Principles of Software Construction: Objects, Design, and Concurrency

A Quick Tour of all 23 GoF Design Patterns

Claire Le Goues **Bogdan Vasilescu**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

But first:

Last of static analysis

We left off: …Google had done experiments with FindBugs, and none of them had worked!

1. **A dashboard:** run FindBugs overnight, report results in a centralized location
   Failed because: dashboard is outside the developer's workflow
2. **Recurring FixIt events:** company-wide one-week effort to fix warnings
   Failed because: FindBugs is too imprecise (44% of issues were "bugs", but only 16% mattered)
3. **Add to Code Review:** run on every change, allow toggling warnings
   Failed because: too imprecise and inconsistent

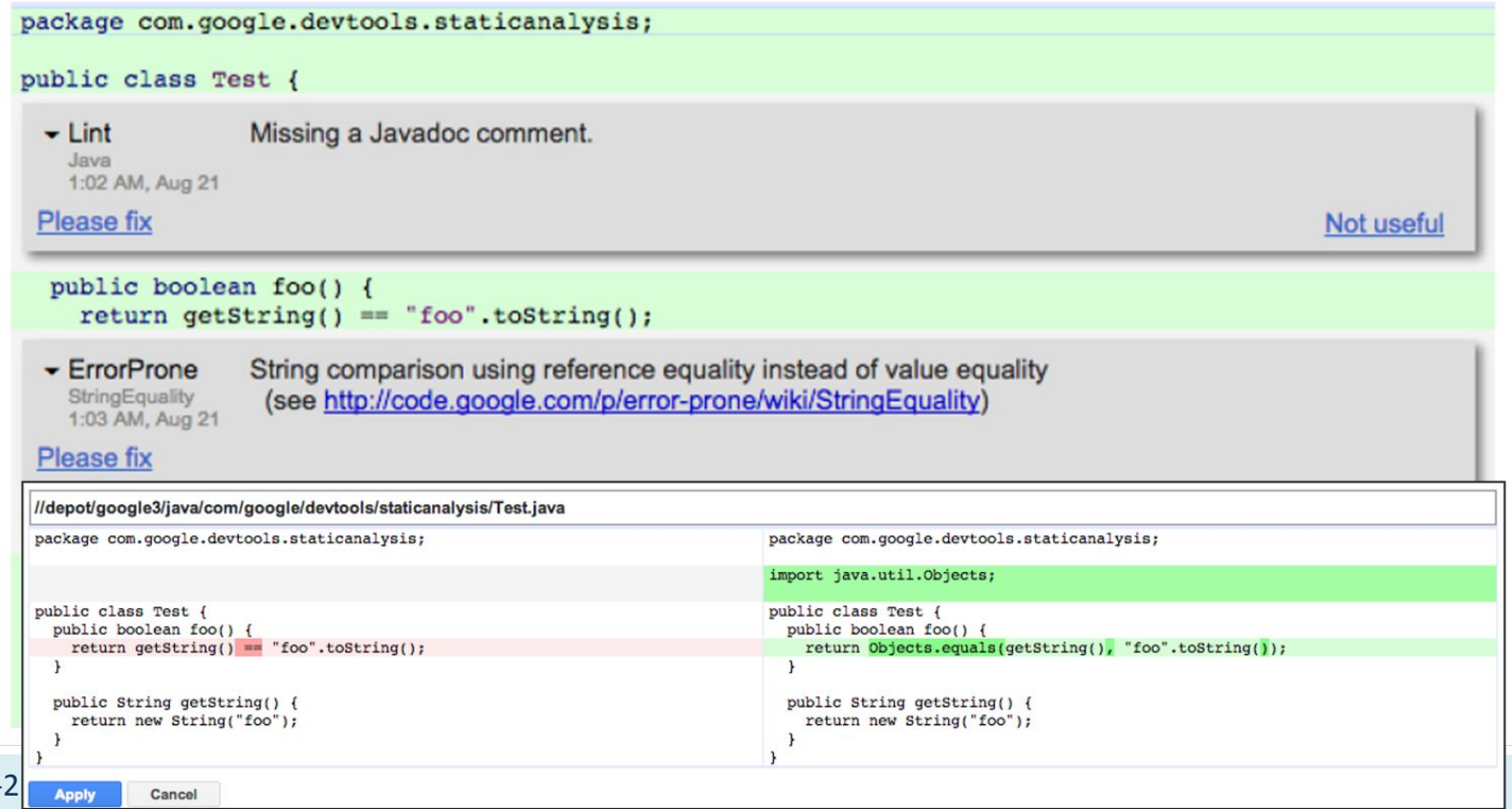
What went wrong / what do we need?

1. Precision is key -- developers lose faith in inaccurate tools
2. Provide timely warnings -- in-IDE or rapidly on builds
   a. Checkers are way more useful during coding
3. Make a platform -- allow adding useful checks

Google built its own tools for compile and review time

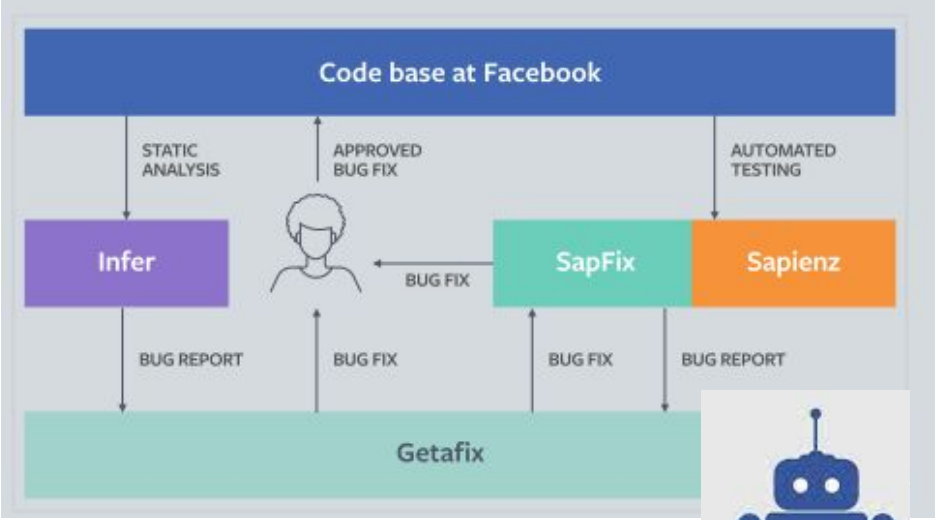
- At compile-time:
  - Perfectly Precise: only run checkers that have **NO** false-positives; never halt a build incorrectly
  - Simple and actionable, ideally to the point of **auto-fix suggestions**
- At review time: TriCoder
  - Must be 90%+ precise; if not checker gets disabled and it's on the checker authors to fix it.
  - Actionable, but may require some work; improve correctness or quality
  - Some compile-time checks moved to review-time!
- Ran 50K times per day -- in 2018

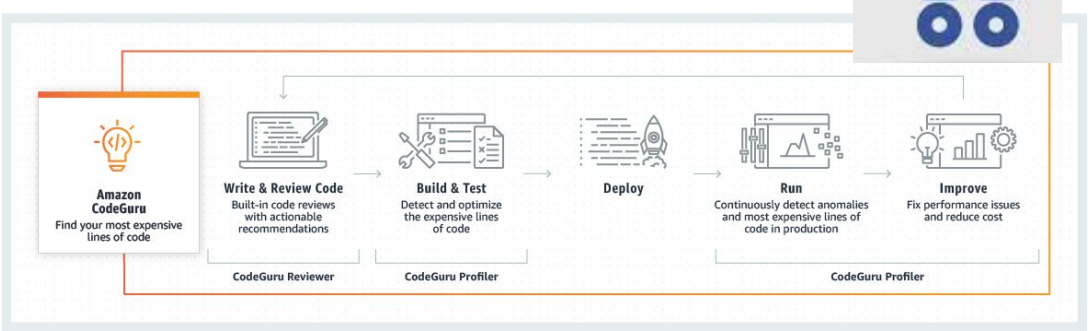
TriCoder


package com.google.devtools.staticanalysis;
public class Test {
Lint
Java
1:02 AM, Aug 21
Missing a Javadoc comment.
Please fix
Not useful
public boolean foo() {
return getString() == "foo".toString();
ErrorProne
StringEquality
1:03 AM, Aug 21
String comparison using reference equality instead of value equality
(see http://code.google.com/p/error-prone/wiki/StringEquality)
Please fix
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
package com.google.devtools.staticanalysis;
public class Test {
public boolean foo() {
return getString() == "foo".toString();
}
public String getString() {
return new String("foo");
}
}
package com.google.devtools.staticanalysis;
import java.util.Objects;
public class Test {
public boolean foo() {
return Objects.equals(getString(), "foo".toString());
}
public String getString() {
return new String("foo");
}
}
Apply
Cancel

What else could we do?

- Use more complicated logic
  - One example: Infer, at Facebook (Google claims this won't (easily) scale to their mono-repo.)
- Use AI?
  - Facebook: Getafix, also integrates with SapFix
  - Amazon: CodeGuru
  - Microsoft: IntelliSense in VSCode, mostly refactoring/code completion, trained on large volumes of code
  - Mostly fairly simple ML (details limited)



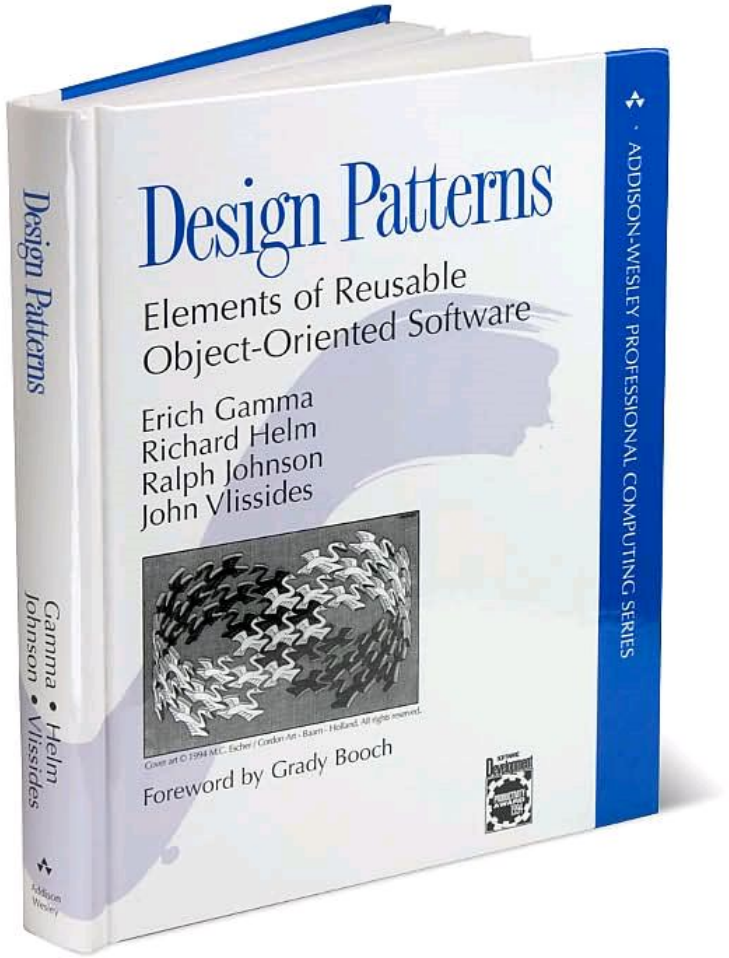


Summary

- We all constantly make mistakes
  - Static analysis captures common issues
  - Choose suitable abstractions; consider trade-offs
    - E.g., dynamic vs. static typing; sound vs. precise
- At big-tech-scale, automated checks are key
  - Help normalize coding standards
  - Even rare bugs are common at scale
  - But: social factors are very important

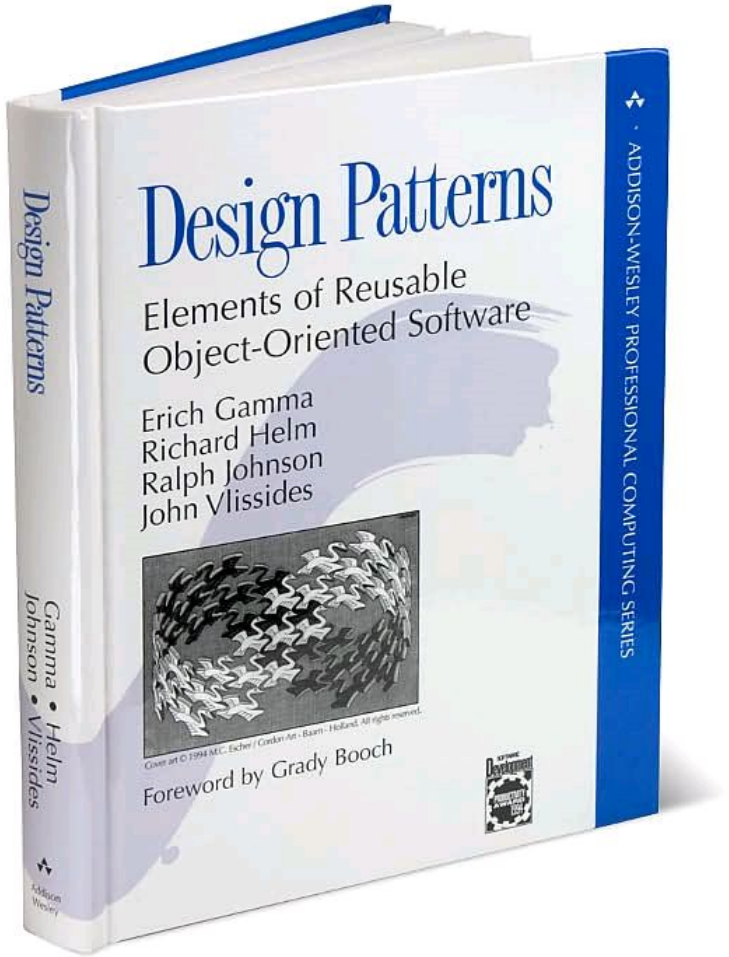
Back to our main topic

- Published 1994
- 23 Patterns
- Widely known



Quiz due tomorrow!

- Published 1994
- 23 Patterns
- Widely known



Our course so far

- Composite
- Strategy
- Template Method
- Iterator
- Decorator
- Observer
- Factory Method
- ?

Not in the book:

- Model view controller
- Promise

Why?

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

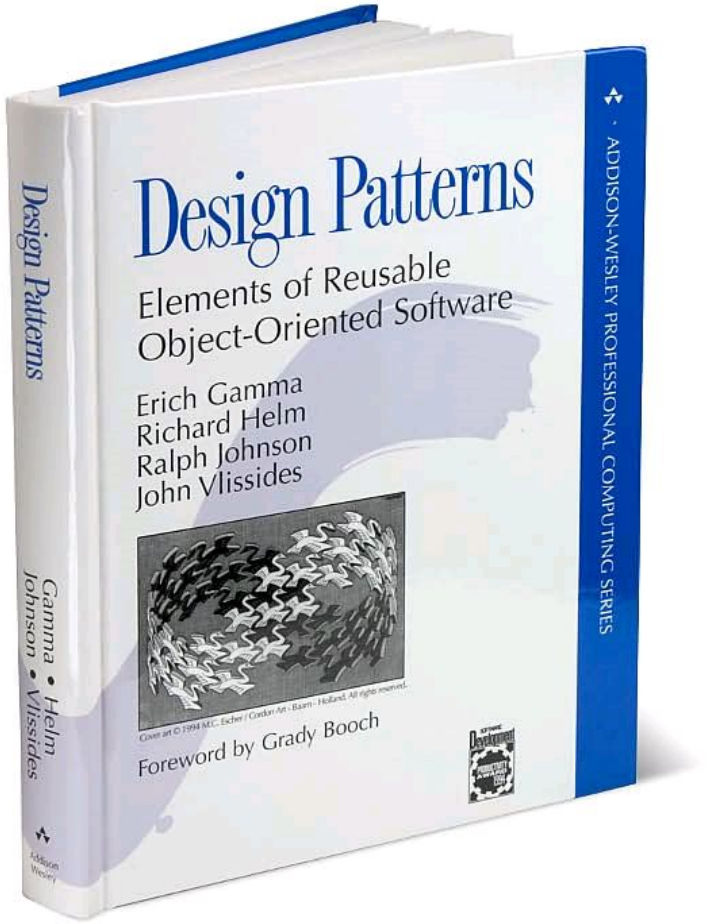
- At least know where to look up when somebody mentions the “Bridge pattern”

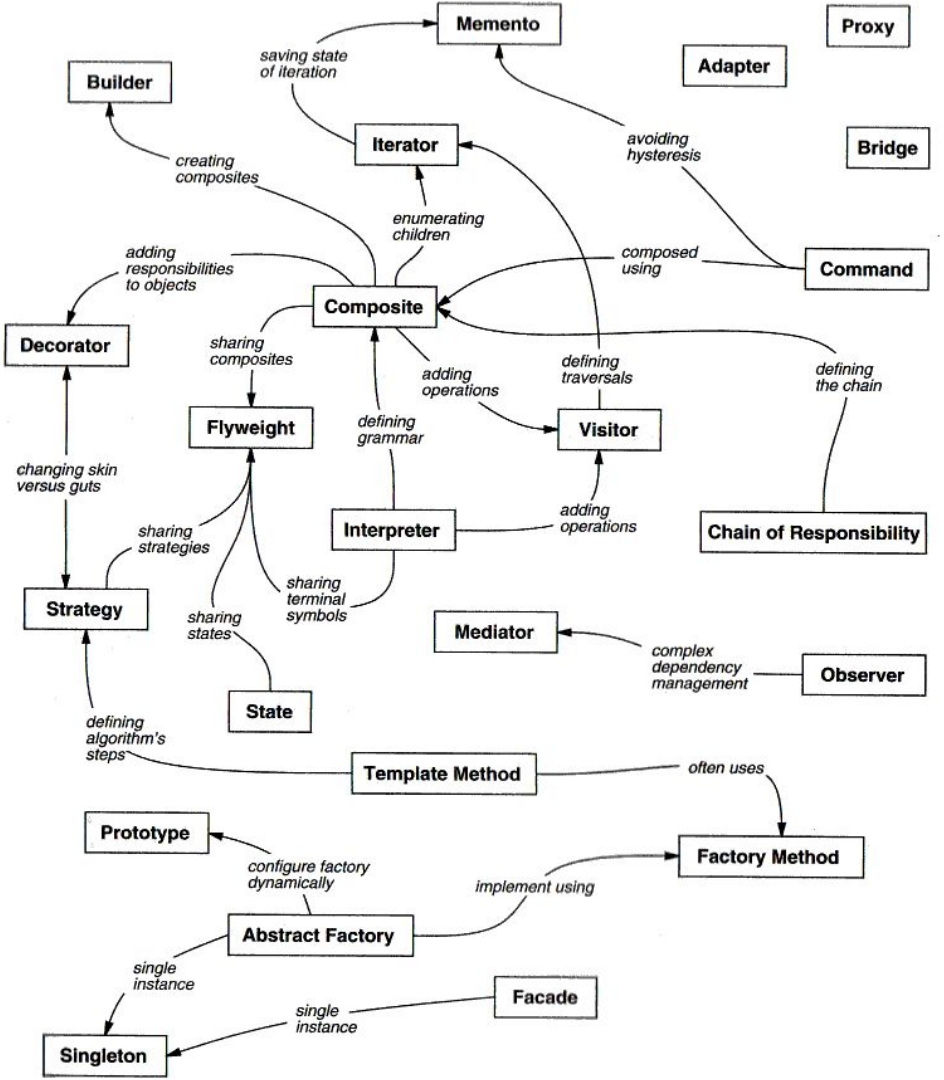
Grouping Patterns

I. Creational Patterns

II. Structural Patterns

III. Behavioral Patterns



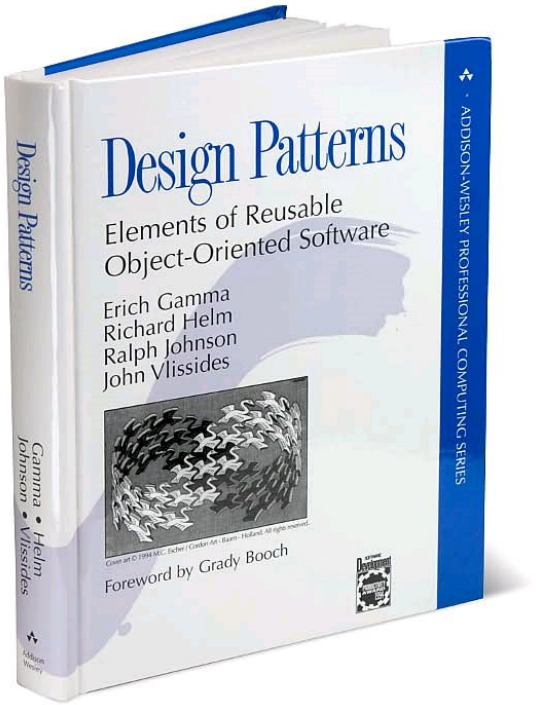

Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

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**

I. Creational Patterns

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

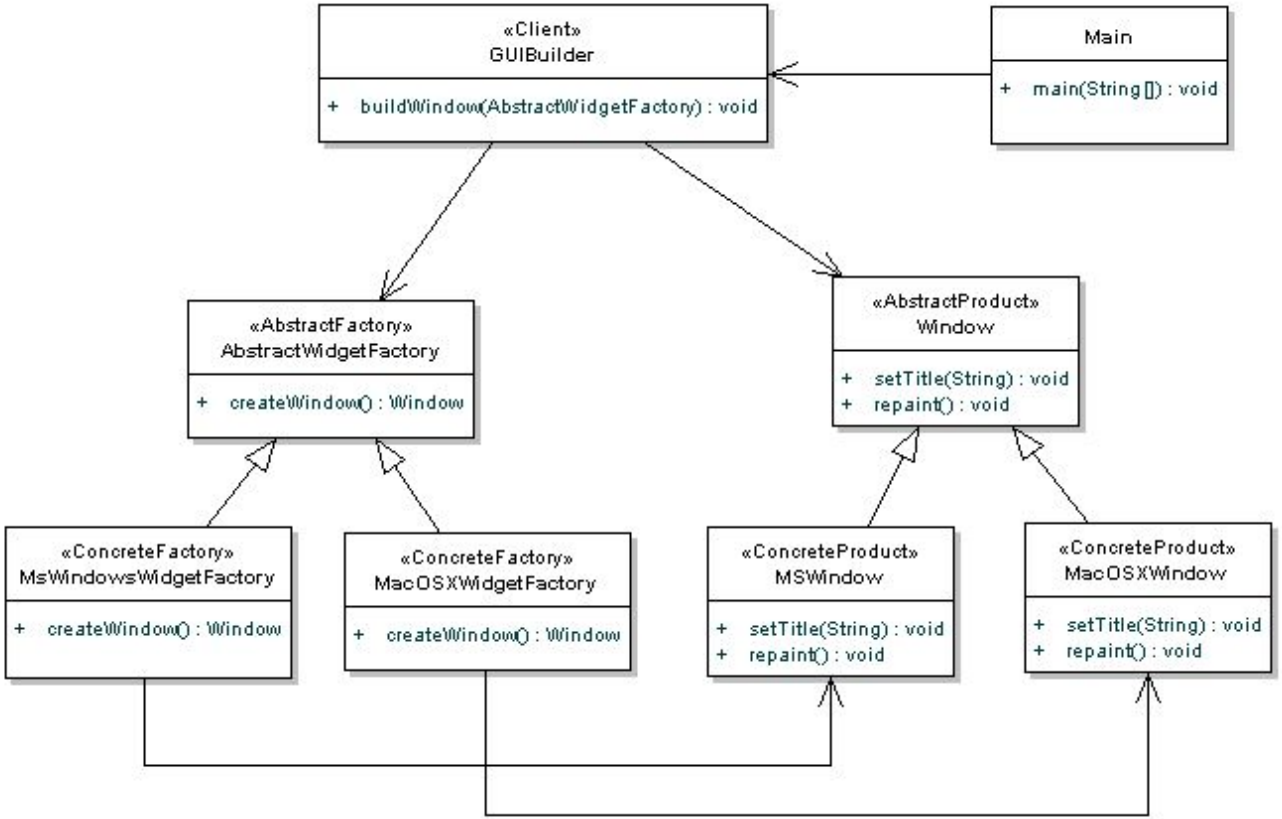


Problem:

- We want to support multiple platforms with our code (e.g., Mac and Windows)
- We want our code to be platform independent
- Suppose we want to create `Window` with `setTile(String text)` and `repaint()`

How can we write code that will create the correct `Window` for the correct platform, without using conditionals?

Abstract Factory Pattern



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

Problem:

- How to handle all combinations of fields when constructing?

```
public class User {
  private final String firstName;    //required
    private final String lastName;    //required
    private final int age;    //optional
    private final String phone;    //optional
    private final String address;    //optional
    ...
}
```

Solution 1

```
public User(String firstName, String lastName) {
  this(firstName, lastName, 0);
}

public User(String firstName, String lastName, int age) {
  this(firstName, lastName, age, "");
}

public User(String firstName, String lastName, int age, String phone) {
  this(firstName, lastName, age, phone, "");
}

public User(String firstName, String lastName, int age, String phone, String address) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.phone = phone;
  this.address = address;
}
```

Bad (code becomes harder to read and maintain with many attributes)

Solution 2: default no-arg constructor plus setters and getters for every attribute

```
public class User {
  private String firstName; // required
  private String lastName; // required
  private int age; // optional
  private String phone; // optional
  private String address;  //optional

  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public String getPhone() {
    return phone;
  }
  public void setPhone(String phone) {
    this.phone = phone;
  }
  public String getAddress() {
    return address;
  }
  public void setAddress(String address) {
    this.address = address;
  }
}
```

Bad (potentially inconsistent state, mutable)

Solution 3

```
public class User {
  private final String firstName; // required
  private final String lastName; // required
  private final int age; // optional
  private final String phone; // optional
  private final String address; // optional

  private User(UserBuilder builder) {
    this.firstName = builder.firstName;
    this.lastName = builder.lastName;
    this.age = builder.age;
    this.phone = builder.phone;
    this.address = builder.address;
  }

  public String getFirstName() { ... }

  public String getLastName() { ... }

  public int getAge() { ... }

  public String getPhone() { ... }

  public String getAddress() { ... }
```

```
public static class UserBuilder {
  private final String firstName;
  private final String lastName;
  private int age;
  private String phone;
  private String address;

  public UserBuilder(String firstName,
                     String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public UserBuilder age(int age) {
    this.age = age;
    return this;
  }

  public UserBuilder phone(String phone) {
    this.phone = phone;
```

```
public User getUser() {
  return new
    User.UserBuilder("Jhon", "Doe")
    .age(30)
    .phone("1234567")
    .address("Fake address 1234")
    .build();
}
```

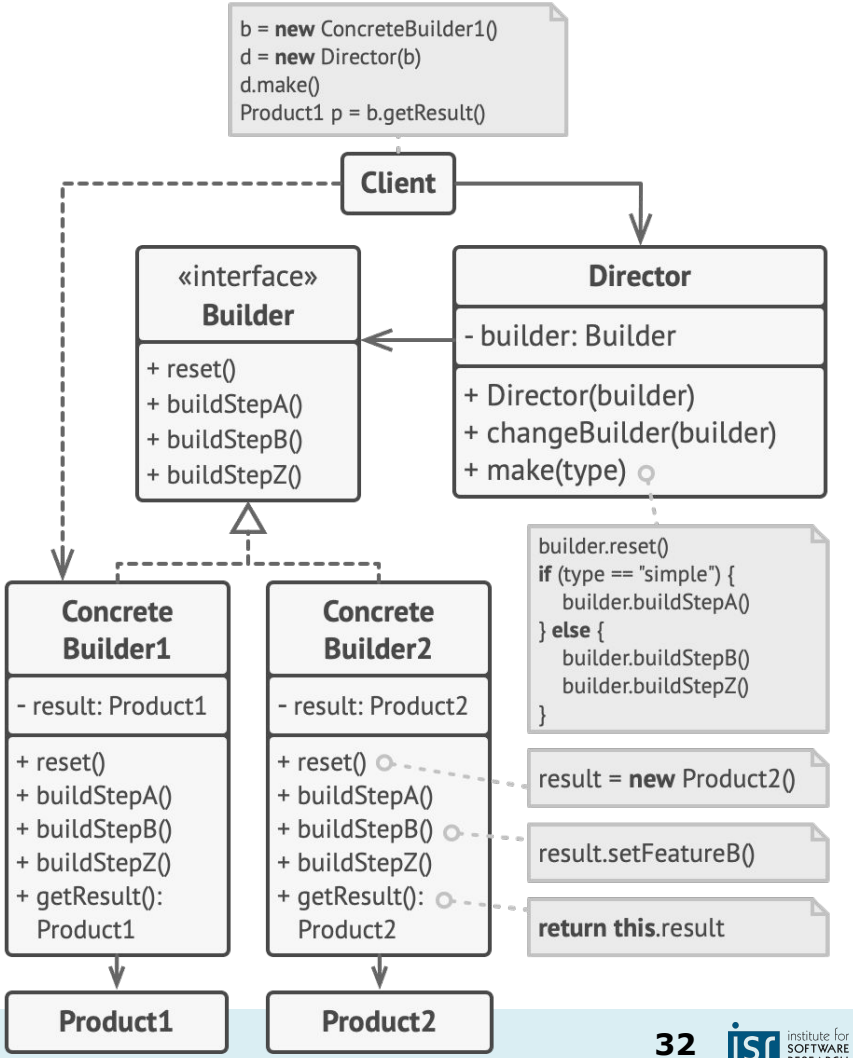
Builder Pattern

- 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)`

Builder Discussion

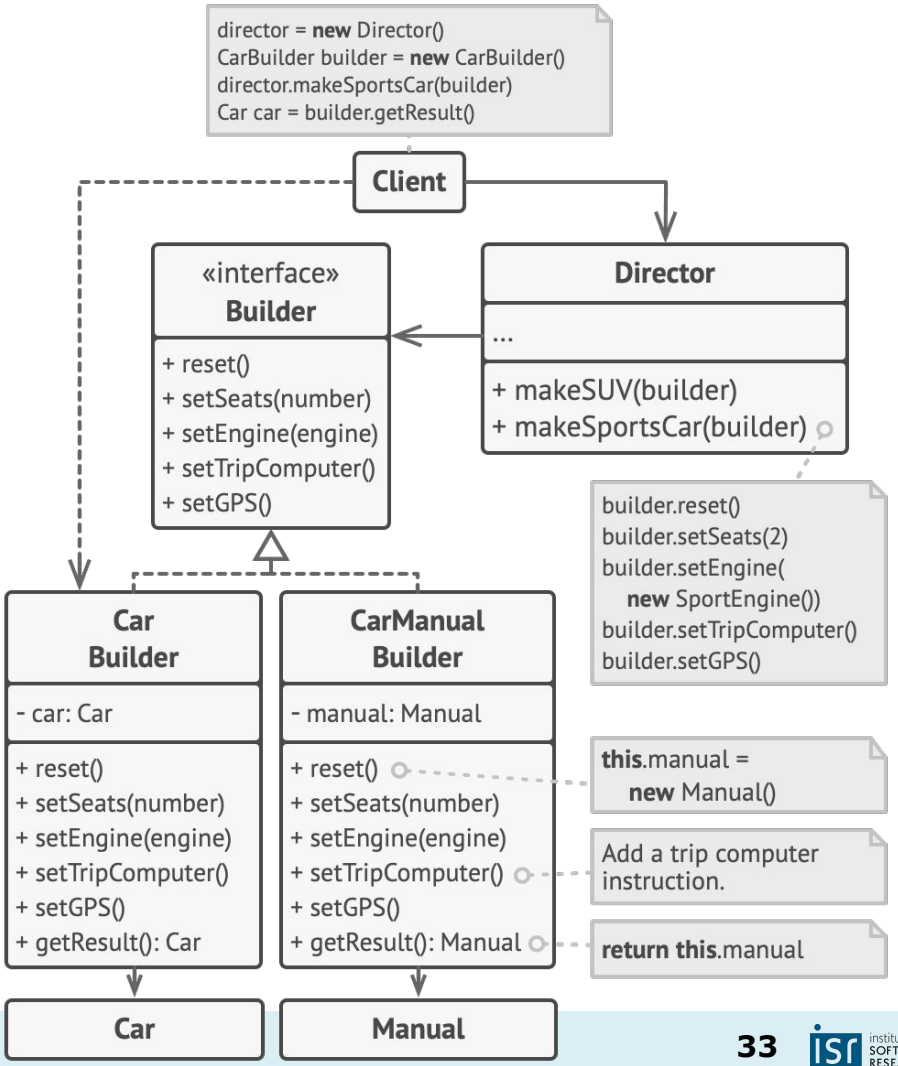
- Emulates named parameters in languages that don't support them
- Emulates 2^n constructors or factories with n builder methods, by allowing them to be combined freely
- Cost is an intermediate (Builder) object

Gof4 Builder Illustration



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

Builder Example



Another Builder Code Example

```
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) { ... }
}
```

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

```
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

Problem:

- Ensure there is only a single instance of a class (e.g., java.lang.Runtime)
- Provide global access to that class

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

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

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

export { getElvis }
```

Singleton Discussion

Singleton = global variable

No flexibility for change or extension

Tends to be overused

These were the creational patterns

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

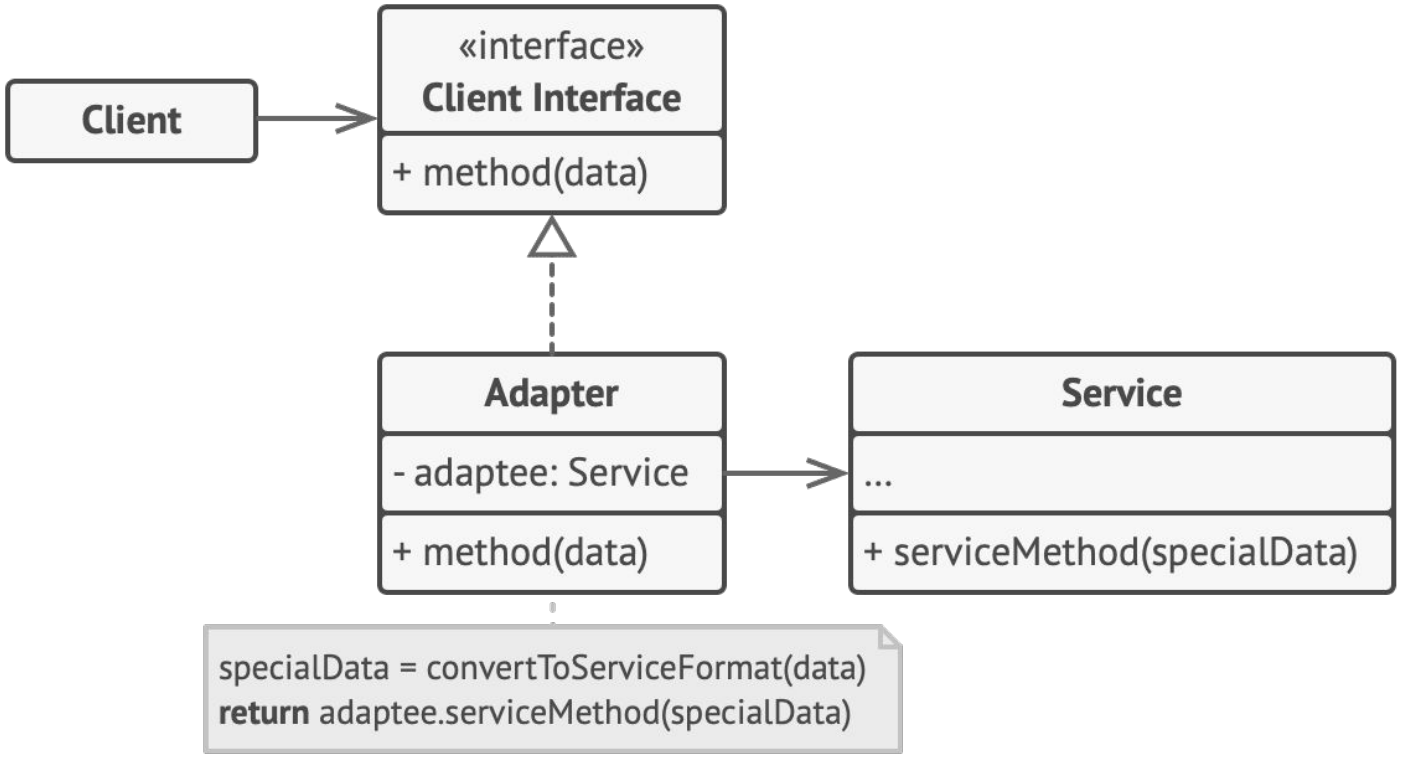
II. Structural Patterns

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

Adapter

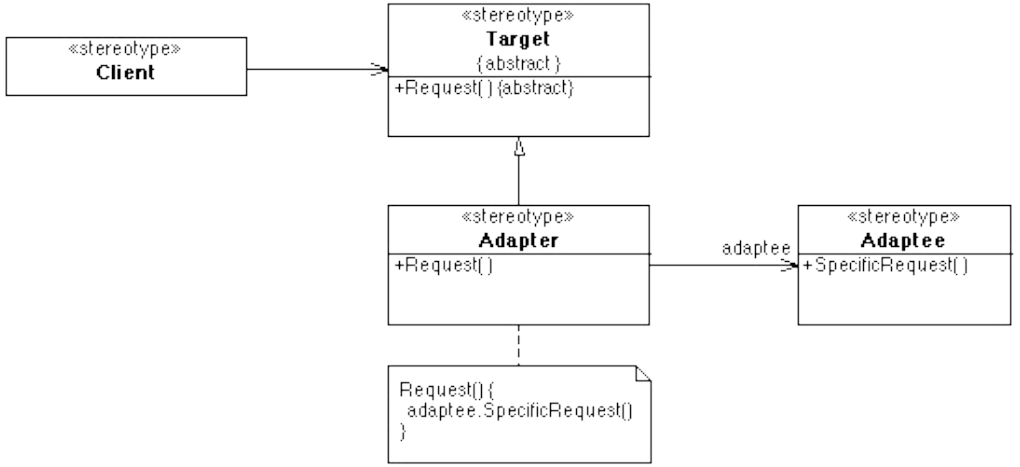
- **Intent** – convert interface of a class into one that another class requires, allowing interoperability
- **Use case** – numerous, e.g., arrays vs. collections
- **Key types** – Target, Adaptee, Adapter
- **JDK** – `Arrays.asList(T[])`

Recall: The *Adapter* Design Pattern



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

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**

Problem: There are two types of thread schedulers, and two types of operating systems or "platforms".

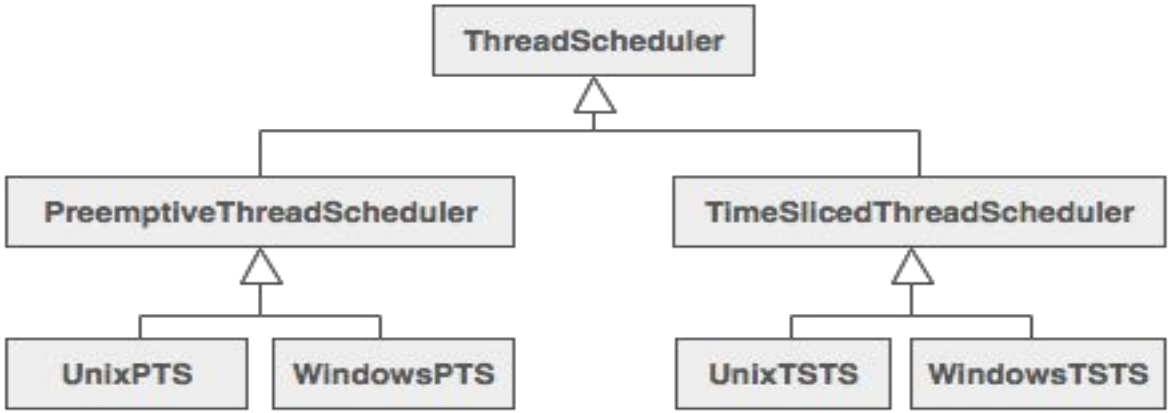
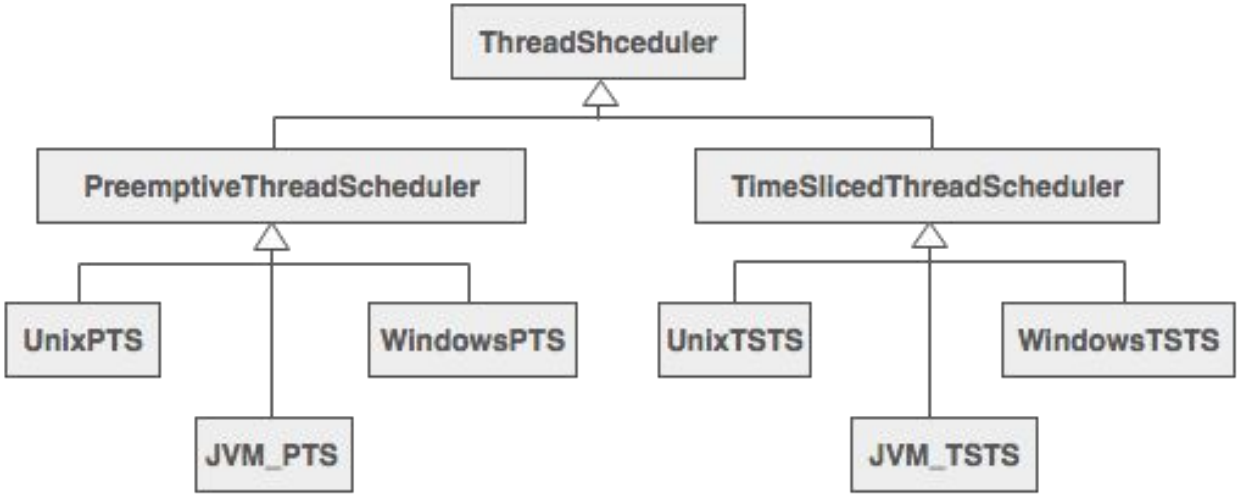


image source: https://sourcemaking.com

Problem: we have to define a class for each permutation of these two dimensions



- How would you redesign this?

image source: https://sourcemaking.com

Bridge Pattern: Decompose the component's interface and implementation into orthogonal class hierarchies.

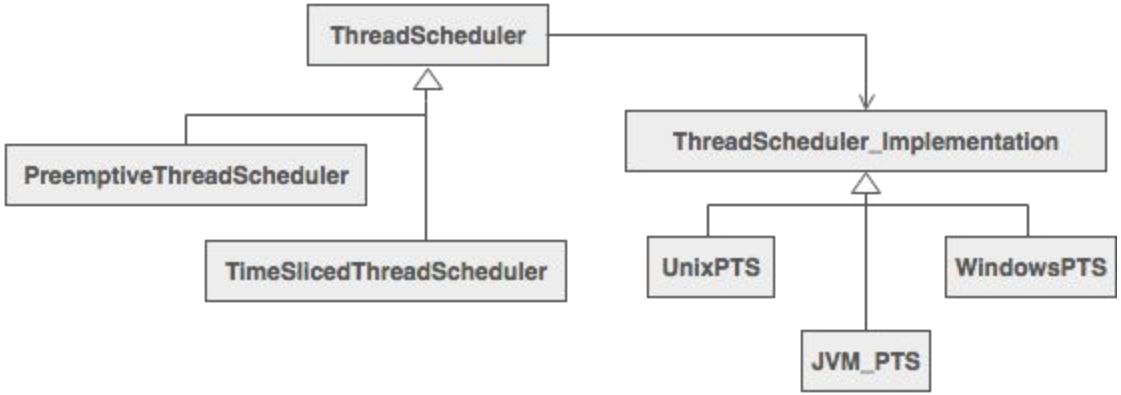


image source: https://sourcemaking.com

Bridge

- **Intent** – decouple an abstraction from its implementation so they can vary independently
- **Use case** – portable windowing toolkit
- **Key types** – Abstraction, *Implementor*
- **JDK** – JDBC, Java Cryptography Extension (JCE), Java Naming & Directory Interface (JNDI)
- Bridge pattern is *very* similar to Service Provider
  - Abstraction ~ API, *Implementer* ~ SPI

Adapter vs Bridge

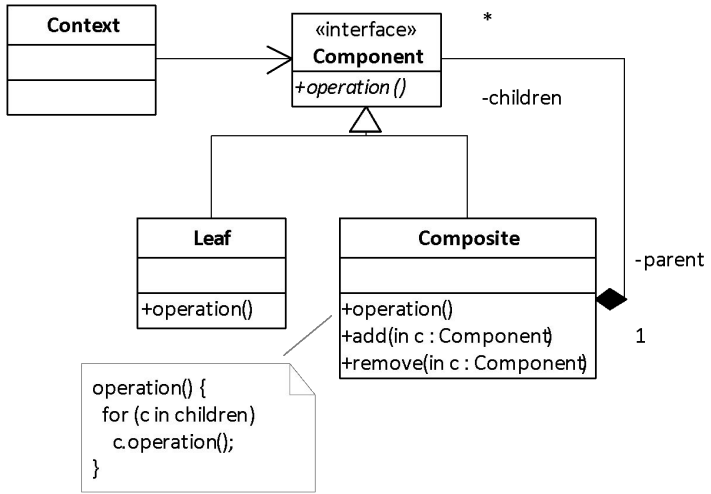
- Adapter makes things work together after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.

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`

The Composite Design Pattern

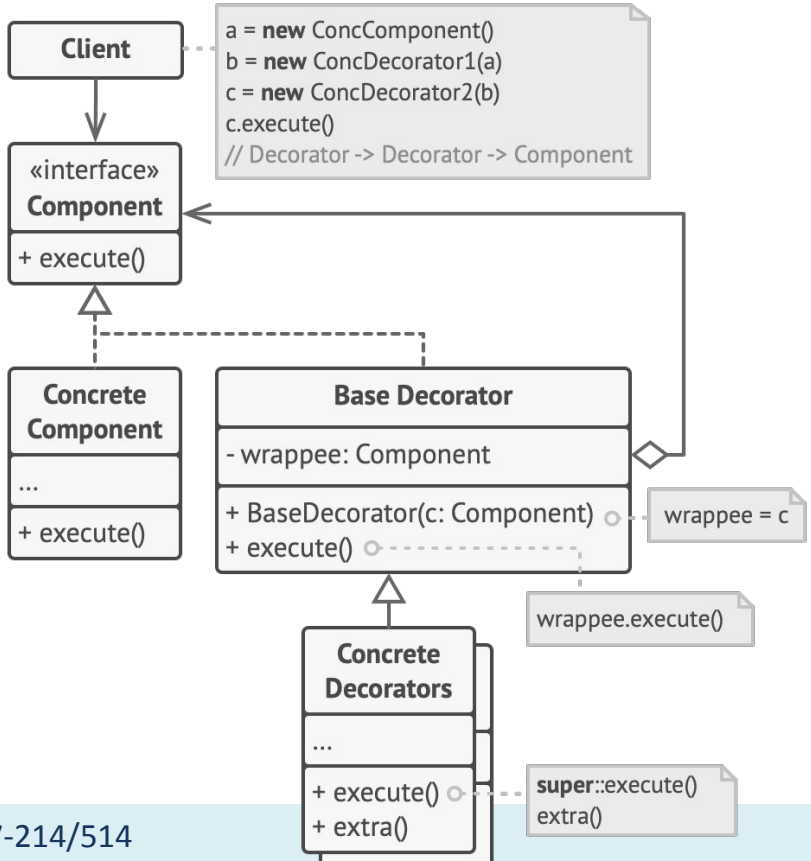
- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add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

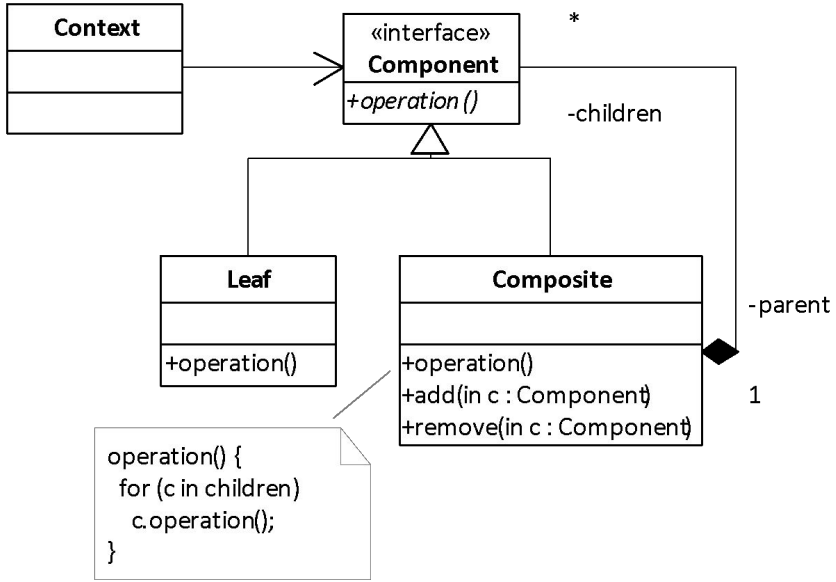


Recall: Decorator Pattern

- 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 vs Composite?

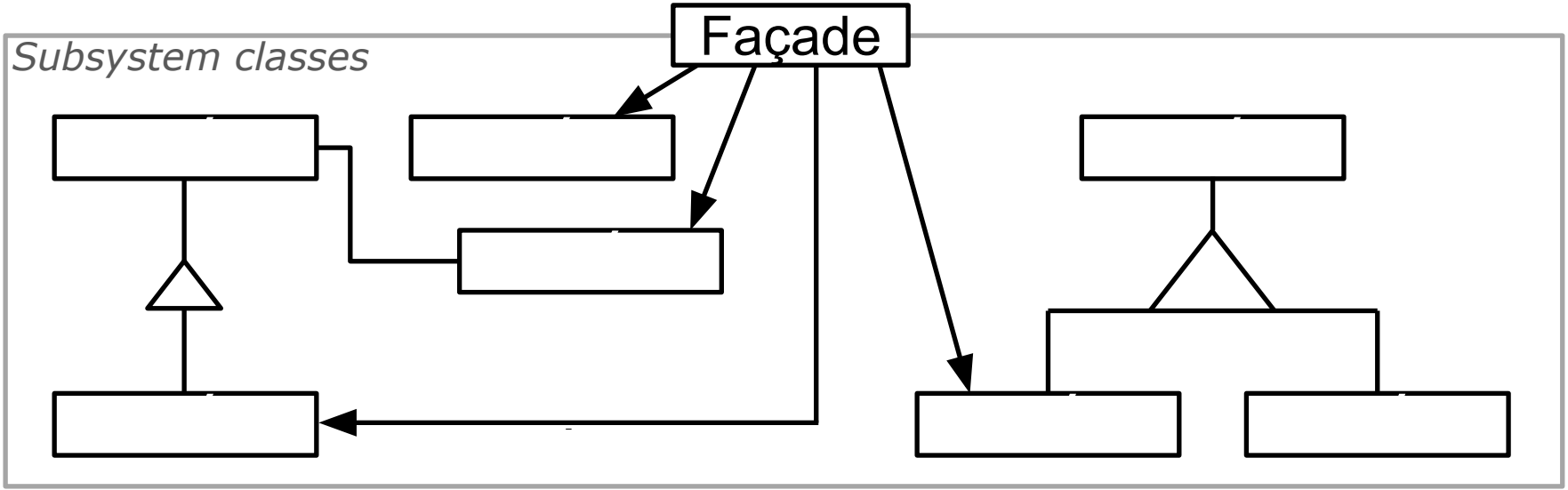




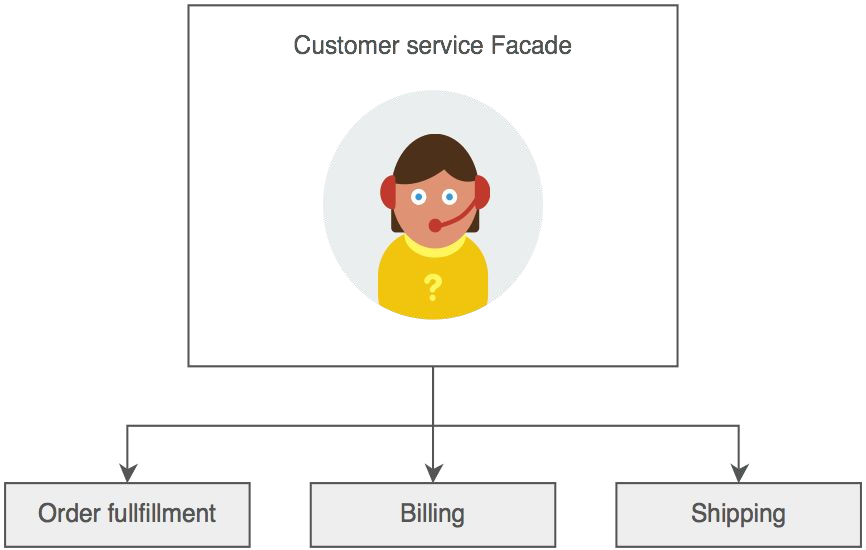
Façade Pattern

- 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



Façade example



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

Discussion

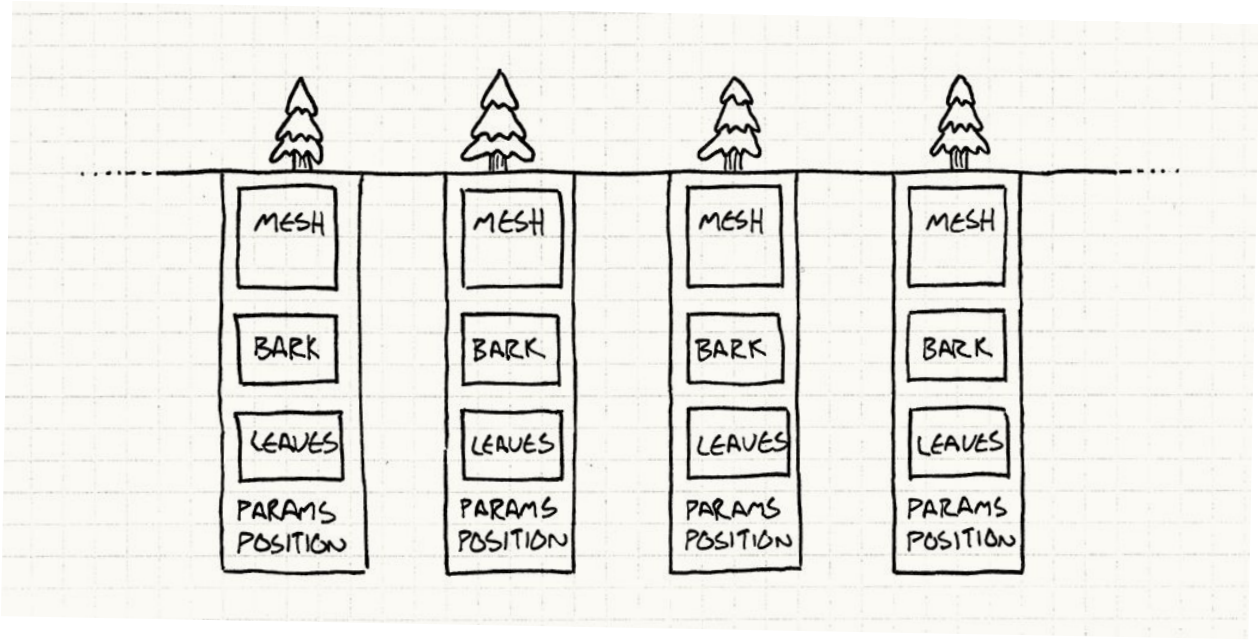
Facade vs Controller Heuristic

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

Facade vs Singleton

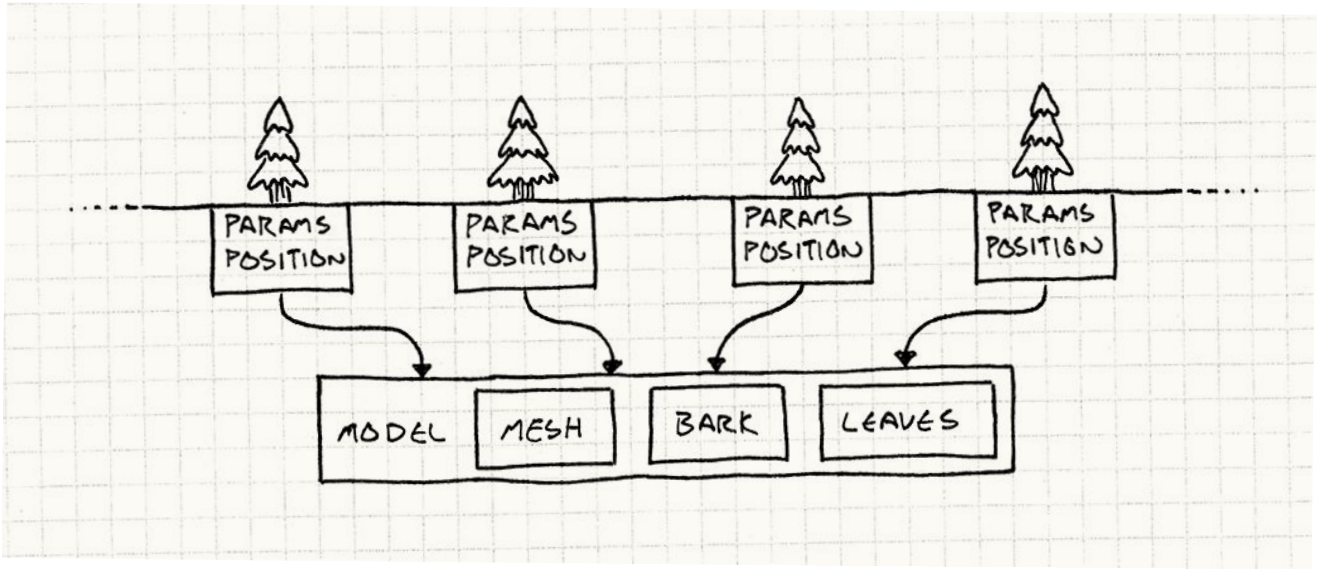
- Facade sometimes a global variable
- Typically little design for change/extension

Problem: Imagine implementing a forest of individual trees in a realtime game



Source: http://gameprogrammingpatterns.com/flyweight.html

Trick: most of the fields in these objects are the *same* between all of those instances



Source: http://gameprogrammingpatterns.com/flyweight.html

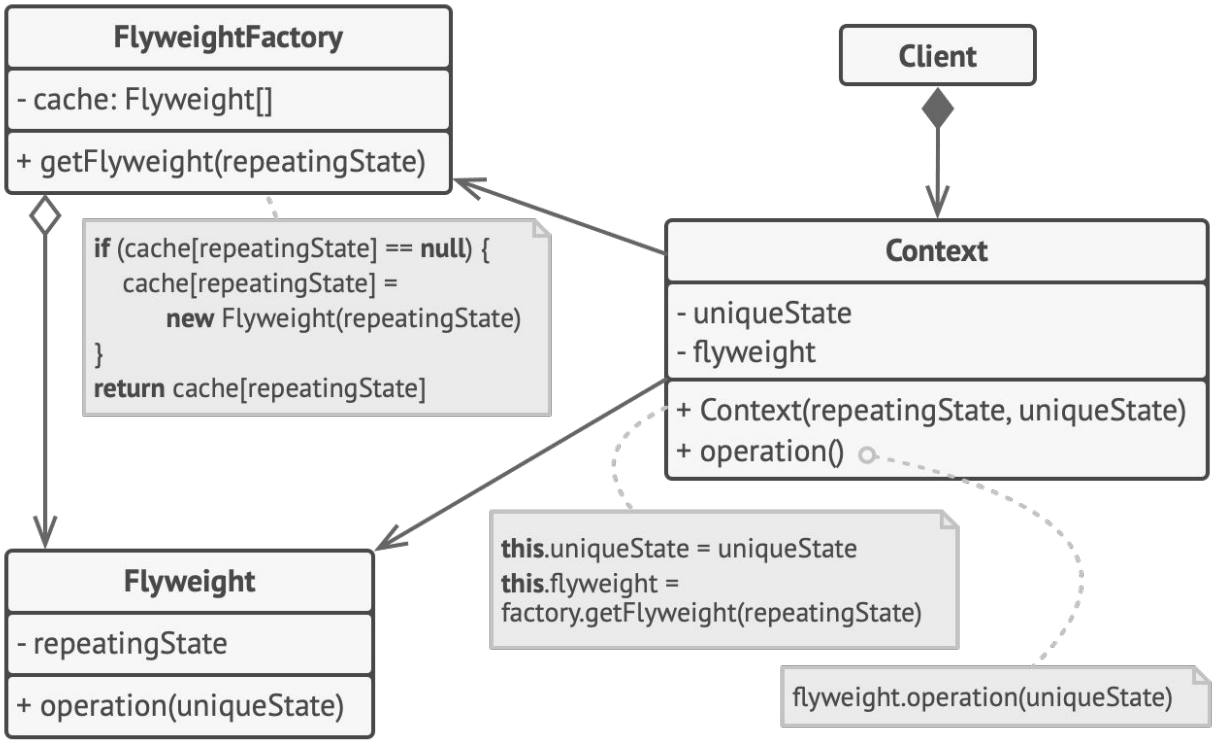
Flyweight

- 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
- JDK – String literals (JVM feature)

Flyweight

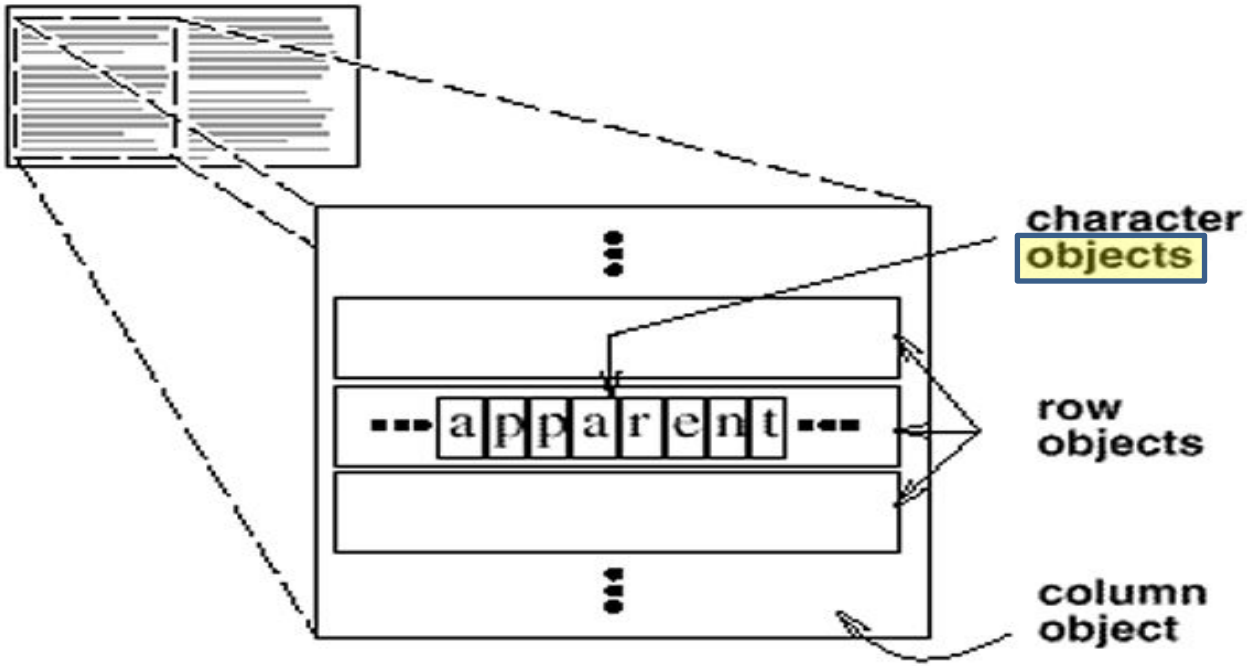
Key idea: Avoid copies of structurally equal objects, reuse object

Requires immutable objects and factory with caching



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

Flyweight Illustration



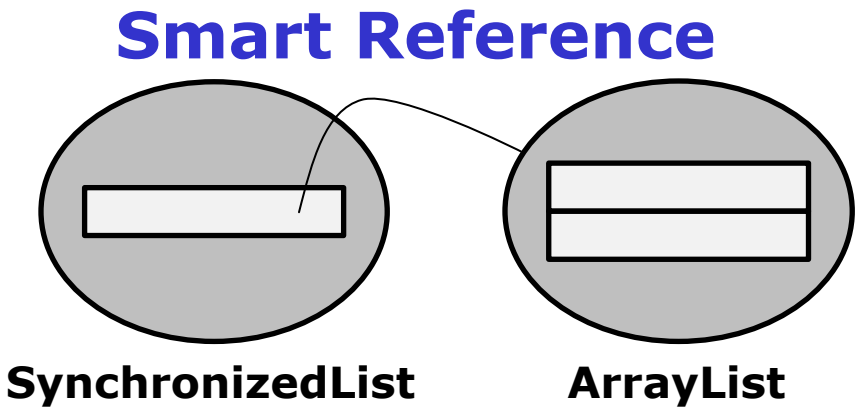
Proxy Pattern

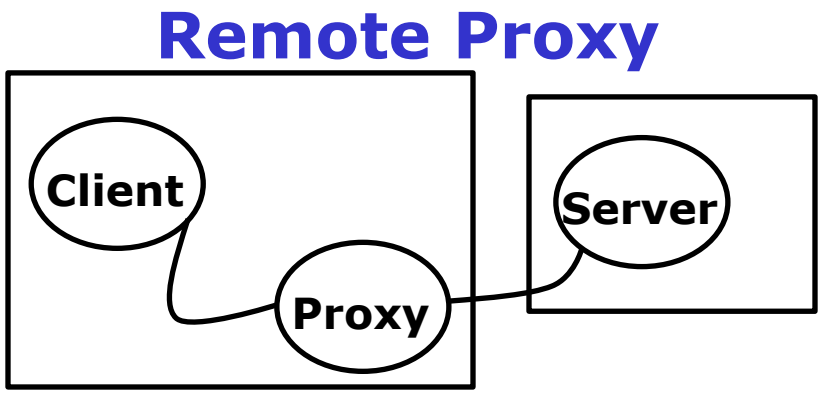
- Intent – surrogate for another object
- Use case – delay loading of images till needed
- Key types – *Subject*, Proxy, RealSubject
- Gof mention several flavors
  - virtual proxy – stand-in that instantiates lazily
  - remote proxy – local representative for remote obj
  - protection proxy – denies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

Proxy

- Decorator vs Proxy:
  - Decorator adds responsibilities to object (w/t inheritance).
  - Proxy is used to “control access” to an object.

Proxy Illustrations





These were the structural patterns

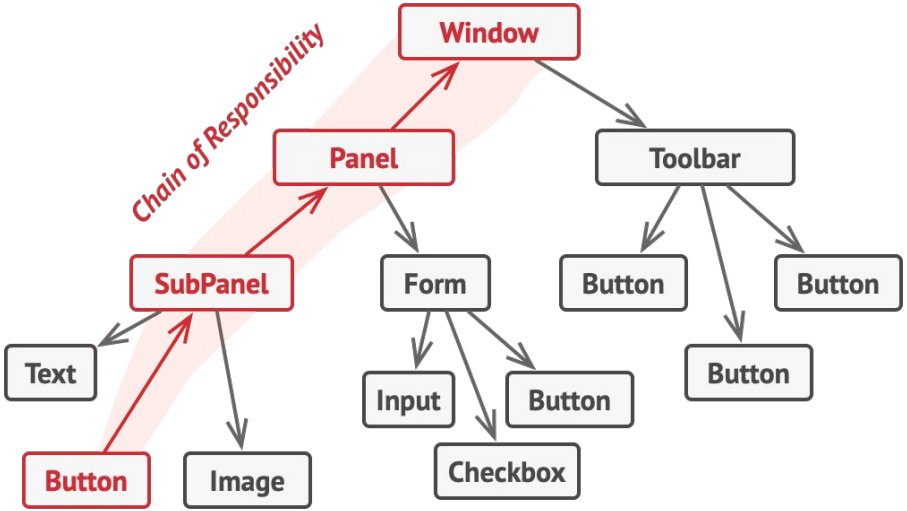
1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

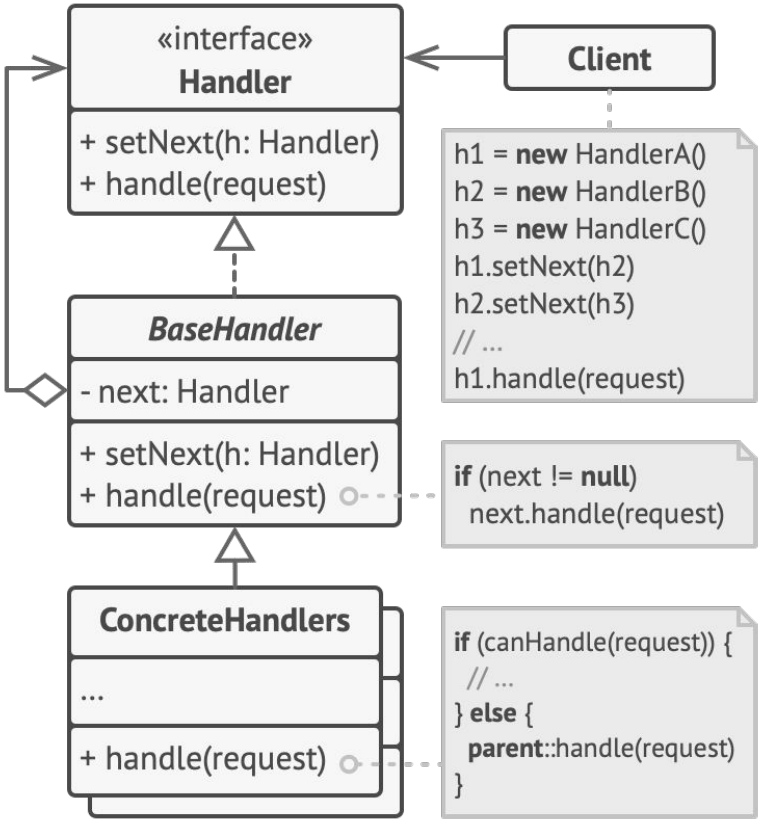
III. Behavioral Patterns

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

- 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





https://refactoring.guru/design-patterns/chain-of-responsibility

Command

- Intent – encapsulate a request 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.

```
public static void main(String[] args) {
   SwingUtilities.invokeLater(() -> new Demo().setVisible(true));
}
```

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!

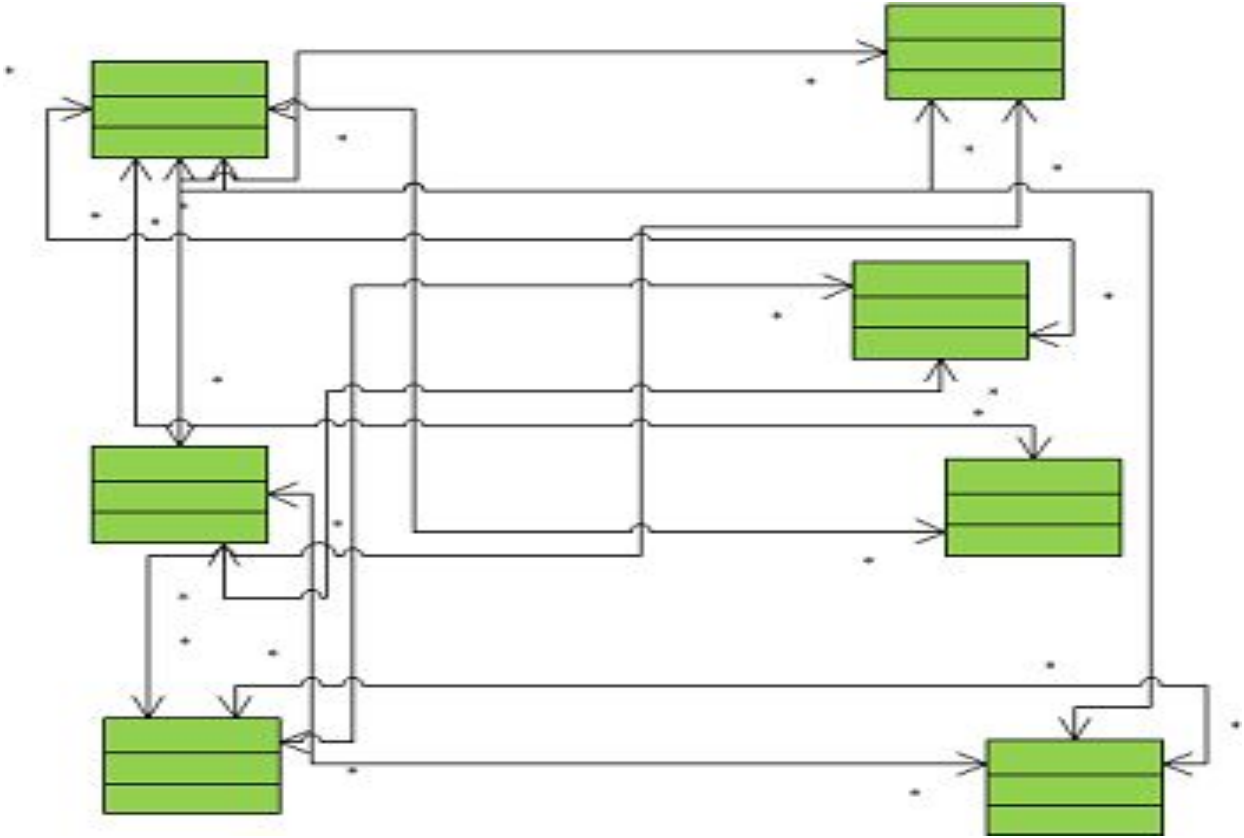
Iterator Pattern

- 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 ..

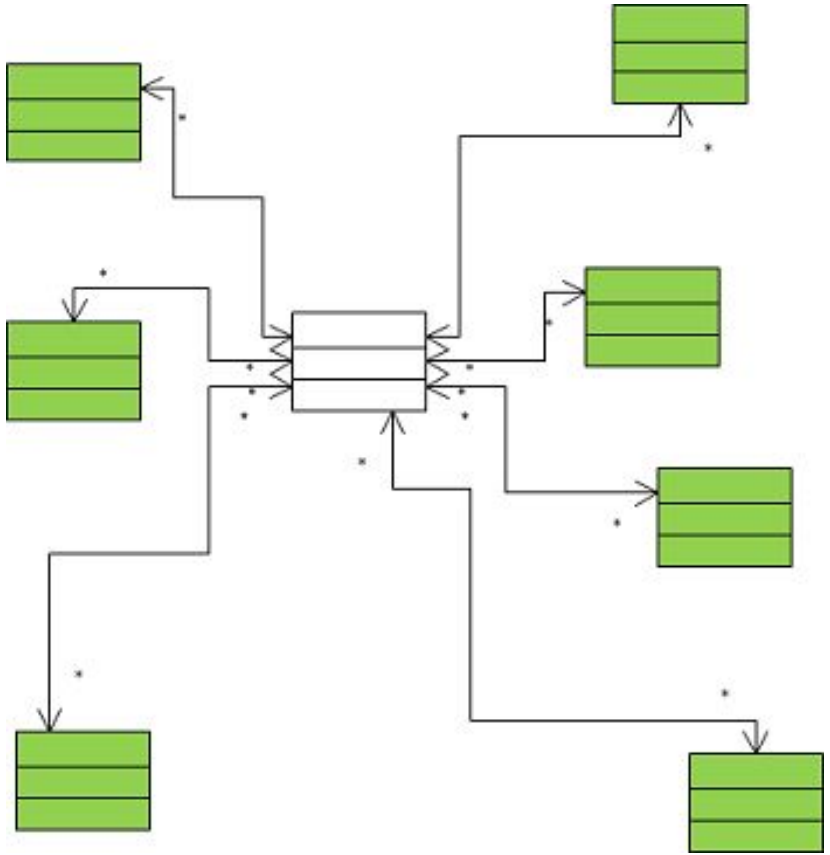
Iterator Illustration

```
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);
```

Problem:



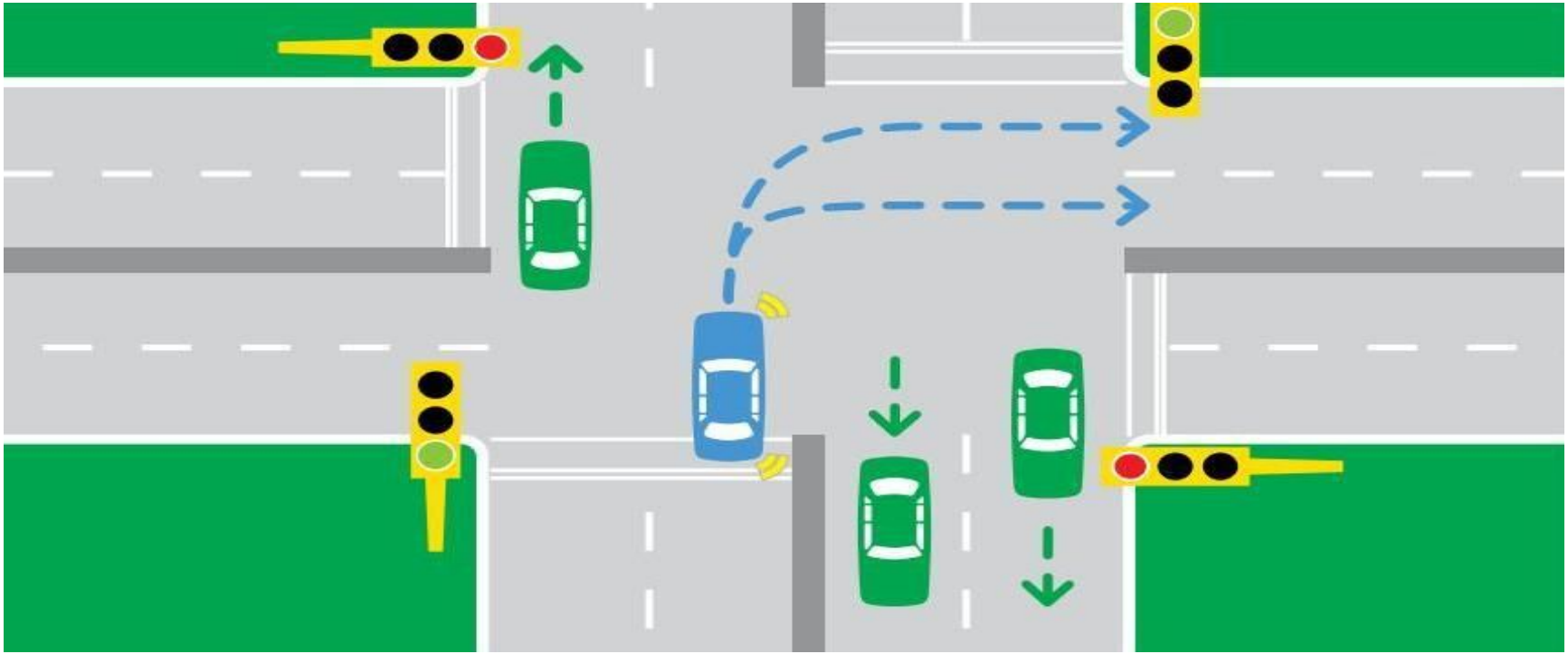
Mediator Pattern



Mediator

- 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



Problem: without violating encapsulation, allow client of Editor to capture the object's state and restore later

```
public class Editor {
  //state
  public String editorContents;
  public void setState(String contents) {
    this.editorContents = contents;
  }

}
```

Provide save and restoreToState methods
Hint: define custom type (Memento)

Problem: without violating encapsulation, allow client of Editor to capture the object's state and restore later

```
public class Editor {
  //state
  public String editorContents;
  public void setState(String contents) {
    this.editorContents = contents;
  }
  public EditorMemento save() {
    return new EditorMemento(editorContents);
  }
  public void restoreToState(EditorMemento memento) {
    editorContents = memento.getSavedState();
  }
}
```

Problem: without violating encapsulation, allow client of Editor to capture the object's state and restore later

```
public class EditorMemento {
  private final String editorState;
  public EditorMemento(String state) {
    editorState = state;
  }
  public String getSavedState() {
    return editorState;
  }
}
```

Memento

- Intent – without violating encapsulation, allow client to capture an object's state, and restore later
- Use case – when you need to provide an undo mechanism in your applications, when the internal state of an object may need to be restored at a later stage (e.g., text editor)
- Key type – Memento (opaque state object)
- JDK – none that I'm aware of (*not* serialization)

Observer

- 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!
- JDK – Swing, left and right

Problem: allow object to behave in different ways depending on internal state

```
class Document
    string state;
    // ...
    method publish() {
        switch (state) {
            "draft":
                state = "moderation";
                break;
            "moderation":
                if (currentUser.role == 'admin')
                    state = "published"
                break;
            "published":
                // Do nothing.
        }
    }
    // ...
```

```
class Document
    string state;
    // ...
    method publish() {
        switch (state) {
            "draft":
                state = "moderation";
                break;
            "moderation":
                if (currentUser.role == 'admin')
                    state = "published"
                break;
            "published":
                // Do nothing.
        }
    }
    // ...
```

```
interface State {
    void publish(Document wrapper);
}

class Document {
    private State currentState;

    public Document() {
        currentState = new Draft();
    }

    public void set_state(State s) {
        currentState = s;
    }

    public void publish() {
        currentState.publish(this);
    }
}

class Draft implements State {
    public void publish(Document wrapper) {
        wrapper.set_state(new Moderation());

    }
}

// ...
```

State

- 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…
  - Works *great* in Java
  - Use enums as states
  - Use `AtomicReference<State>` to store it [EJ]

State

- State can be considered as an extension of Strategy
- Both patterns use composition to change the behavior of the main object by delegating the work to the helper objects.
  - Strategy makes these objects completely independent
  - State allows state objects to alter the current state of the context with another state, making them interdependent

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

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

Template Method

- 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`)

Problem:

- It should be possible to define a new operation for (some) classes of an object structure without changing the classes.
  - Example: Calculate shipping for different regions for all items in shopping cart. Be able to add new shipping cost formulas without changing existing code.

The Visitable interface

```java
//Element interface
public interface Visitable{
  public void accept(Visitor visitor);
}
```

```java
//concrete element
public class Book implements Visitable{
  private double price;
  private double weight;

  //accept the visitor
  public void accept(Visitor vistor) {
    visitor.visit(this);
  }
  public double getPrice() {
    return price;
  }
  public double getWeight() {
    return weight;
  }
}
```

The Visitor interface

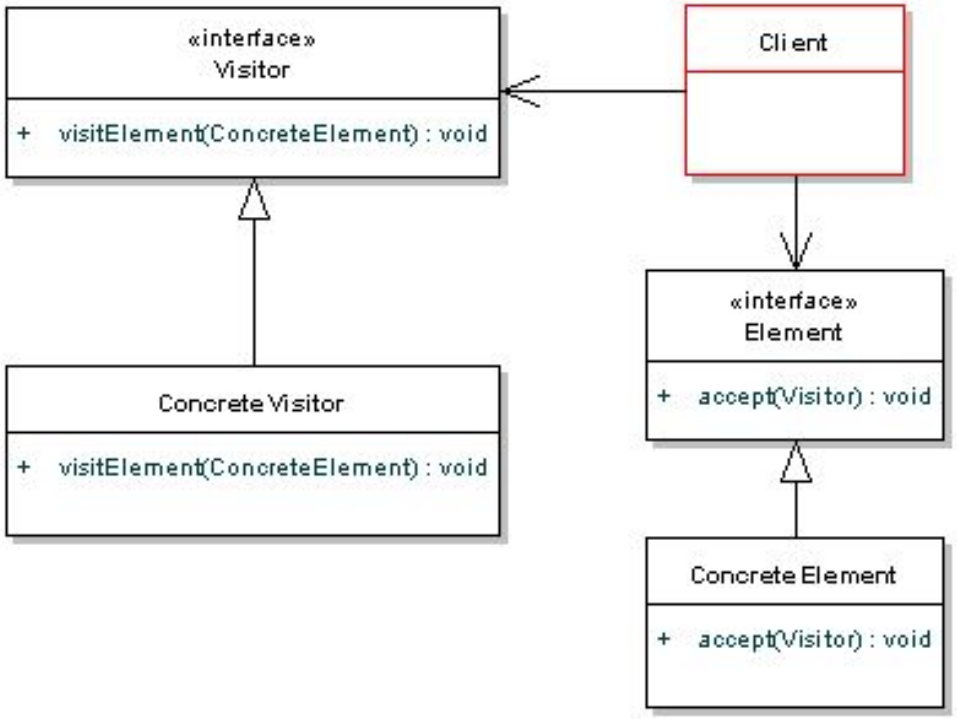
```java
public interface Visitor{
  public void visit(Book book);

  //visit other concrete items
  public void visit(CD cd);
  public void visit(DVD dvd);
}
```

```java
public class PostageVisitor implements Visitor {
  private double totalPostageForCart;
  //collect data about the book
  public void visit(Book book) {
    //assume we have a calculation here related to weight and price
    //free postage for a book over 10
    if(book.getPrice() < 10.0) {
      totalPostageForCart += book.getWeight() * 2;
    }
  }

  //add other visitors here
  public void visit(CD cd) {...}
  public void visit(DVD dvd) {...}

  //return the internal state
  public double getTotalPostage() {
    return totalPostageForCart;
  }
}
```

Driving the visitor

```
1  public class ShoppingCart {
2    //normal shopping cart stuff
3    private ArrayList<Visitable> items;
4    public double calculatePostage() {
5      //create a visitor
6      PostageVisitor visitor = new PostageVisitor();
7      //iterate through all items
8      for(Visitable item: items) {
9        item.accept(visitor);
10     }
11     double postage = visitor.getTotalPostage();
12     return postage;
13   }
14 }
```

Visitor


«interface»
Visitor
+ visitElement(ConcreteElement) : void
Client
«interface»
Element
+ accept(Visitor) : void
Concrete Visitor
+ visitElement(ConcreteElement) : void
Concrete Element
+ accept(Visitor) : void

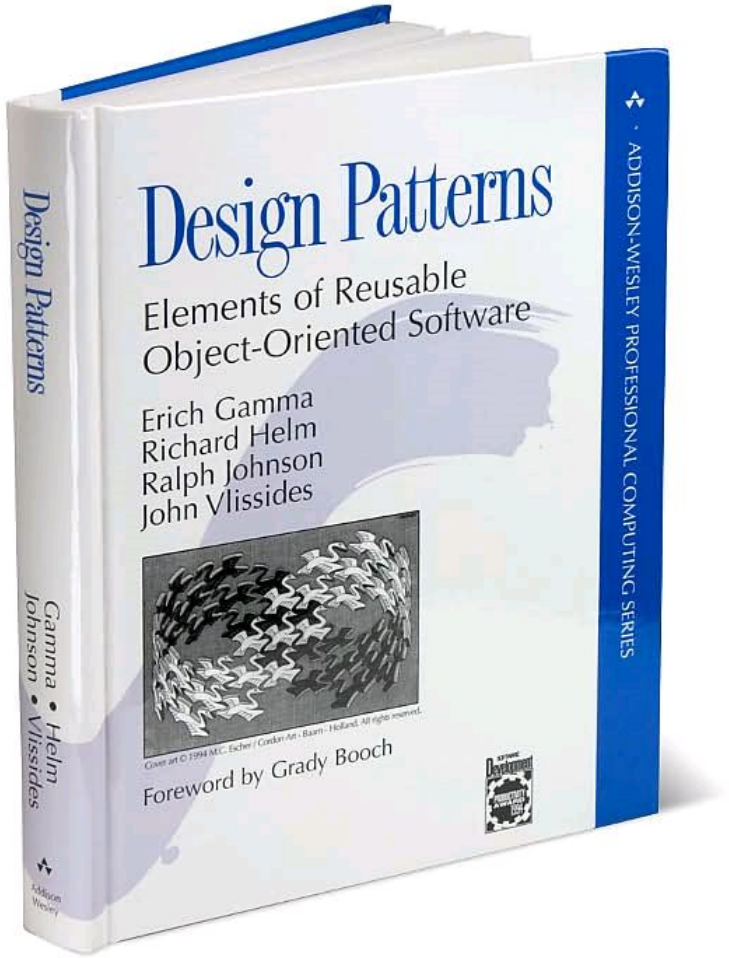
Visitor

- 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 – none that I'm aware of; very common in compilers

These were the 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

- Published 1994
- 23 Patterns
- Widely known



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