Principles of Software Construction: Objects, Design, and Concurrency

All GoF Design Patterns

Charlie Garrod **Bogdan Vasilescu**

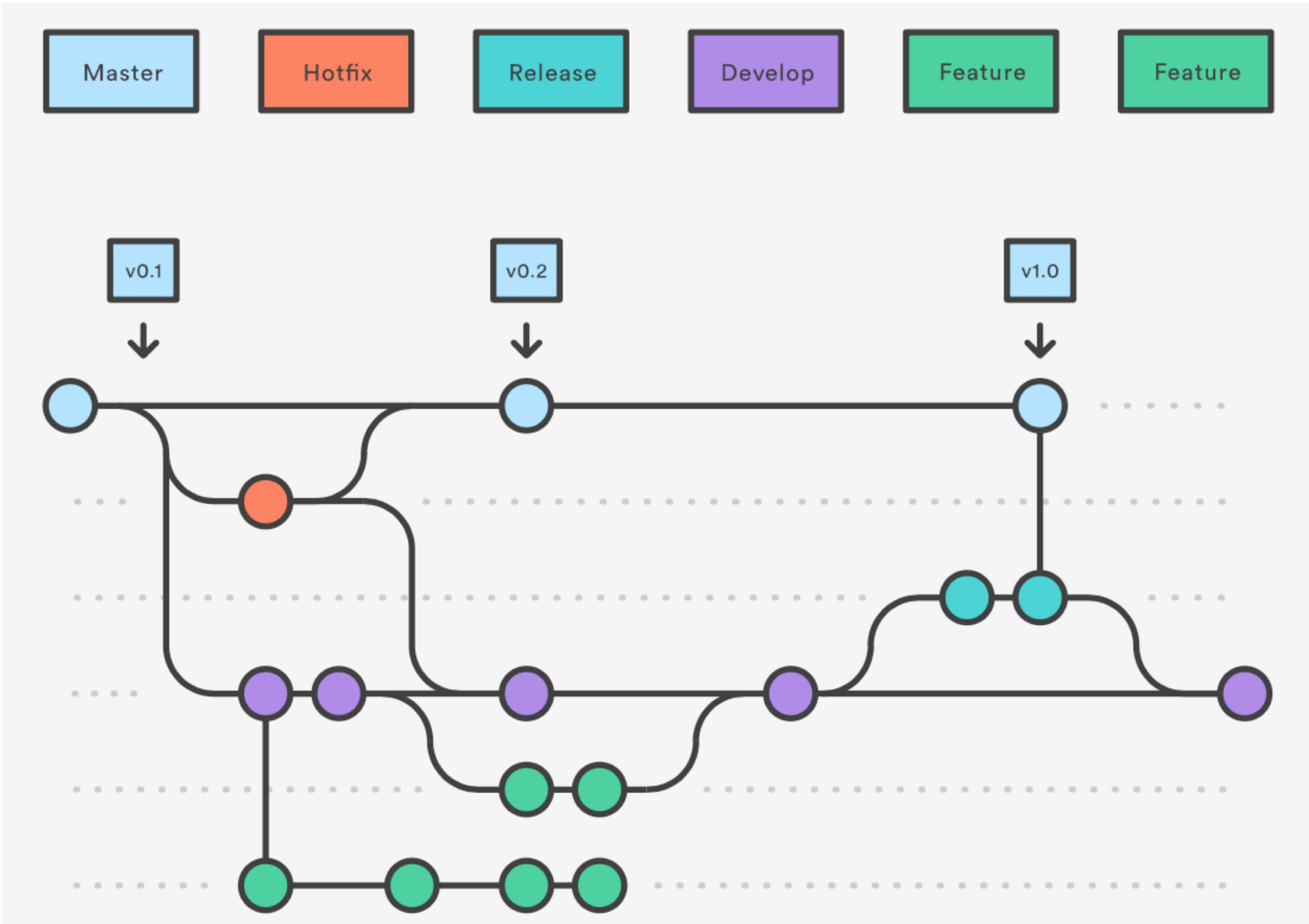
School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

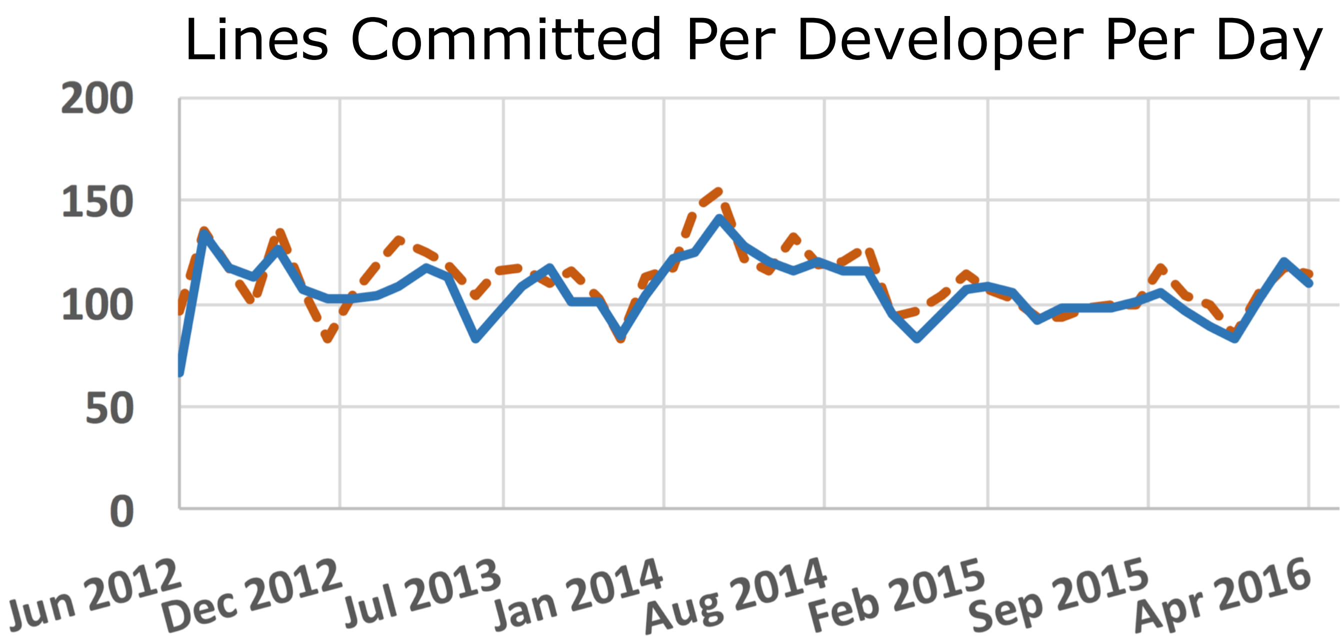
- Final exam Monday May 7th 5:30-8:30 PH 100
- Review session Saturday May 5th 2pm WH 5403

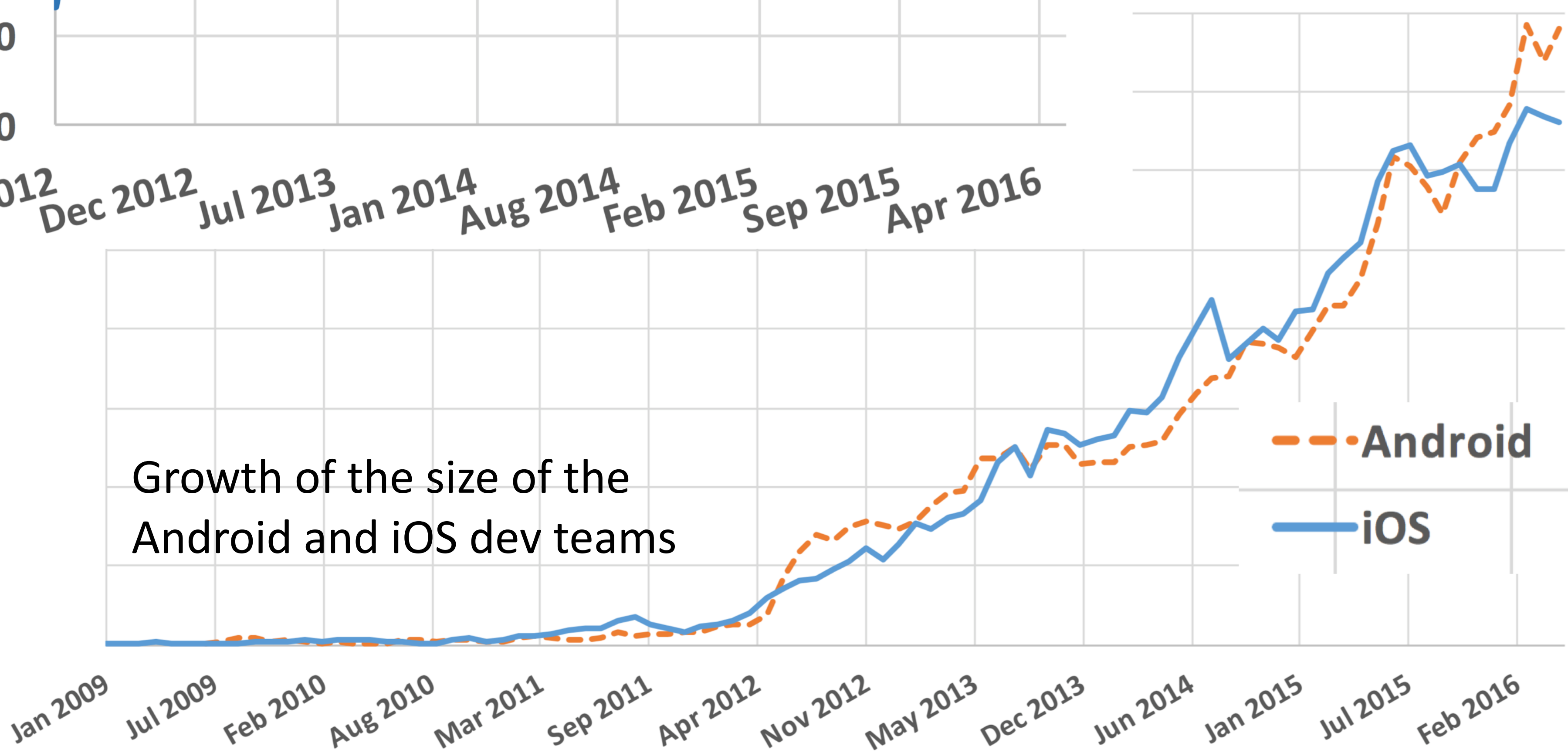
Key concepts from Thursday

GitFlow branch workflow



Coping with scale at Facebook





MONOREPO VS MANY REPOS

What is a monolithic repository (monorepo)?

- A **single** version control repository containing multiple
  - Projects
  - Applications
  - Libraries
- often using a common build system.

Monorepos in industry

Google (computer science version)

TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS | ACM.org | Join ACM | About Communications | ACM Resources | Alerts & Feeds

SIGN IN

COMMUNICATIONS OF THE ACM

Search

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | ARCHIVE | VIDEOS

Home / Magazine Archive / July 2016 (Vol. 59, No. 7) / Why Google Stores Billions of Lines of Code in a Single... / Full Text

CONTRIBUTED ARTICLES

Why Google Stores Billions of Lines of Code in a Single Repository

By Rachel Potvin, Josh Levenberg
Communications of the ACM, Vol. 59 No. 7, Pages 78-87
10.1145/2854146
Comments (3)

VIEW AS: SHARE:

Early Google employees decided to work with a shared codebase managed through a centralized source control system. This approach has served Google well for more than 16 years, and today the vast majority of Google's software assets continues to be stored in a single, shared repository. Meanwhile, the number of Google software developers has steadily increased, and the size of the Google codebase has grown exponentially (see Figure 1). As a result, the technology used to host the codebase has also evolved significantly.

Back to Top

Key Insights

SIGN IN for Full Access
User Name
Password
» Forgot Password?
» Create an ACM Web Account
SIGN IN

ARTICLE CONTENTS:
Introduction
Key Insights
Google-Scale
Background
Analysis
Alternatives

Monorepos in industry

Code

Search

Open Source | Platforms ▾ | Infrastructure Systems ▾ | Hardware Infrastructure ▾ | Video & VR ▾ | Artificial Intelligence ▾

7 January 2014 INFRA · OPEN SOURCE · PERFORMANCE · OPTIMIZATION

Scaling Mercurial at Facebook

Durham Goode Siddharth P Agarwal

With thousands of commits a week across hundreds of thousands of files, Facebook's main source repository is enormous--many times larger than even the Linux kernel, which checked in at 17 million lines of code and 44,000 files in 2013. Given our size and complexity—and Facebook's practice of shipping code twice a day--improving our source control is one way we help our engineers move fast.

Choosing a source control system

Two years ago, as we saw our repository continue to grow at a staggering rate, we sat down and extrapolated our growth forward a few years. Based on those projections, it appeared likely that our then-current technology, a Subversion server with a Git mirror, would become a productivity bottleneck very soon. We looked at the available options and found none that were both fast and easy to use at scale.

Our code base has grown organically and its internal dependencies are very complex. We could have spent a lot of time making it more modular in a way that would be friendly to a source control tool, but there are a number of benefits to using a single repository. Even at our current scale, we often make large changes throughout our code base, and having a single repository is useful for continuous

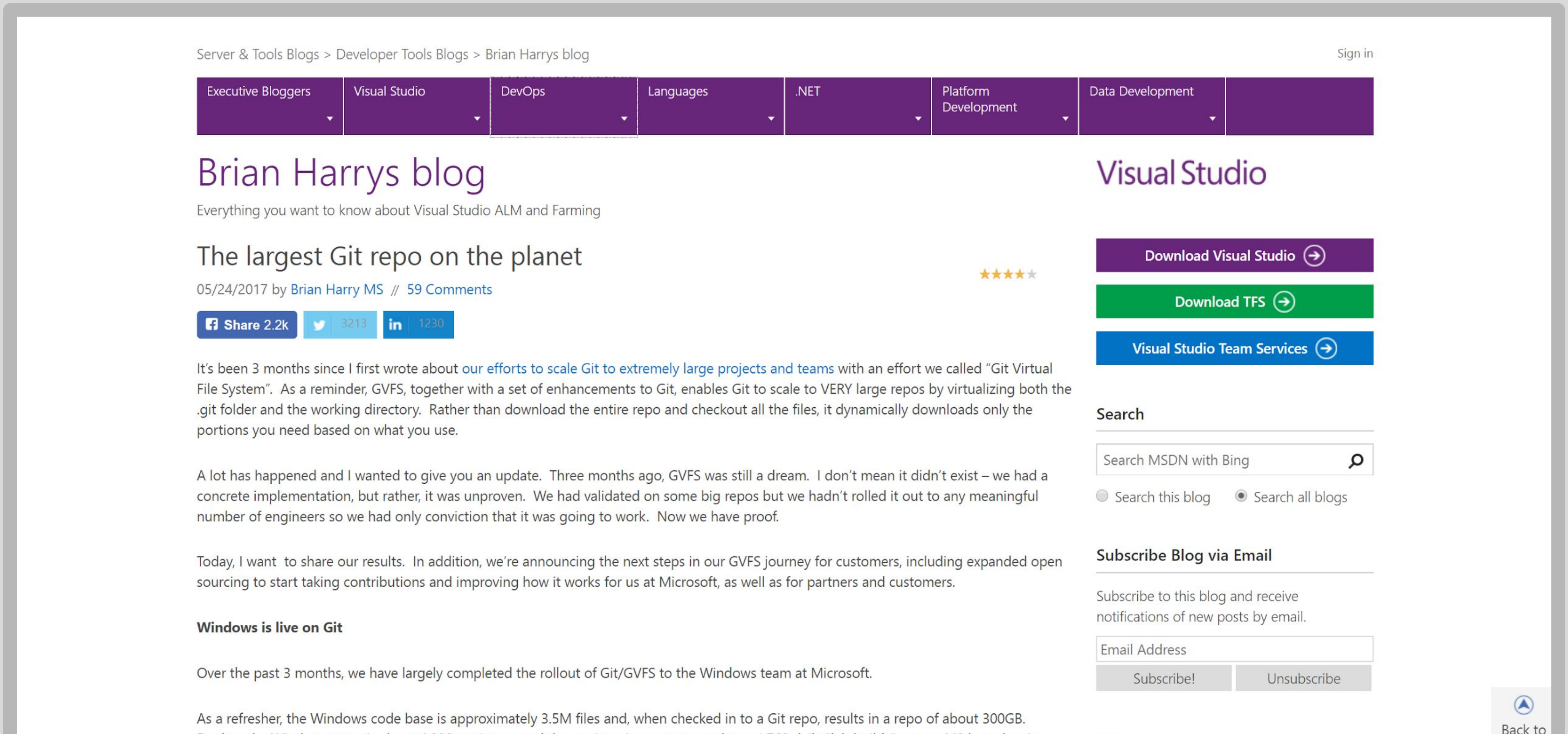
Recommended

Scaling memcached at Facebook

Flashcache at Facebook: From 2010 to 2013 and beyond

Monorepos in industry

Microsoft claim the largest git repo on the planet



Server & Tools Blogs > Developer Tools Blogs > Brian Harrys blog

Sign in

Executive Bloggers | Visual Studio | DevOps | Languages | .NET | Platform Development | Data Development

Brian Harrys blog

Everything you want to know about Visual Studio ALM and Farming

The largest Git repo on the planet

05/24/2017 by Brian Harry MS // 59 Comments

Share 2.2k 3213 in 1230

It's been 3 months since I first wrote about our efforts to scale Git to extremely large projects and teams with an effort we called "Git Virtual File System". As a reminder, GVFS, together with a set of enhancements to Git, enables Git to scale to VERY large repos by virtualizing both the .git folder and the working directory. Rather than download the entire repo and checkout all the files, it dynamically downloads only the portions you need based on what you use.

A lot has happened and I wanted to give you an update. Three months ago, GVFS was still a dream. I don't mean it didn't exist – we had a concrete implementation, but rather, it was unproven. We had validated on some big repos but we hadn't rolled it out to any meaningful number of engineers so we had only conviction that it was going to work. Now we have proof.

Today, I want to share our results. In addition, we're announcing the next steps in our GVFS journey for customers, including expanded open sourcing to start taking contributions and improving how it works for us at Microsoft, as well as for partners and customers.

Windows is live on Git

Over the past 3 months, we have largely completed the rollout of Git/GVFS to the Windows team at Microsoft.

As a refresher, the Windows code base is approximately 3.5M files and, when checked in to a Git repo, results in a repo of about 300GB.

Visual Studio

Download Visual Studio

Download TFS

Visual Studio Team Services

Search

Search MSDN with Bing

Search this blog   Search all blogs

Subscribe Blog via Email

Subscribe to this blog and receive notifications of new posts by email.

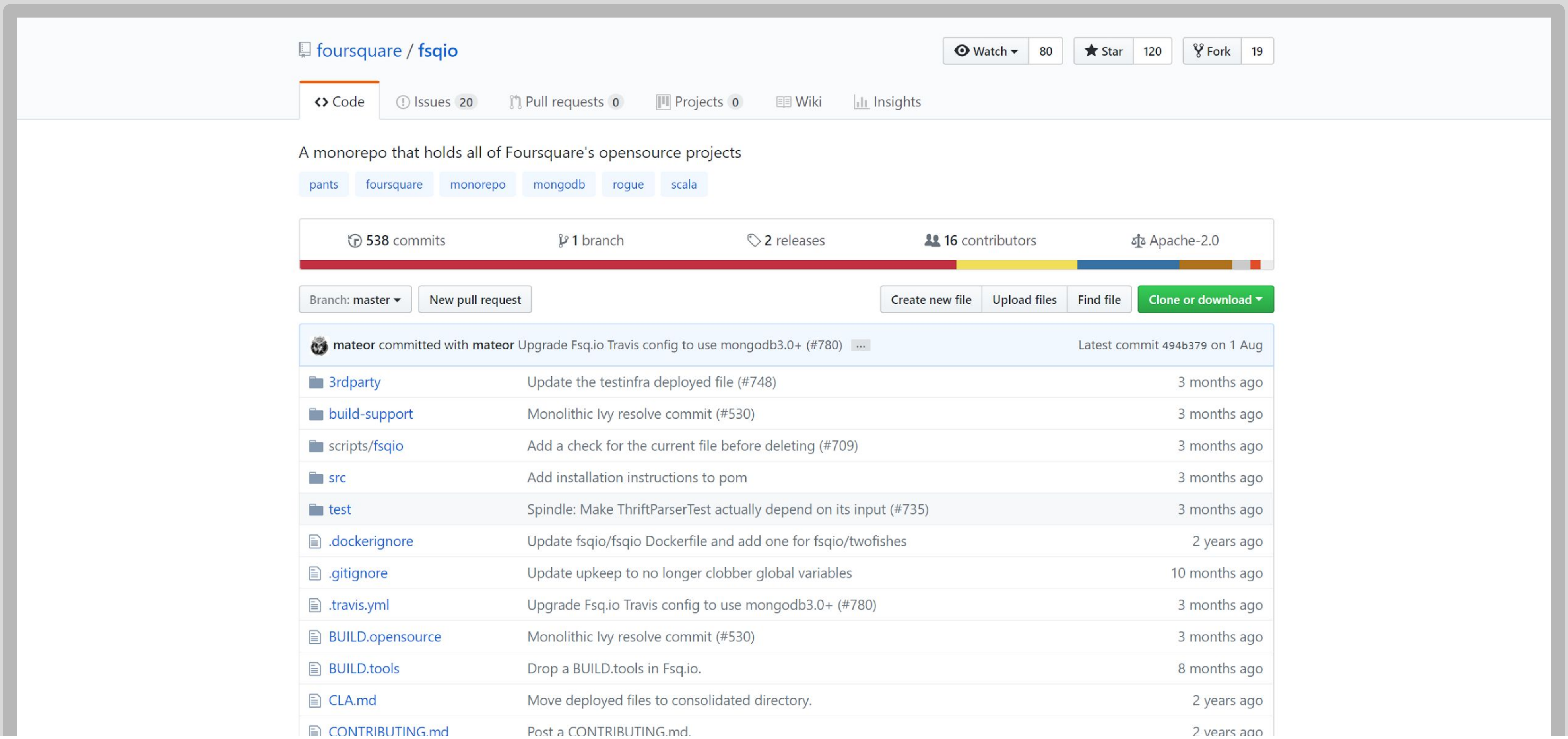
Email Address

Subscribe!   Unsubscribe

Back to

Monorepos in open-source

foresquare public monorepo



Monorepos in open-source

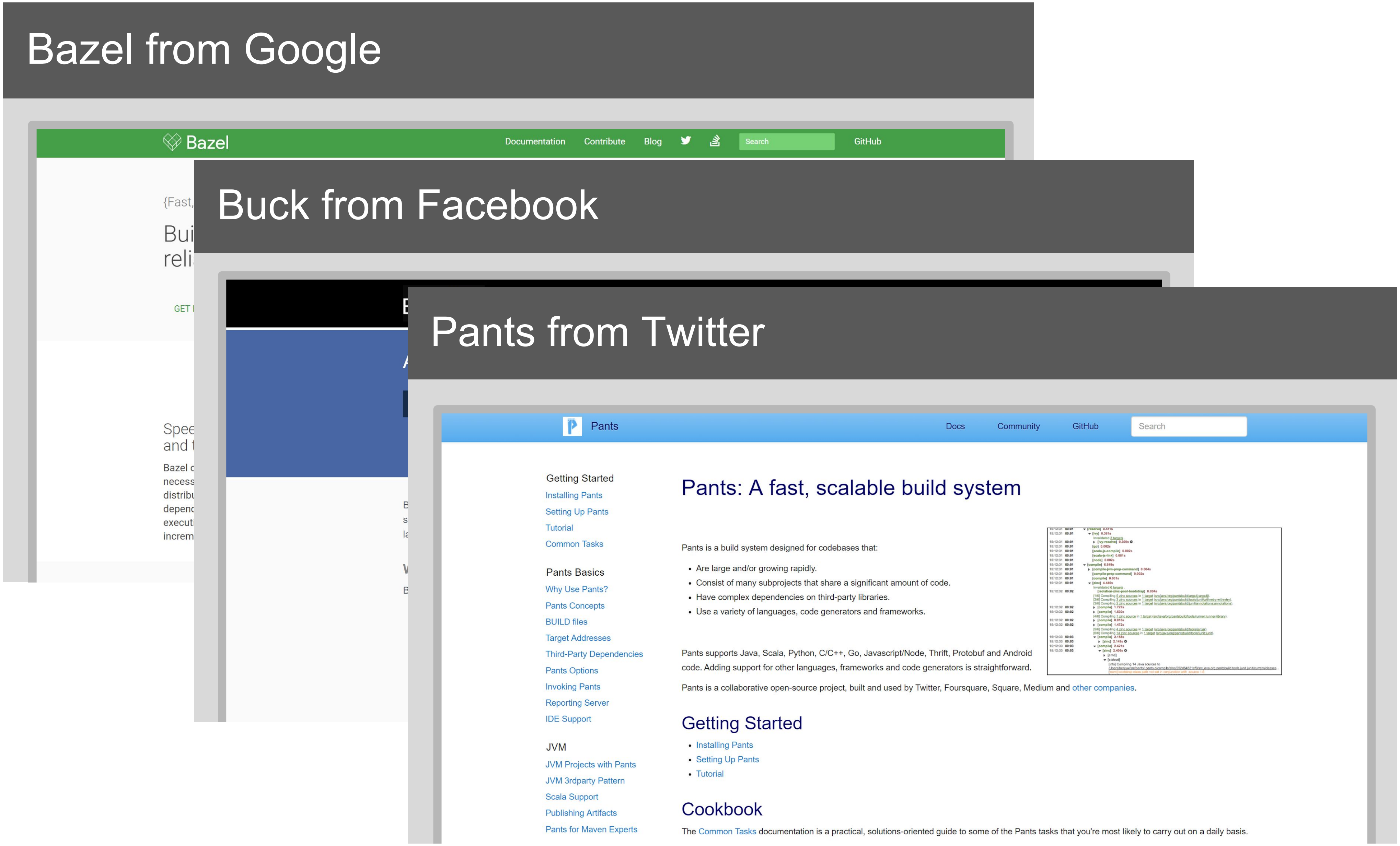
The Symfony monorepo

43 projects, **25 000** commits, and **400 000** LOC

https://github.com/symfony/symfony

```
Bridge/
    5 sub-projects
Bundle/
    5 sub-projects
Component/
    33 independent sub-projects like Asset, Cache,
    CssSelector, Finder, Form, HttpKernel, Ldap,
    Routing, Security, Serializer, Templating,
    Translation, Yaml, ...
```

Common build system



Some advantages of monorepos

High Discoverability For Developers

- Developers can read and explore the whole codebase
- `grep`, IDEs and other tools can search the whole codebase
- IDEs can offer auto-completion for the whole codebase
- Code Browsers can links between all artifacts in the codebase

Code-Reuse is cheap

Almost zero cost in introducing a new library

- Extract library code into a new directory/component
- Use library in other components
- Profit!

Refactorings in one commit

Allow large scale refactorings with one single, atomic, history-preserving commit

- Extract Library/Component
- Rename Functions/Methods/Components
- Housekeeping (phpcs-fixer, Namespacing, ...)

Another refactoring example

- Make large backward incompatible changes easily... especially if they span different parts of the project

- For example, old APIs can be removed with confidence
  - Change an API endpoint code **and** all its usages in **all** projects in **one** pull request

Some more advantages

- Easy continuous integration and code review for changes spanning several projects
- (Internal) dependency management is a non-issue
- Less context switching for developers
- Code more reusable in other contexts
- Access control is easy

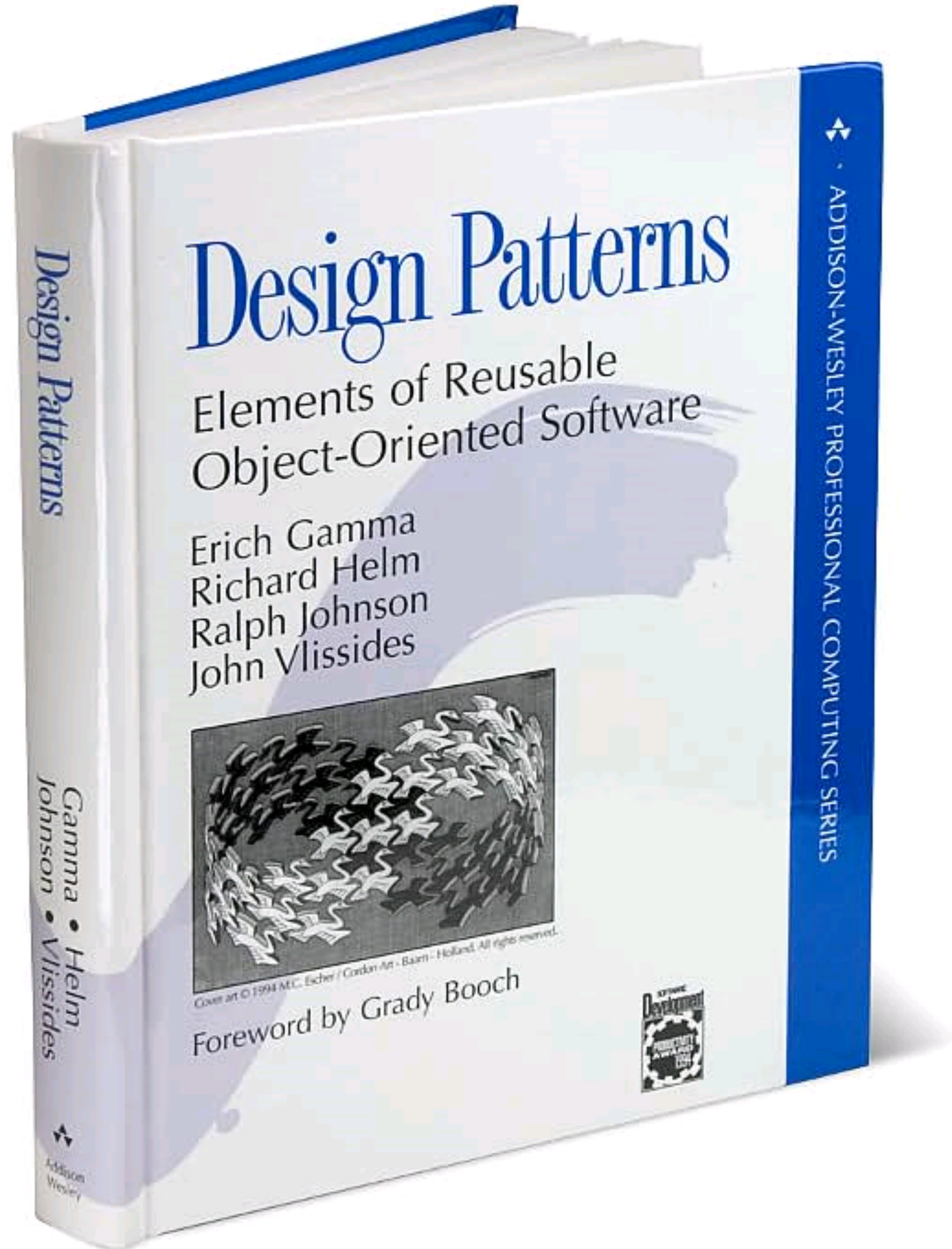
Some downsides

- Require collective responsibility for team and developers
- Require trunk-based development
  - Feature toggles are technical debt (recall financial services example)
- Force you to have only one version of everything
- Scalability requirements for the repository
- Can be hard to deal with updates around things like security issues
- Build and test bloat without very smart build system
- Slow VCS without very smart system
- Permissions?

Summary

- Configuration management
  - Treat infrastructure as code
  - Git is powerful
- Release management: versioning, branching, …
- Software development at scale requires a lot of infrastructure
  - Version control, build managers, testing, continuous integration, deployment, …
- It's hard to scale development
  - Move towards heavy automation (DevOps)
- Continuous deployment increasingly common
- Opportunities from quick release, testing in production, quick rollback

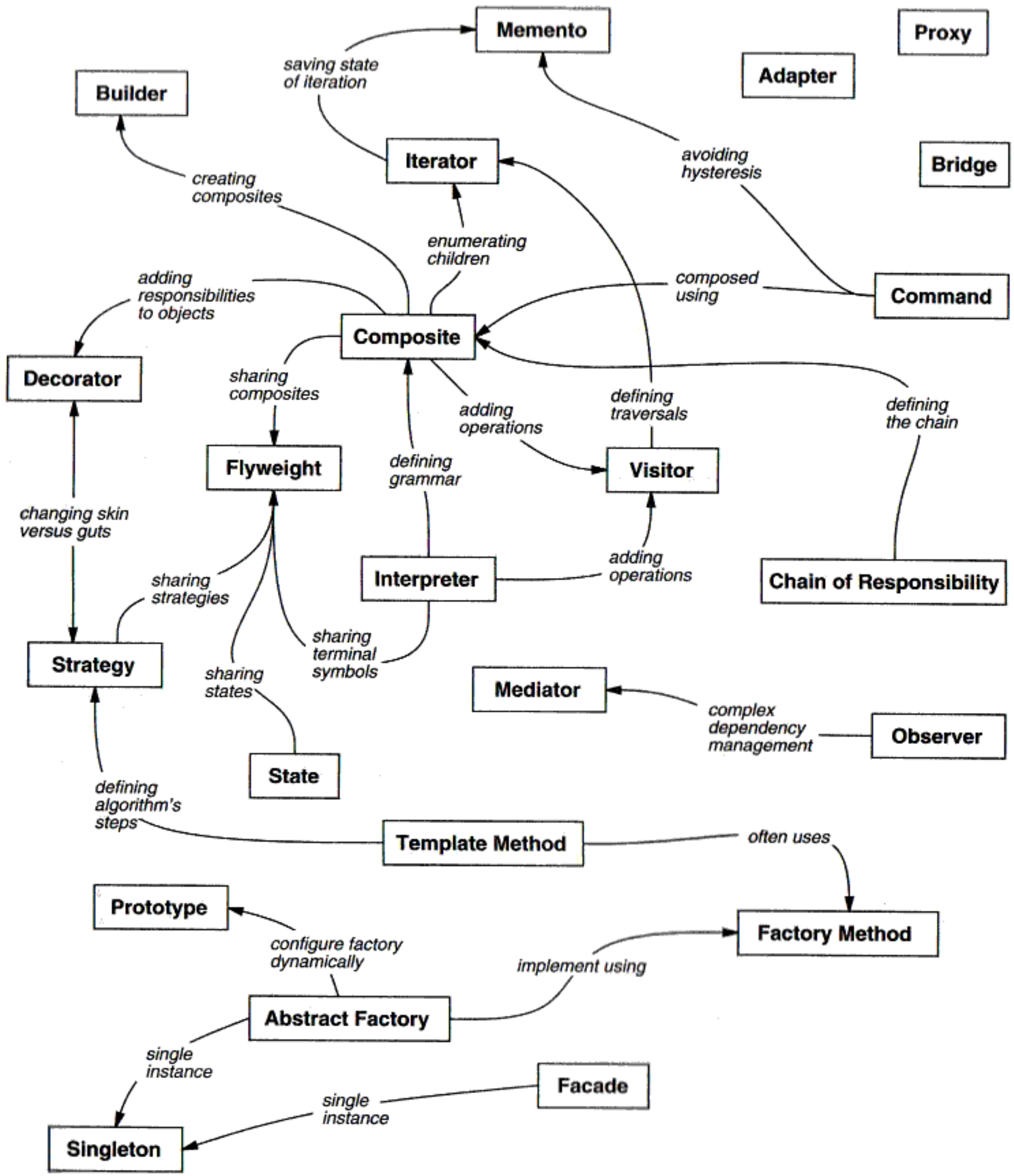
Today:



- Published 1994
- 23 Patterns
- Widely known

Why?

- GOF book is seminal and canonical list of well-known patterns
- At least know where to look up when somebody mentions the "Bridge pattern"


Memento
Proxy
Adapter
Builder
saving state of iteration
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
Interpreter
adding operations
Chain of Responsibility
sharing strategies
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
- **JDK** – example(s) of this pattern in the JDK

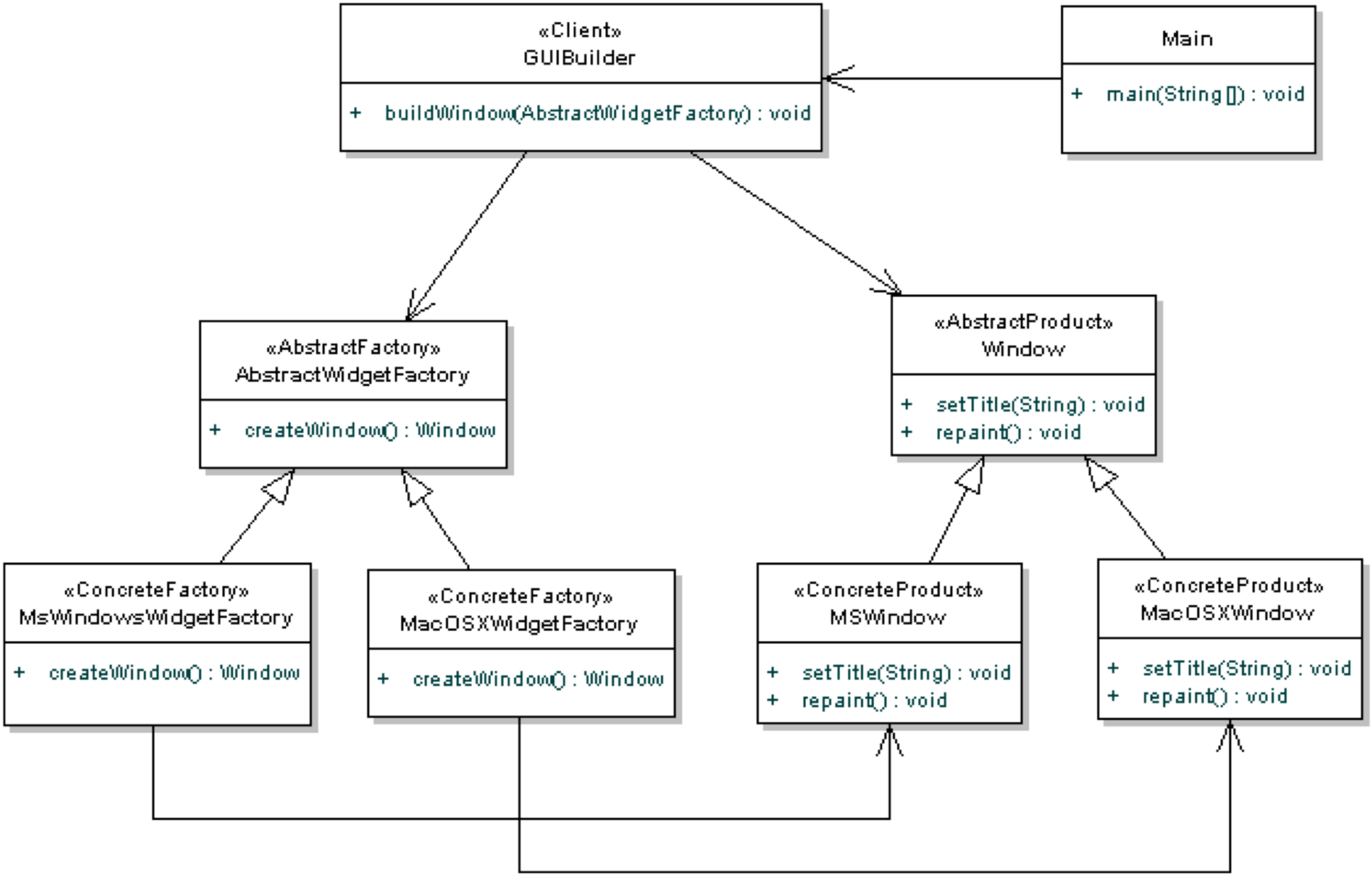
I. Creational Patterns

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

Problem:

- 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*
- JDK – not common

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

```java
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

```java
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.me/2012/12/13/the-builder-pattern-in-practice/

Solution 3

```java
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() { ... }
```

```java
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;
    return this;
  }

  // ...
```

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


https://jlordiales.me/2012

Builder

- 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
- JDK – java.lang.StringBuilder, java.lang.StringBuffer

Factory Method

- 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
- JDK – `Iterable.iterator()`

Prototype

- 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*
- JDK – Cloneable, but avoid (except on arrays)
  - Java and Prototype pattern are a poor fit

Problem:

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

Singleton

- 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
- JDK – `java.lang.Runtime.getRuntime()`, `java.util.Collections.emptyList()`
- Used for instance control

Singleton Illustration

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


// Alternative implementation
public enum Elvis {
    ELVIS;

    sing(Song song) { ... }

    playGuitar(Riff riff) { ... }

    eat(Food food) { ... }

    take(Drug drug) { ... }
}
```

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[])`

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

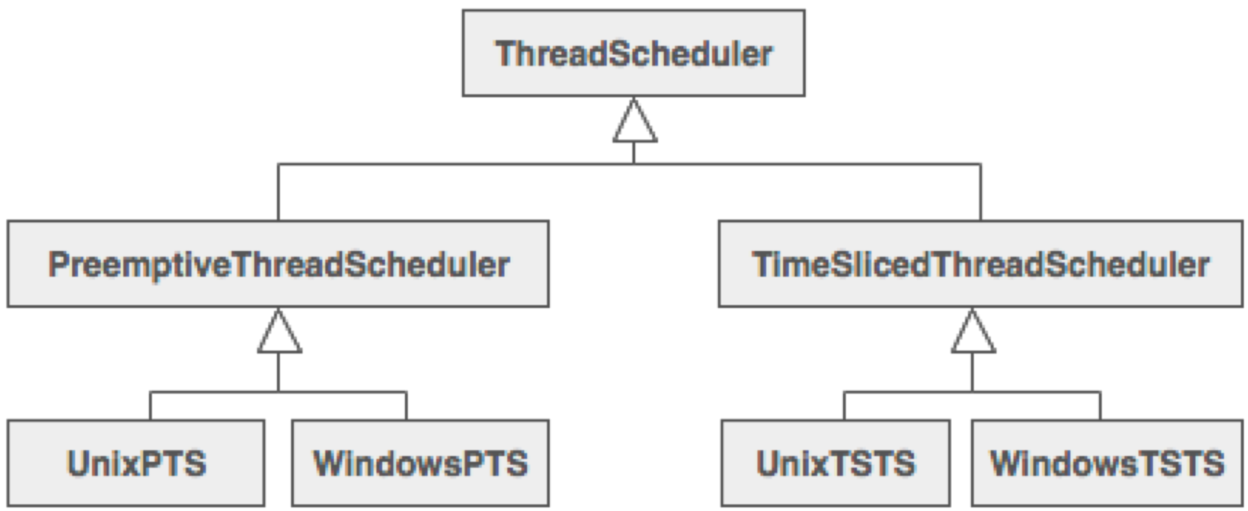
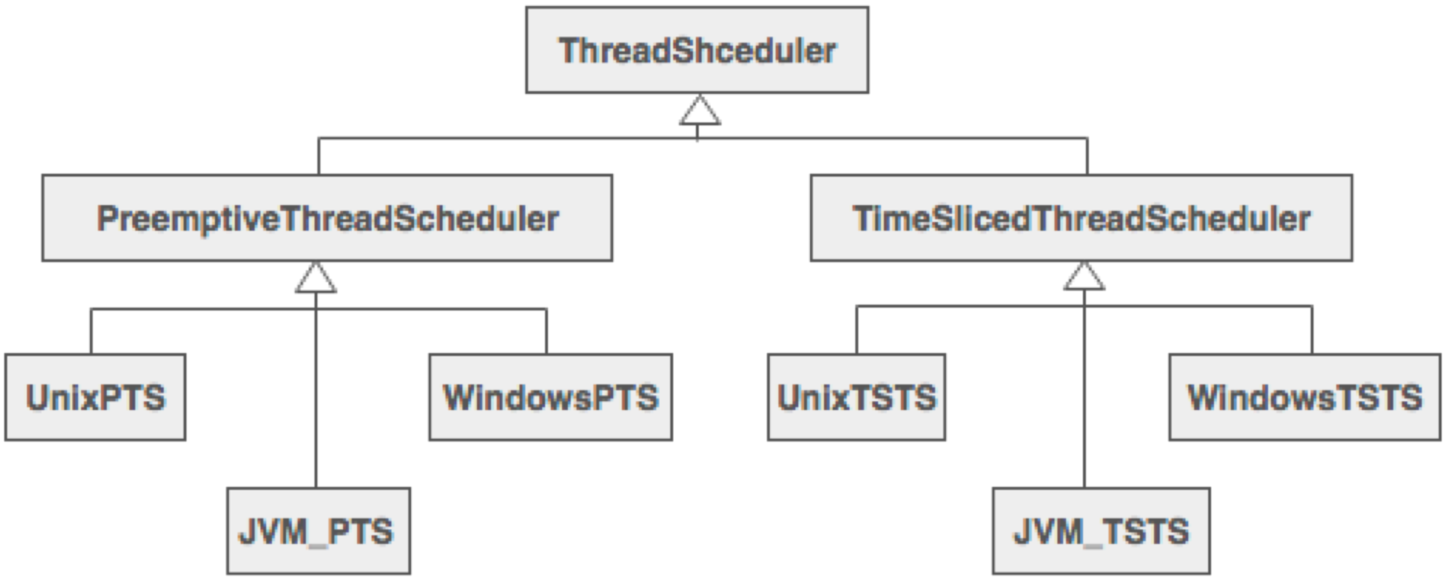


image source: https://sourcemaking.com

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.

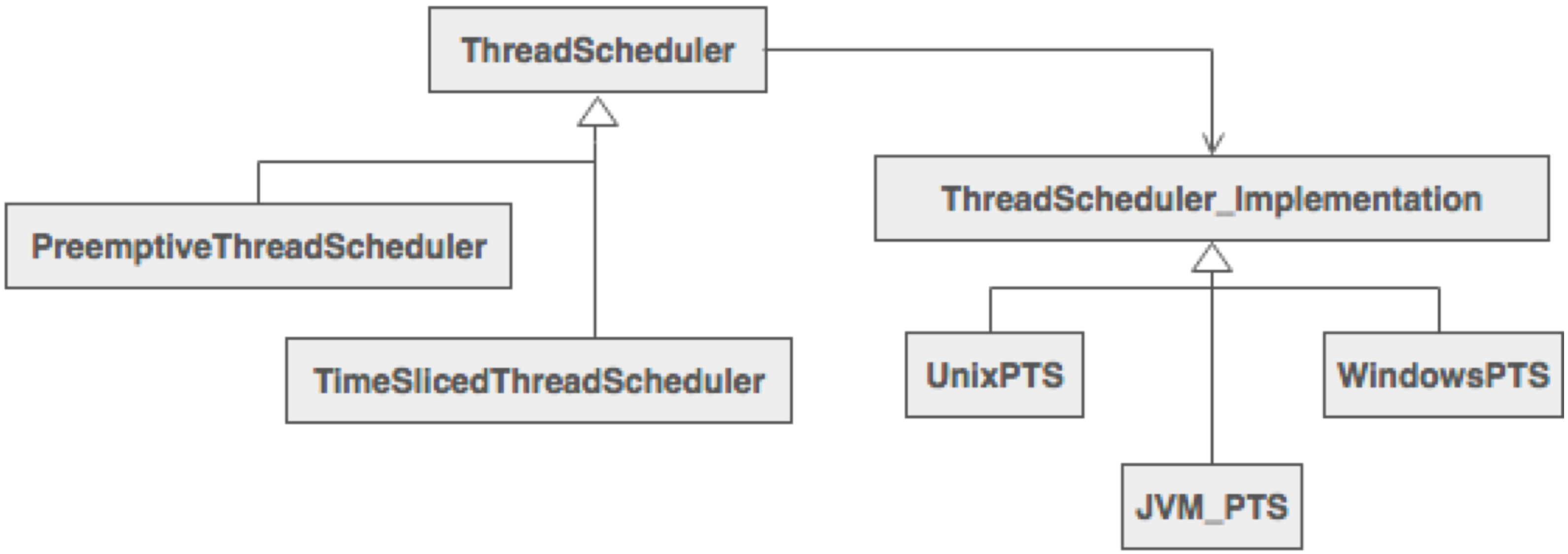


image source: https://sourcemaking.com

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)
- 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.

Composite

- 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
- JDK – `javax.swing.JComponent`

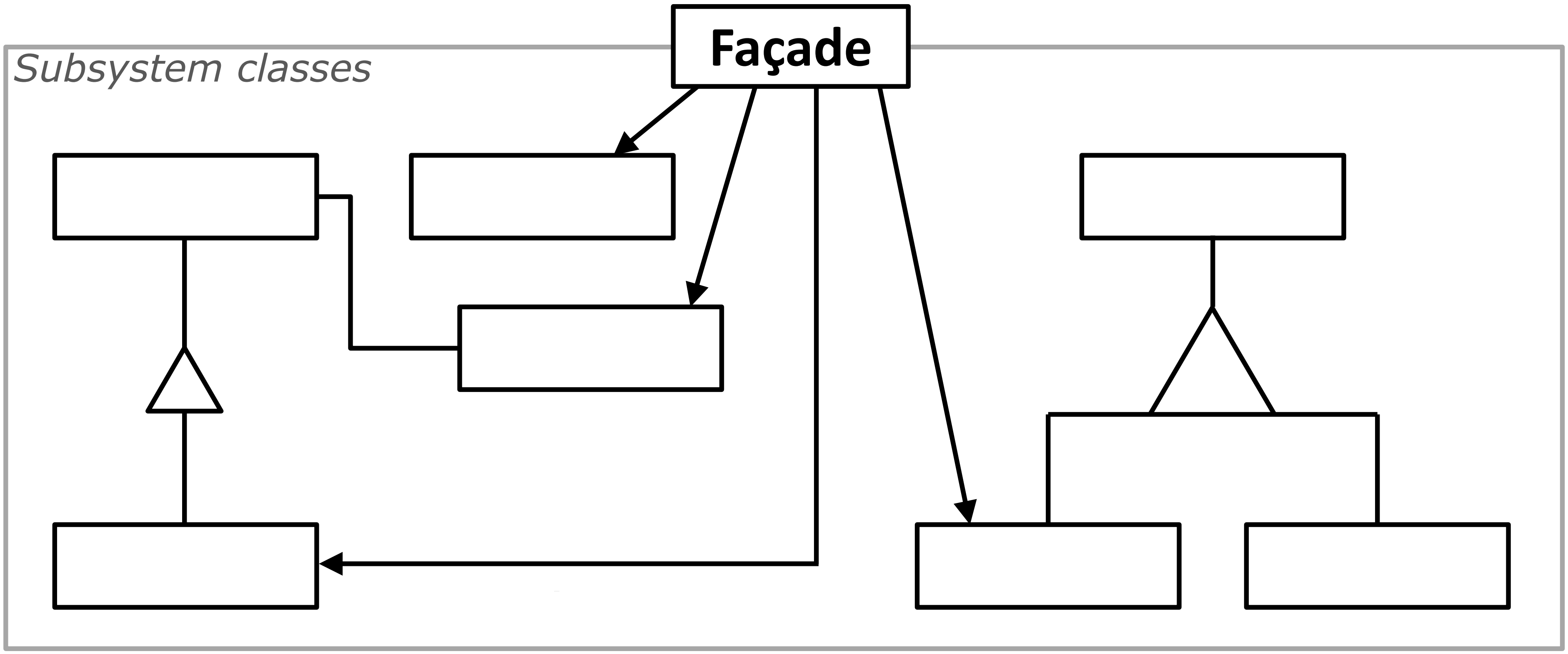
Decorator

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

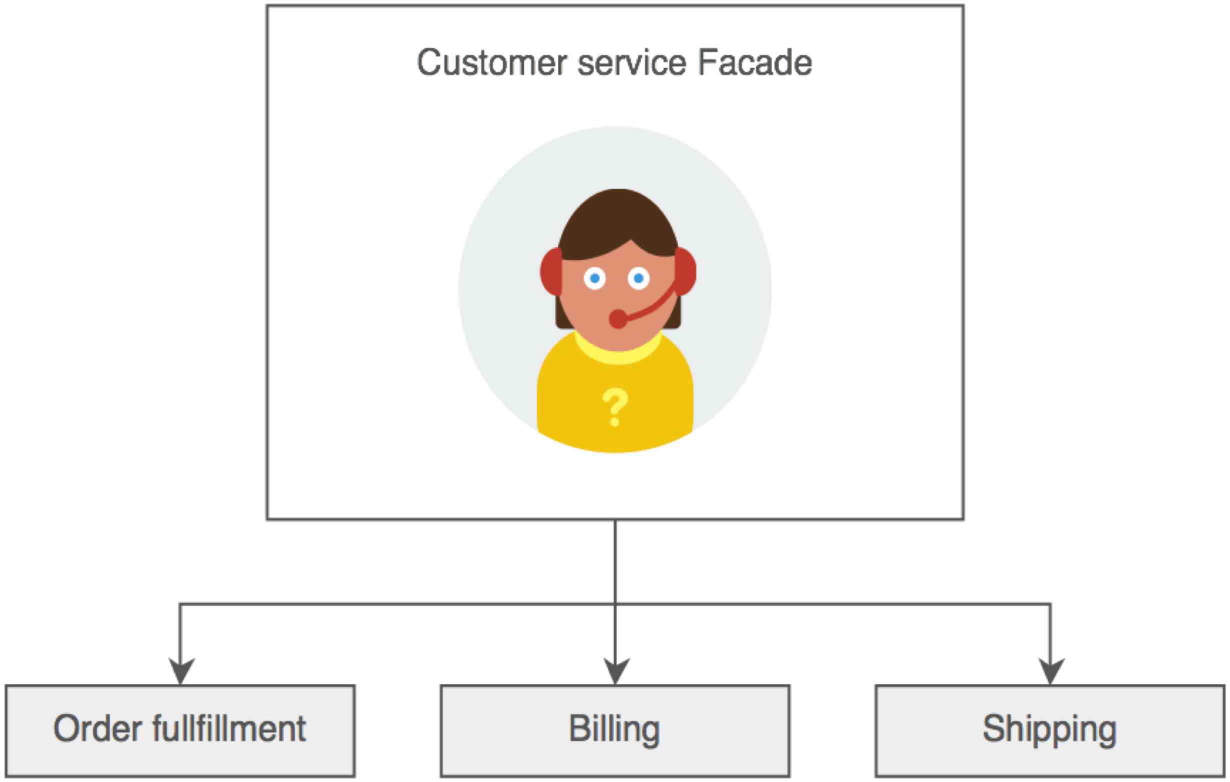
Façade

- 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`

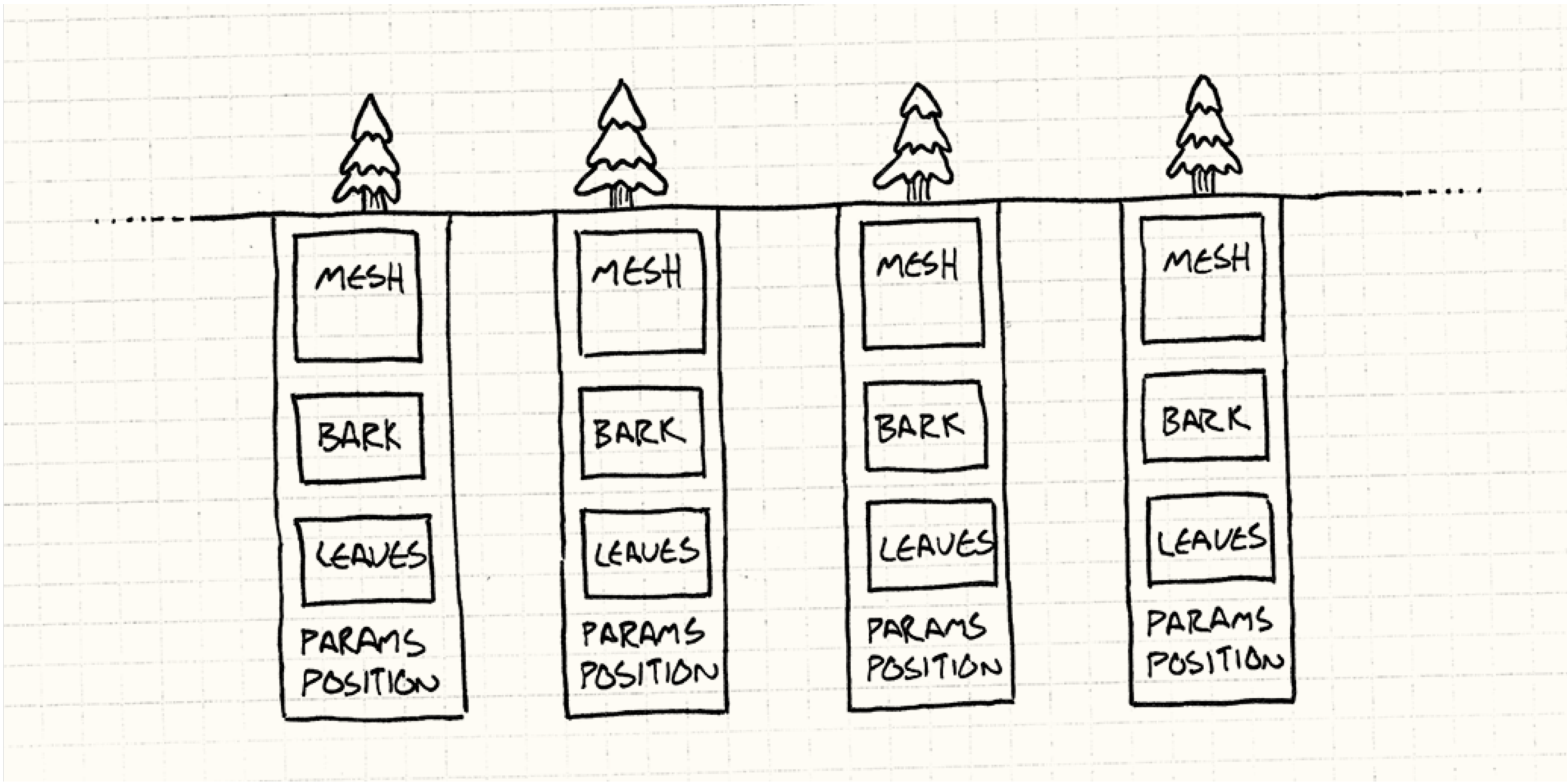
Façade Illustration



Façade example

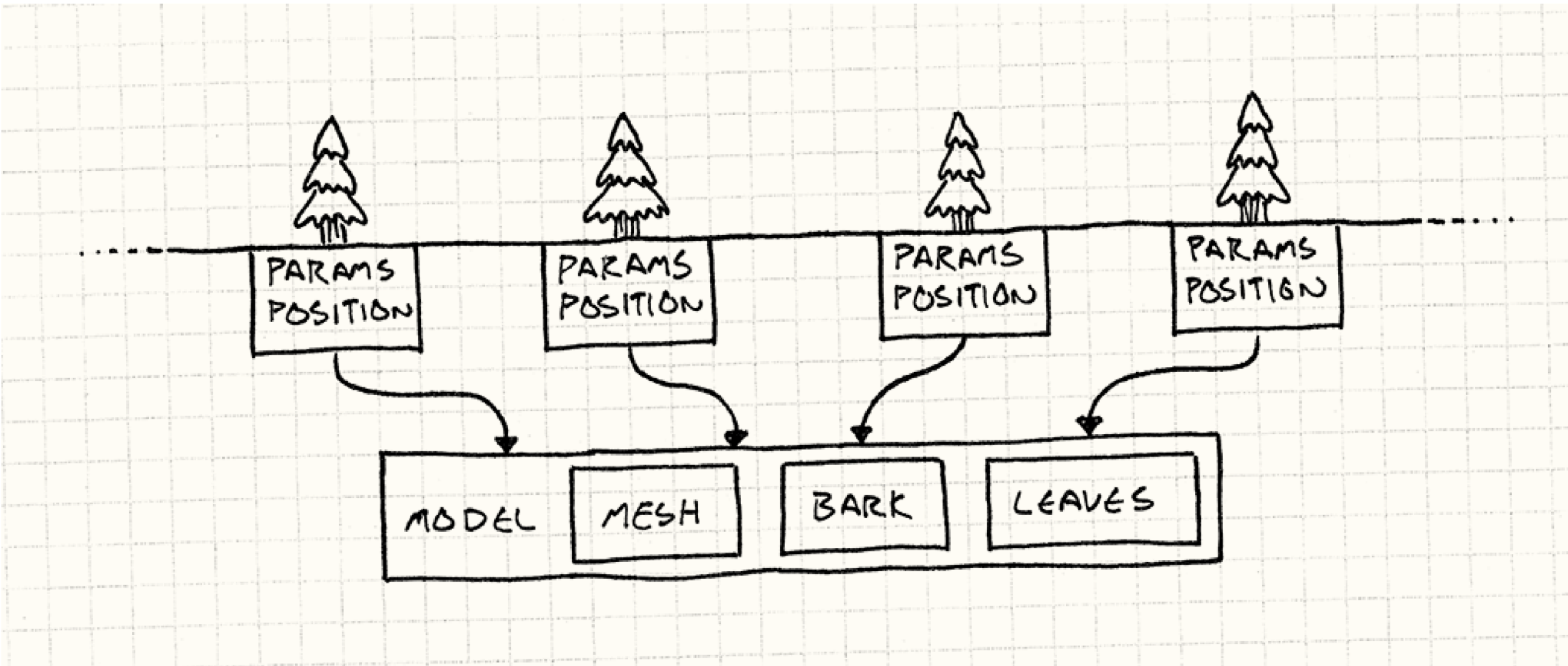


Problem: Imagine implementing a forest of individual trees in a realtime game



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

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

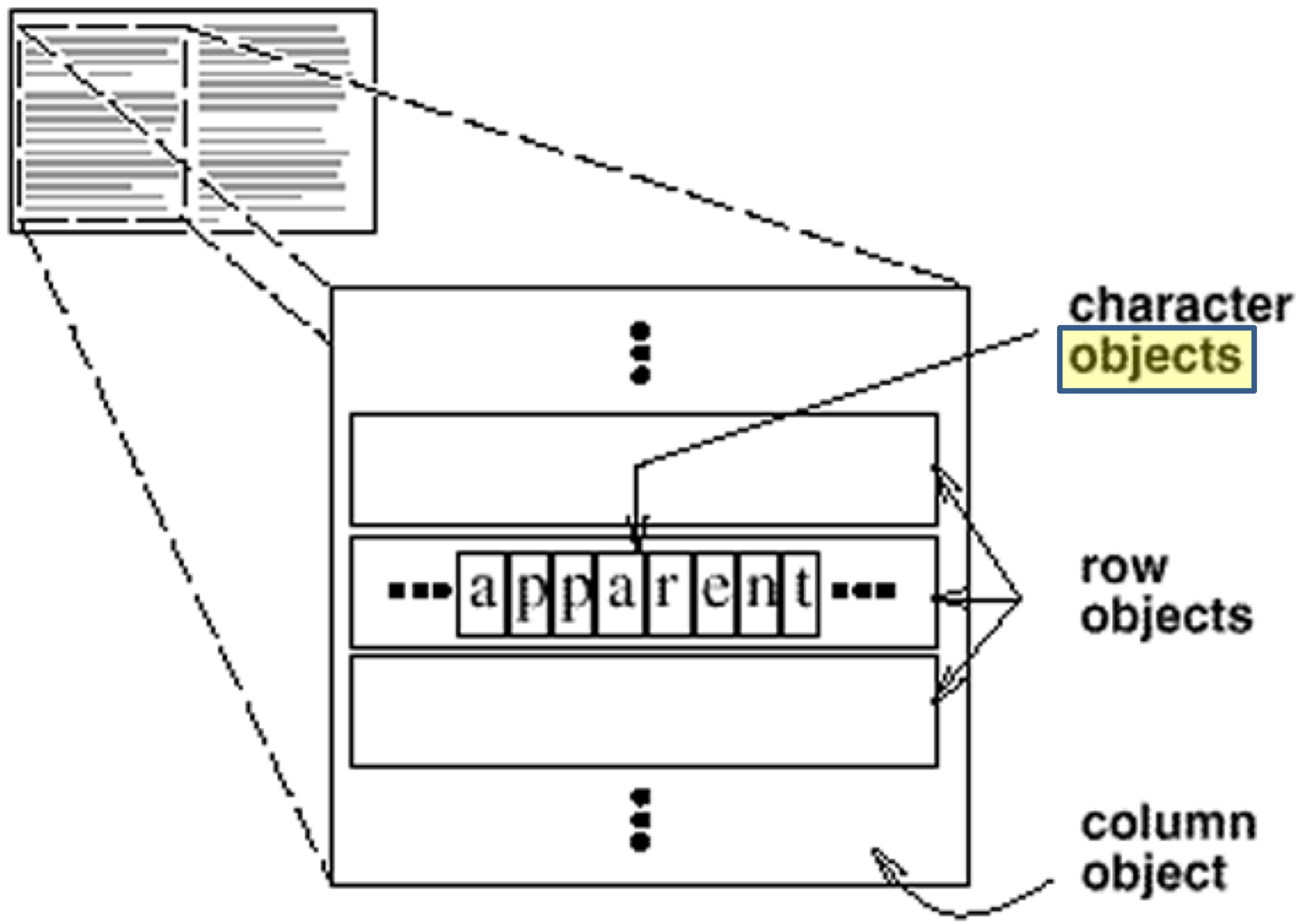


Source: http://gameprogrammingpatterns.com/flyweight.html

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 Illustration

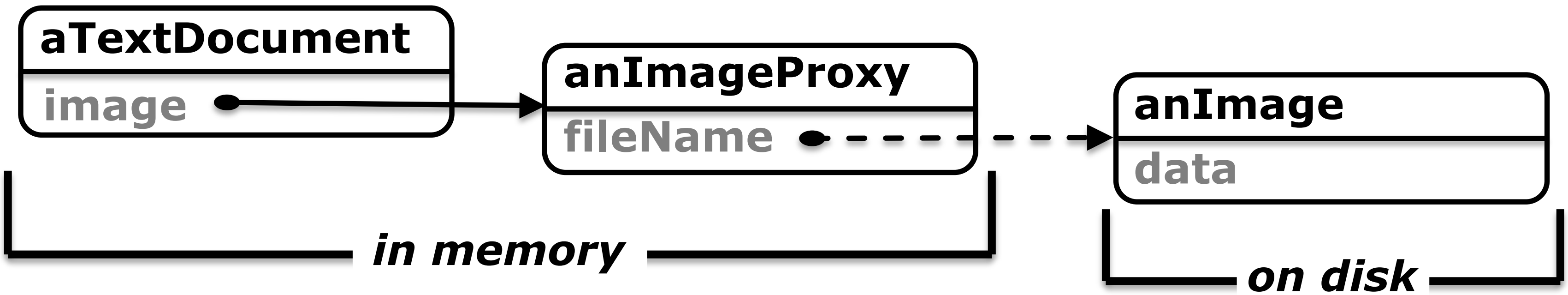


Proxy

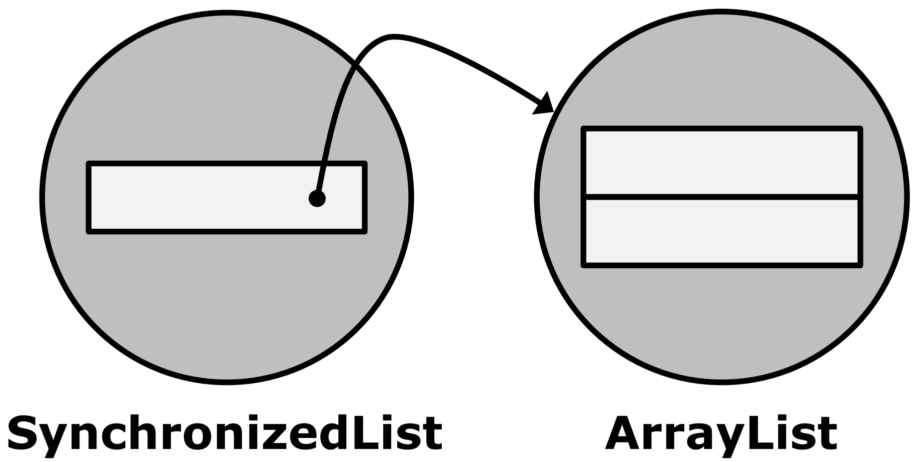
- Intent – surrogate for another object
- Use case – delay loading of images till needed
- Key types – *Subject*, Proxy, RealSubject
- Gof mention several flavors
  - virtual proxy – stand-in that instantiates lazily
  - remote proxy – local representative for remote obj
  - protection proxy – denies some ops to some users
  - smart reference – does locking or ref. counting, e.g.
- JDK – collections wrappers
- Decorator vs Proxy:
  - Decorator adds responsibilities to object (w/t inheritance).
  - Proxy is used to “control access” to an object.

Proxy Illustrations

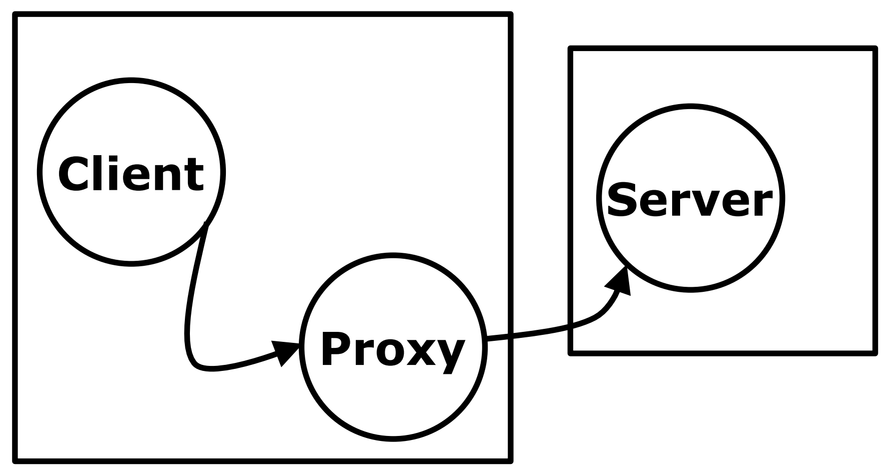
Virtual Proxy



Smart Reference



Remote Proxy



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

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));
}
```

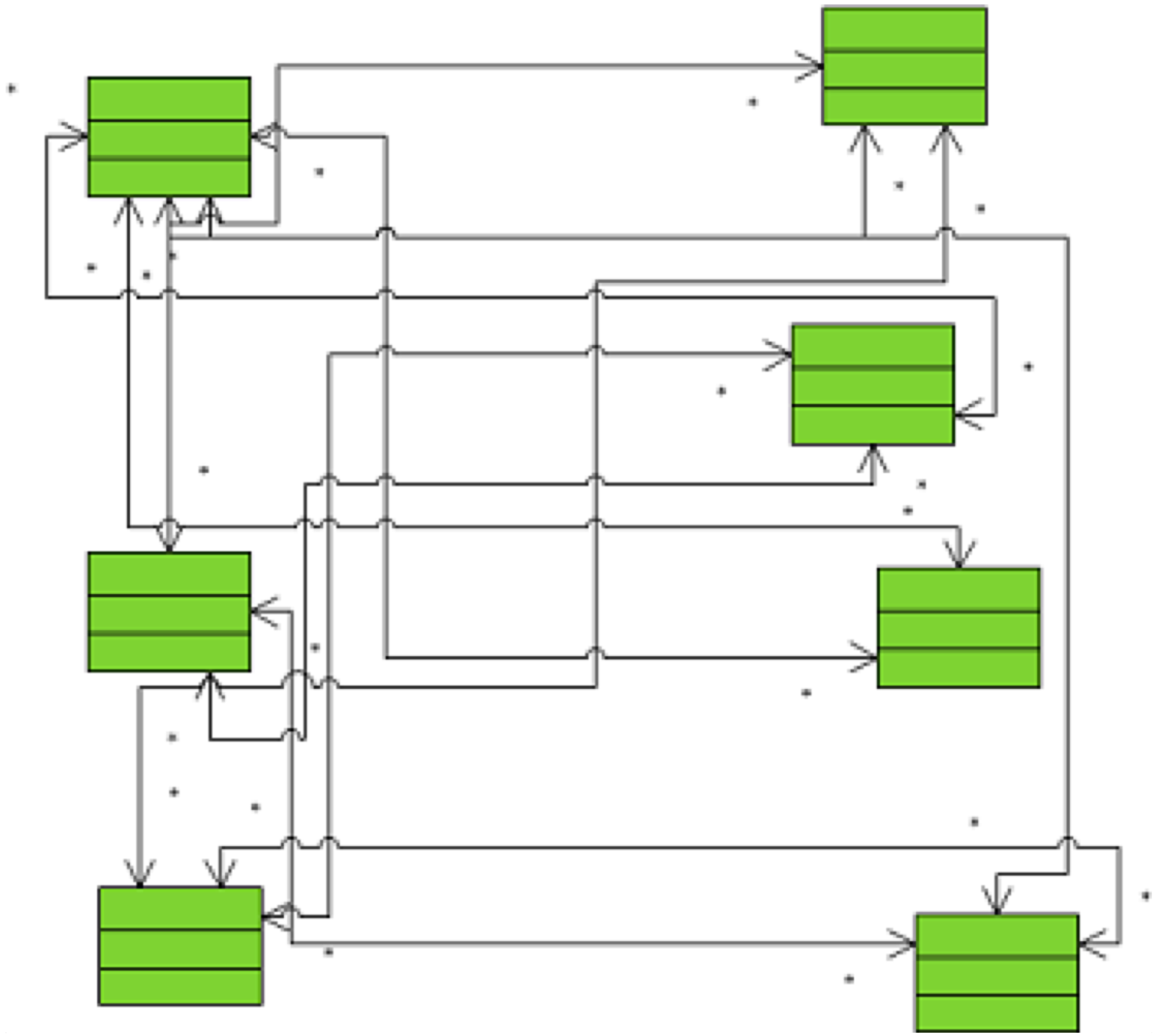
Interpreter

- Intent – given a language, define class hierarchy for parse tree, recursive method to interpret it
- Use case – regular expression matching, compiler
- Key types – *Expression*, *NonterminalExpression*, *TerminalExpression*
- JDK – no uses I'm aware of
  - Our expression evaluator (HW2) is a classic example
- Necessarily uses Composite pattern!

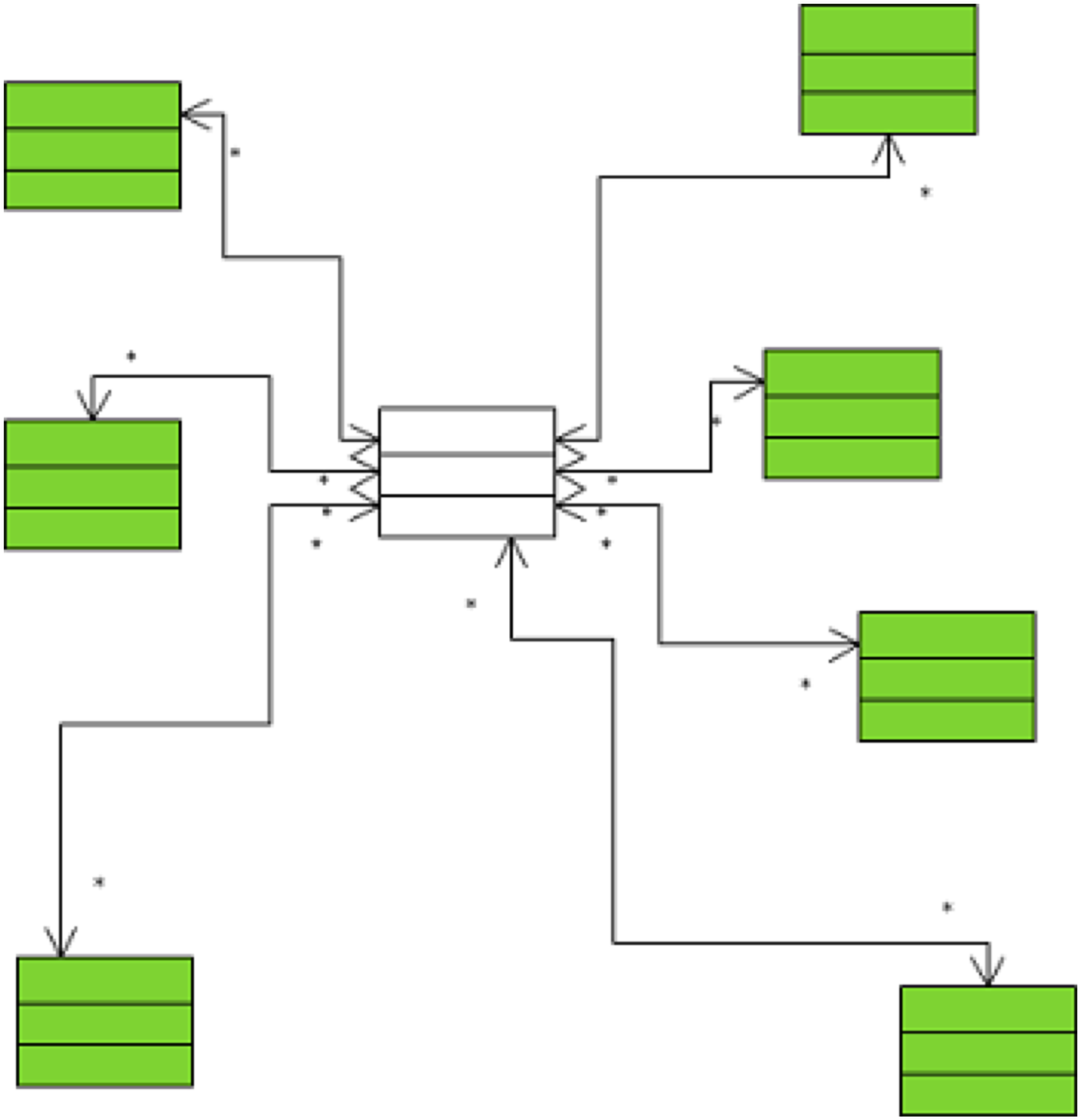
Iterator

- 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
- JDK – collections, for-each statement, etc.

Problem:



Mediator Pattern



institute for SOFTWARE RESEARCH

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

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

```java
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.
        }
    }
    // ...
```

```java
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

- 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

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

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

```
//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

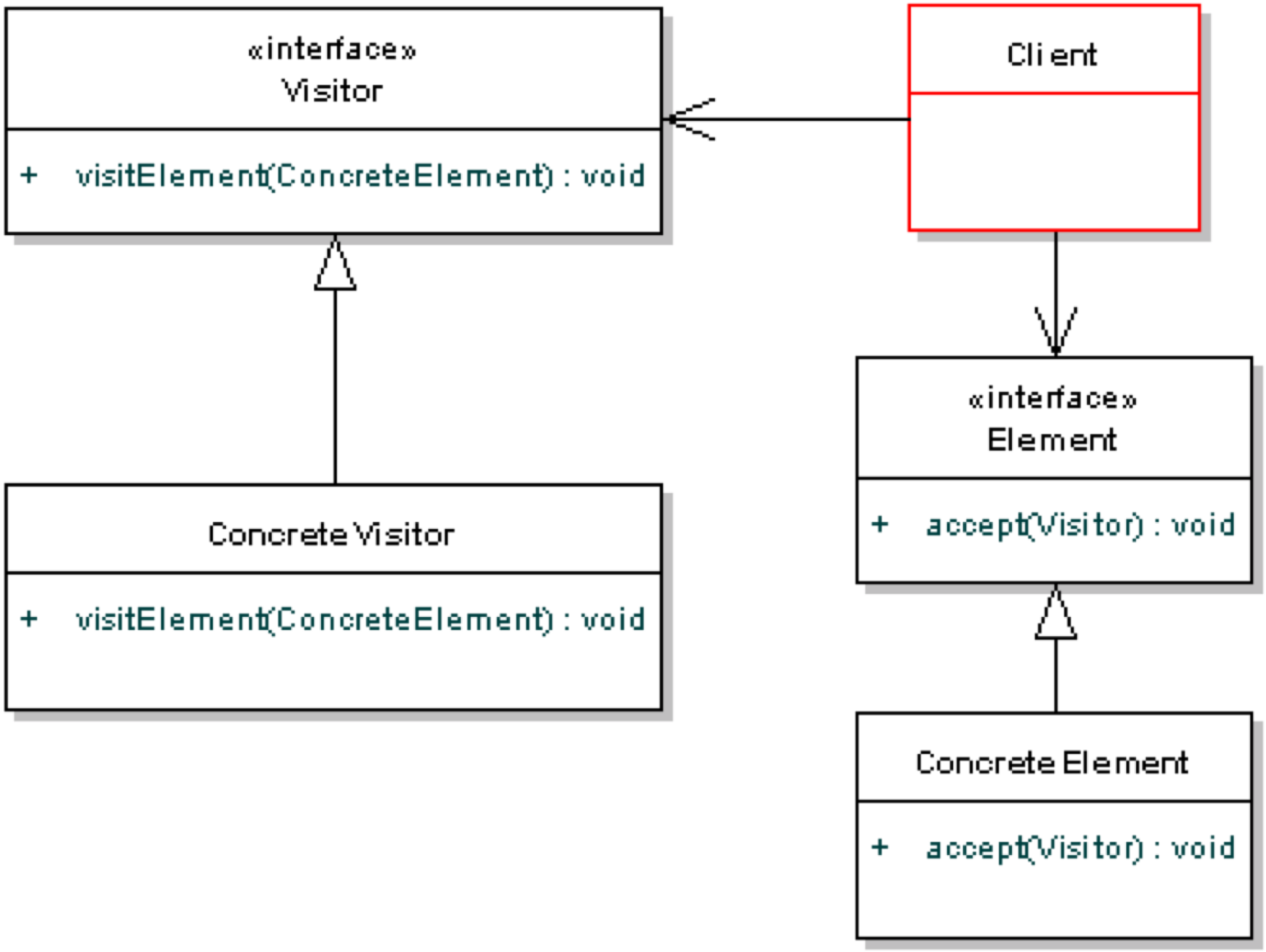
```
public interface Visitor{
  public void visit(Book book);

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

```
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

```java
public class ShoppingCart {
  //normal shopping cart stuff
  private ArrayList<Visitable> items;
  public double calculatePostage() {
    //create a visitor
    PostageVisitor visitor = new PostageVisitor();
    //iterate through all items
    for(Visitable item: items) {
      item.accept(visitor);
    }
    double postage = visitor.getTotalPostage();
    return postage;
  }
}
```

Visitor



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

All GoF Design Patterns

1. Abstract factory
2. Builder
3. Factory method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Façade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template method
23. Visitor