Principles of Software Construction: Objects, Design, and Concurrency

A Tour of the 23 GoF Design Patterns

Bogdan Vasilescu **Jonathan Aldrich**

Registration is soon! Consider:

Do you want to gain experience in SE issues above the level of class design…

- …particularly working in teams on long-running systems?
  - 17-313, Foundations of Software Engineering, Padhye.

- …but make it ML?
  - 17-400: Machine Learning and Data Science at Scale, Miller.

Are you an undergrad CS student looking to fill Logic and Languages, OR do you want to learn more about programming languages and compiler, OR do you really like Jonathan?

- 17-363/17-663/17-819, Programming Language Pragmatics, Aldrich & Titzer

Forget all that, do you really just want to go very deep on web app development?

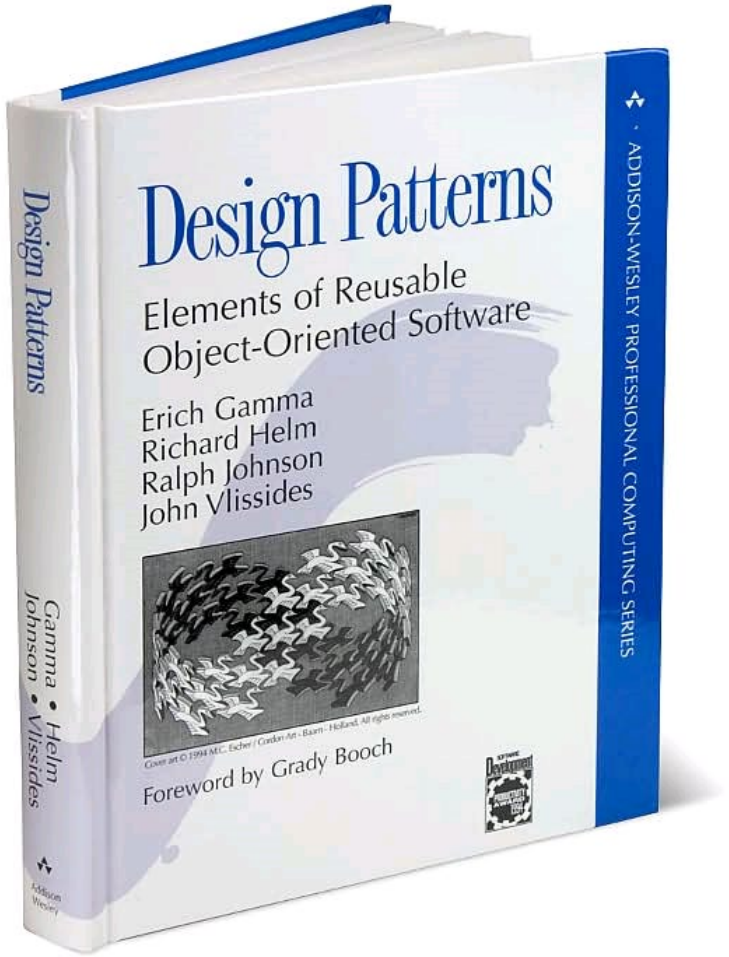
- 17-437/17-637, Web Application Development, Eppinger

Midterm Question: Decorator

```
abstract class OrderDecorator extends CharSorter {

  public OrderDecorator(               ) {

  }

  public boolean comesBefore(char i, char j) {


  }

}
```

```
class SpacesFirstSorter extends OrderDecorator {

  public SpacesFirstSorter(                ) {

  }

  @Override
  public boolean comesBefore(char i, char j) {


  }

}
```

- Published 1994, widely known
- 23 Patterns; considered canonical, BUT:
  - not all patterns commonly used
  - not all common patterns included
- Good to where to look up when somebody mentions the “Bridge pattern”



Today's goal is **not** to cover all 23 patterns.

Instead, touch on a bunch of them, especially the ones that are still useful, so you recognize the words when you're out in the Real World.

And, will practice quickly reasoning about design situations/alternatives.

Key takeaways:

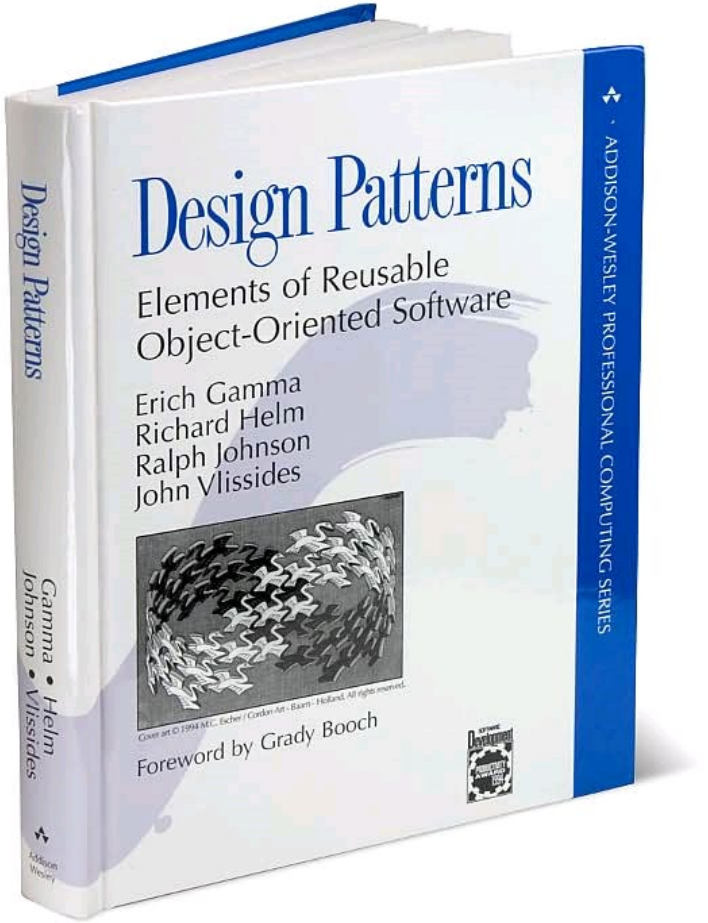
- Design patterns capture a shared vocabulary; knowing/recognizing them makes it easier for you to design new systems; understand existing systems; and write systems that other people can understand.
- The key distinguishing feature *between* patterns is **intent.**

Grouping Patterns

I. Creational Patterns

II. Structural Patterns

III. Behavioral Patterns



All GoF Design Patterns

Creational:

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

Structural:

1. Adapter
2. Bridge
3. Composite
9. Decorator
10. Façade
11. Flyweight
12. Proxy

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

Course so far…

Creational:

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

Structural:

1. **Adapter**
2. Bridge
3. **Composite**
9. **Decorator**
10. Façade
11. Flyweight
12. **Proxy**

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. **Iterator**
17. Mediator
18. Memento
19. **Observer**
20. State
21. **Strategy**
22. **Template method**
23. Visitor

Course so far…

Creational:

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

Structural:

1. **Adapter**
2. Bridge
3. **Composite**
9. **Decorator**
10. Façade
11. Flyweight
12. **Proxy**

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. **Iterator**
17. Mediator
18. Memento
19. **Observer**
20. State
21. **Strategy**
22. **Template method**
23. Visitor

Not in the book:

- Model view controller
- Promise
- Module (JS)

Patterns we will mostly skip

Creational:

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

Structural:

1. *Adapter*
2. Bridge
3. *Composite*
9. *Decorator*
10. Façade
11. Flyweight
12. *Proxy*

Behavioral:

9. Chain of Responsibility
10. Command
11. **Interpreter**
16. *Iterator*
17. **Mediator**
18. **Memento**
19. *Observer*
20. **State**
21. *Strategy*
22. *Template method*
23. **Visitor**

Warm Up 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.

All GoF Design Patterns

Creational:

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

Structural:

1. Adapter
2. Bridge
3. Composite
9. Decorator
10. Façade
11. Flyweight
12. Proxy

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

(New) Problem:

Imagine you want to write code that supports multiple platforms (e.g., Mac and Windows)

- We want code to be platform independent

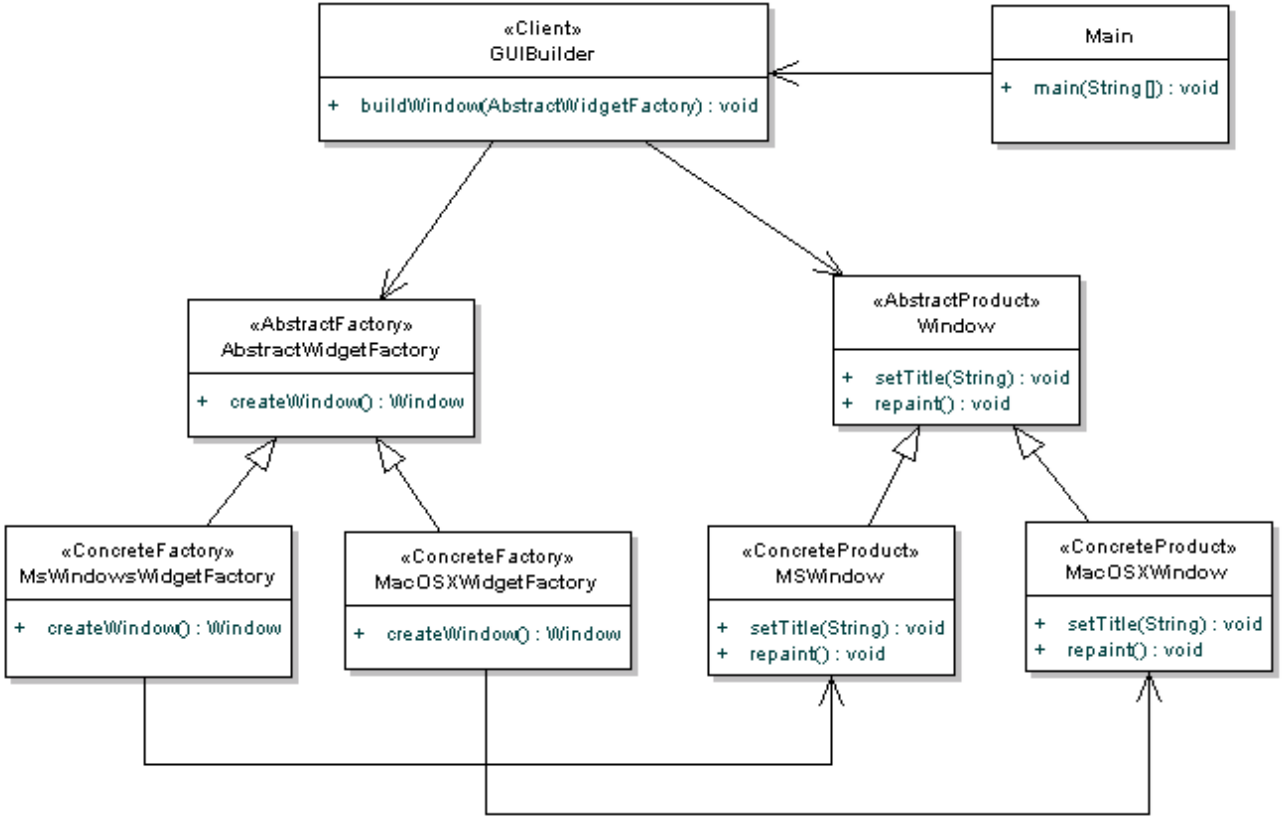
Suppose we want to create a Window with setTile(String text) and repaint()

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

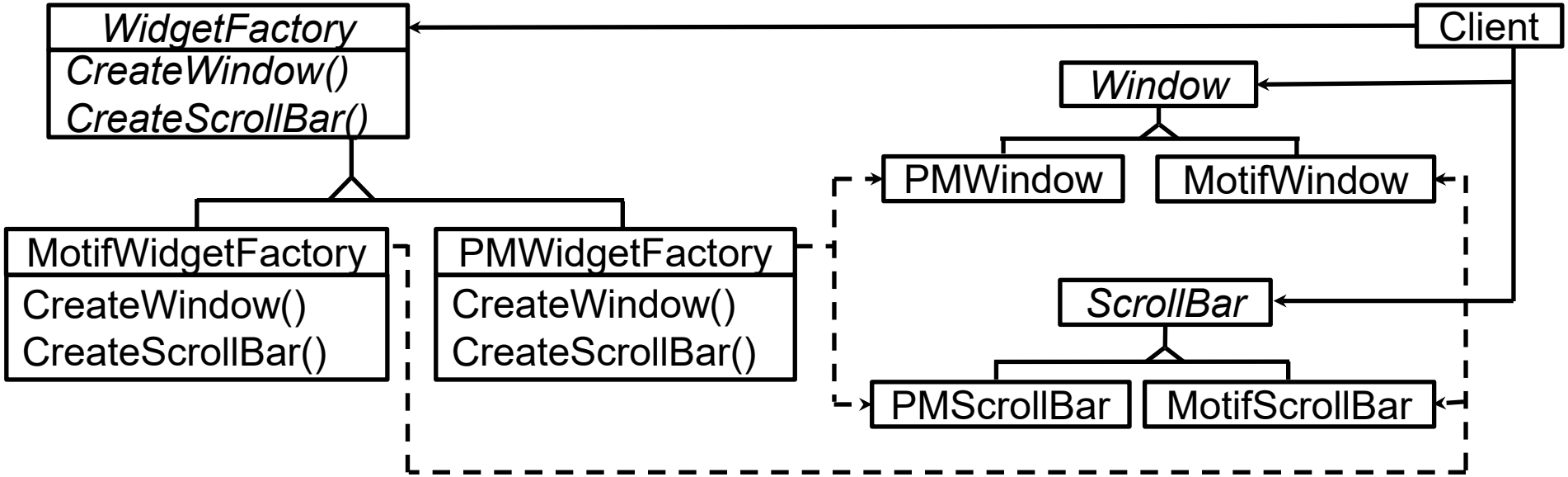
Abstract Factory

- 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 Pattern



Abstract Factory Illustration



Abstract factory compared to?

Creational:

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

Structural:

1. Adapter
2. Bridge
3. Composite
9. Decorator
10. Façade
11. Flyweight
12. Proxy

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. ***Strategy***
22. Template method
23. Visitor

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");
```

(New) 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
    ...
}
```

Related problems:

- How can a class (the same construction process) create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

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)

https://jlordiales.wordpress.com/2012/12/13/the-builder-pattern-in-practice/

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)

https://jlordiales.wordpress.com/2012/12/13/the-builder-pattern-in-practice/

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.phor
    this.address = builder.ac
  }

  public String getFirstName(

  public String getLastName()

  public int getAge() { ... }

  public String getPhone() { ... }

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

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

```
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;
  }

  erBuilder age(int age) {
  e = age;
  this;

  erBuilder phone(String phone) {
  one = phone;
  this;

  // ...

  }
}
```

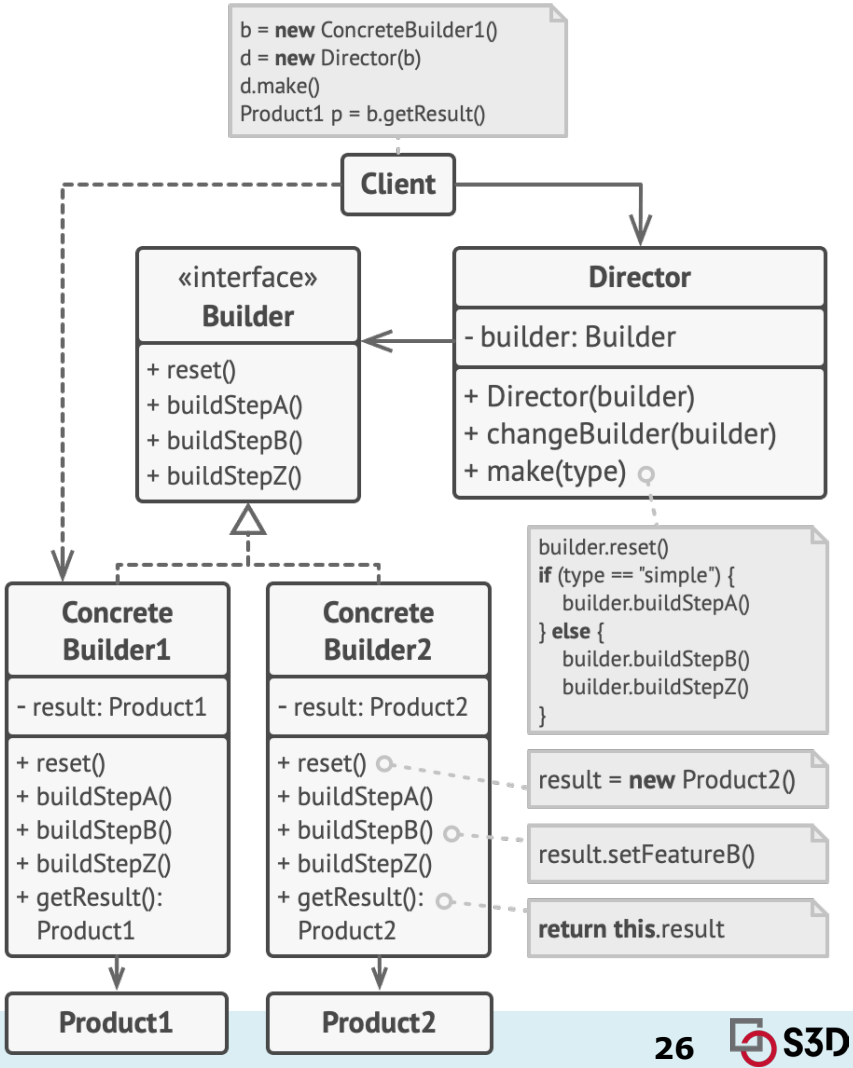

https://jlordiales.wordpress.com/2012/12/13/the-builder-pattern-in-practice/

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

- 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



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

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) { ... }
}
```

(New) 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

Course so far…

Creational:

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

Structural:

1. Adapter
2. Bridge
3. Composite
9. Decorator
10. Façade
11. Flyweight
12. Proxy

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

Course so far…

Creational:

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

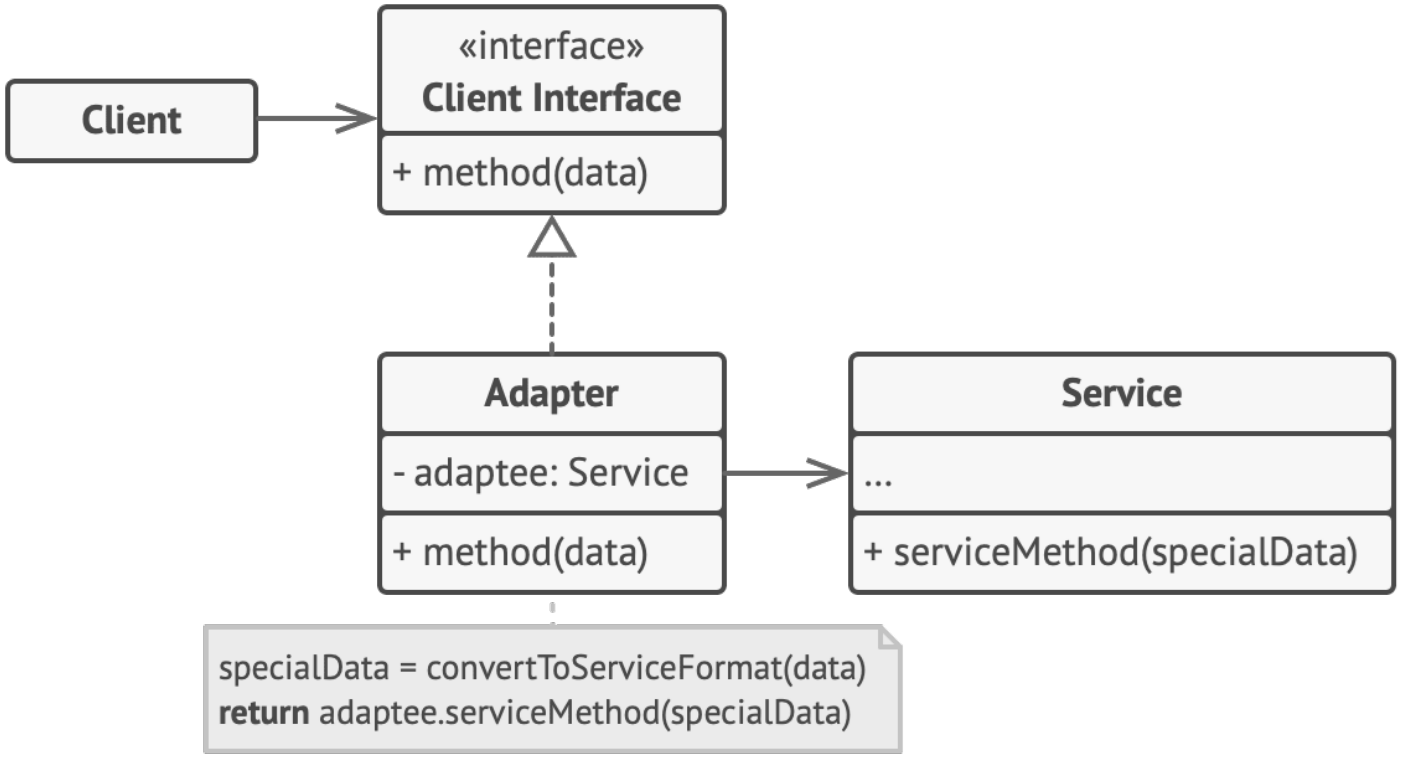
Structural:

1. *Adapter*
2. Bridge
3. *Composite*
9. *Decorator*
10. Façade
11. Flyweight
12. *Proxy*

Behavioral:

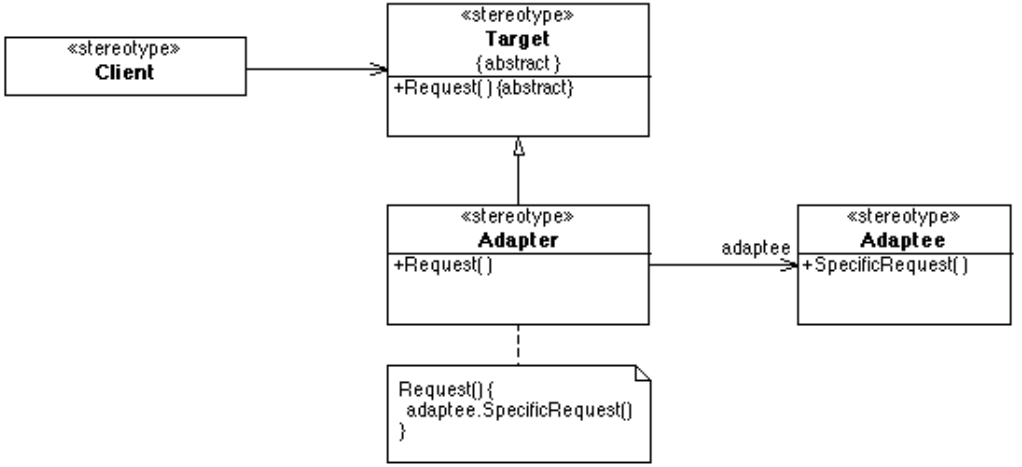
9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

Recall: The *Adapter* Design Pattern



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

Recall: The *Adapter* Design Pattern



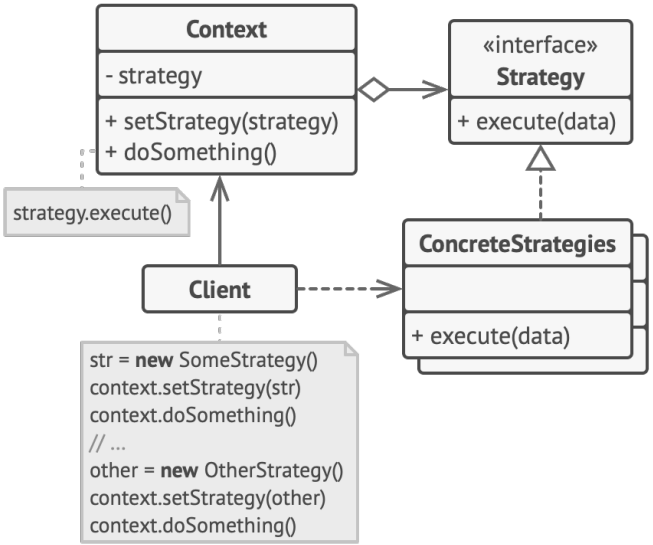
Applicability

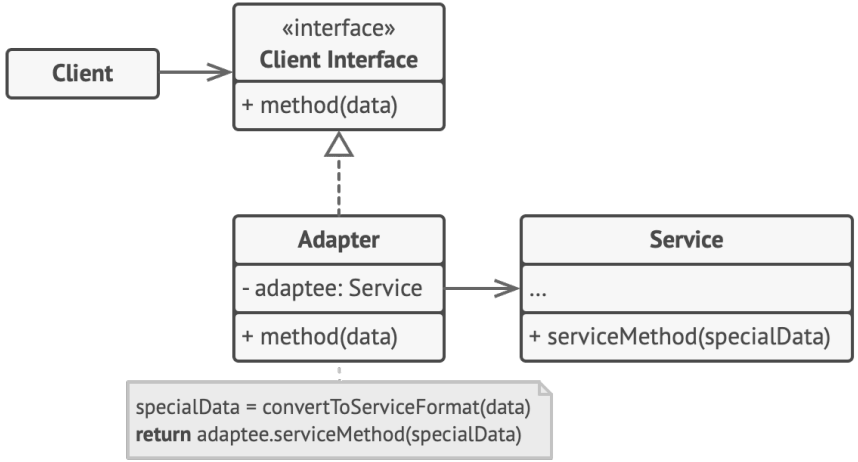
- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
- You need to use several subclasses, but it's impractical to adapt their interface by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

Adapter vs Strategy?





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

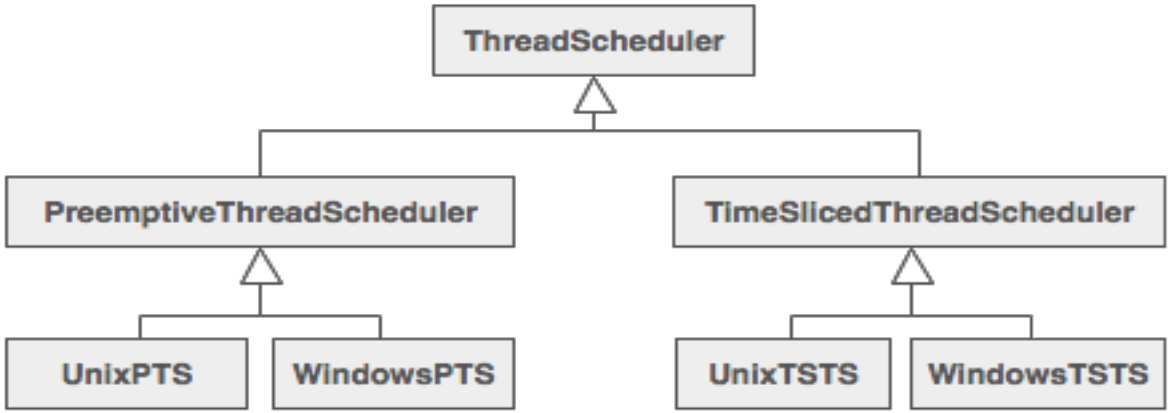
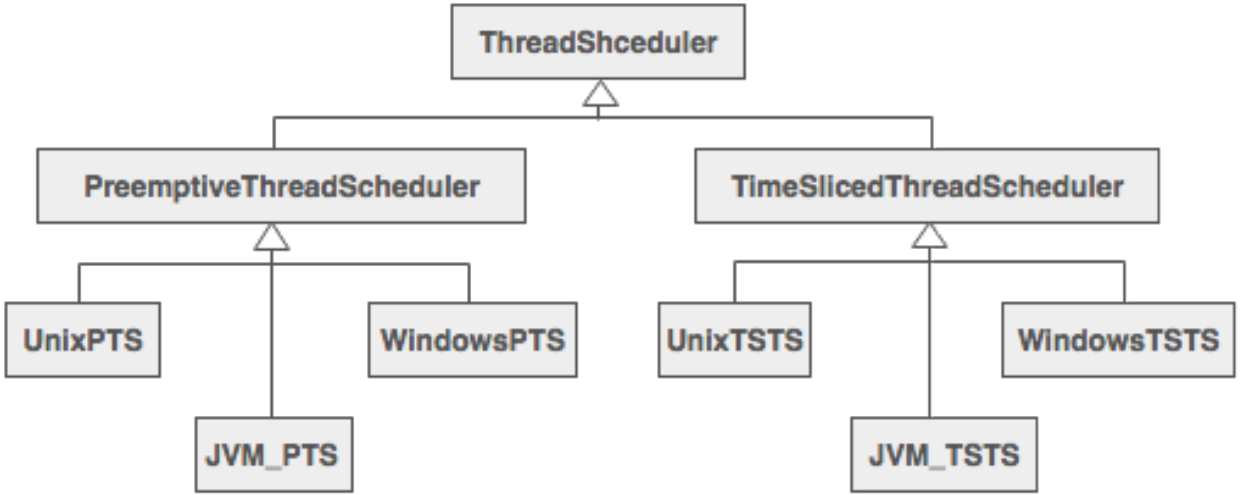


image source: https://sourcemaking.com

(New) 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.

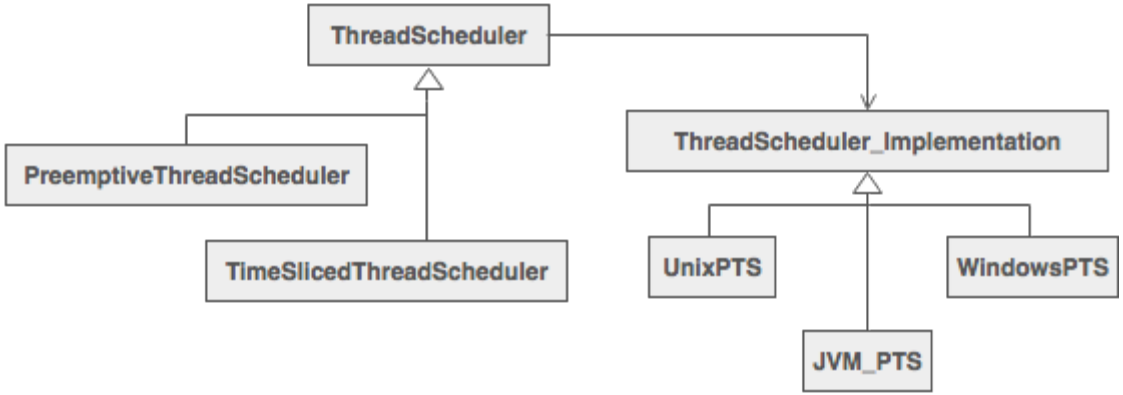


image source: https://sourcemaking.com

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 compared to…

Strategy?

Adapter?

Course so far…

Creational:

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

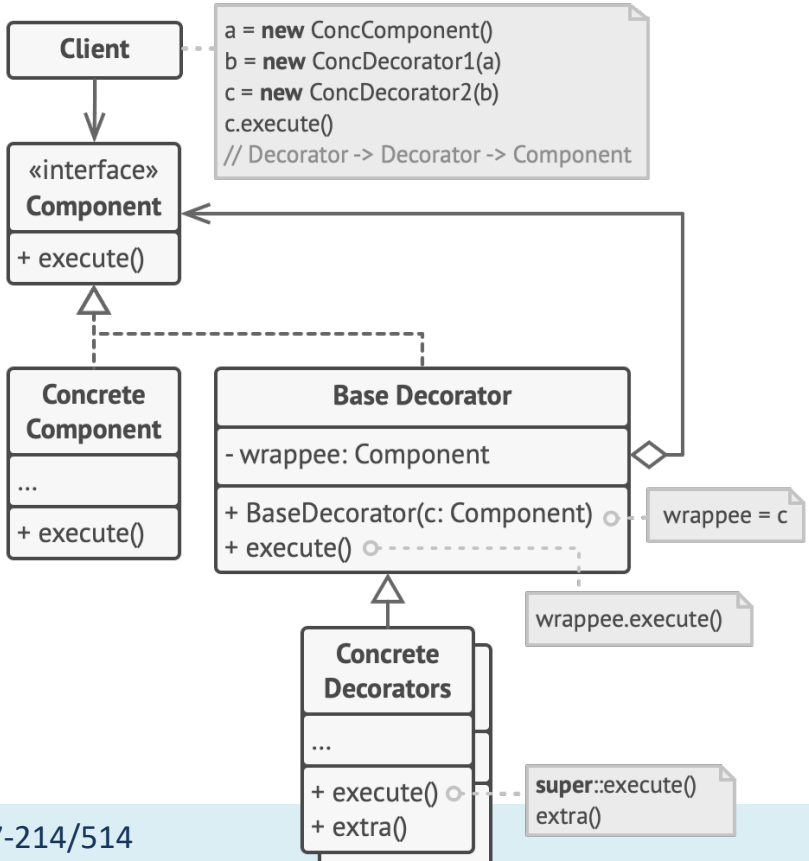
Structural:

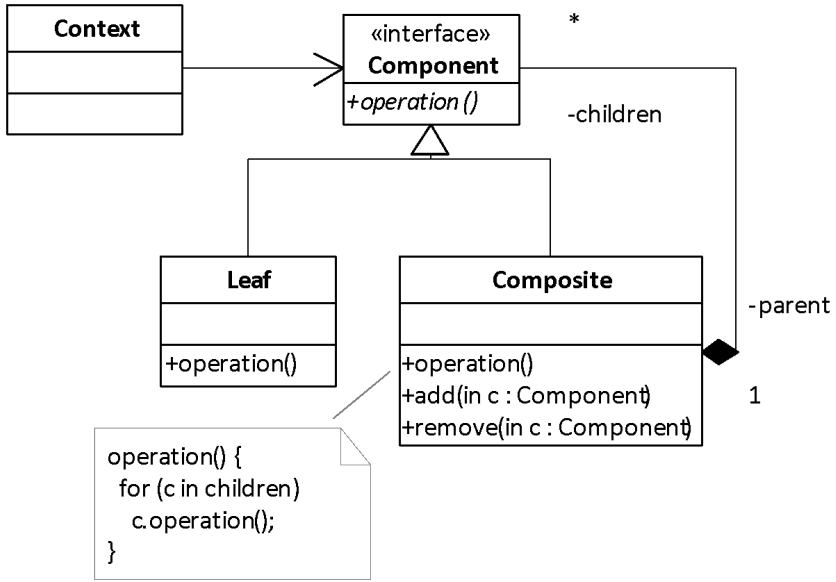
1. *Adapter*
2. Bridge
3. ***Composite***
9. ***Decorator***
10. Façade
11. Flyweight
12. *Proxy*

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

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

Context: 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.

Problem: 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.

Course so far…

Creational:

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

Structural:

1. ***Adapter***
2. Bridge
3. *Composite*
9. *Decorator*
10. **Façade**
11. **Flyweight**
12. *Proxy*

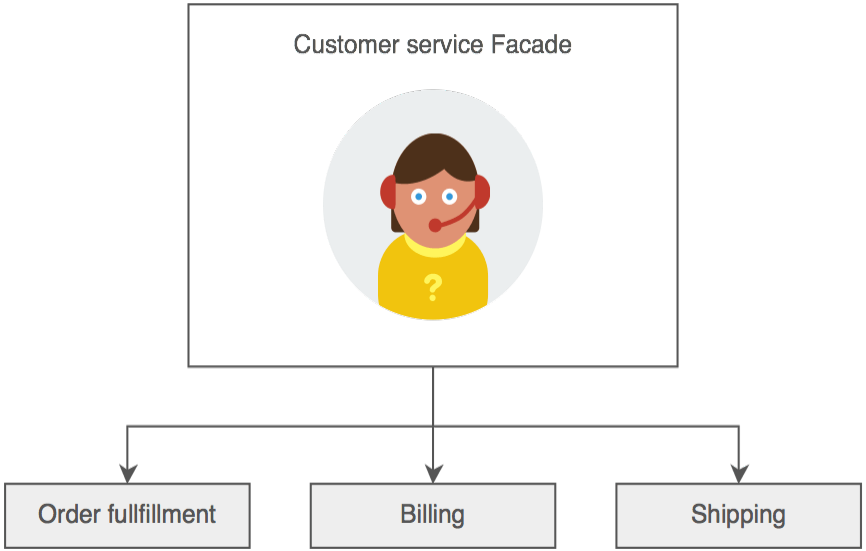
Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

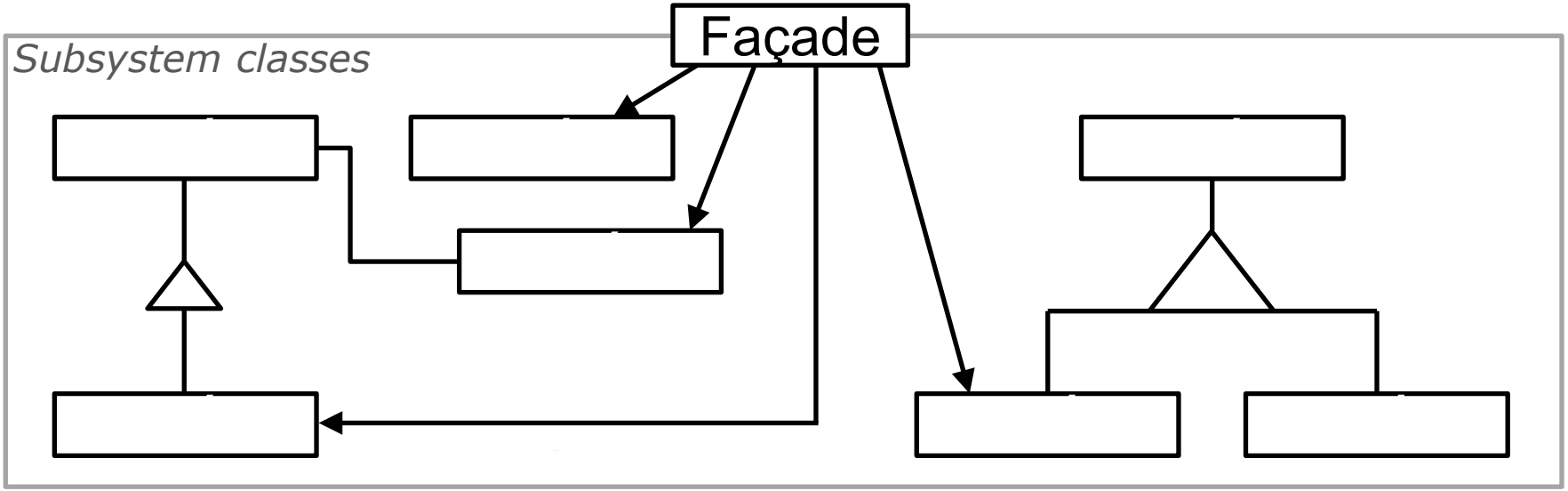
Façade Pattern

- 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 example



Façade Illustration



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

Facade vs…

…Controller Heuristic

Same idea

Facade for subsystem, controller for use case

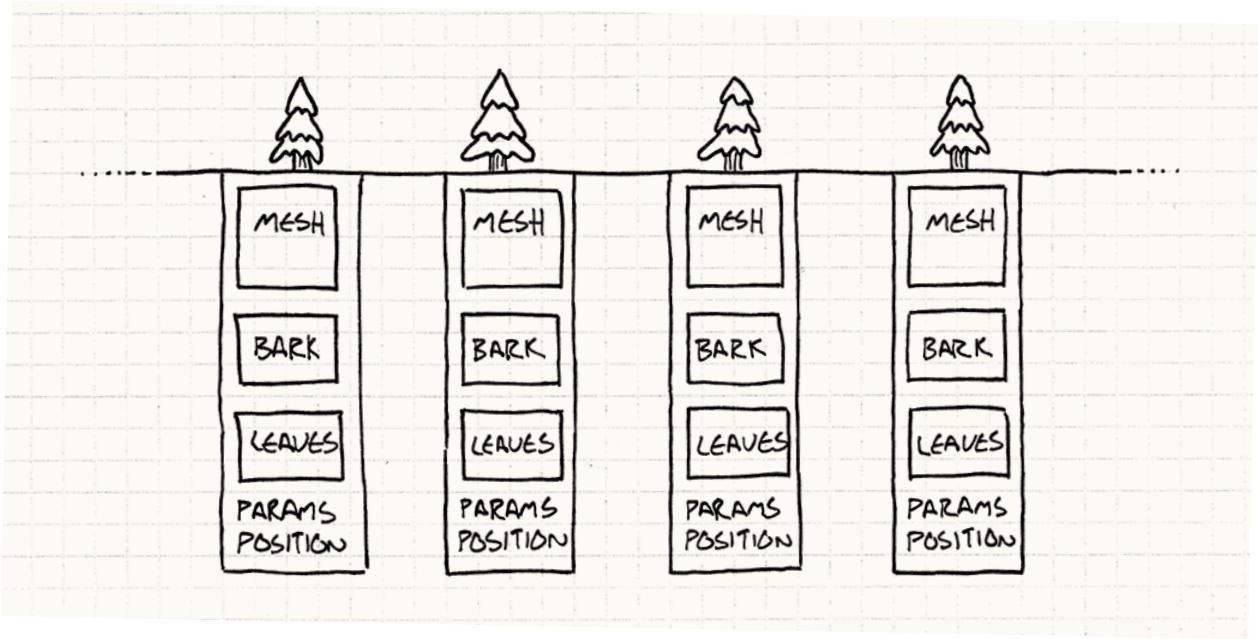
…Singleton

Facade sometimes a global variable

Typically little design for change/extension

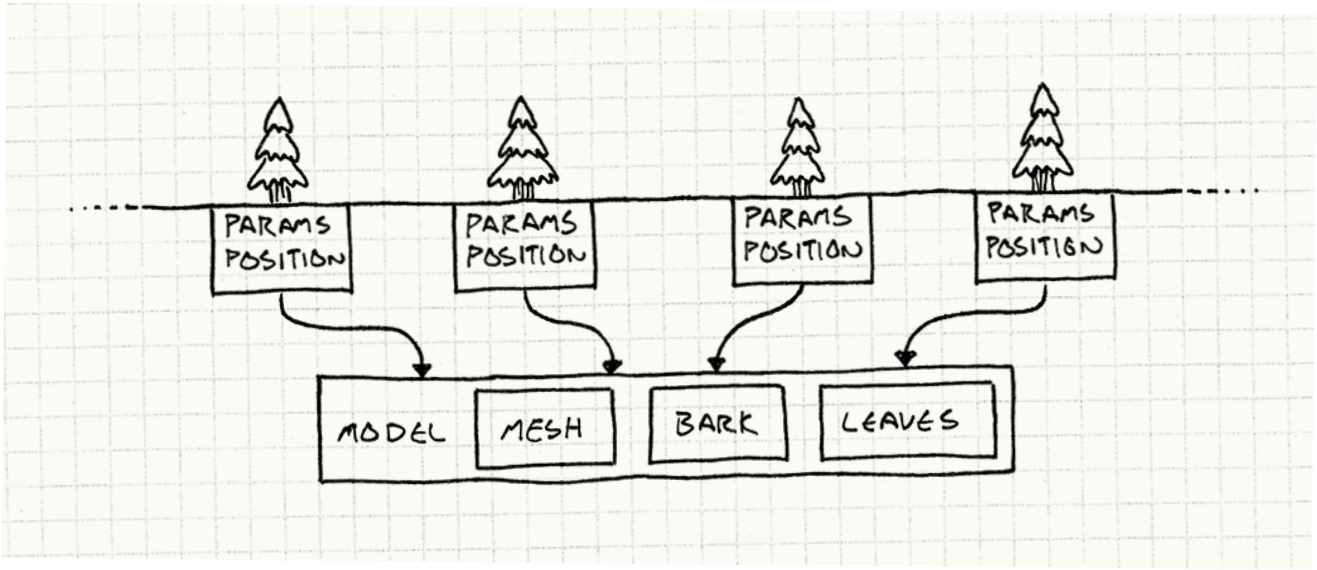
…Adapter?

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

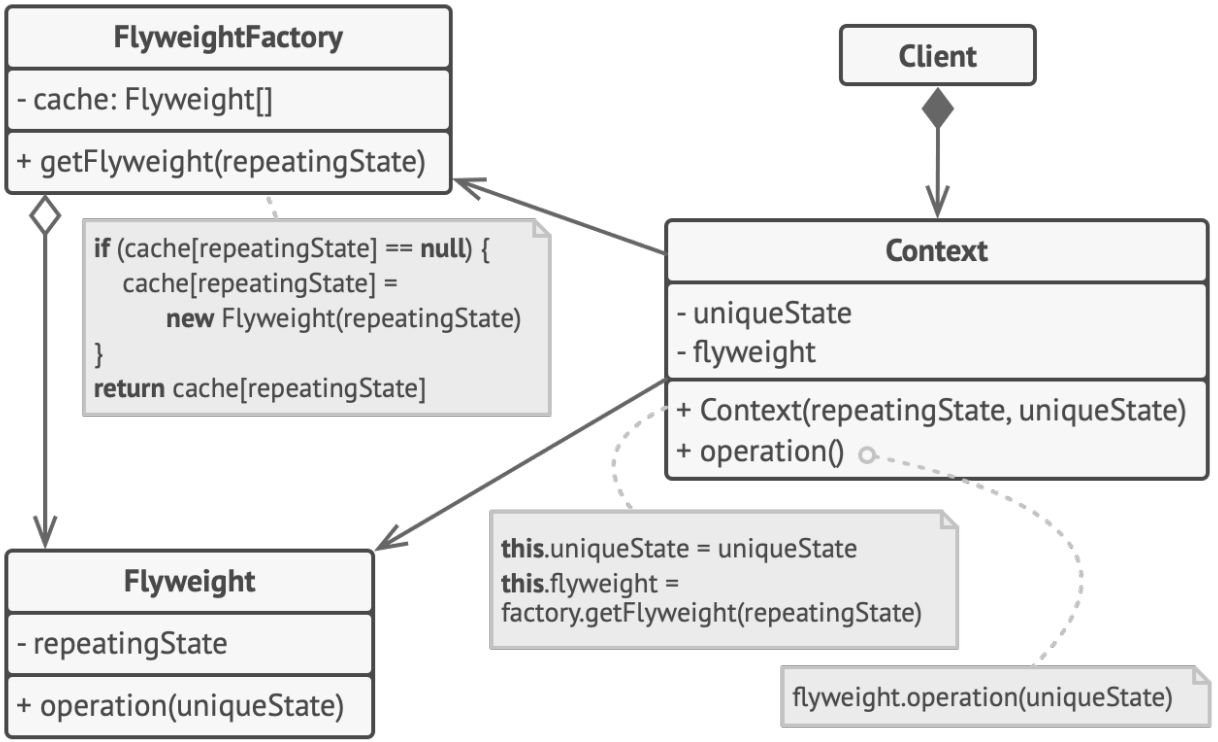
Flyweight Pattern

- 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

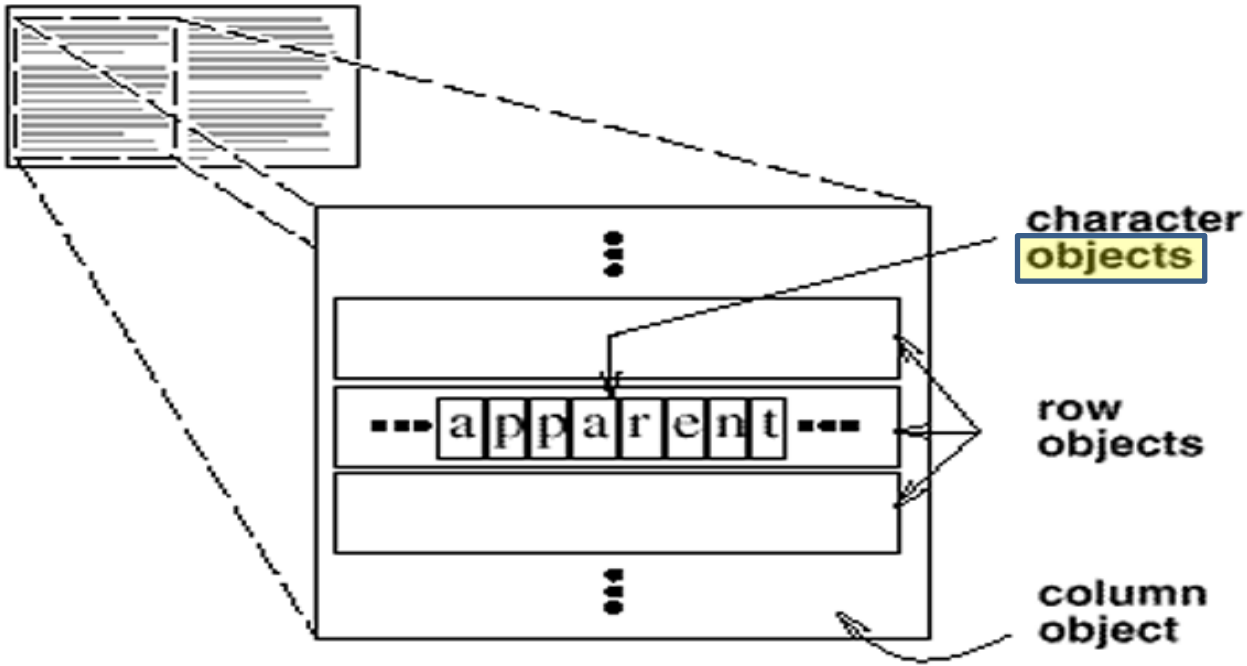
Key idea: Avoid copies of structurally equal objects, reuse object

Requires immutable objects and factory with caching



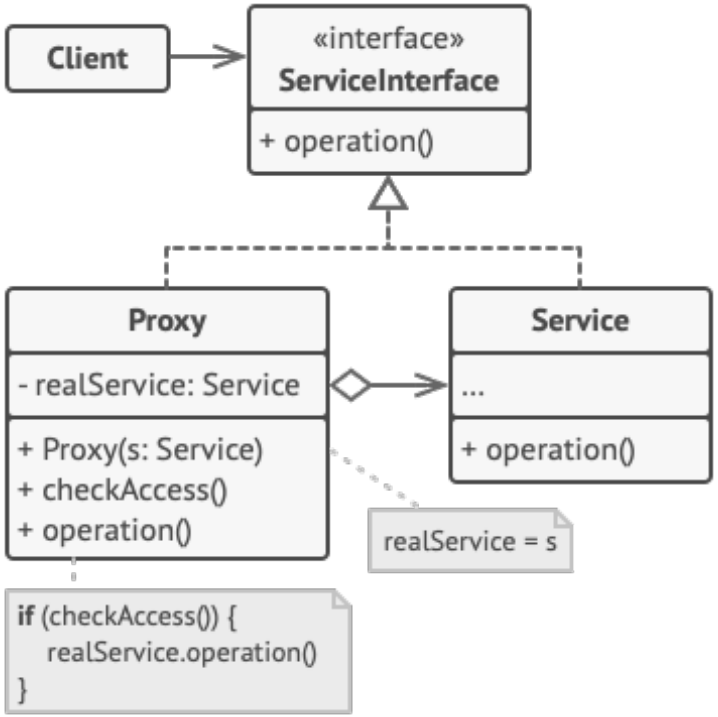
https://refactoring.guru/design-patterns/flyweight

Flyweight Illustration



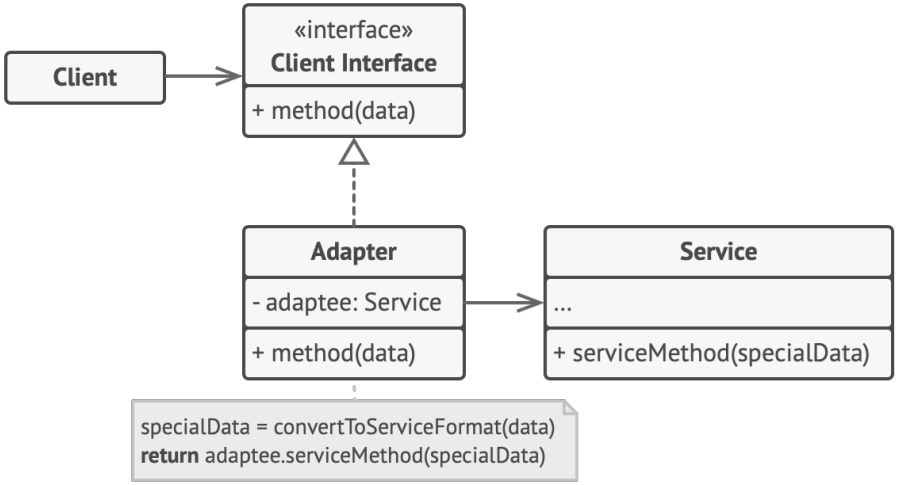
Recall: Proxy Design Pattern

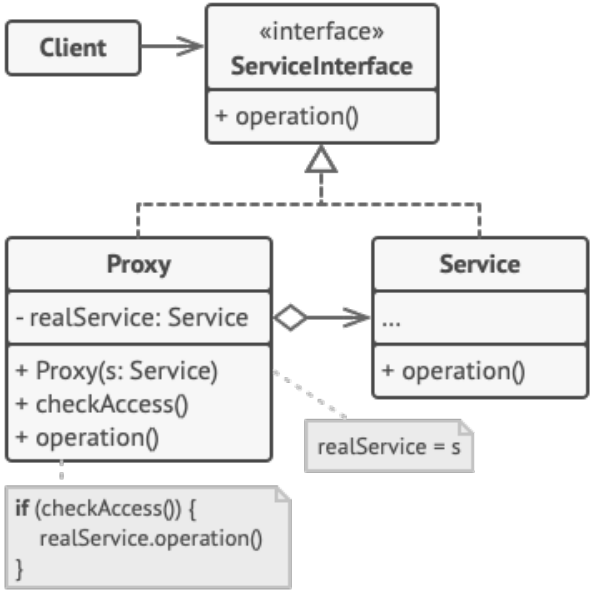
- Local representative for remote object
  - Create expensive obj on-demand
  - Control access to an object
- Hides extra “work” from client
  - Add extra error handling, caching
  - Uses *indirection*



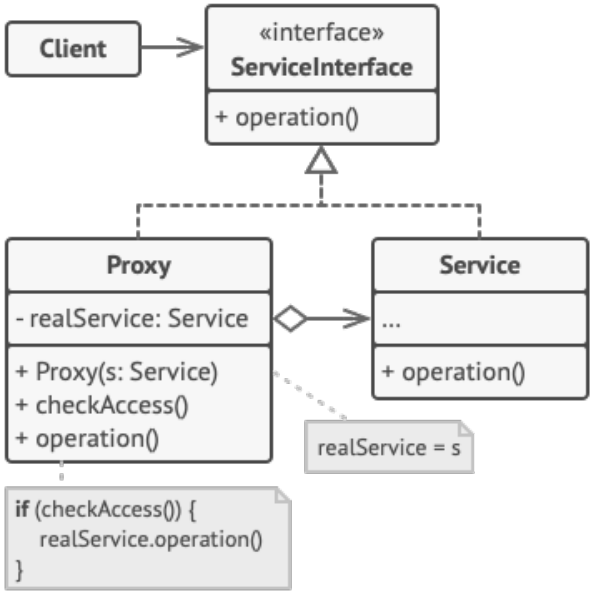
https://refactoring.guru/design-patterns/proxy

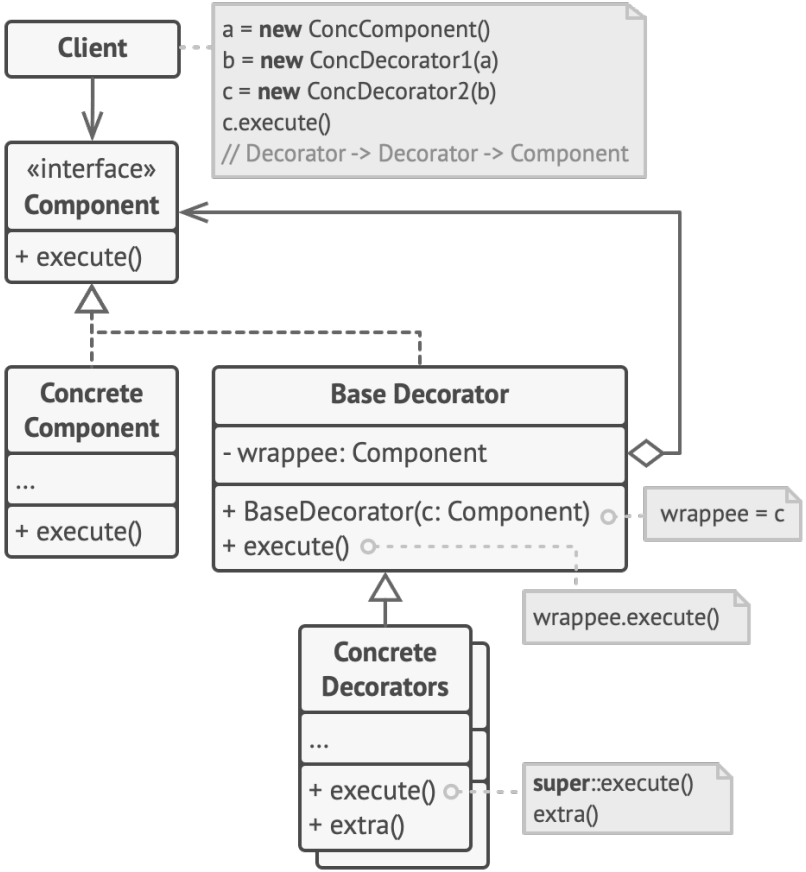
Proxy vs Adapter?





Proxy vs Decorator?





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.

Course so far…

Creational:

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

Structural:

1. *Adapter*
2. Bridge
3. *Composite*
9. *Decorator*
10. Façade
11. Flyweight
12. *Proxy*

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

Course so far…

Creational:

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

Structural:

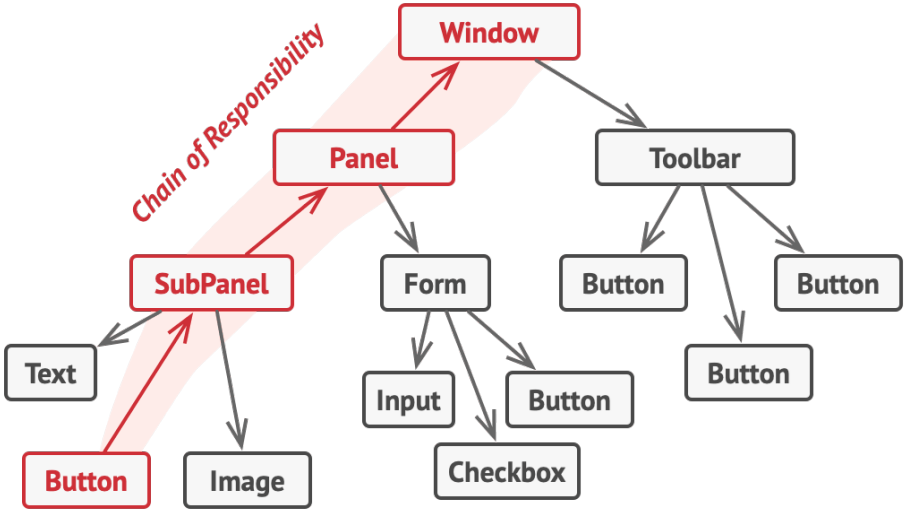
1. *Adapter*
2. Bridge
3. *Composite*
9. *Decorator*
10. Façade
11. Flyweight
12. *Proxy*

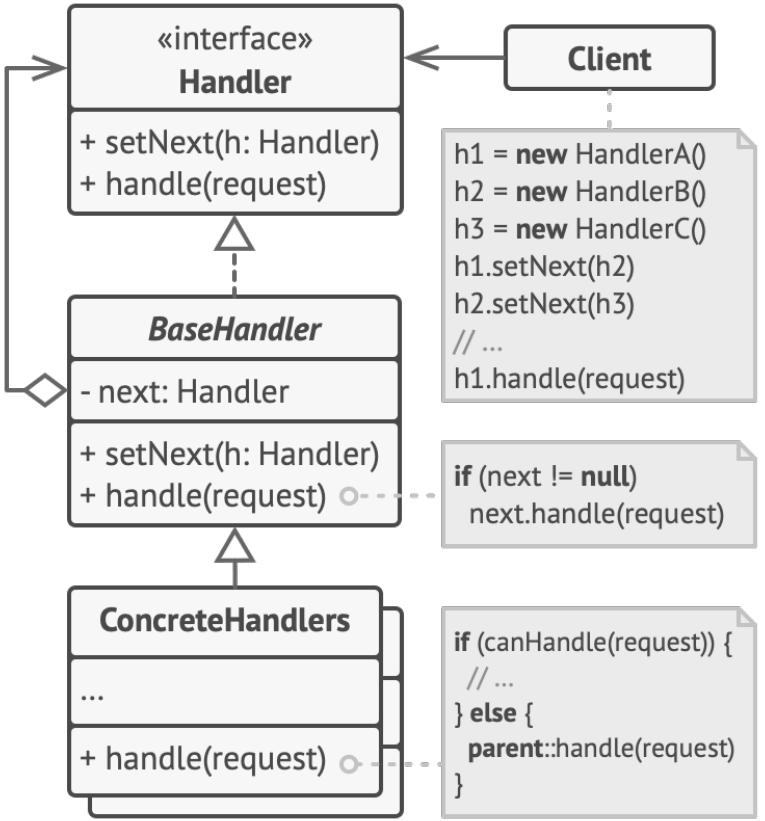
Behavioral:

9. Chain of Responsibility
10. Command
11. ~~Interpreter~~
16. *Iterator*
17. ~~Mediator~~
18. ~~Memento~~
19. *Observer*
20. ~~State~~
21. *Strategy*
22. *Template method*
23. ~~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





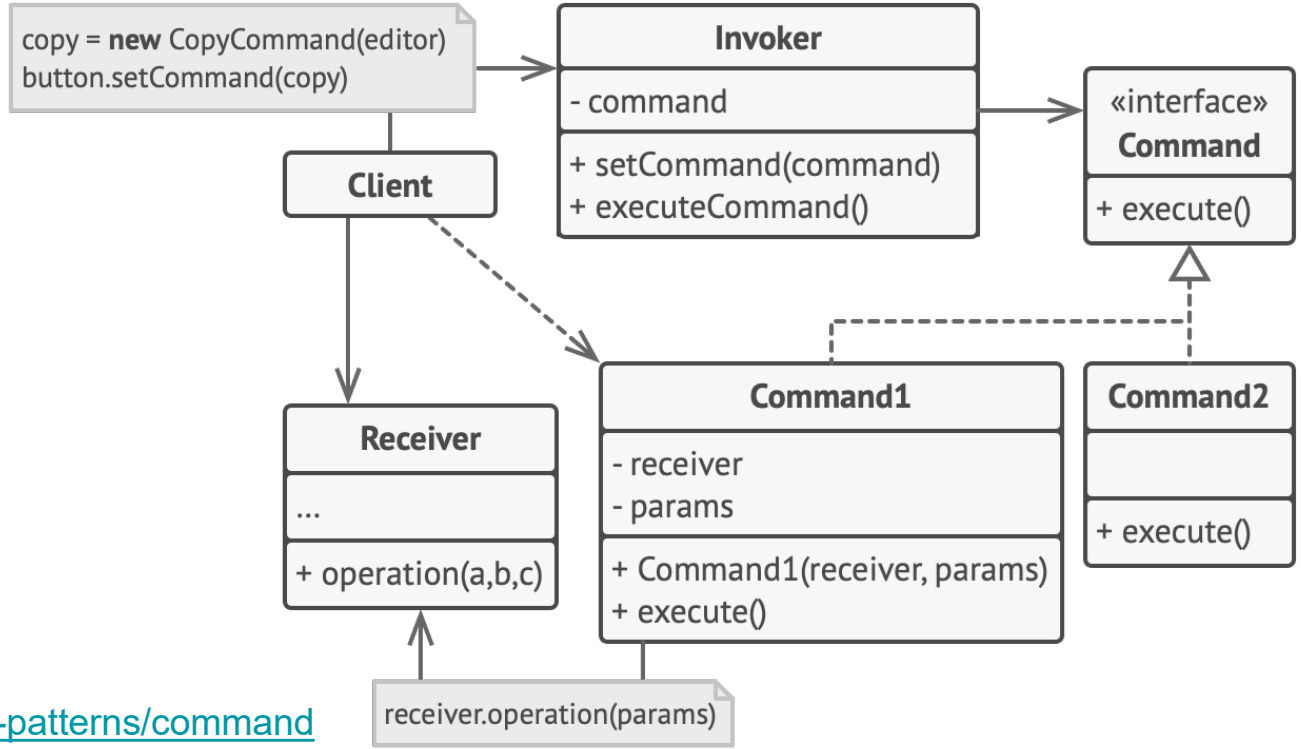
https://refactoring.guru/design-patterns/chain-of-responsibility

S3D

Command Pattern

- 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. -- see higher order function
- Commands may:
  - be run repeatedly
  - take an argument
  - return a value

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, ...

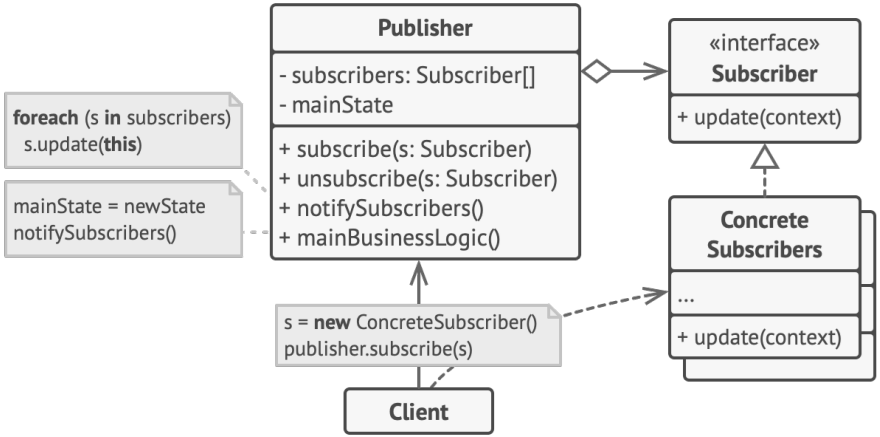
Reminder: Iterator Pattern

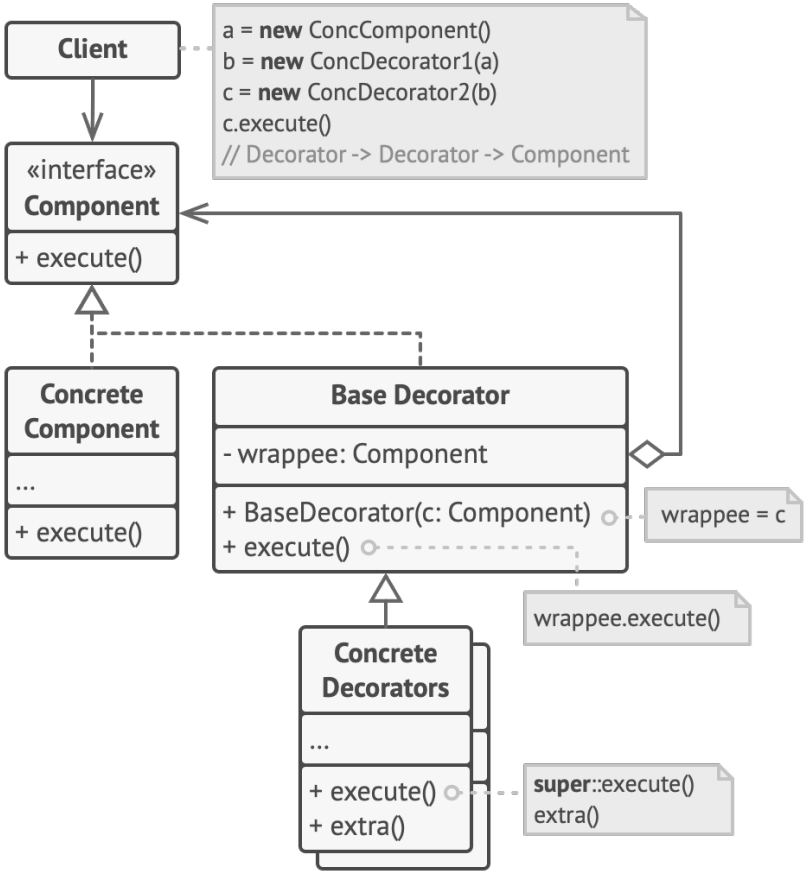
- Intent – provide a way to access elements of a collection without exposing representation
- Use case – collections
- Key types – *Iterable*, *Iterator*
  - But GoF discuss internal iteration, too
- Java and JavaScript: collections, for-each statement ..

Reminder: 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);
```

Observer vs. Decorator?





Observer vs. Promise?

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.

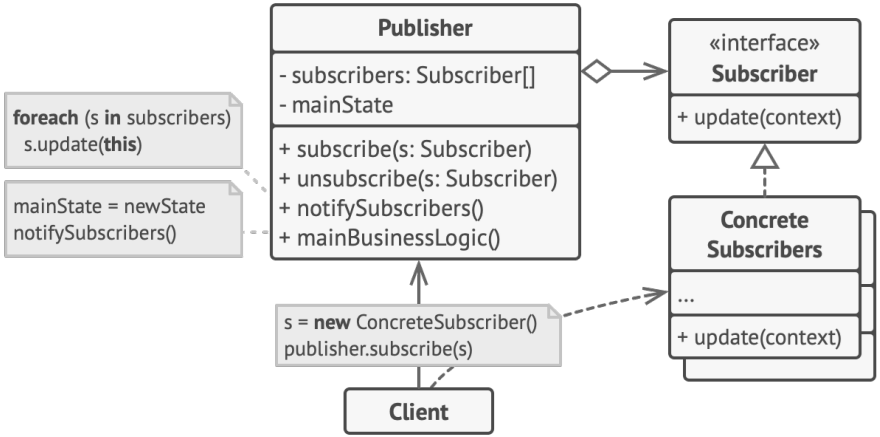
Design problem 6: Every time a report is resolved, one of multiple actions should be taken (email, text message, …). The action is selected by the person creating the report.

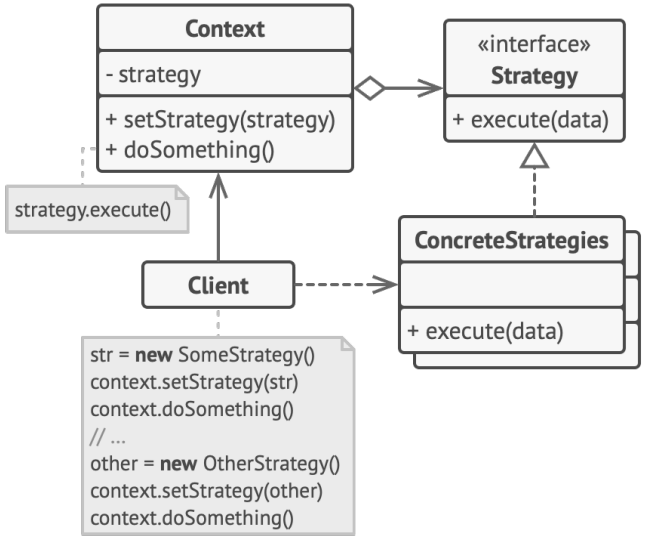
Design Problem

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.

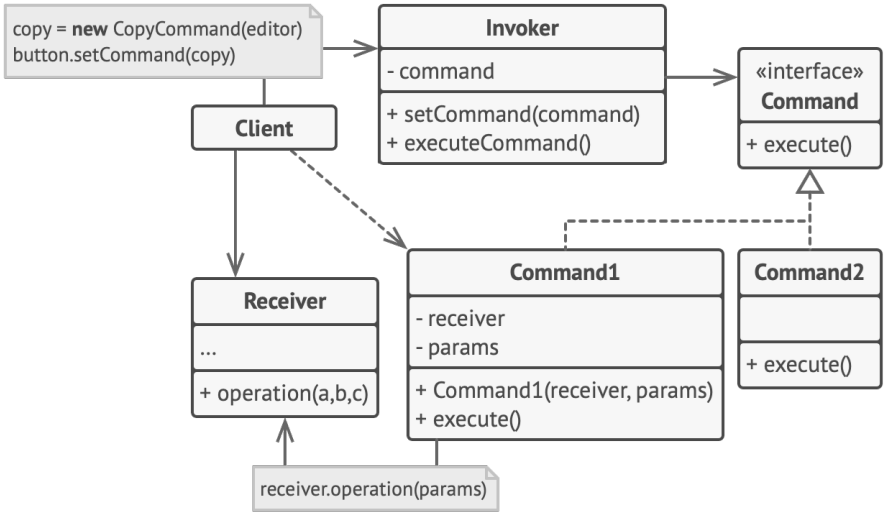
Design problem 7: Every time a report is resolved, multiple follow-up actions should be performed. Results should be added to a database, an email should be sent, a supervisor should be informed, etc. More actions might be added later.

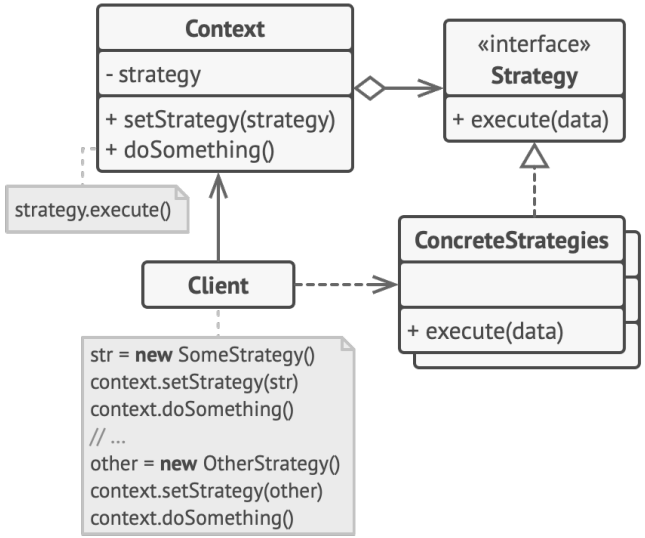
Observer vs. Strategy





Command vs. Strategy





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

Course so far…

Creational:

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

Structural:

1. **Adapter**
2. **Bridge**
3. **Composite**
9. **Decorator**
10. **Façade**
11. **Flyweight**
12. **Proxy**

Behavioral:

9. **Chain of Responsibility**
10. **Command**
11. Interpreter
16. **Iterator**
17. Mediator
18. Memento
19. **Observer**
20. State
21. **Strategy**
22. **Template method**
23. Visitor

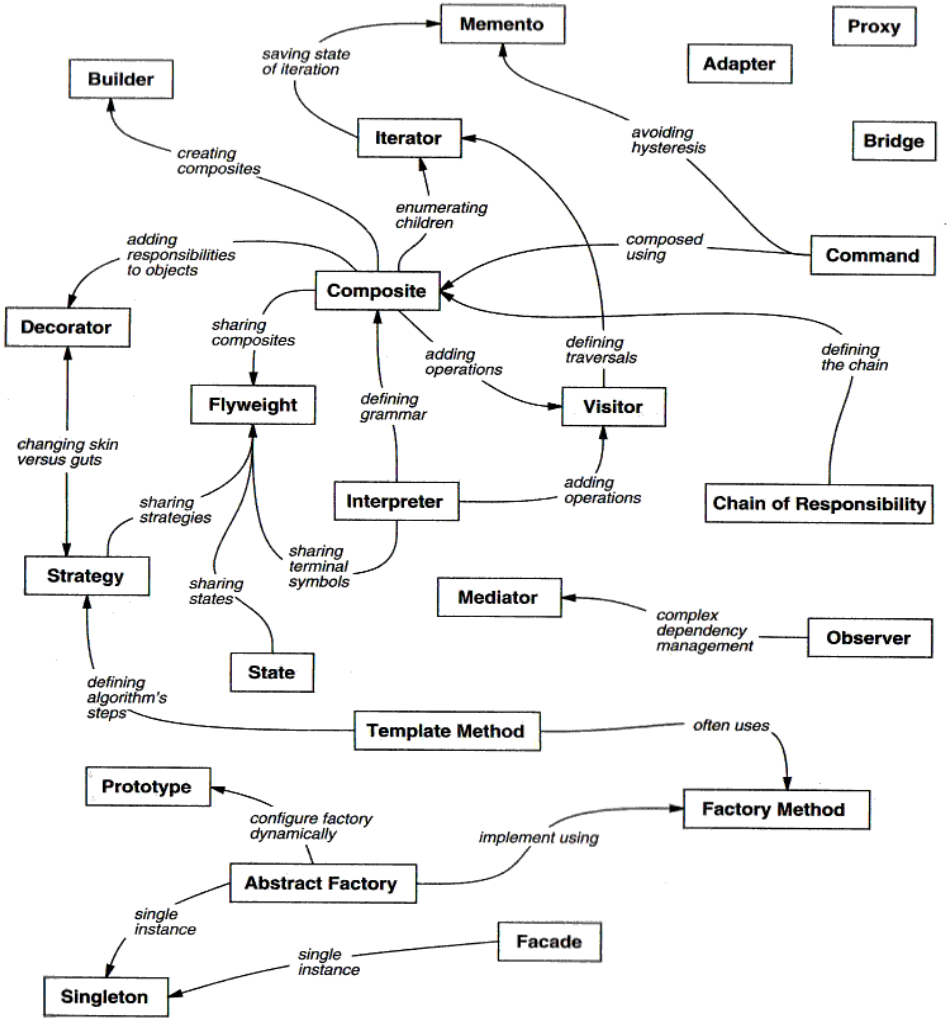


Figure 1.1: Design pattern relationships

Patterns I am discussing only very briefly for various reasons

Creational: Prototype Pattern

- Intent – create an object by cloning another and tweaking as necessary
- Key types – *Prototype*
- Java: `Cloneable`, but avoid (except on arrays)
- JavaScript: Builtin language feature
- *Not discussing it because it's powerfully error-prone when it's not built-in.*

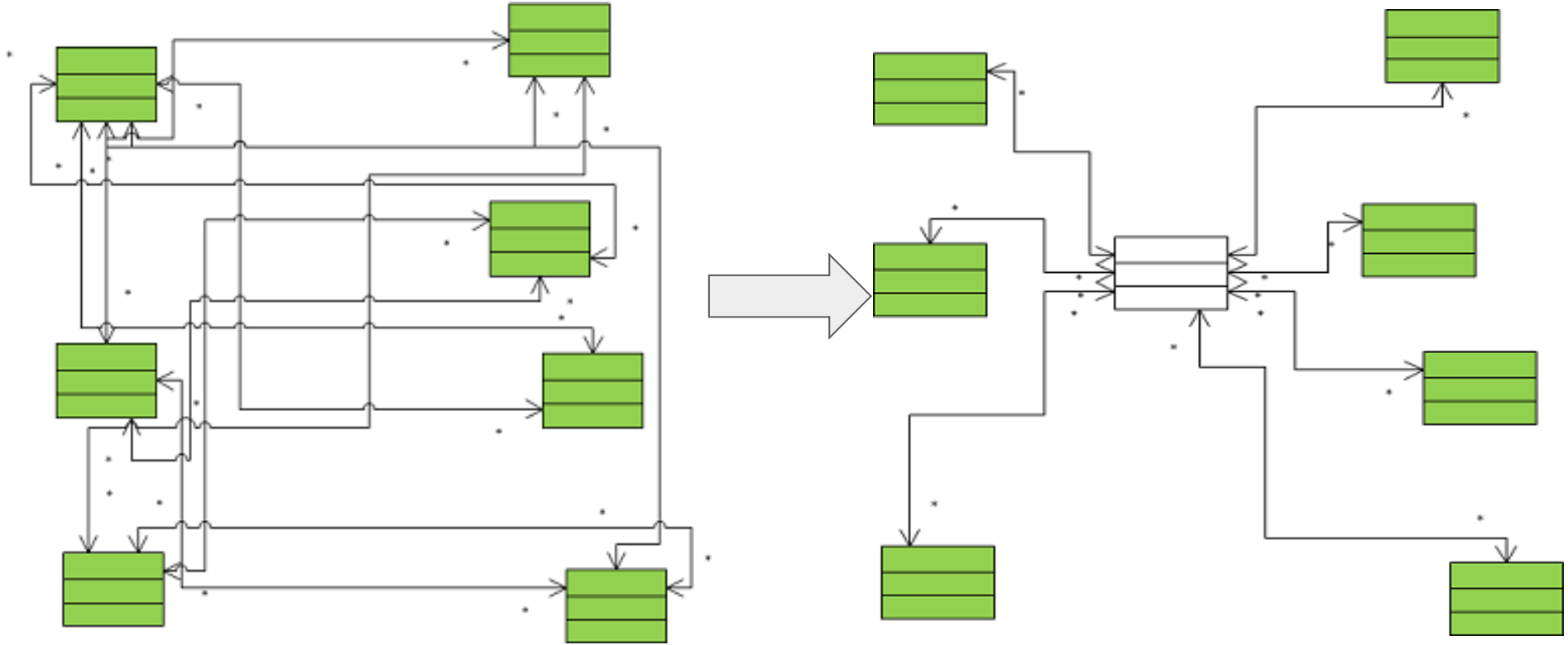
Behavioral: Interpreter Pattern

- 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*
- *Discussing only briefly because it's kind of a specialization of the Composite pattern. Also, take a PL class.*

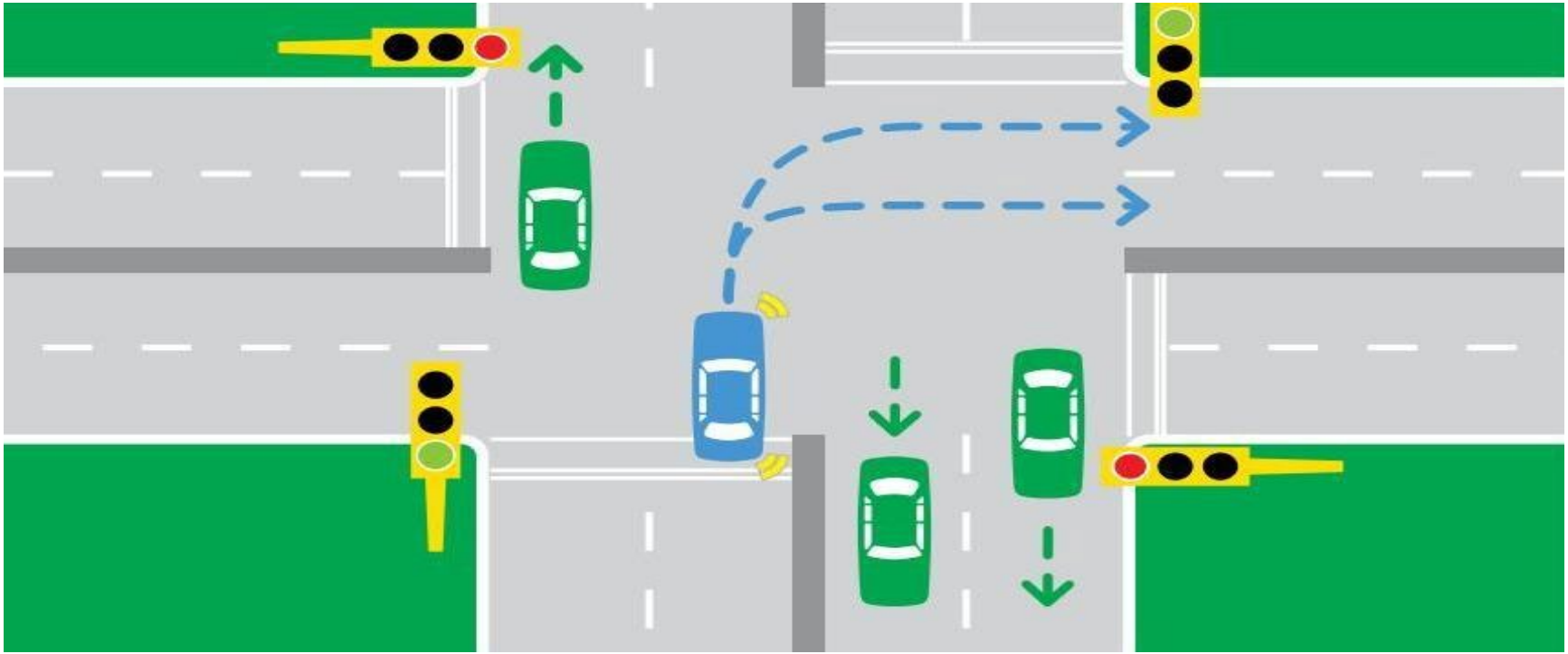
Mediator Pattern

- 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

Problem:



Mediator Illustration



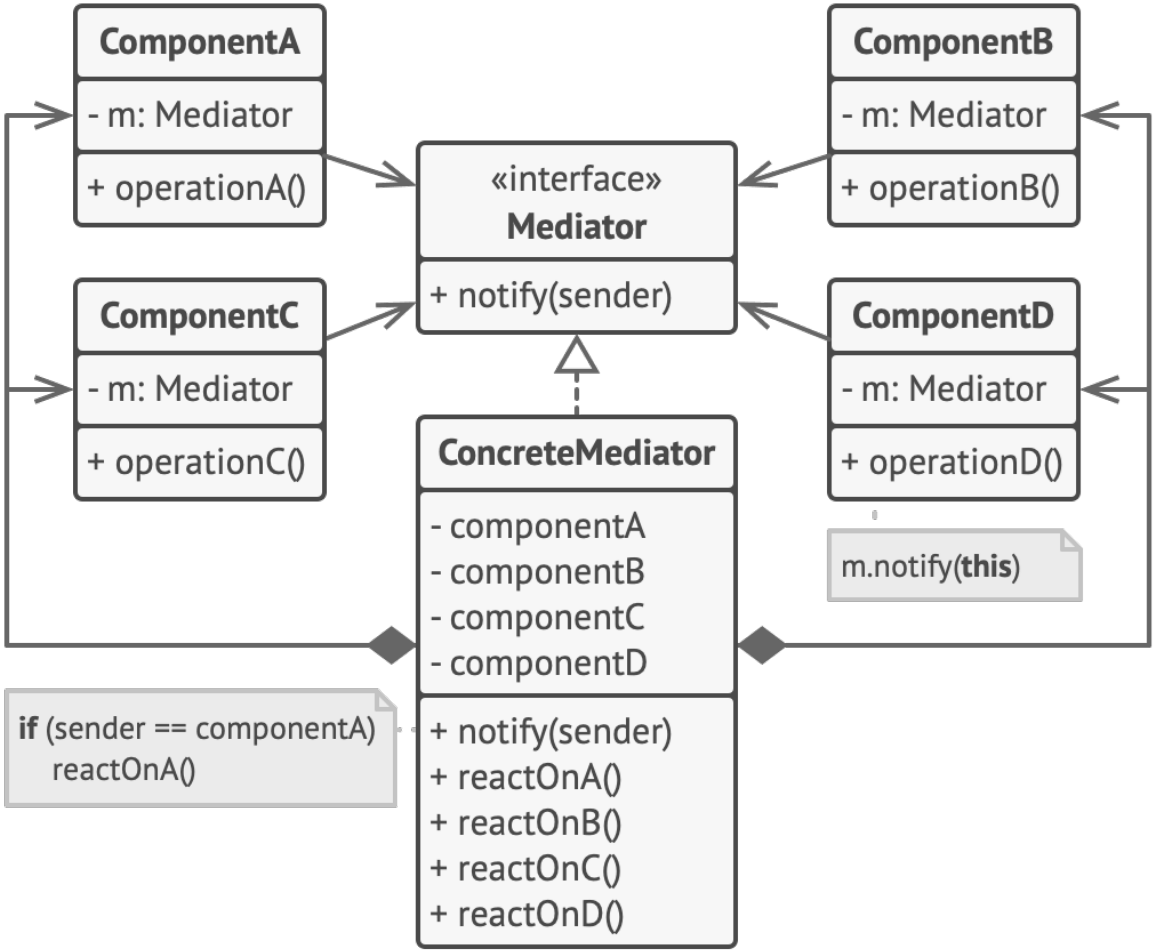
Single responsibility at mediator

- Coupling to single component

Intent – define an object that encapsulates how a set of objects interact, to reduce coupling.

- $\mathcal{O}$(n) couplings instead of $\mathcal{O}$(n2)

Discussing it only briefly because it's intuitive, and also turns into a god object if you're not careful.



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

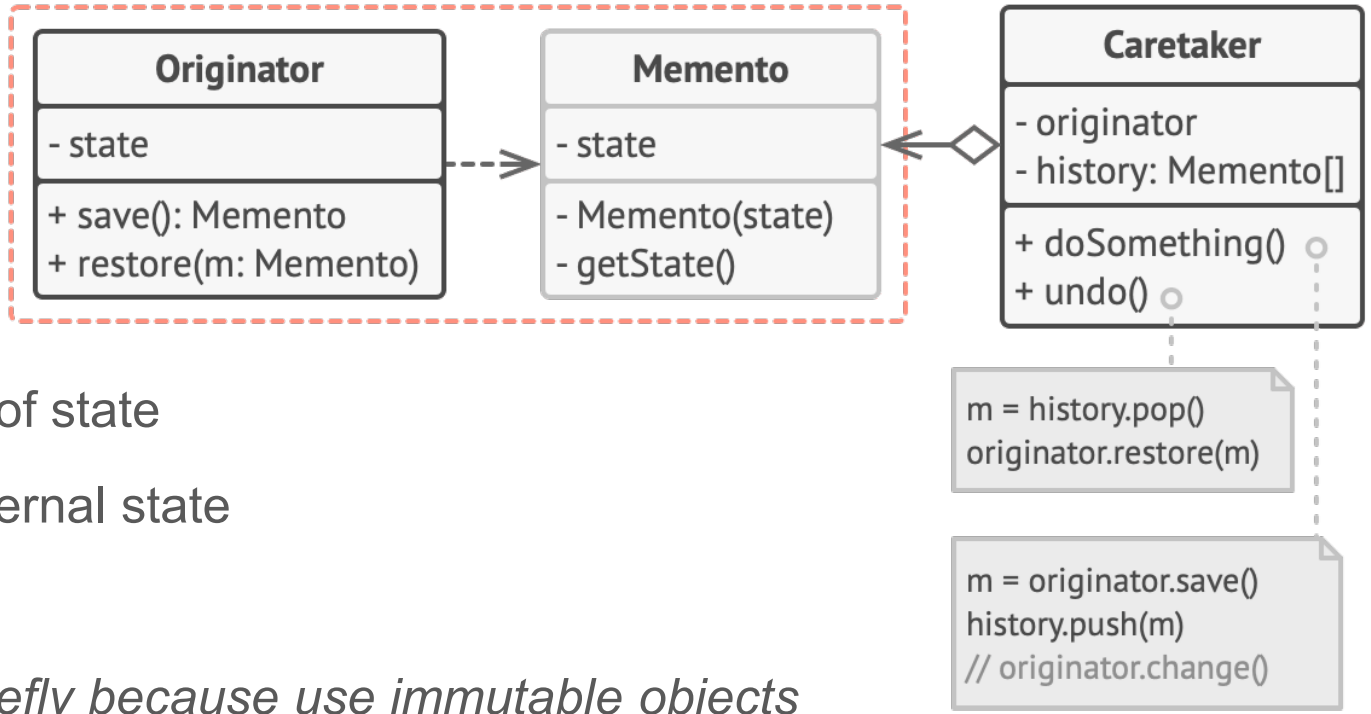
```
public class Editor {
  //state
  public String editorContents;
  public void setState(String contents) {
    this.editorContents = contents;
  }
  public EditorMemento save() {
    return new EditorMemento(editorContents);
  }
  public void restoreToState(EditorMemento memento) {
    editorContents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


https://dzone.com/articles/design-patterns-memento

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 Pattern



Record snapshots of state

Avoid access to internal state

Allows undo

Discussing only briefly because use immutable objects instead when you can.

Problem:

- It should be possible to change the behavior of a class's methods when its internal state changes.
  - Example: TCP/IP connections go through various states. The methods of a TCPConnection should do different things depending on the protocol state.

State Pattern 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 { … }
```

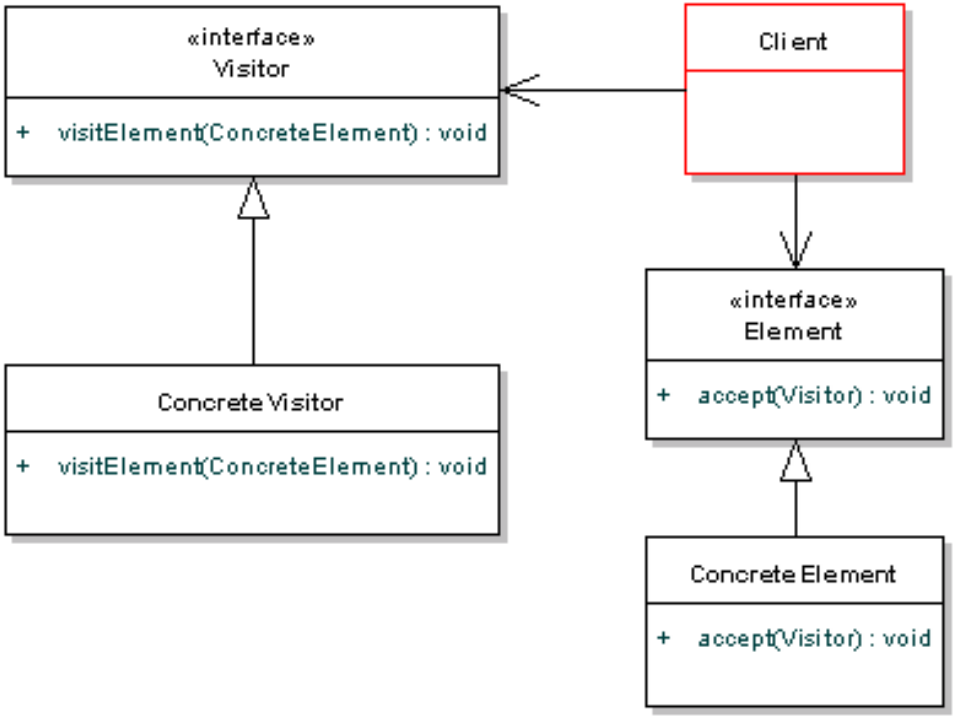
State Pattern

- Intent – allow an object to alter its behavior when internal state changes. “Object will appear to change class.”
- Use case – TCPConnection (which is stateful)
- Key type – *State* (Object delegates to state!)
- *Discussing only briefly because state machines are fairly intuitive.*

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
- *Discussing only briefly because describing it well enough that you actually could understand it would take longer than it's worth given that it's only ever used by program analysis/compiler types. If you're one of those people, go learn it.*

Visitor


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

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;
  }
}
```


https://dzone.com/articles/design-patterns-visitor

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 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

All GoF Design Patterns

Creational:

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

Structural:

1. Adapter
2. Bridge
3. Composite
9. Decorator
10. Façade
11. Flyweight
12. Proxy

Behavioral:

9. Chain of Responsibility
10. Command
11. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor

Bonus: Other Design Principles

Where we are

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