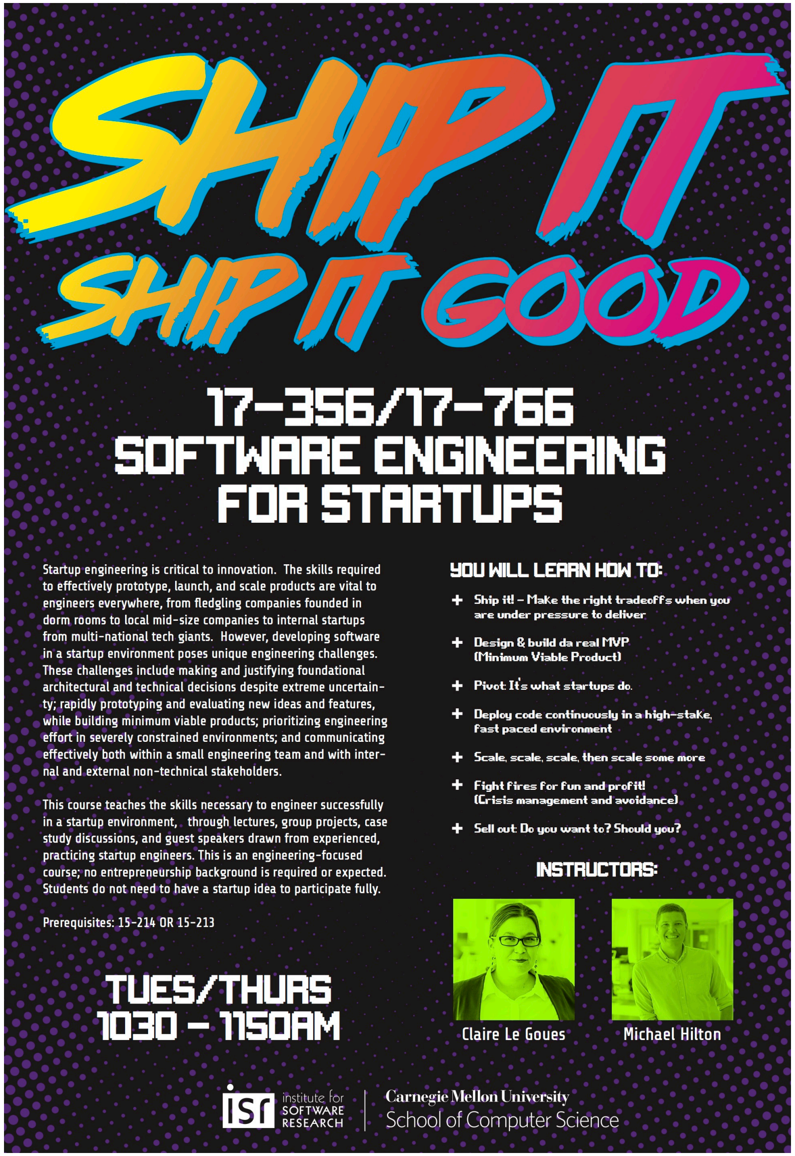
23 GoF Design Patterns – an interactive tour

Charlie Garrod **Michael Hilton**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- (NEW) Homework 6 out soon.
- SE for Startups
- No recitation tomorrow
- Happy Thanksgiving



17-356/17-766
SOFTWARE ENGINEERING FOR STARTUPS

Startup engineering is critical to innovation. The skills required to effectively prototype, launch, and scale products are vital to engineers everywhere, from fledgling companies founded in dorm rooms to local mid-size companies to internal startups from multi-national tech giants. However, developing software in a startup environment poses unique engineering challenges. These challenges include making and justifying foundational architectural and technical decisions despite extreme uncertainty; rapidly prototyping and evaluating new ideas and features, while building minimum viable products; prioritizing engineering effort in severely constrained environments; and communicating effectively both within a small engineering team and with internal and external non-technical stakeholders.

This course teaches the skills necessary to engineer successfully in a startup environment, through lectures, group projects, case study discussions, and guest speakers drawn from experienced, practicing startup engineers. This is an engineering-focused course; no entrepreneurship background is required or expected. Students do not need to have a startup idea to participate fully.

Prerequisites: 15-214 OR 15-213

TUES/THURS
1030 – 1150AM

YOU WILL LEARN HOW TO:

- Ship it! – Make the right tradeoffs when you are under pressure to deliver
- Design & build da real MVP (Minimum Viable Product)
- Pivot: It's what startups do.
- Deploy code continuously in a high-stake, fast paced environment
- Scale, scale, scale, then scale some more
- Fight fires for fun and profit! (Crisis management and avoidance)
- Sell out: Do you want to? Should you?

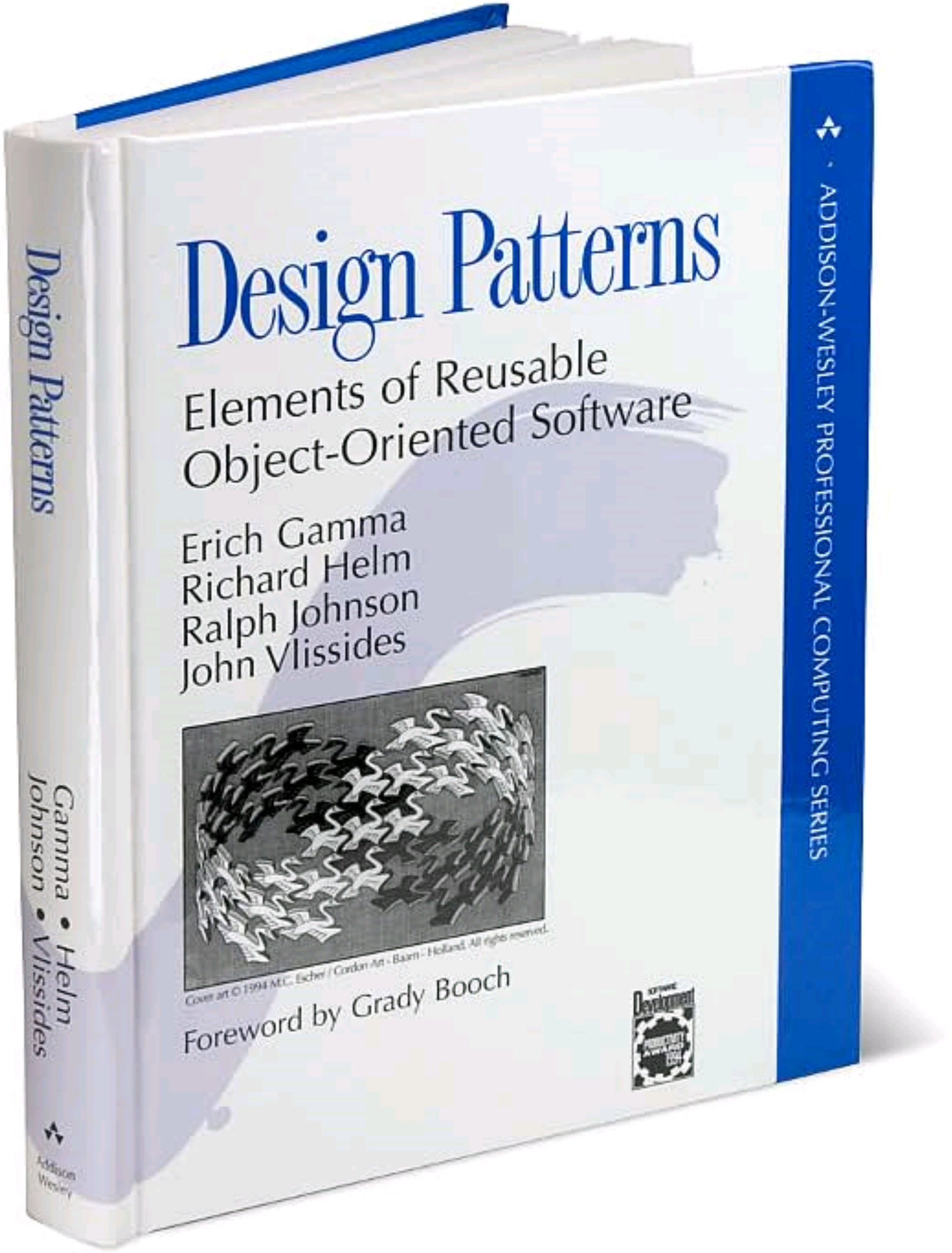
INSTRUCTORS:

Claire Le Goues
Michael Hilton

isr institute for SOFTWARE RESEARCH | Carnegie Mellon University School of Computer Science

Last Time:

- Monster interface creates challenges for users
- Java is not a functional language
  - “Bolted on” features are difficult to integrate well



- Published 1994
- 23 Patterns
- Widely known

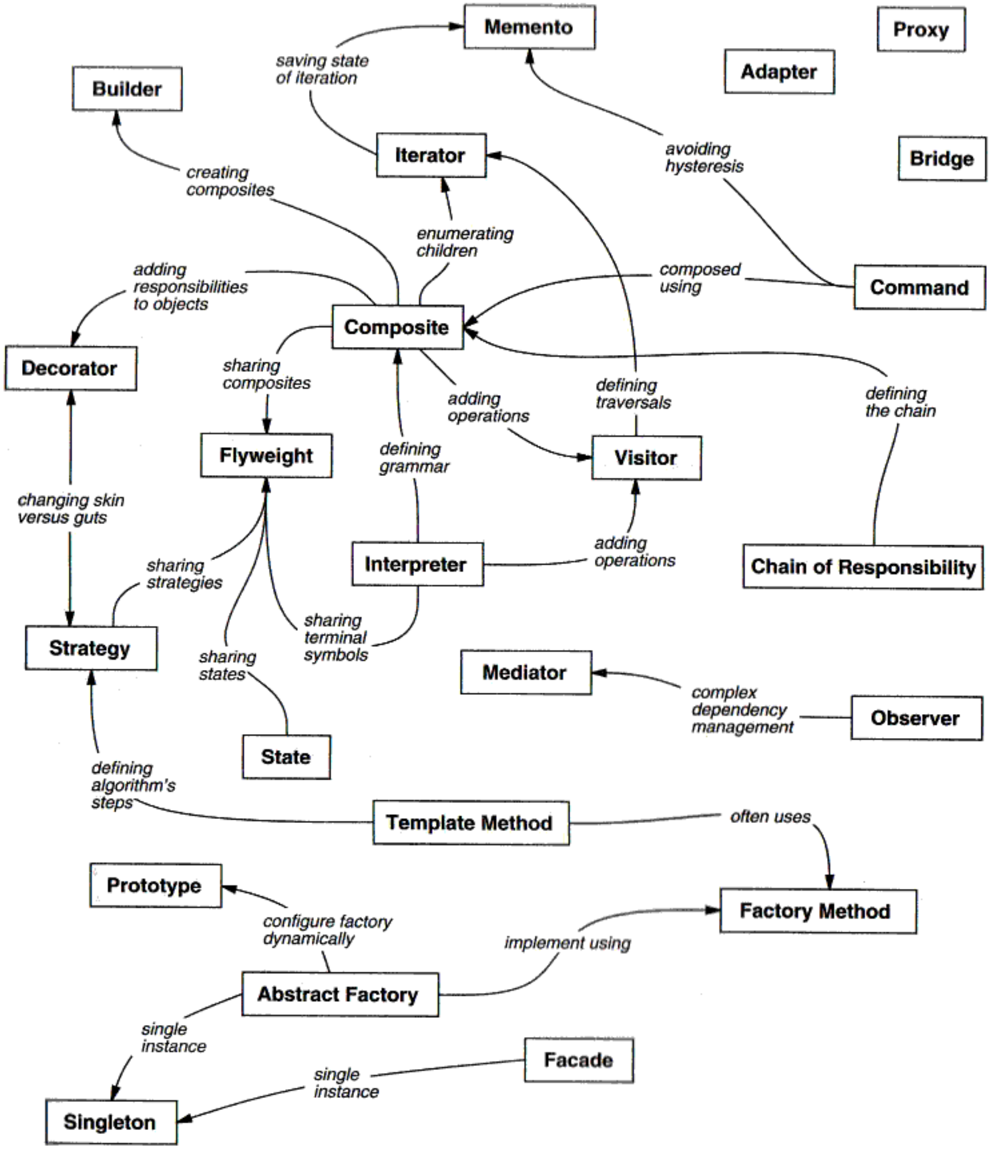


Figure 1.1: Design pattern relationships

Object Oriented Design Principles:

- Program to an interface, not an implementation
- Favor object composition over class inheritance

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

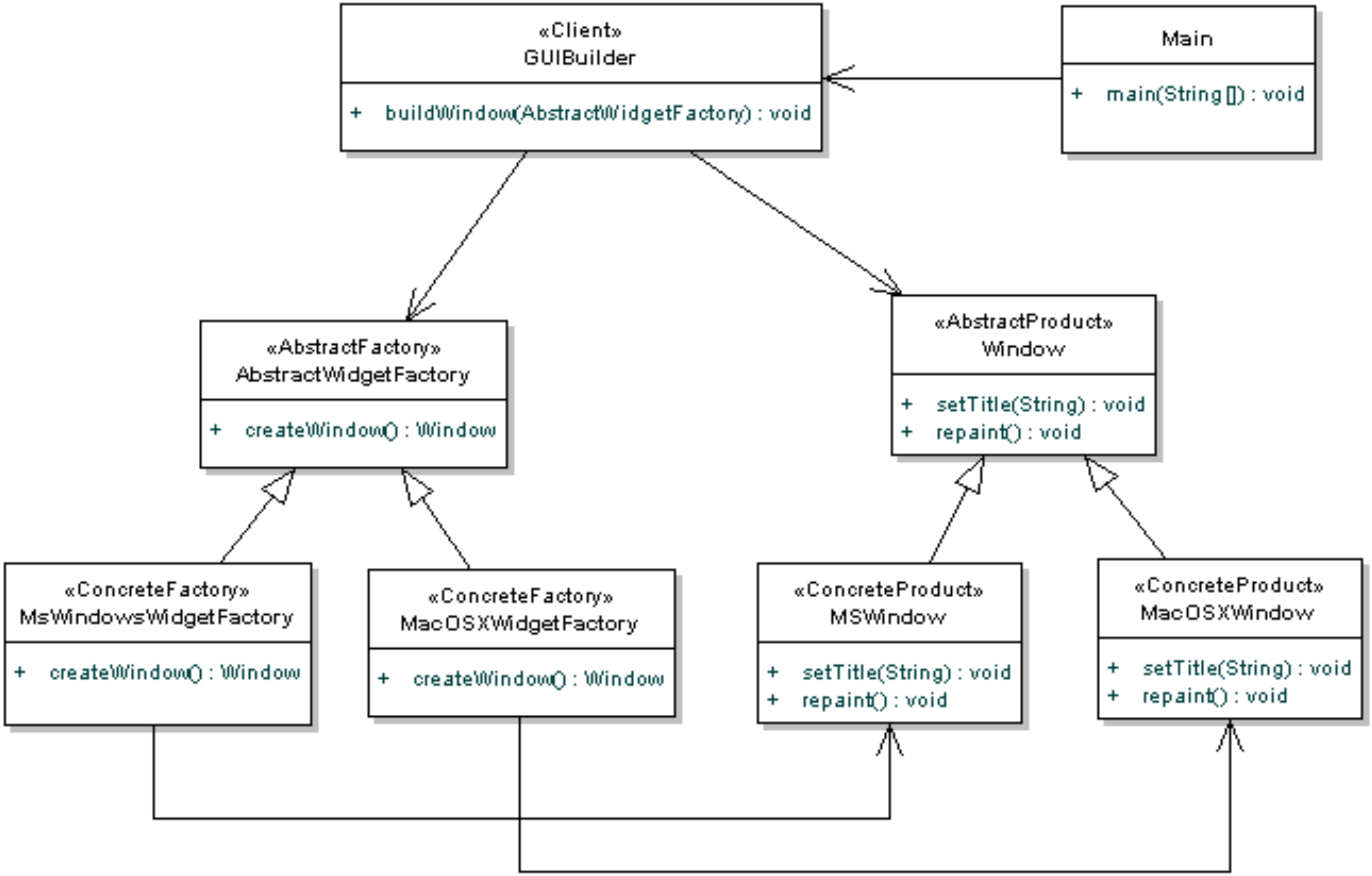
Plan for today

1. Problem
2. Discussion
3. Example Solution
4. Pattern

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



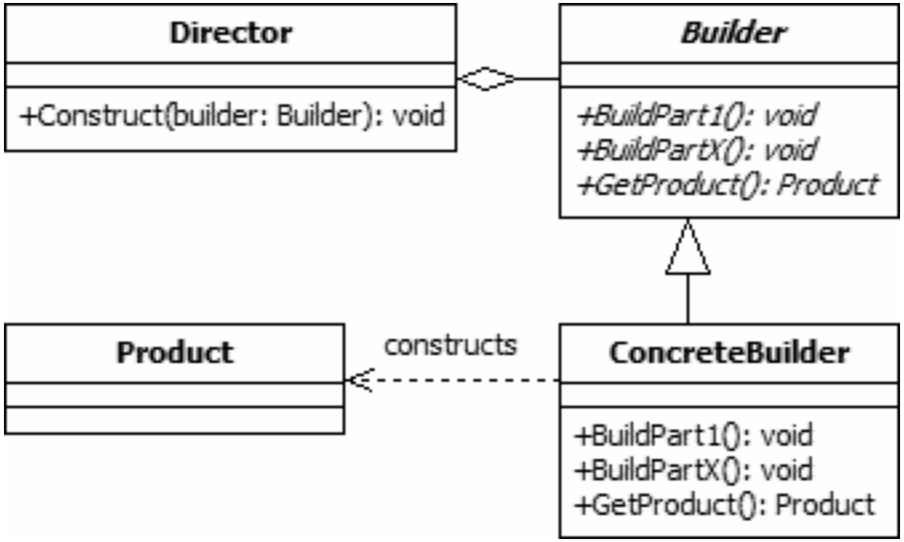
Abstract Factory

- 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 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?

Builder Pattern



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

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) { ... }
}
```

Creational Patterns

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

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[])`

Problem:

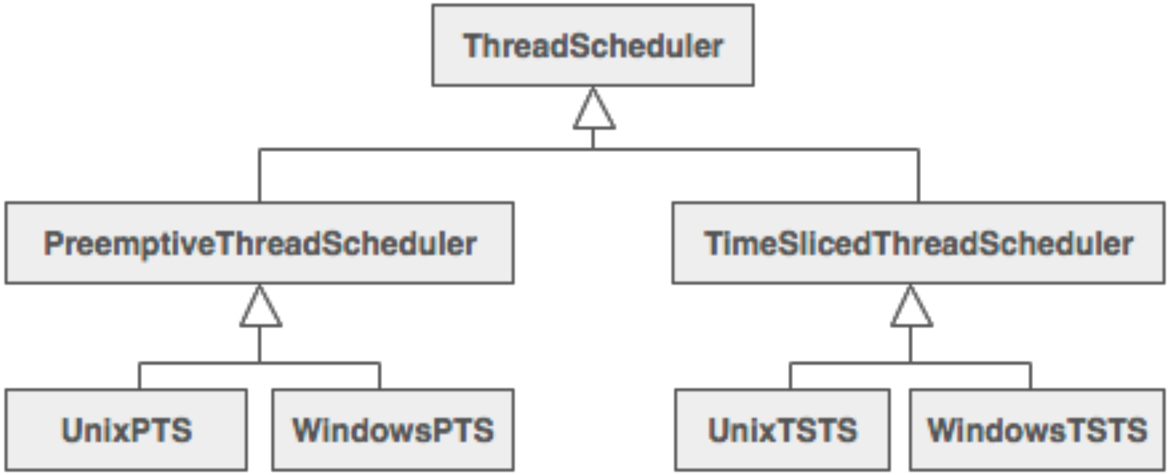


image source: https://sourcemaking.com

Problem:

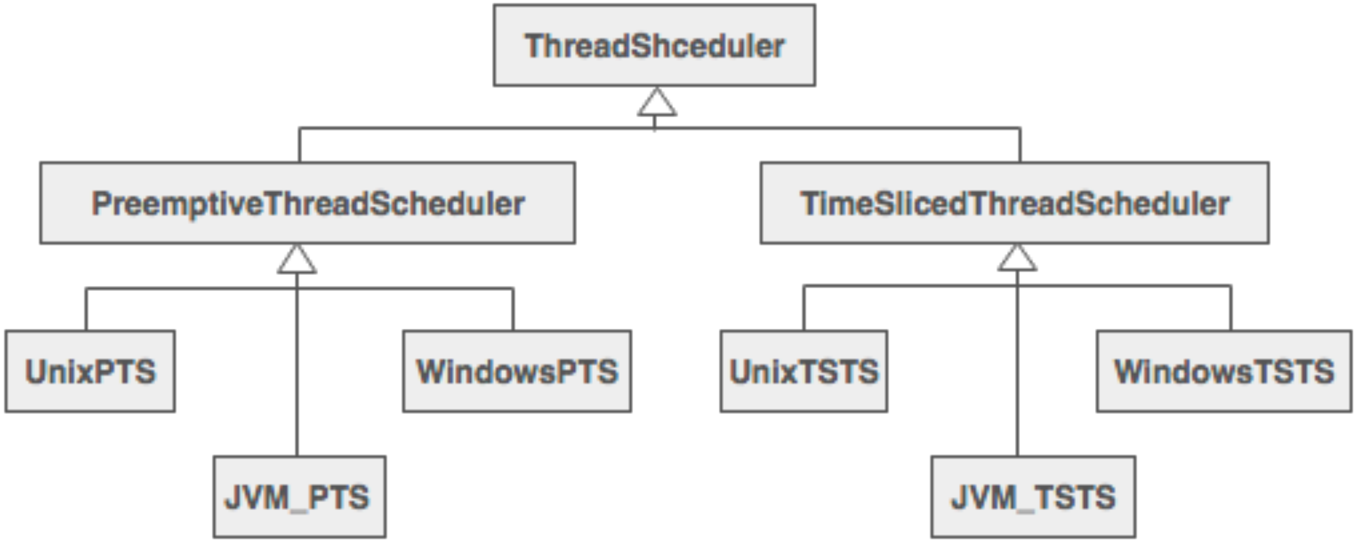


image source: https://sourcemaking.com

Bridge Pattern

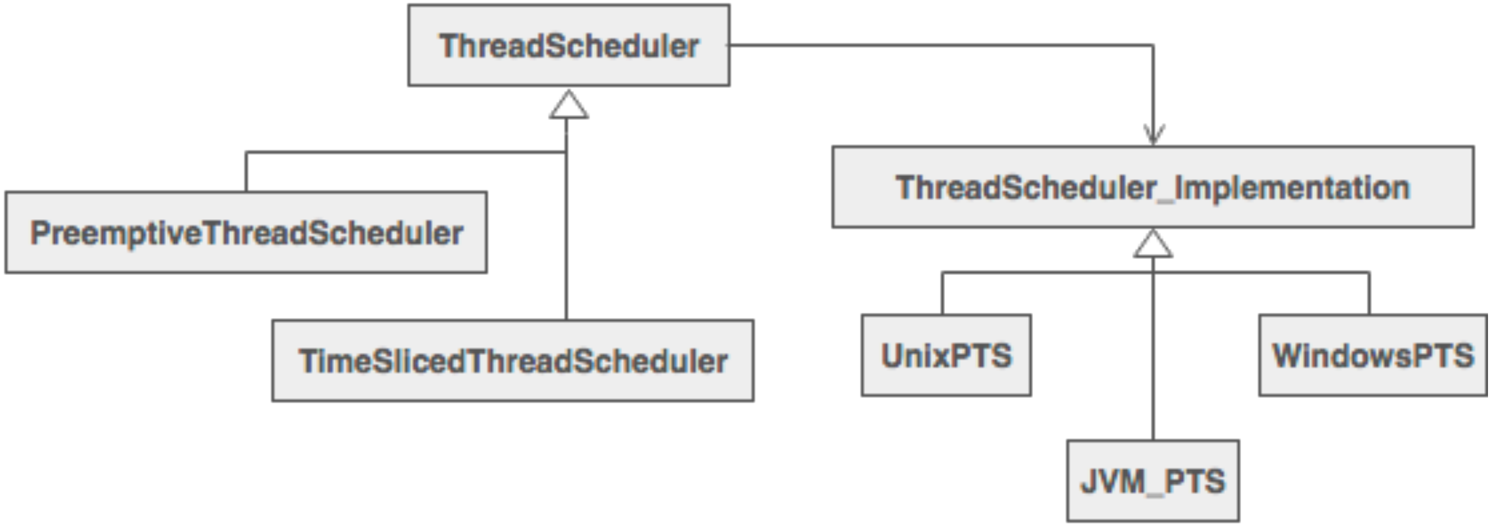

ThreadScheduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
ThreadScheduler_Implementation
UnixPTS
WindowsPTS
JVM_PTS

image source: https://sourcemaking.com

Bridge

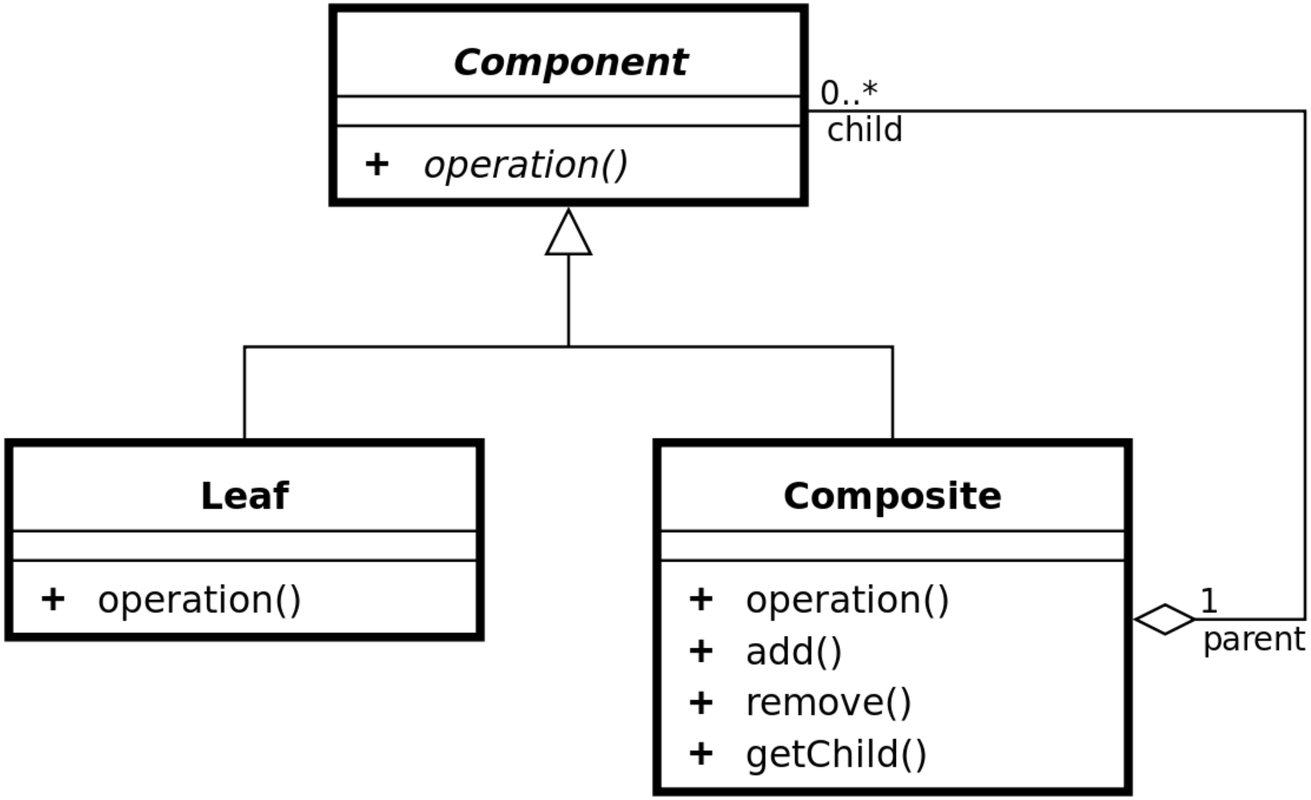
- Intent – decouple an abstraction from its implementation so they can vary independently
- Use case – portable windowing toolkit
- Key types – Abstraction, *Implementor*
- JDK – JDBC, Java Cryptography Extension (JCE), Java Naming & Directory Interface (JNDI)
- Bridge pattern *very* similar to Service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

Problem:

```
Graphic ::= ellipse | GraphicList
GraphicList ::= empty | Graphic GraphicList
```

We want to print all Graphics (ellipse, or list).

Composite Pattern



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`

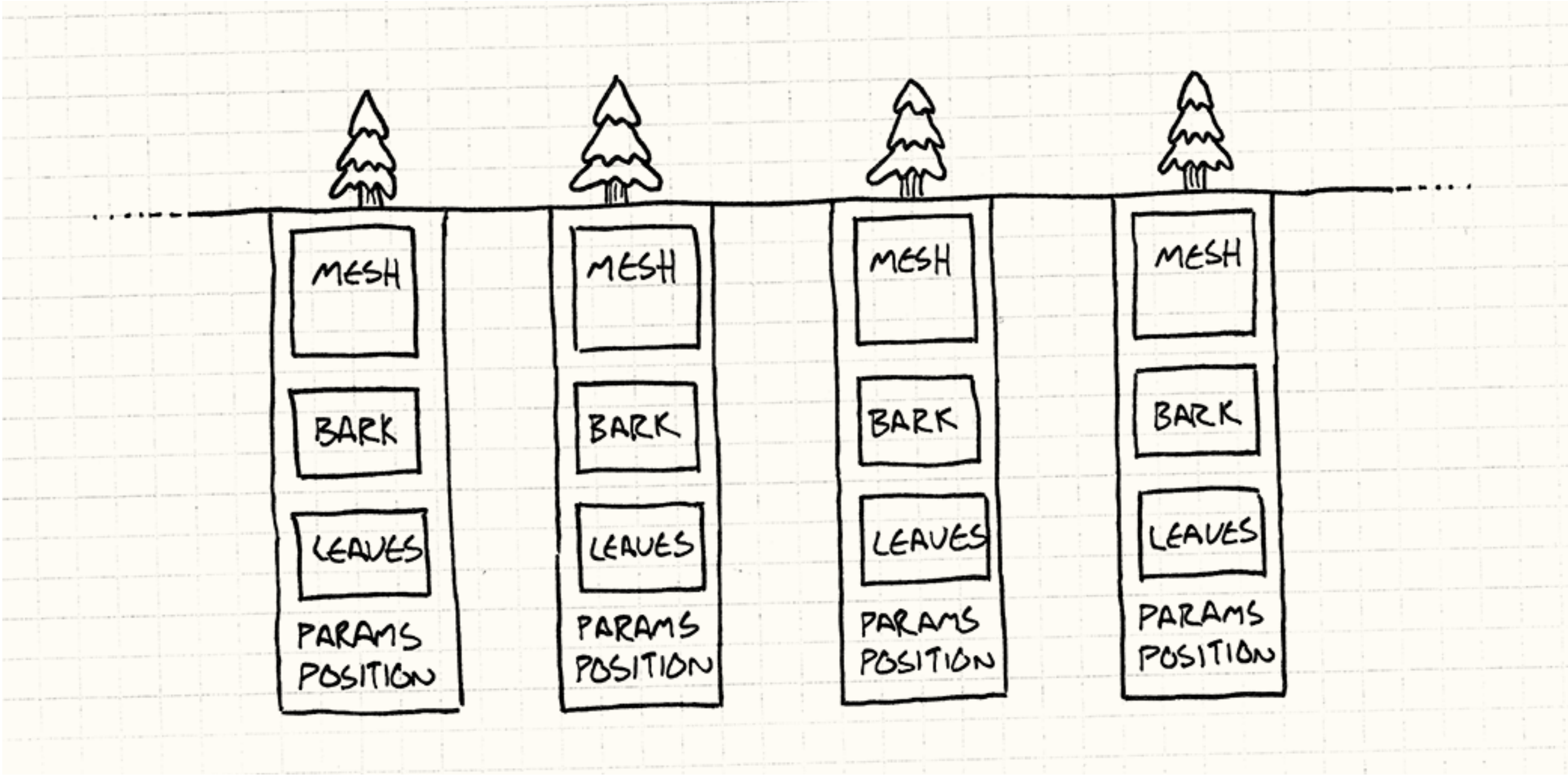
Decorator

- 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

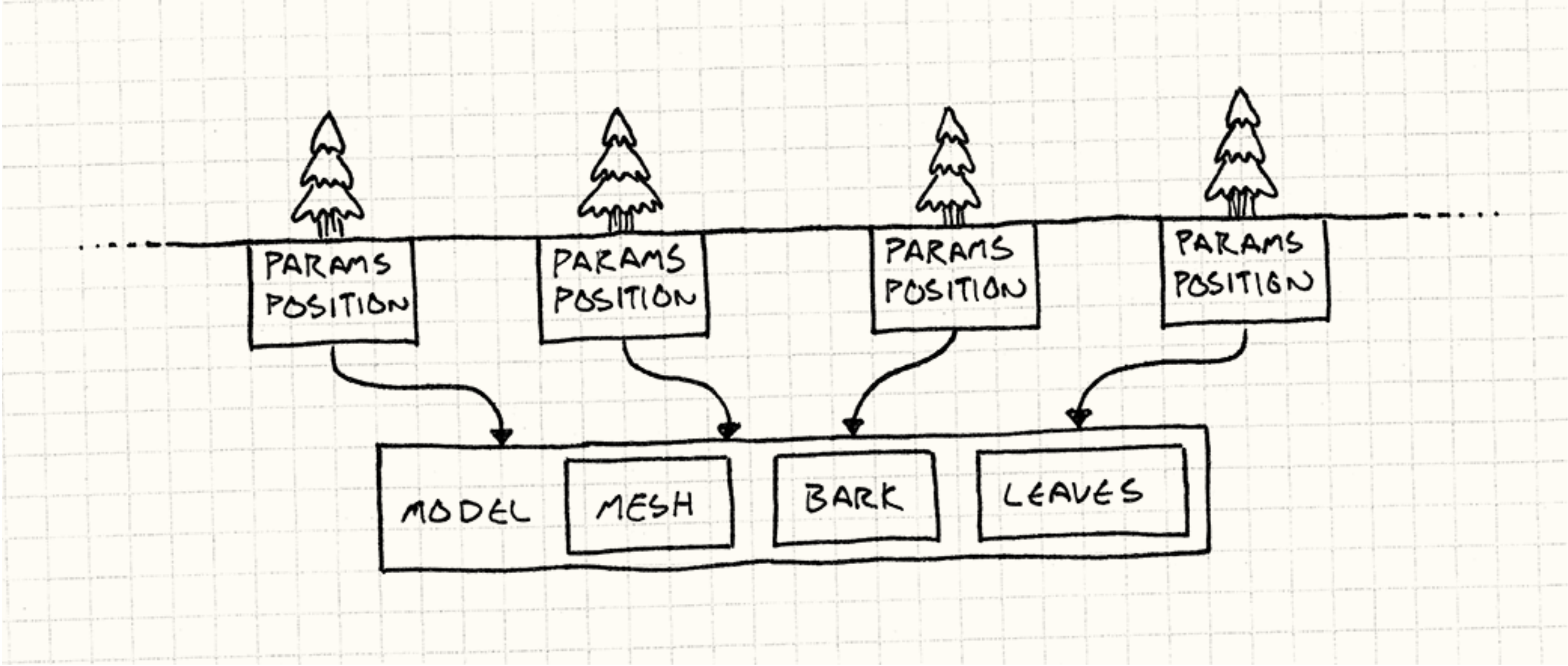
- Intent – provide a simple unified interface to a set of interfaces in a subsystem
  - GoF allow for variants where the complex underpinnings are exposed and hidden
- Use case – any complex system; GoF use compiler
- Key types – Façade (the simple unified interface)
- JDK – `java.util.concurrent.Executors`

Problem:



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

Problem:



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)

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

Structural Patterns

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