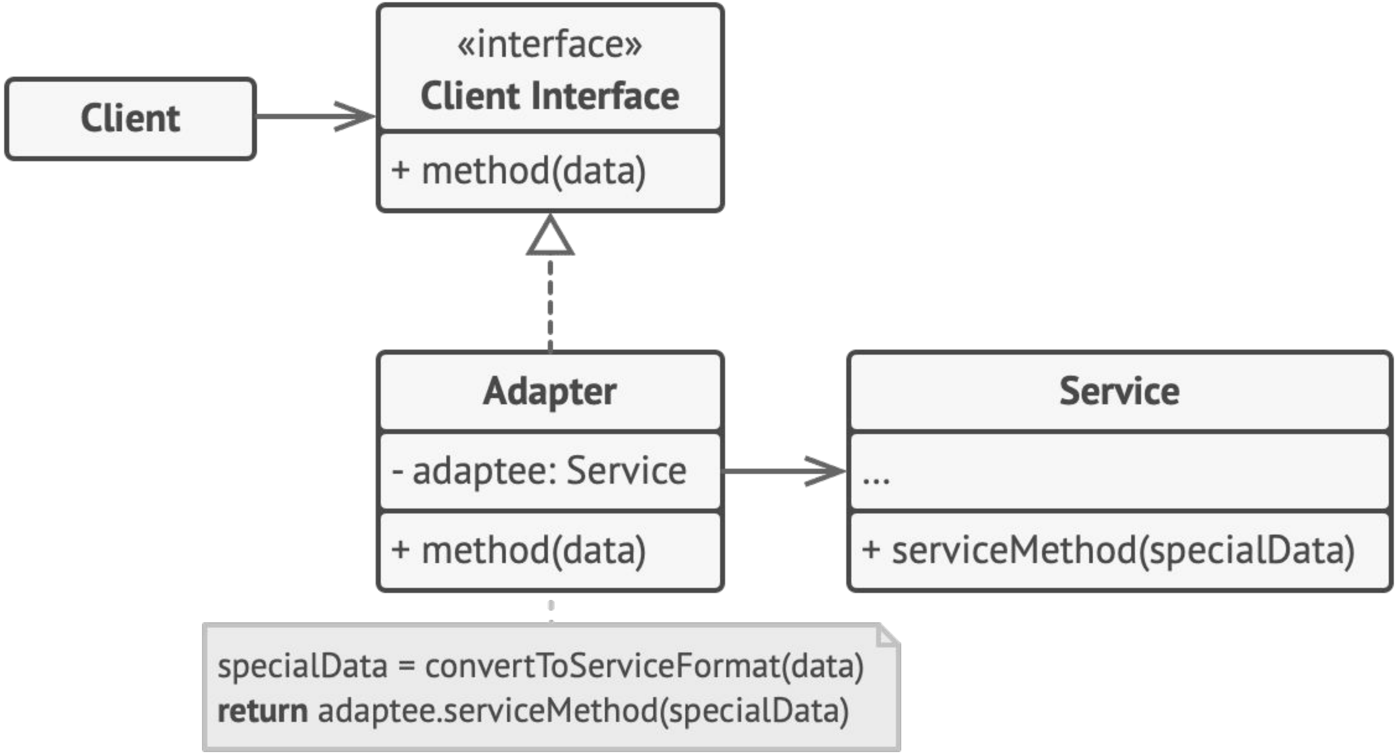
II. Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

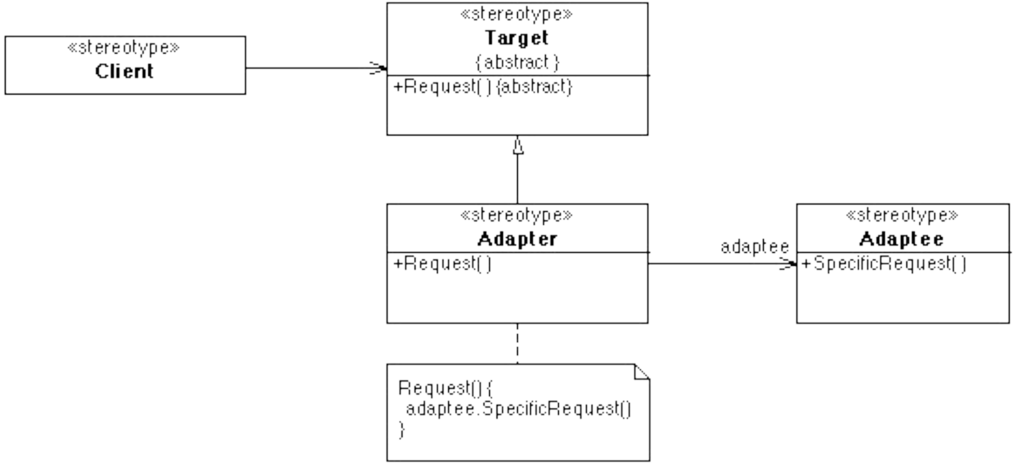
Recall: Adapter Pattern

- Intent – convert interface of a class into one that another class requires, allowing interoperability
- Use case – numerous, e.g., arrays vs. collections
- Key types – Target, Adaptee, Adapter
- JDK – `Arrays.asList(T[])`

Recall: The *Adapter* Design Pattern



Recall: The *Adapter* Design Pattern



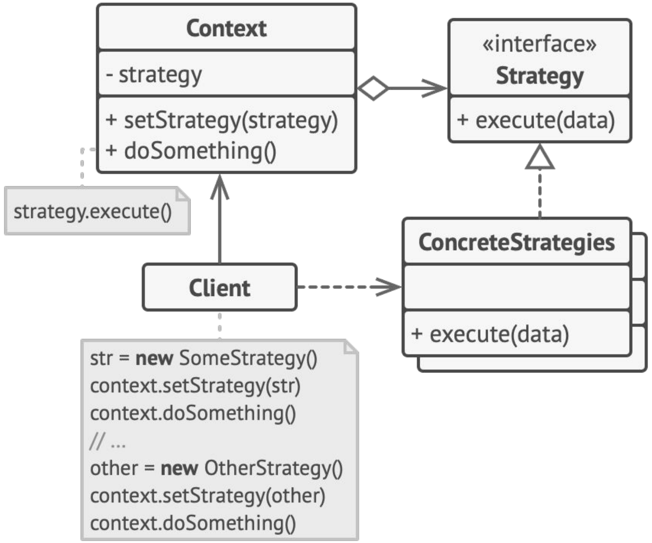
Applicability

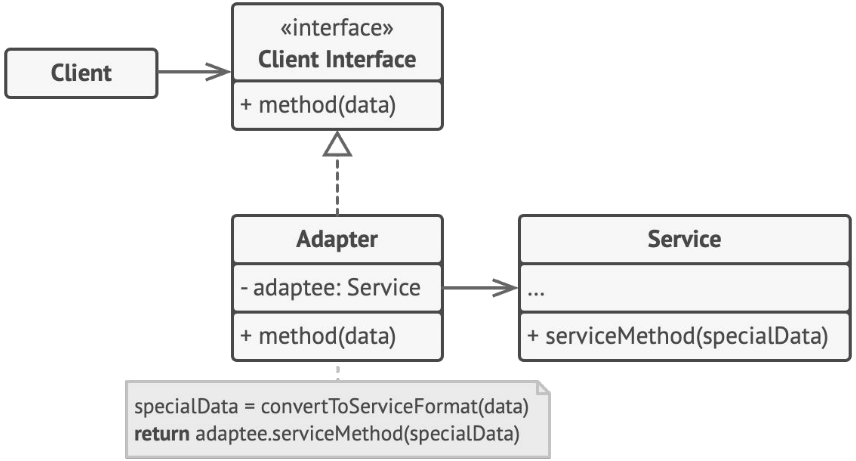
- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
- You need to use several subclasses, but it's impractical to adapt their interface by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

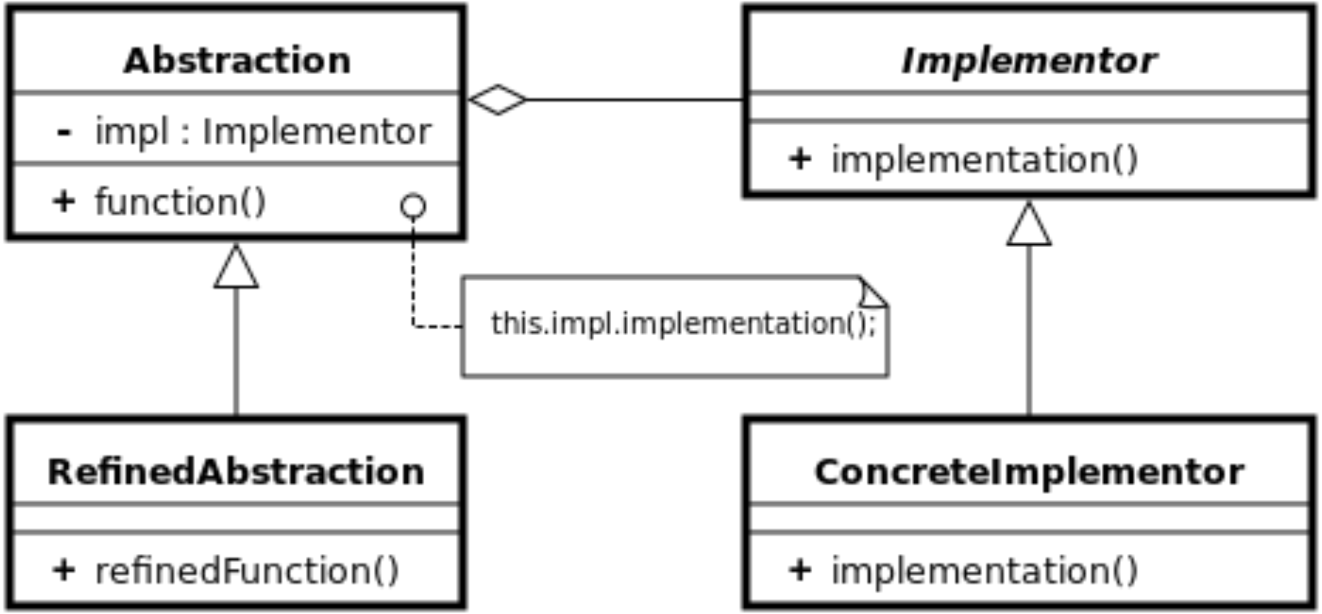
Adapter vs Strategy?





2. Bridge

- Intent – decouple an abstraction from its implementation so they can vary independently
- Use case – portable windowing toolkit
- Key types – Abstraction, *Implementor*
- Java: JDBC, Java Cryptography Extension (JCE), Java Naming & Directory Interface (JNDI)

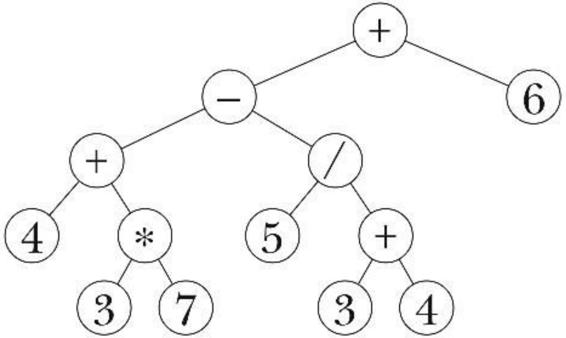


Bridge is very similar to Adapter: In Bridge you define both the abstract interface and the underlying implementation; you do not adapt to some legacy or third-party code. The goal is to swap out implementations, not to ensure compatibility after the fact.
In addition, abstraction and implementation can vary independently.

Recall: Composite Pattern

- Intent – compose objects into tree structures. **Let clients treat primitives & compositions uniformly.**
- Use case – GUI toolkit (widgets and containers)
- Key type – *Component* that represents both primitives and their containers
- Java: `javax.swing.JComponent`

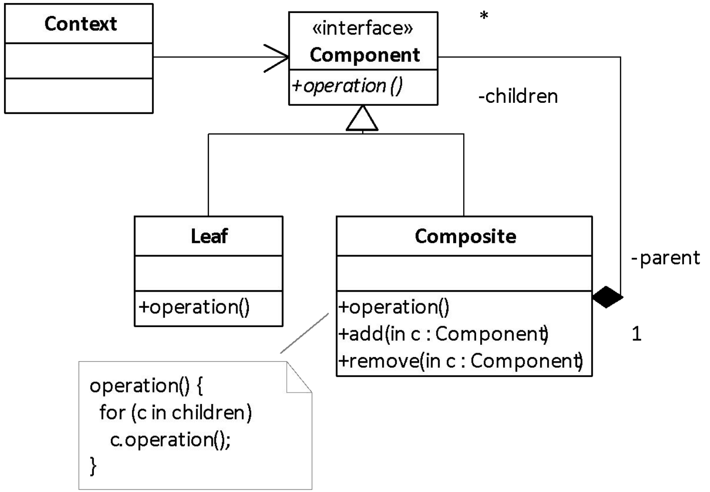
Composite Illustration




```java
public interface Expression {
    double eval();      // Returns value
    String toString(); // Returns infix expression string
}
public class UnaryOperationExpression implements Expression {
    public UnaryOperationExpression(
            UnaryOperator operator, Expression operand);
}
public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
            Expression operand1, Expression operand2);
}
public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
```

The Composite Design Pattern

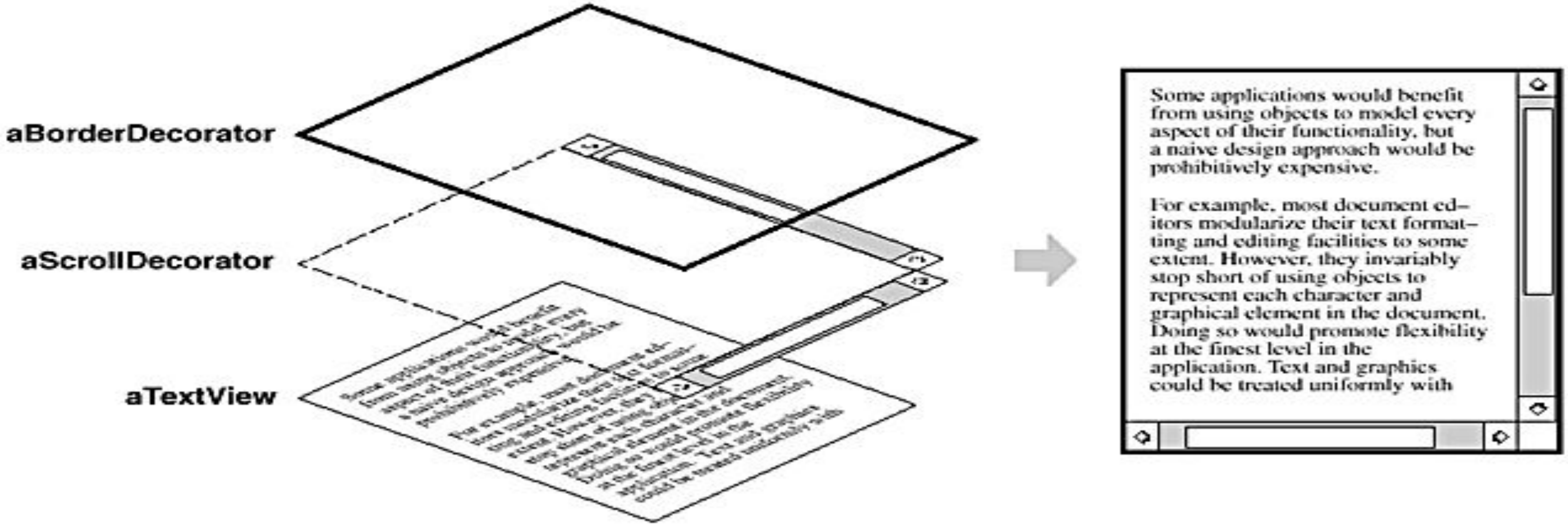
- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



Recall: Decorator Pattern

- Intent – attach features to an object dynamically
- Use case – attaching borders in a GUI toolkit
- Key types – *Component*, implement by decorator *and* decorated
- Java: Collections (e.g., `Synchronized` wrappers), `java.io` streams, Swing components

Decorator Illustration



Decorator

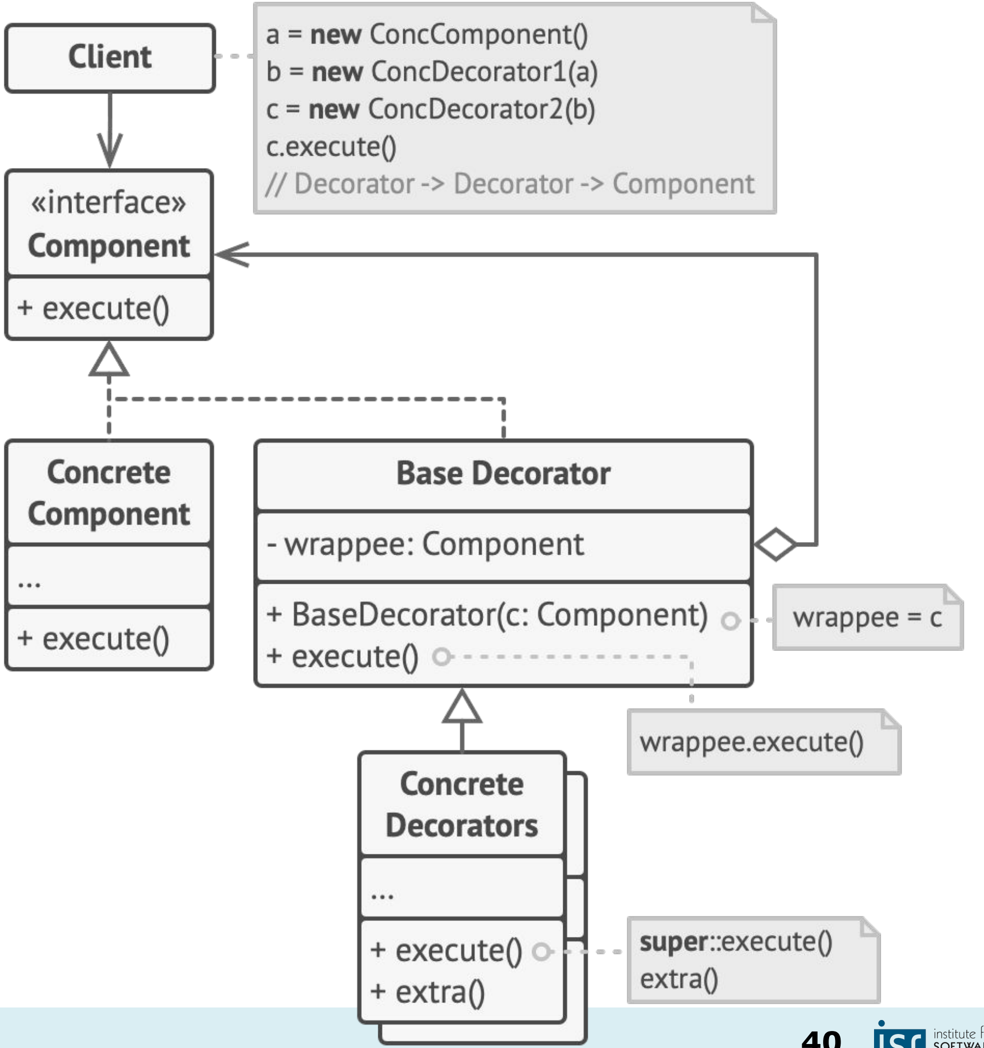
One or multiple base classes (same interface)

Any number of decorators

Adding decorators at runtime

Base decorator provides default forwarding logic

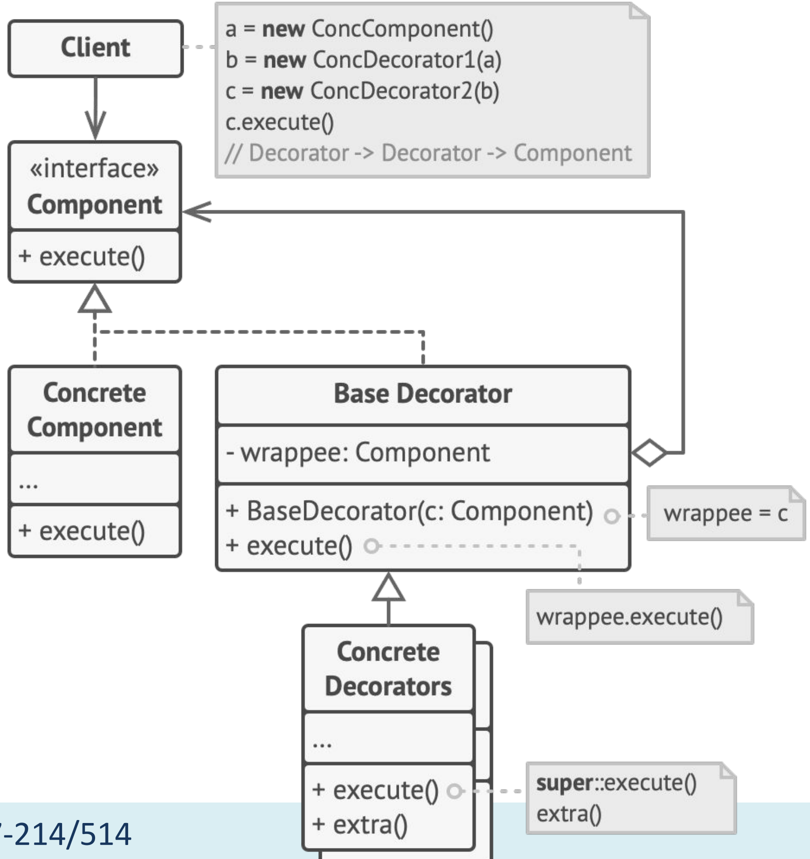
No open recursion

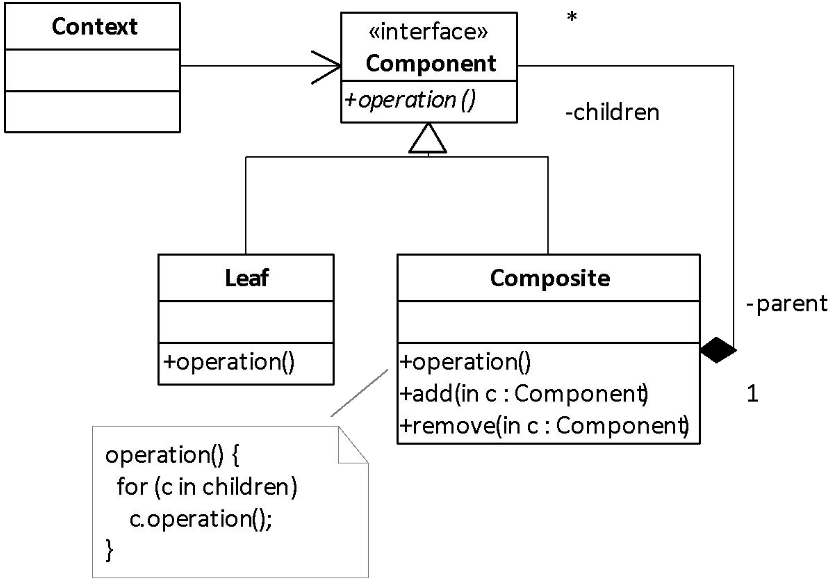


```
interface GameLogic {
     isValidMove(w, x, y)
     move(w, x, y)
}
class BasicGameLogic implements GameLogic {
     constructor(board) { … }
     isValidMove(w, x, y) { … }
     move(w, x, y) { … }
}
class AbstractGodCardDecorator implements GameLogic {
     readonly gl: GameLogic
     constructor(gameLogic) { this.gl = gameLogic }
     isValidMove(w, x, y} { return this.gl.isValidMove(w, x, y) }
     move(w, x, y} { return this.gl.move(w, x, y) }
}
class PanDecorator extends AbstractGodCardDecorator implements GameLogic {
     move(w, x, y} { /* this.gl.move(w, x, y) + checkWinner */ }
}
```


institute for SOFTWARE RESEARCH

Decorator vs Composite?





Decorator vs Strategy?

```
interface GameLogic {
     isValidMove(w, x, y)
     move(w, x, y)
}

class BasicGameLogic
     implements GameLogic { … }

class AbstractGodCardDecorator
     implements GameLogic { … }

class PanDecorator
     extends AbstractGodCardDecorator
     implements GameLogic { … }
```

```
interface GameLogic {
     isValidMove(w, x, y)
     move(w, x, y)
}
class BasicGameLogic
          implements GameLogic {
     constructor(board) { … }
     isValidMove(w, x, y) { … }
     move(w, x, y) { … }
}
class PanDecorator
          extends BasicGameLogic {
     move(w, x, y} { /* super.move(w,
x, y) + checkWinner */ }
}
```

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 2: City workers after inspecting a problem can mark the problem as high priority, as delegated, or in several other ways. Markers change how issues are shown (e.g., in reports).

Design Problem

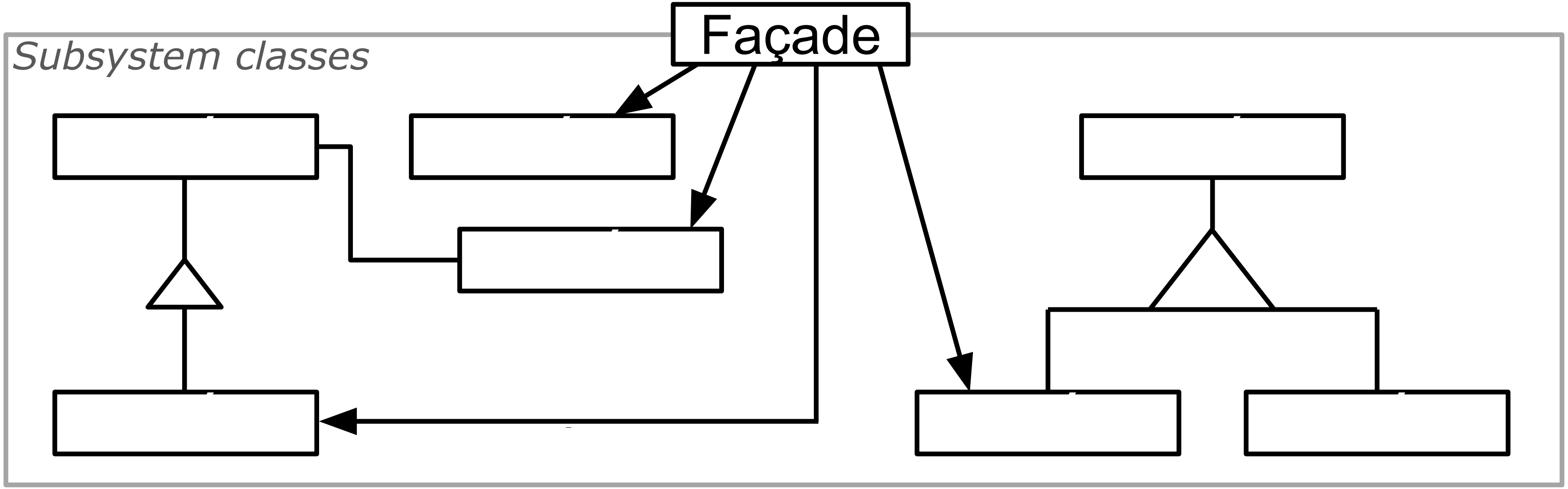
You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 3: You want to group problems that are related into a problem group with a new name, and those might be grouped again, but still count them directly. Those groups should still show up in reports and all scheduling activities.

Façade Pattern

- Intent – provide a simple unified interface to a set of interfaces in a subsystem
  - GoF allow for variants where the complex underpinnings are exposed and hidden
- Use case – any complex system; GoF use compiler
- Key types – Façade (the simple unified interface)
- JDK – `java.util.concurrent.Executors`

Façade Illustration



```
class SantoriniController {
    newGame() { … }
    isValidMove(w, x, y) { … }
    move(w, x, y) { … }
    getWinner() { … }
}
```

Discussion

Facade vs Controller Heuristic

- Same idea
- Facade for subsystem, controller for use case

Facade vs Singleton

- Facade sometimes a global variable
- Typically little design for change/extension

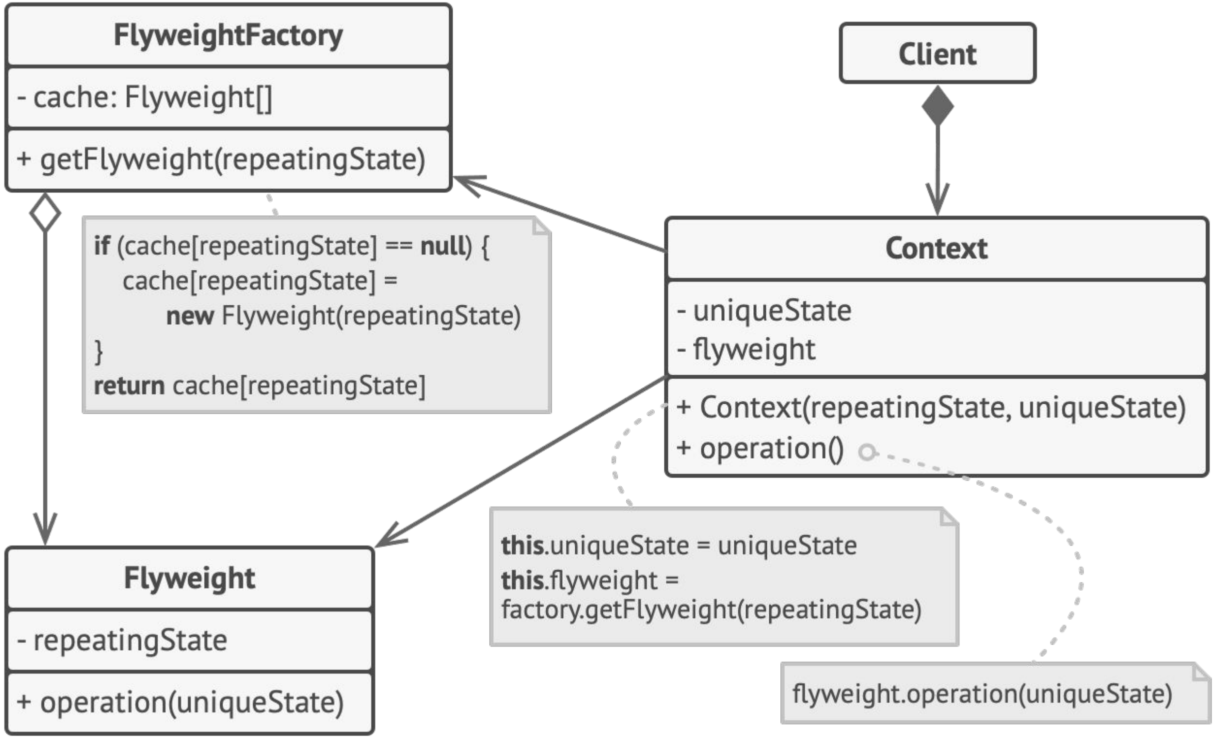
Flyweight Pattern

- Intent – use sharing to support large numbers of fine-grained objects efficiently
- Use case – characters in a document
- Key types – Flyweight (instance-controlled!)
    - Some state can be *extrinsic* to reduce number of instances
- Java: String literals (JVM feature), Integer
- “Hash Consing” in functional programming

Flyweight

Key idea: Avoid copies of structurally equal objects, reuse object

Requires immutable objects and factory with caching



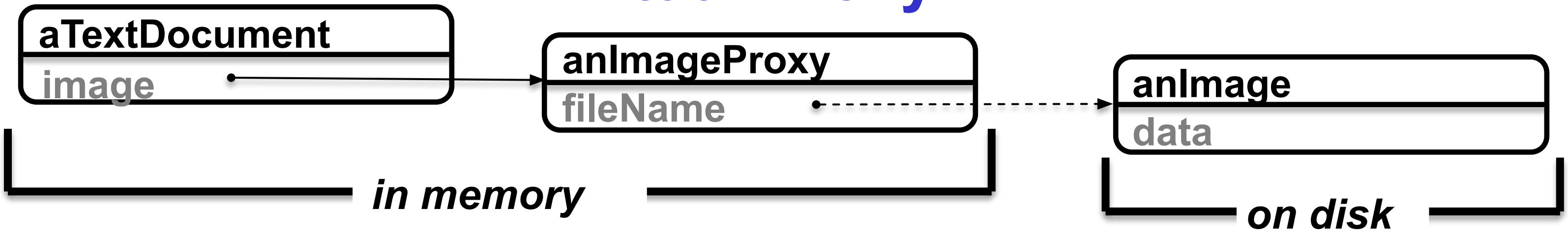
https://refactoring.guru/design-patterns/flyweight

Proxy Pattern

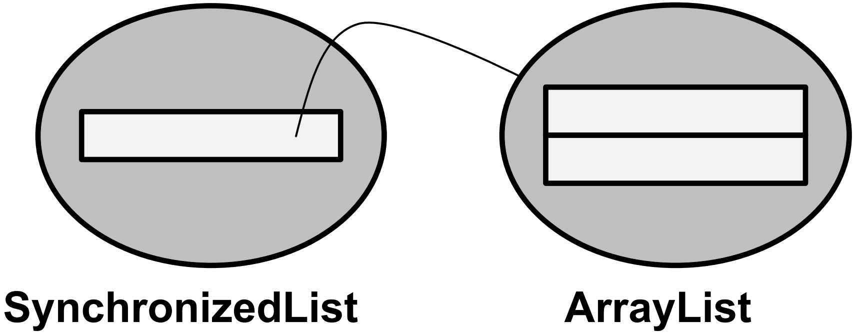
- Intent – surrogate for another object
- Use case – delay loading of images till needed
- Key types – *Subject*, Proxy, RealSubject
- Gof mention several flavors
  - virtual proxy – stand-in that instantiates lazily
  - remote proxy – local representative for remote obj
  - protection proxy – denies some ops to some users
  - smart reference – does locking or ref. counting, e.g.
- JDK – RMI, collections wrappers

Proxy Illustrations

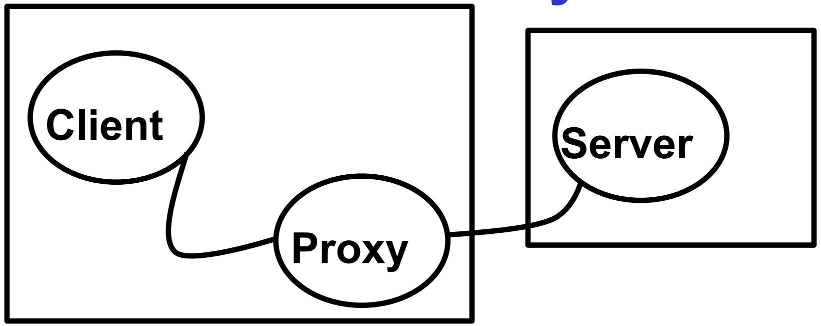
Virtual Proxy



Smart Reference

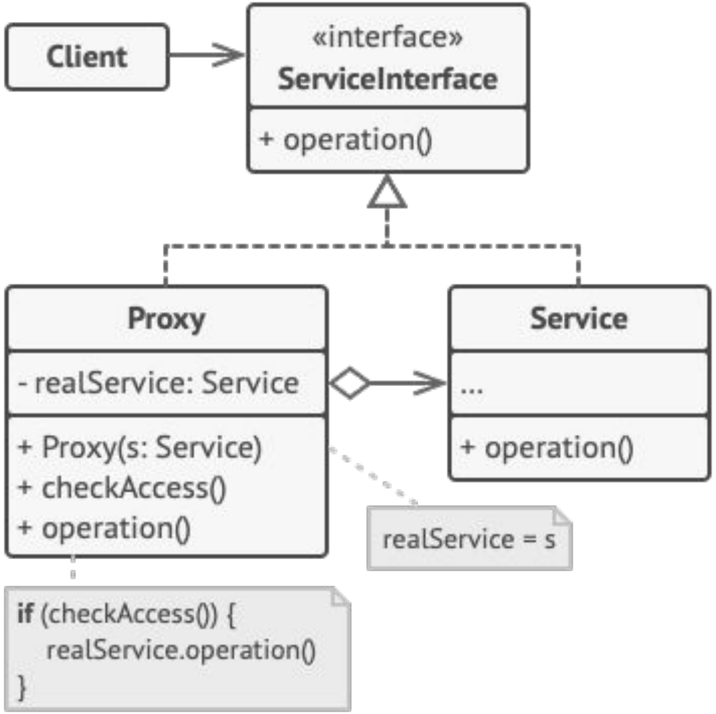


Remote Proxy



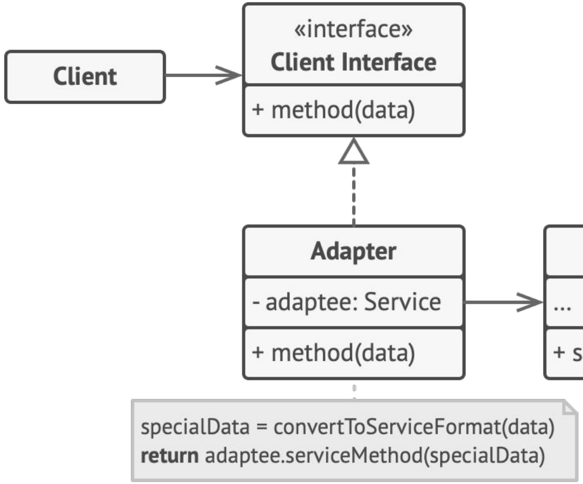
Proxy Design Pattern

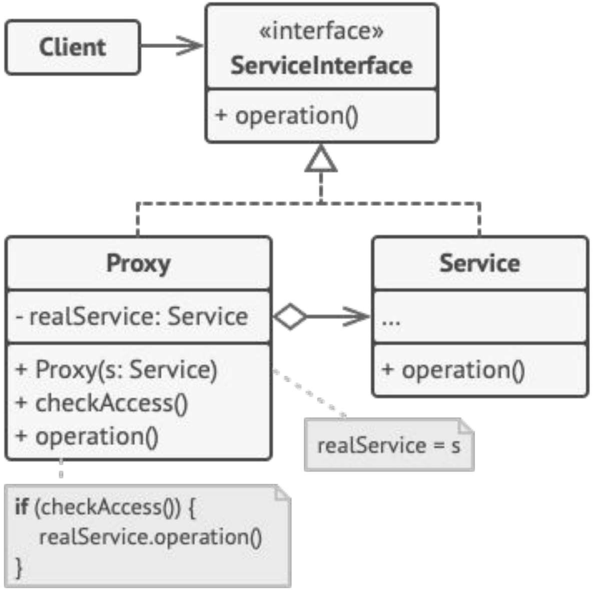
- Local representative for remote object
  - Create expensive obj on-demand
  - Control access to an object
- Hides extra "work" from client
  - Add extra error handling, caching
  - Uses *indirection*



https://refactoring.guru/design-patterns/proxy

Proxy vs Adapter?





Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 4: Some problems point to large pictures stored in another database and you do not want to keep them in memory, but load them only when needed.

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

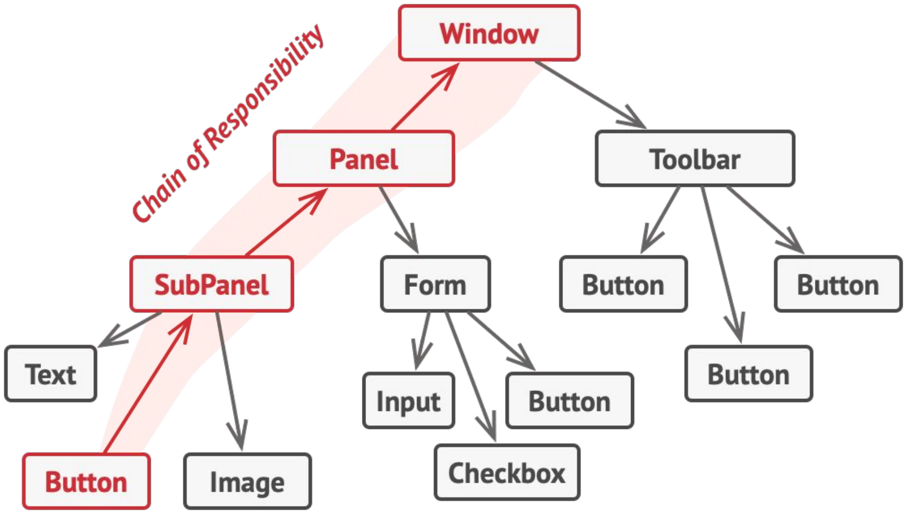
Design problem 5: The county has a different system that records potholes in a different format. You want to include them in your reports regardless.

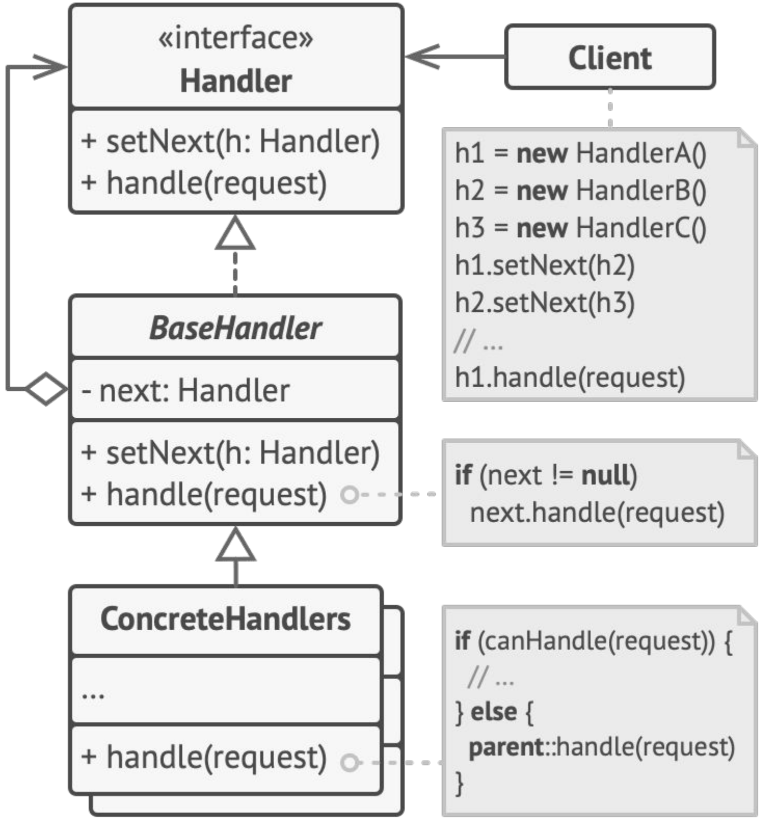
III. Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template method
11. Visitor

Chain of Responsibility Pattern

- Intent – avoid coupling sender to receiver by passing request along until someone handles it
- Use case – context-sensitive help facility
- Key types – *RequestHandler*
- JDK – `ClassLoader`, `Properties`
- Exception handling could be considered a form of Chain of Responsibility pattern

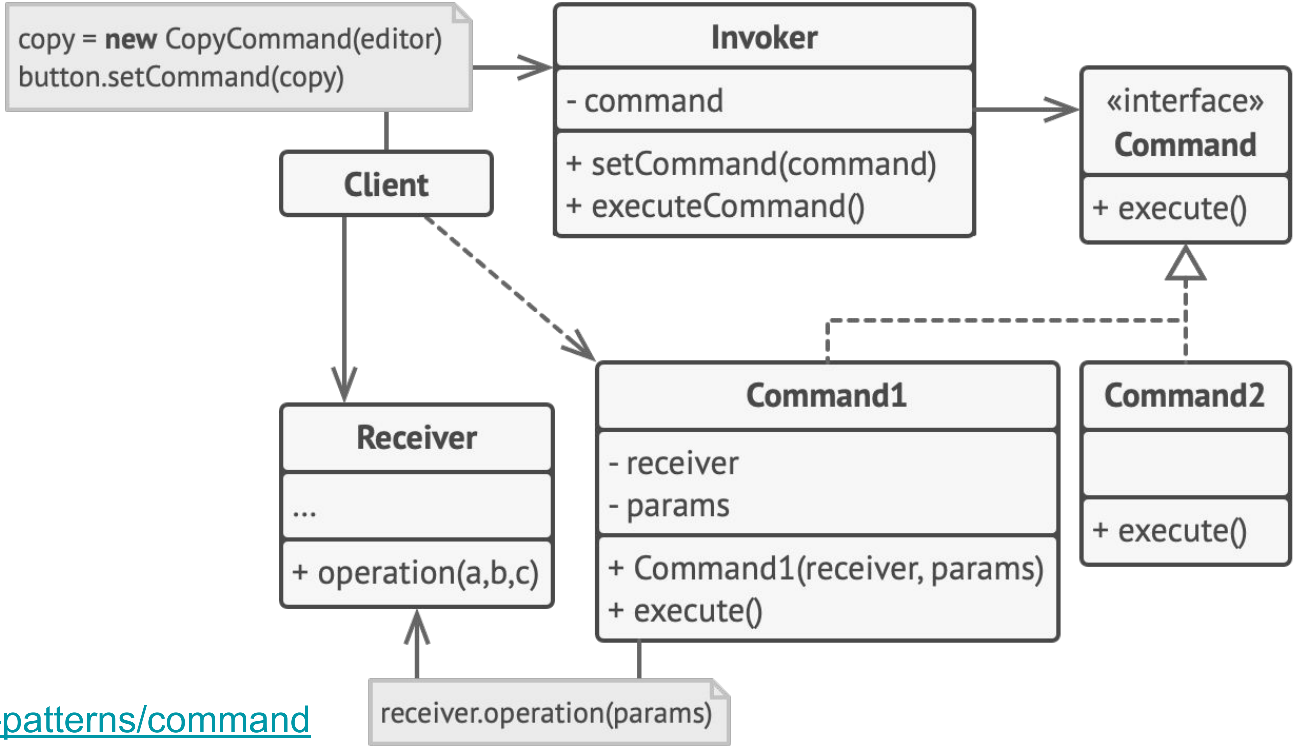

Chain of Responsibility
Window
Panel
Toolbar
SubPanel
Form
Button
Button
Text
Input
Button
Button
Button
Image
Checkbox


«interface»
Handler
+ setNext(h: Handler)
+ handle(request)
Client
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
BaseHandler
- next: Handler
+ setNext(h: Handler)
+ handle(request)
if (next != null)
next.handle(request)
ConcreteHandlers
...
+ handle(request)
if (canHandle(request)) {
// ...
} else {
parent::handle(request)
}

Command Pattern

- Intent – encapsulate a request as as an object, letting you parameterize one action with another, queue or log requests, etc.
- Use case – menu tree
- Key type – *Command* (`Runnable`)
- JDK – Common! Executor framework, etc. -- see higher order function
- Is it Command pattern if you run it repeatedly? If it takes an argument? Returns a val?

Command Pattern



https://refactoring.guru/design-patterns/command

Command Illustration

```
class ClickAction {
    constructor(name) { this.name = name }
    execute() { /* … update based on click event */ }
}
let c = new ClickAction("Restart Game")
getElementById("menu").addEventListener("click", c.execute)
getElementById("btn").addEventListener("click", c.execute)
setTimeout(c.execute, 2000)
```

Object (or function) represents an action, execution deferred, arguments possibly configured early. Can be reused in multiple places. Can be queued, logged, ...

Interpreter Pattern

- Intent – given a language, define class hierarchy for parse tree, recursive method to interpret it
- Use case – regular expression matching
- Key types – *Expression*, *NonterminalExpression*, *TerminalExpression*
- JDK – no uses I'm aware of
- Necessarily uses Composite pattern!

Iterator Pattern

- Intent – provide a way to access elements of a collection without exposing representation
- Use case – collections
- Key types – *Iterable*, *Iterator*
  - But GoF discuss internal iteration, too
- Java and JavaScript: collections, for-each statement ..

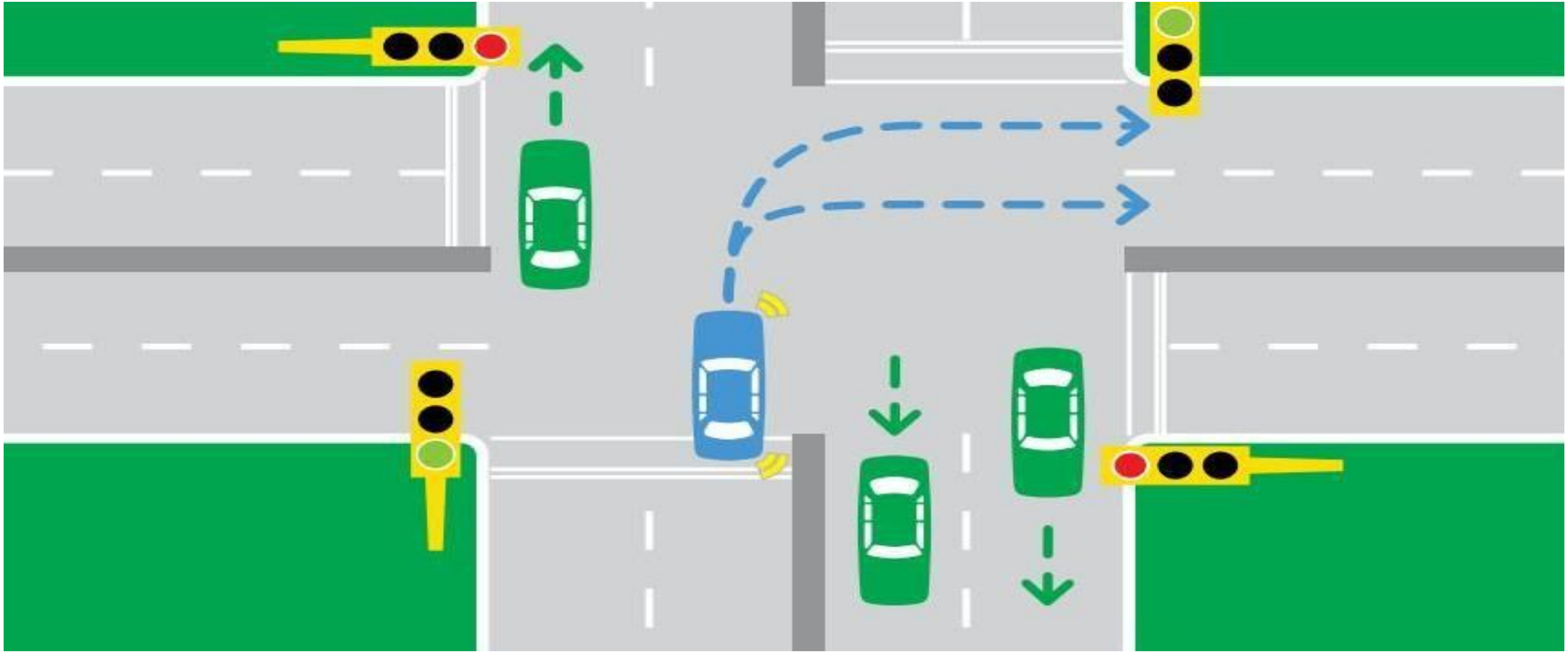
Iterator Illustration

```
public interface Iterable<E> {
    public abstract Iterator<E> iterator();
}
public class ArrayList<E> implements List<E> {
    public Iterator<E> iterator() { ... }
    ...
}
public class HashSet<E> implements Set<E> {
    public Iterator<E> iterator() { ... }
    ...
}
Collection<String> c = ...;
for (String s : c)  // Creates an Iterator appropriate to c
    System.out.println(s);
```

Mediator Pattern

- Intent – define an object that encapsulates how a set of objects interact, to reduce coupling.
  - $\mathcal{O}(n)$ couplings instead of $\mathcal{O}(n^2)$
- Use case – dialog box where change in one component affects behavior of others
- Key types – Mediator, Components
- JDK – Unclear

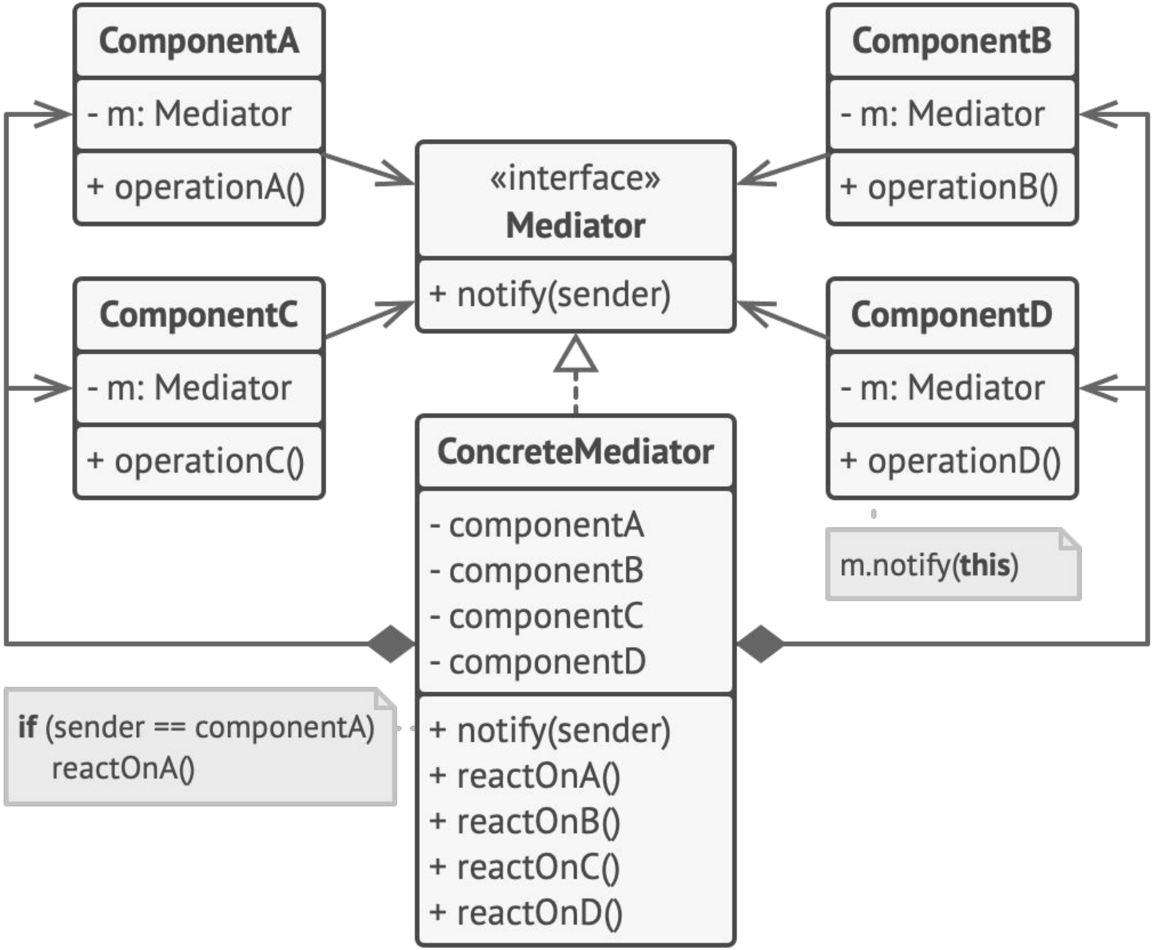
Mediator Illustration



Single responsibility at mediator

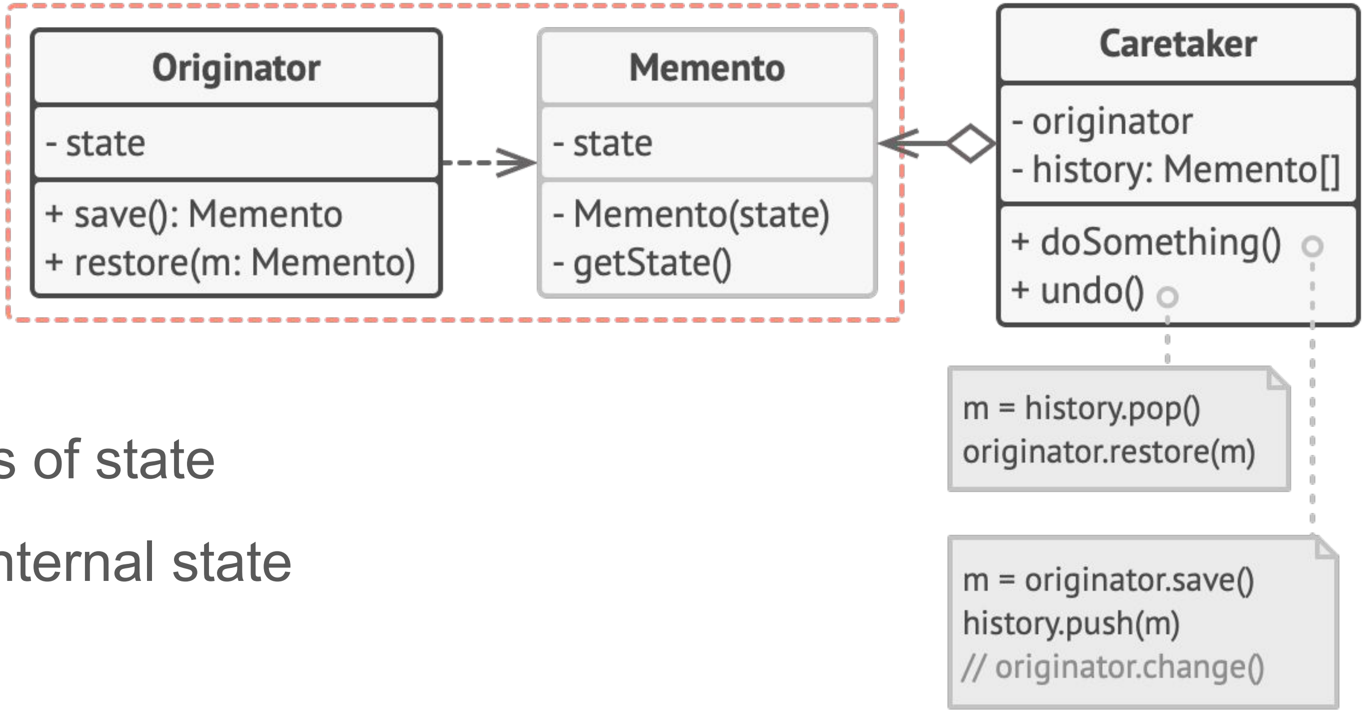
Coupling to single component

God object?



Memento Pattern

- Intent – without violating encapsulation, allow client to capture an object’s state, and restore
- Use case – undo stack for operations that aren’t easily undone, e.g., line-art editor
- Key type – Memento (opaque state object)
- JDK – none that I’m aware of (*not* serialization)



Record snapshots of state

Avoid access to internal state

Allows undo

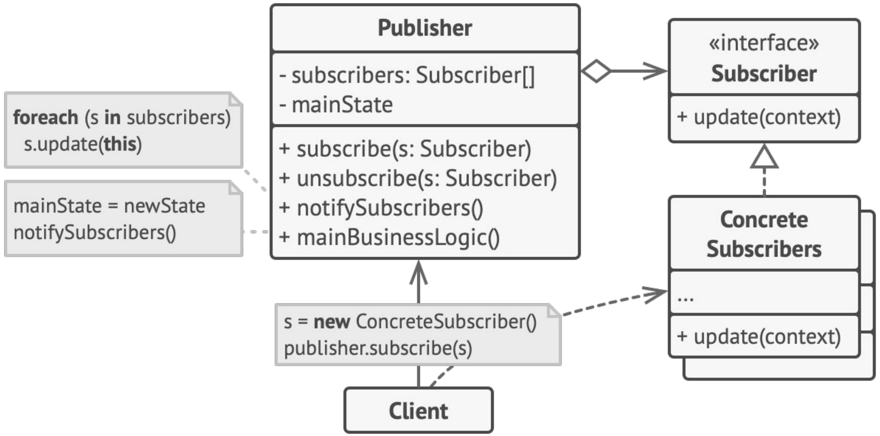
Consider using immutable objects to begin with

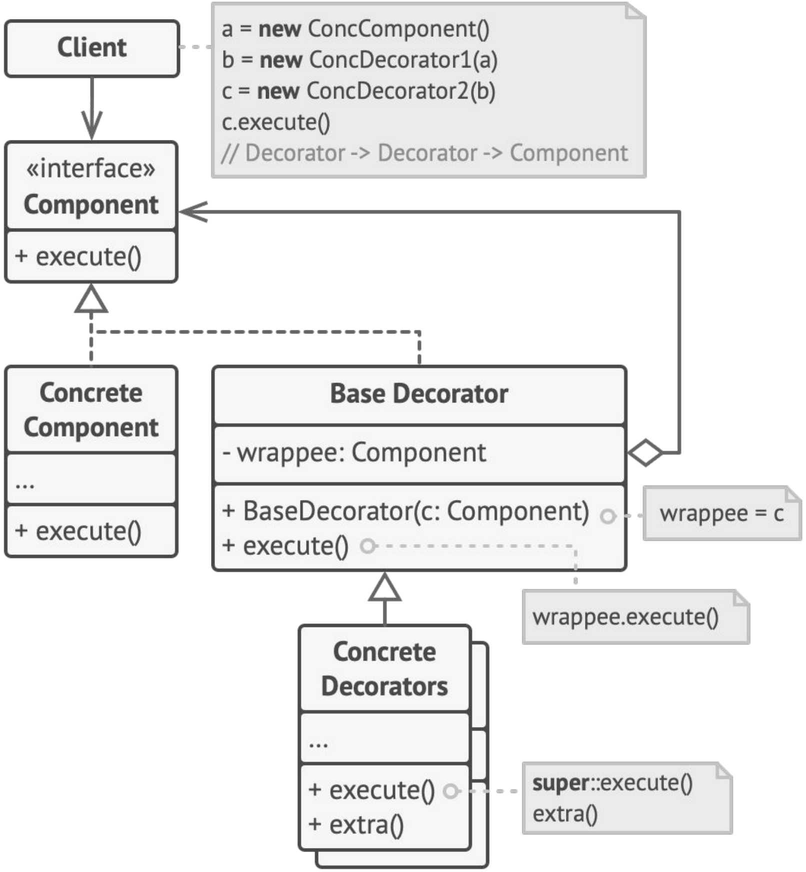
Observer Pattern

- Intent – let objects observe the behavior of other objects so they can stay in sync
- Use case – multiple views of a data object in a GUI
- Key types – *Subject* (“Observable”), *Observer*
    - GoF are agnostic on many details!
- Examples: All GUIs

Observer vs. Promise

Observer vs. Decorator





Observer vs Generator

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 6: Every time a report is resolved, one of multiple actions should be taken (email, text message, …). The action is selected by the person creating the report.

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 7: Every time a report is resolved, multiple follow-up actions should be performed. Results should be added to a database, an email should be sent, a supervisor should be informed, etc. More actions might be added later.

Observer Characteristics

Inversion of control, remove direct dependency, reduce coupling

Listen to events, multiple events

Multiple observers possible

Add and remove observers at runtime

Push model/event-based programming: Observable pushes events to observer

State Pattern

- Intent – allow an object to alter its behavior when internal state changes. “Object will appear to change class.”
- Use case – TCP Connection (which is stateful)
- Key type – *State* (Object delegates to state!)
- JDK: none that I’m aware of, but easy to use

State Example

Without the pattern:

```
class Connection {
  boolean isOpen = false;
  void open() {
    if (isOpen) throw new Inval…
    …//open connection
    isOpen=true;
  }
  void close() {
    if (!isOpen) throw new Inval…
    …//close connection
    isOpen=false;
  }
}
```

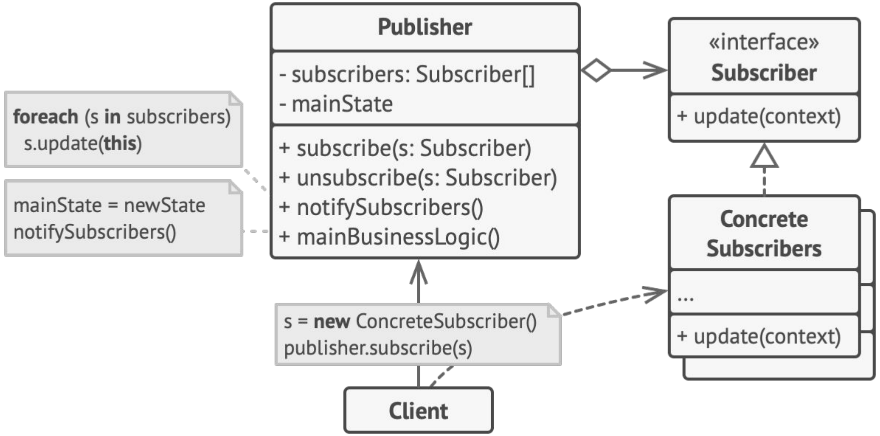
With the pattern:

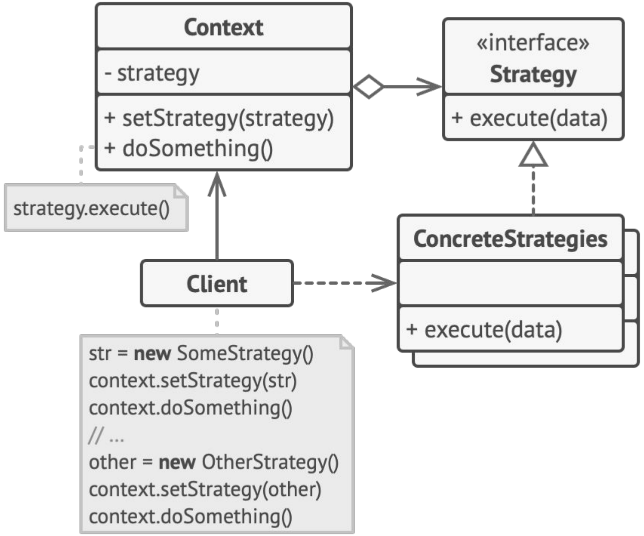
```
class Connection {
  private State state = new Closed();
  public void setState(State s) { … }
  void open() { state.open(this); }
  …
}
interface State {
   void open(Connection c);
   void close(Connection c);
}
class Open implements State {
   void open(Connection c) { throw …}
   void close(Connection c) {
     //…close connection
     c.setState(new Closed());
   }
}
class Closed impl. State { … }
```

Strategy Pattern

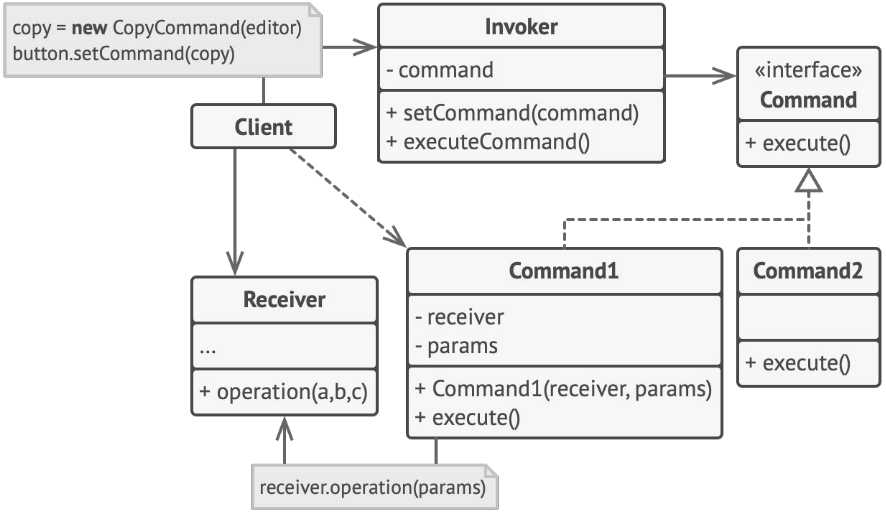
- Intent – represent a behavior that parameterizes an algorithm for behavior or performance
- Use case – line-breaking for text compositing
- Key types – *Strategy*
- JDK – `Comparator`

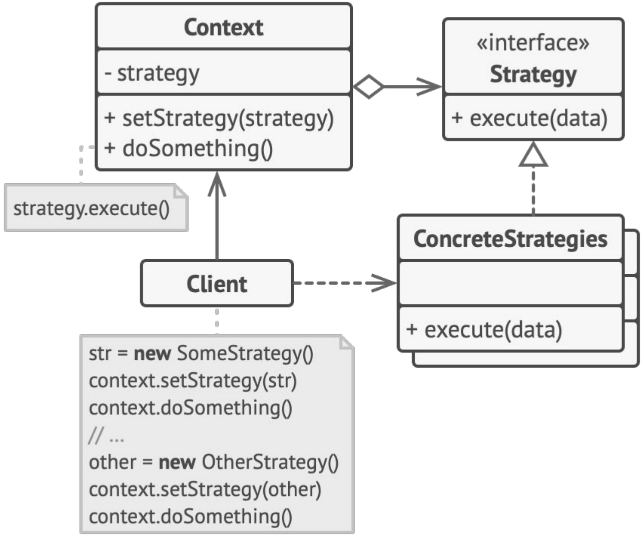
Observer vs. Strategy





Command vs. Strategy



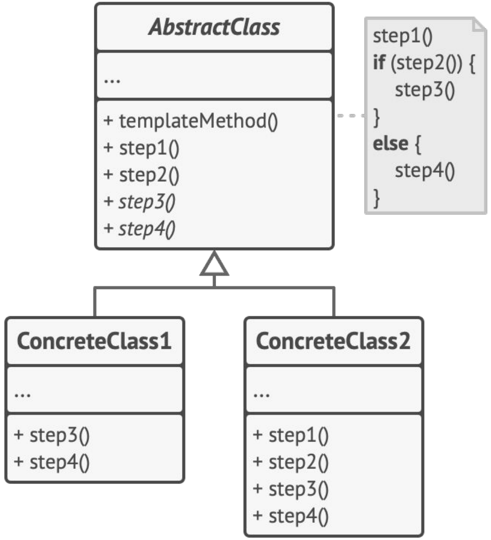


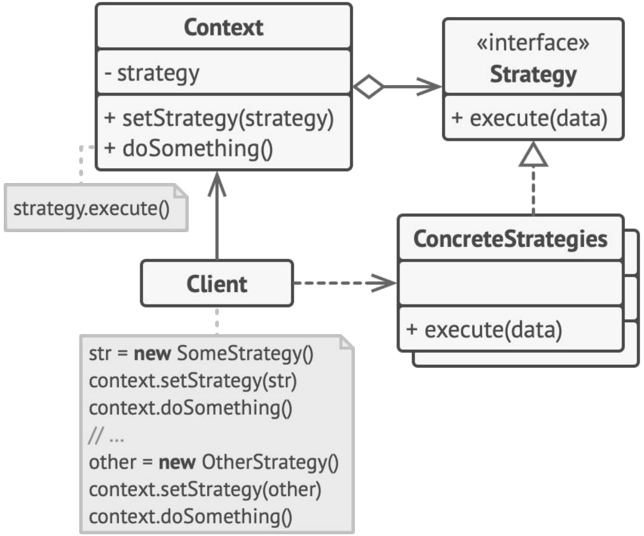
Very similar structure, but different intentions: Command is reusable, delayed function; strategy configures part of algorithm

Template Method Pattern

- Intent – define skeleton of an algorithm or data structure, deferring some decisions to subclasses
- Use case – application framework that lets plugins implement all operations on documents
- Key types – *AbstractClass*, ConcreteClass
- JDK – skeletal collection impls (e.g., `AbstractList`)

Strategy vs Template Method





Strategy vs Template Method

Delegation vs inheritance; context ~~ template method

Template method: Single variation point, configured with constructor call

Strategy: Possibly multiple variation points in context, configured during constructor or dynamically later

Visitor Pattern

- Intent – represent an operation to be performed on elements of an object structure (e.g., a parse tree). Visitor lets you define a new operation without modifying the type hierarchy.
- Use case – type-checking, pretty-printing, etc.
- Key types – *Visitor*, ConcreteVisitors, all the element types that get visited
- JDK – SimpleFileVisitor; AnnotationValueVisitor; very common in compilers

Visitor Pattern Discussion

Double dispatch

Add new operations (like Command pattern)

Iterate over object structure (like Iterator pattern)

Provide object-specific visit methods to avoid dynamic type lookup

Most commonly used in context of compilers and other operations on trees

Different versions exist

Bonus: Other Design Principles

Where we are

	Small scale: One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
Design for understanding change/ext. reuse robustness ...	**Subtype Polymorphism** ✓ **Information Hiding, Contracts** ✓ Immutability ✓ Types Unit Testing ✓	Domain Analysis ✓ **Inheritance & Del.** ✓ **Responsibility Assignment**, Design Patterns, Antipattern ✓ Promises/ Reactive P. ✓ Integration Testing ✓	GUI vs Core ✓ Frameworks and Libraries ✓, APIs ✓ Module systems, microservices ✓ Testing for Robustness ✓ CI ✓, DevOps, Teams

SOLID Principles

Single-responsibility principle: Every class should have only one responsibility -- *cohesion; low coupling; information expert*

The Open–closed principle: "Software entities ... should be open for extension, but closed for modification." -- *encapsulation*

Liskov substitution principle: Program against interface, even with subclassing

Interface segregation principle: Prefer specific small interfaces; multiple interfaces per object okay; cohesion

Dependency inversion principle: "Depend upon abstractions, [not] concretions." -- *prefer interfaces over class types; dynamic dispatch*

Other Common Principles

DRY Principle: Don't Repeat Yourself

KISS Principle: Keep It Simple, Stupid

YAGNI Principle: You Aren't Gonna Need It

Principle of Least Astonishment

Boy Scout Rule: Leave the Code Cleaner than you Found it

Summary

- Now you know all the Gang of Four patterns
- Definitions can be vague
- Coverage is incomplete
- But they're extremely valuable
    - They gave us a vocabulary
    - And a way of thinking about software
- Look for patterns as you read and write software
    - GoF, non-GoF, and undiscovered