Principles of Software Construction: Objects, Design, and Concurrency

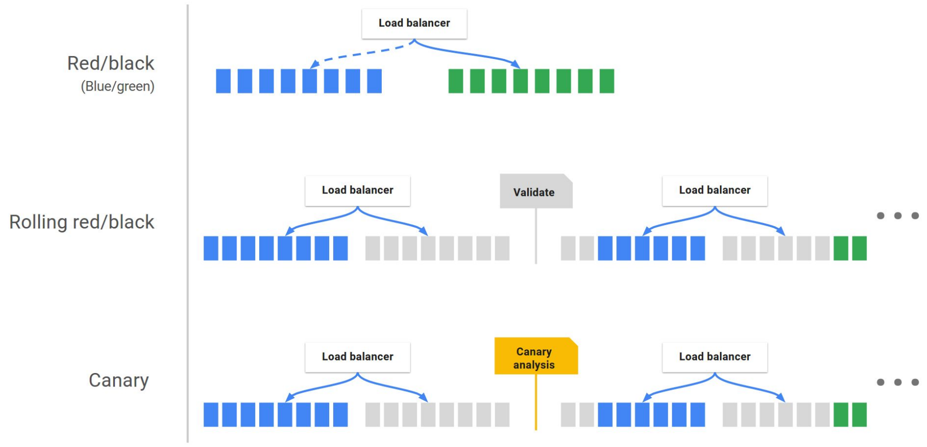
The “Gang of Four” Design Pattern Tour

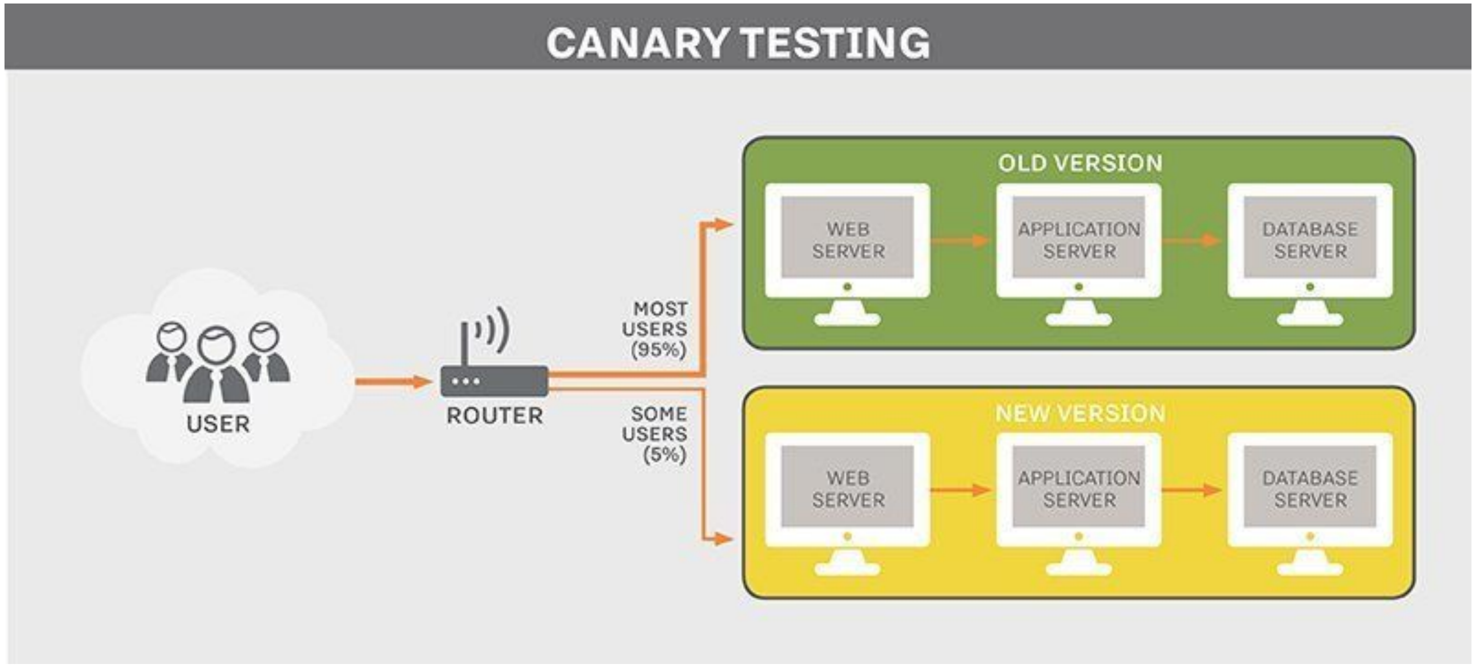
Charlie Garrod **Chris Timperley**

Administrivia

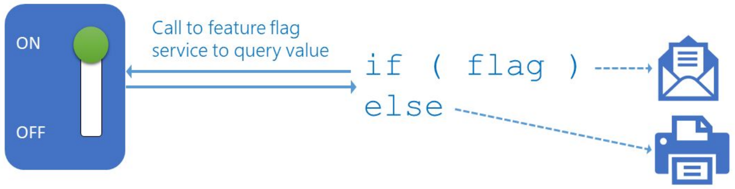
- No recitation this week
- Homework 6 checkpoint due at the end of today
  - Parallel implementation due next Wednesday

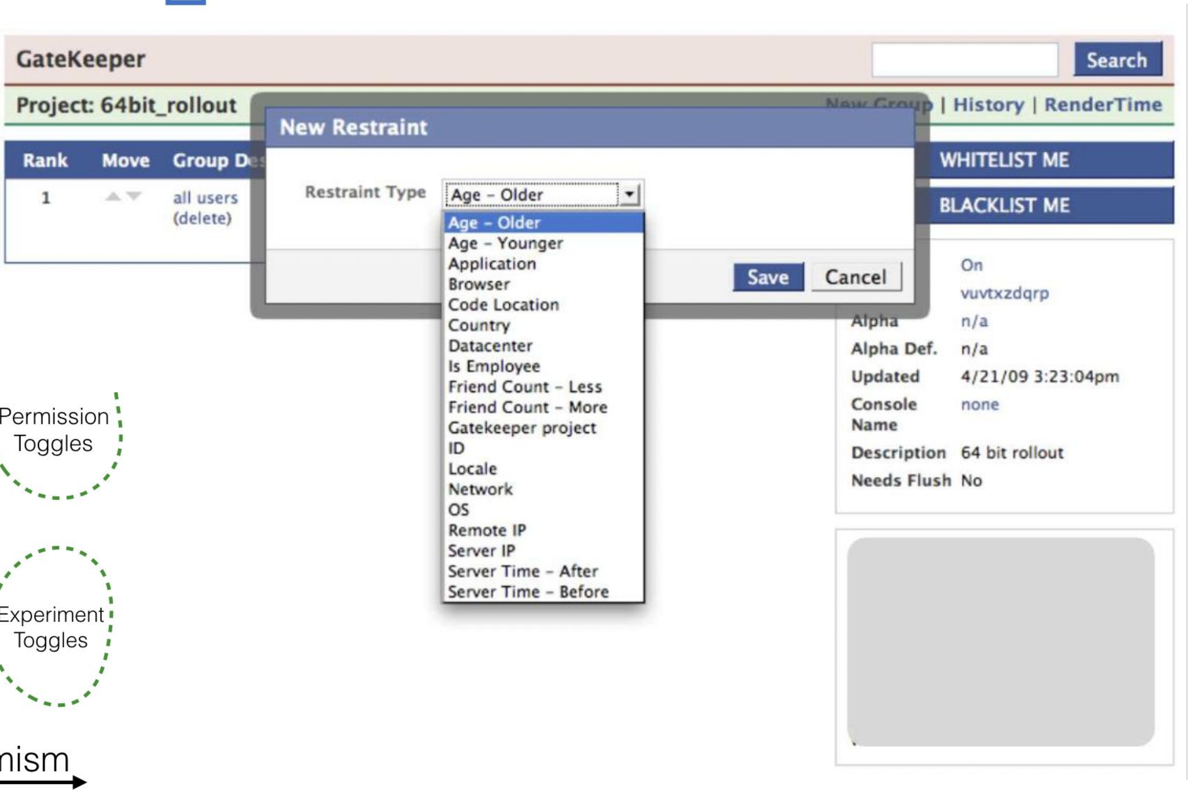
Unfinished Business


Red/black
(Blue/green)
Load balancer
Rolling red/black
Load balancer
Validate
Load balancer
Canary
Load balancer
Canary analysis
Load balancer


CANARY TESTING
USER
ROUTER
MOST USERS (95%)
SOME USERS (5%)
OLD VERSION
WEB SERVER
APPLICATION SERVER
DATABASE SERVER
NEW VERSION
WEB SERVER
APPLICATION SERVER
DATABASE SERVER

Control deployments at run-time using feature flags

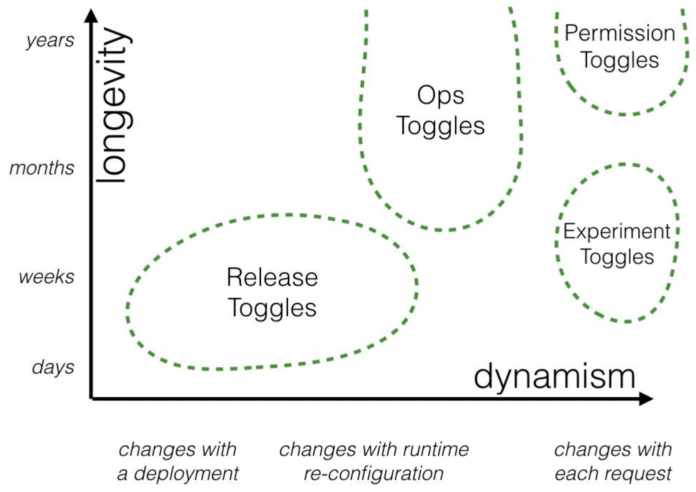




Restraint Type: Age – Older

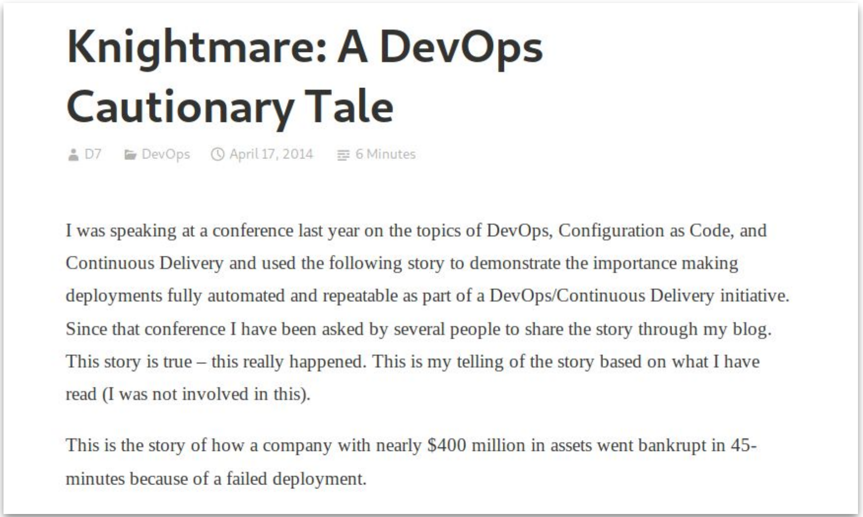
- Age – Older
- Age – Younger
- Application
- Browser
- Code Location
- Country
- Datacenter
- Is Employee
- Friend Count – Less
- Friend Count – More
- Gatekeeper project
- ID
- Locale
- Network
- OS
- Remote IP
- Server IP
- Server Time – After
- Server Time – Before

	On
	vuvtxzdqrp
Alpha	n/a
Alpha Def.	n/a
Updated	4/21/09 3:23:04pm
Console Name	none
Description	64 bit rollout
Needs Flush	No



https://martinfowler.com/articles/feature-toggles.html
https://docs.microsoft.com/en-us/azure/devops/migrate/phase-features-with-feature-flags?view=azure-devops

Warning! Feature flags can be dangerous



Knightmare: A DevOps Cautionary Tale

D7 DevOps April 17, 2014 6 Minutes

I was speaking at a conference last year on the topics of DevOps, Configuration as Code, and Continuous Delivery and used the following story to demonstrate the importance making deployments fully automated and repeatable as part of a DevOps/Continuous Delivery initiative. Since that conference I have been asked by several people to share the story through my blog. This story is true – this really happened. This is my telling of the story based on what I have read (I was not involved in this).

This is the story of how a company with nearly $400 million in assets went bankrupt in 45-minutes because of a failed deployment.

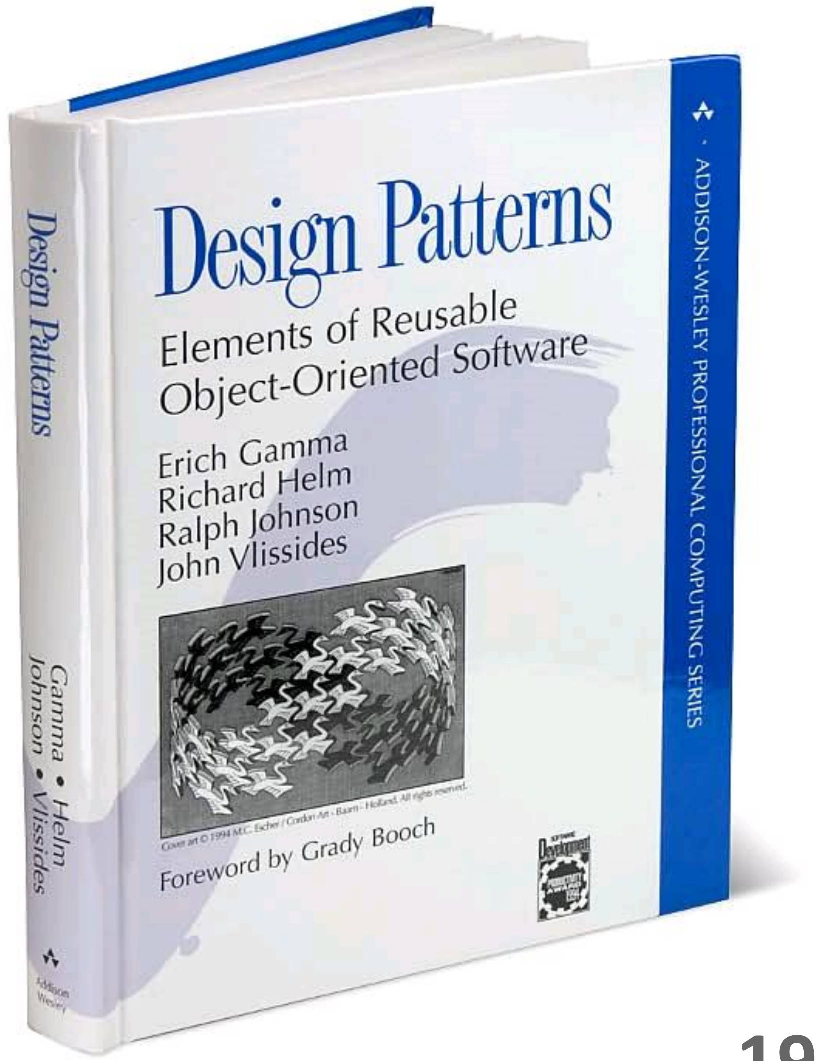
In laymen's terms, Knight Capital Group realized a $460 million loss in 45-minutes. Remember, Knight only has $365 million in cash and equivalents. **In 45-minutes Knight went from being the largest trader in US equities and a major market maker in the NYSE and NASDAQ to bankrupt.**

https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/

Summary

- DevOps brings development and operations together
  - Automation, Automation, Automation
  - Infrastructure as code
- Release management
  - Versioning and branching strategies
- Continuous deployment is increasingly common
- Exploit opportunities of continuous deployment; perform testing in production and quickly rollback
  - Experiment, measure, and improve

Today: A tour of the 23 "Gang of Four" patterns



1994

What patterns have we discussed in class?

How did you use those design patterns?

- Strategy
- Template Method
- Façade
- Iterator
- Adapter
- Composite
- Decorator
- Observer
- Marker interface
- Different forms of factories
- Concurrency patterns as producer-consumer, thread pool
- …

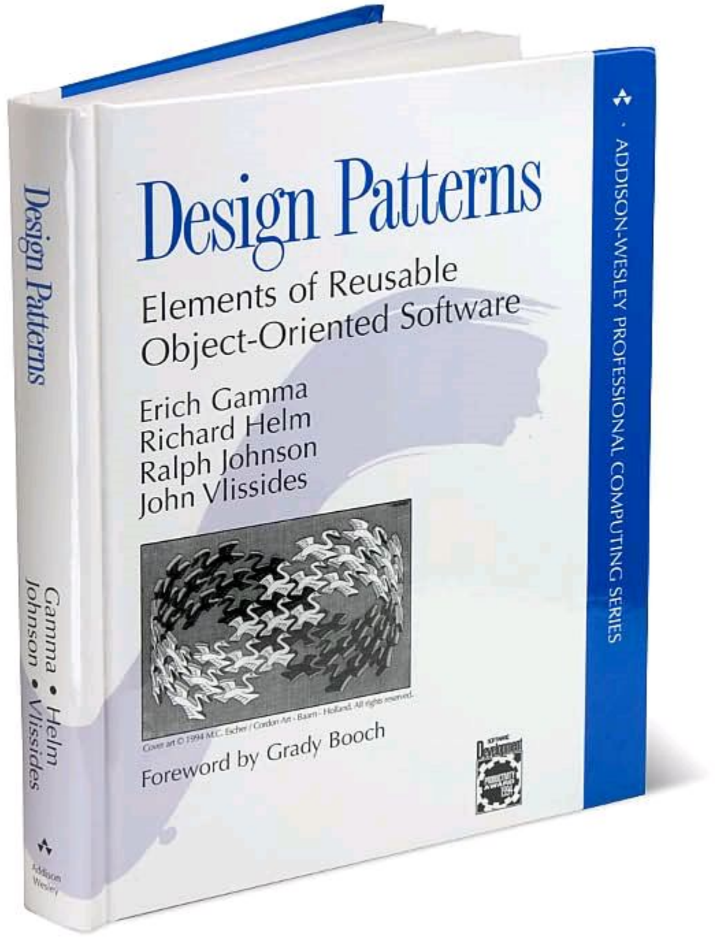
Why learn the Gang of Four design patterns?

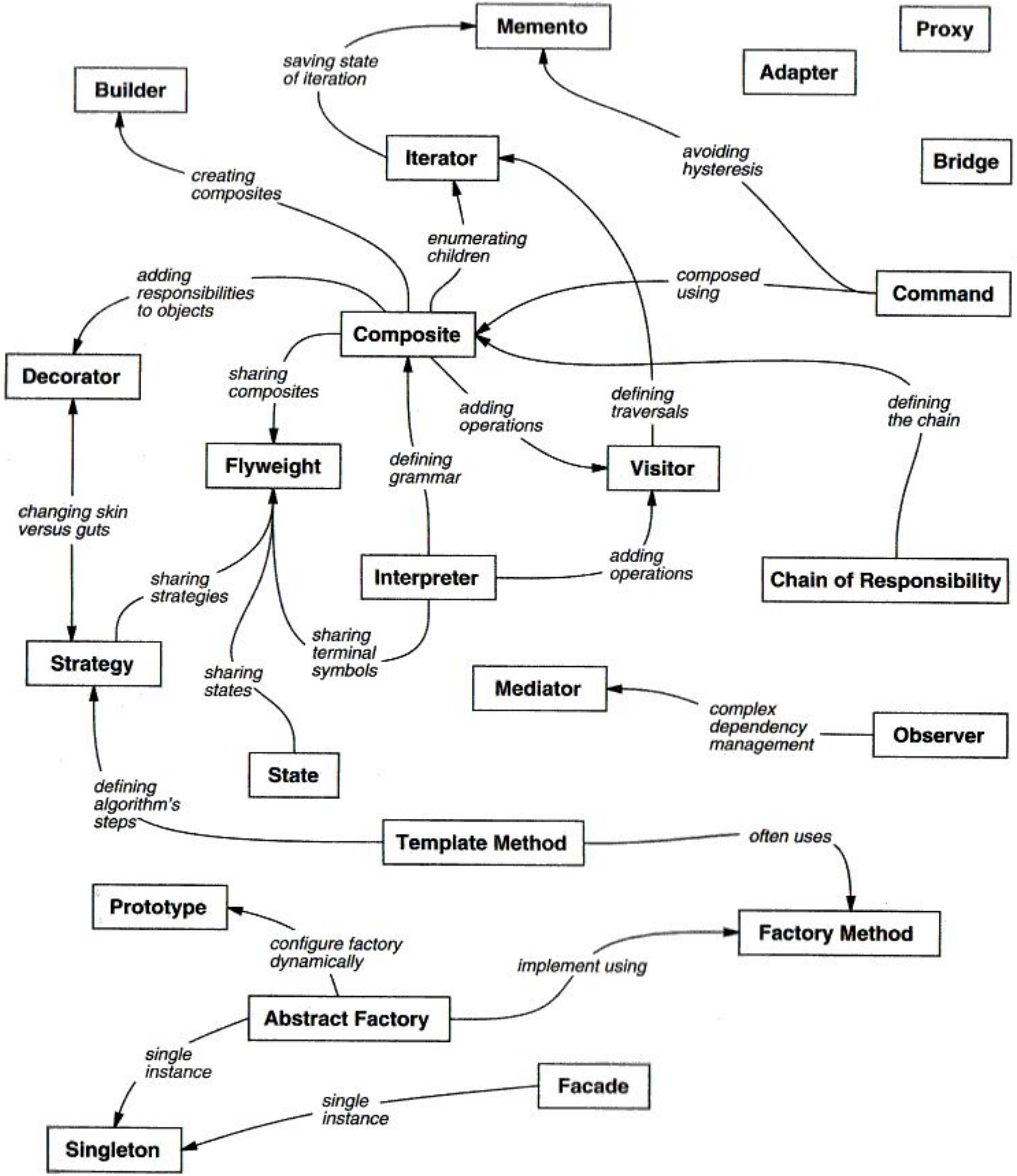
- Seminal and canonical list of well-known patterns
  - Patterns that have stood the test of time!
- Not all patterns are commonly used
- Does not cover all popular patterns

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

Today: A tour of the "Gang of Four" patterns

1. Creational Patterns
2. Structural Patterns
3. Behavioral Patterns




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
adding operations
Chain of Responsibility
sharing strategies
Interpreter
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

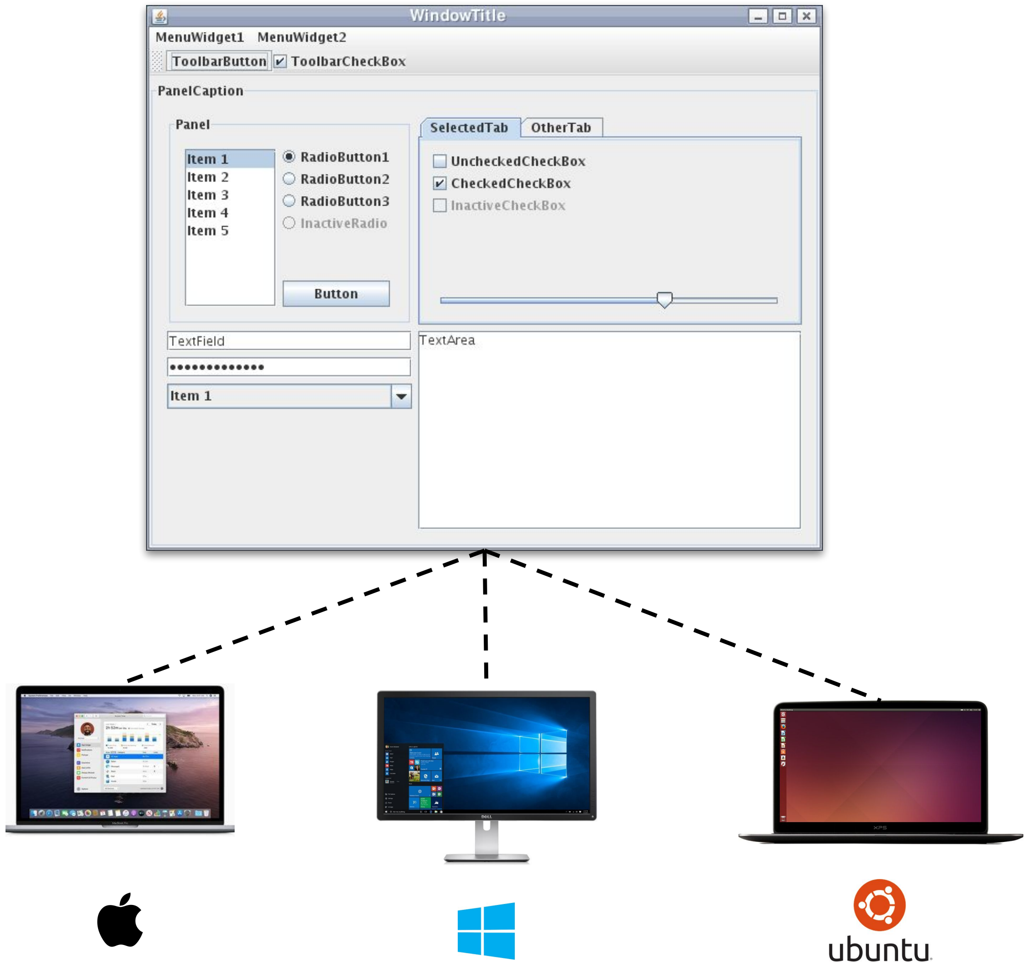
[Pattern Name]

- **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: Build a GUI that supports multiple platforms



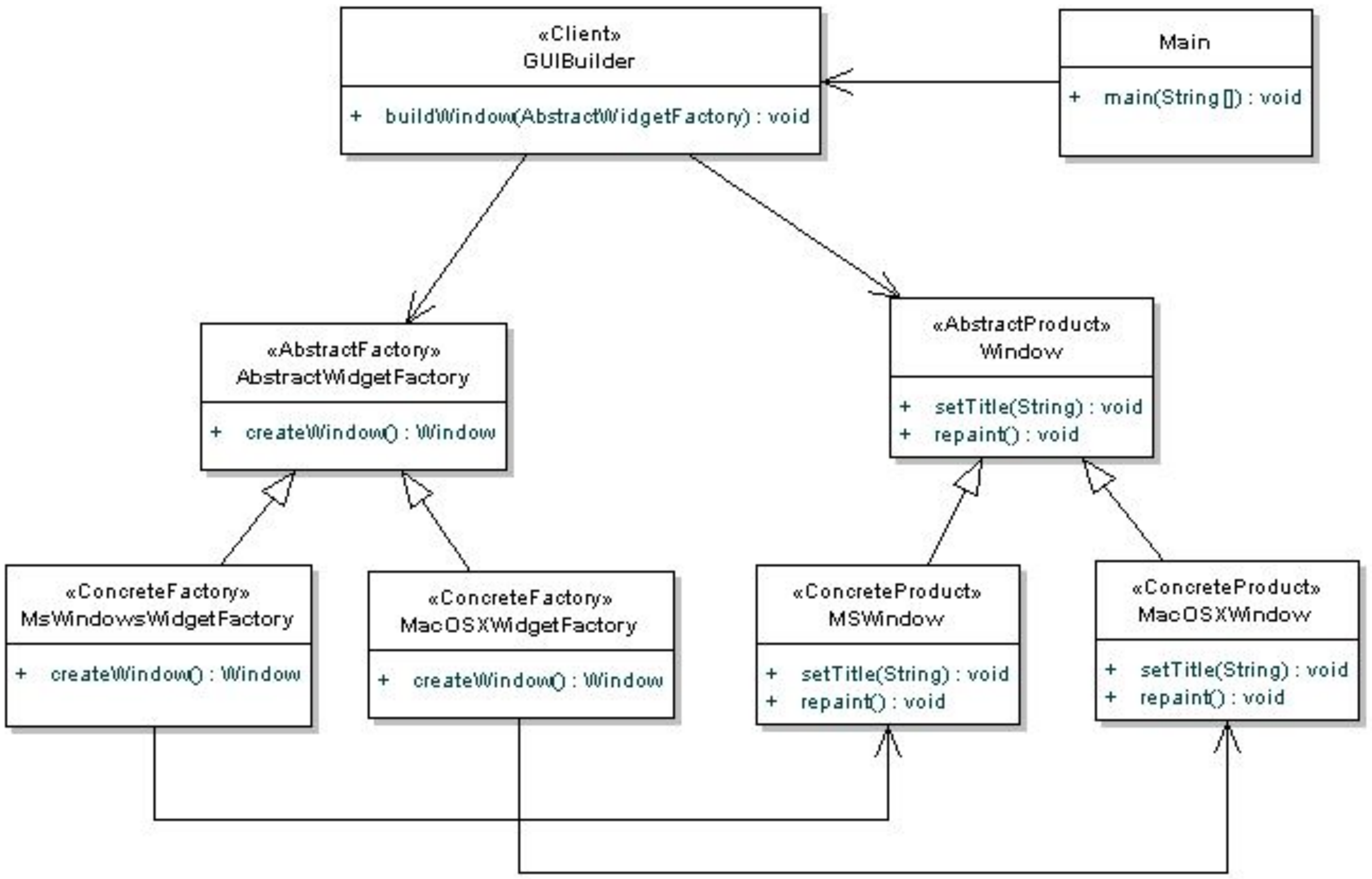
The rest of our code should be platform independent!

Hide platform-specific details behind an interface

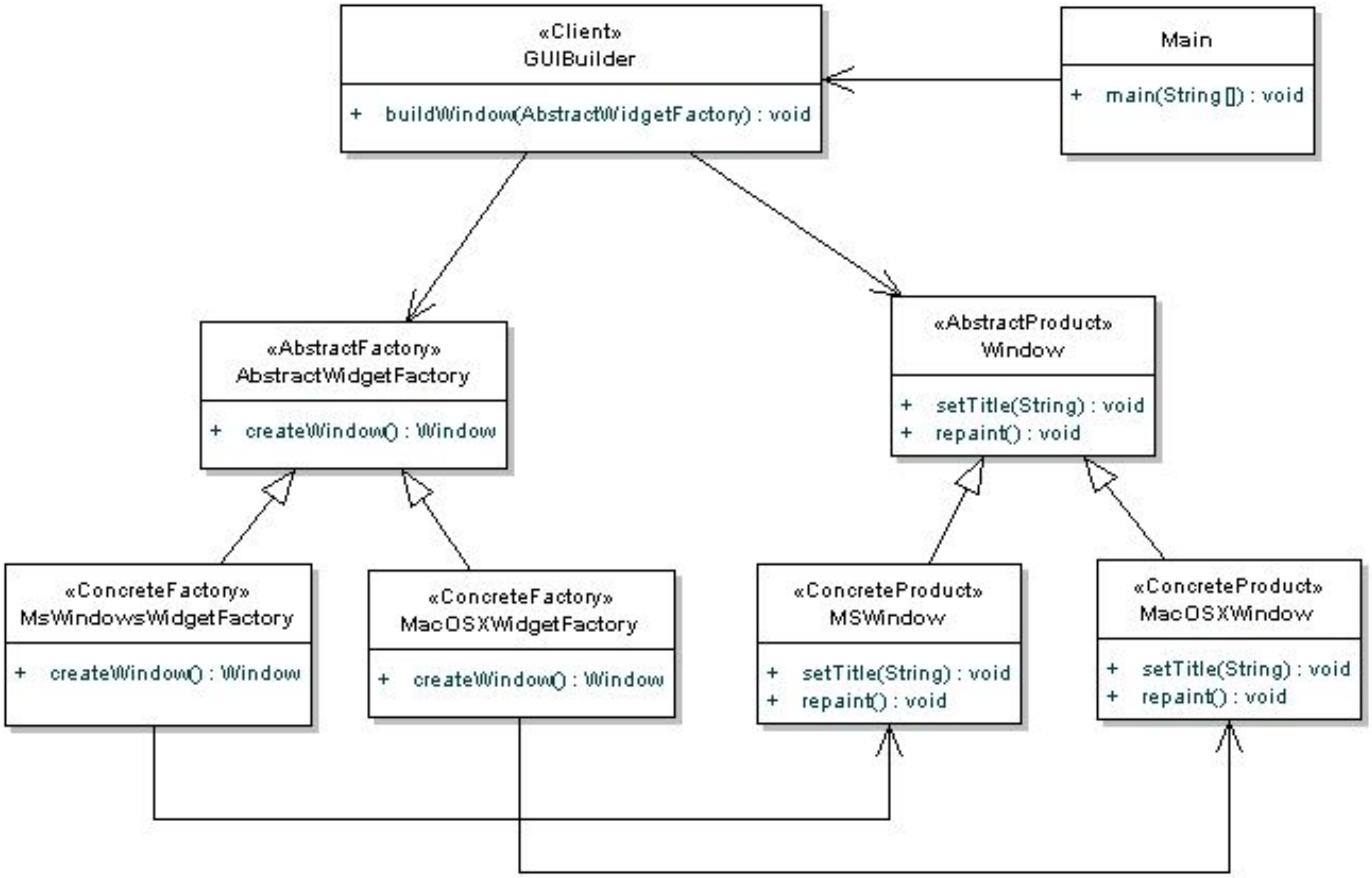
```
public interface Window {
    private void setTitle(String title);
    private void repaint();
}

public class MSWindow implements Window {
    ...
}

public class MacOSXWindow implements Window {
    ...
}
```

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

Abstract Factory Pattern



```
public class GUIBuilder {
  public void buildWindow(AbstractWidgetFactory widgetFactory) {
    Window window = widgetFactory.createWindow();
    window.setTitle("New Window");
```



1. Abstract Factory

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

Problem: Allow the construction to complex objects

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

How can we handle all combinations of fields when constructing User objects?

One solution...

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

What are the problems with this solution?

One solution...

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

Code becomes had to read and maintain with many attributes!

Another solution: Default no-arg constructor plus setters and getters for every attribute

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

  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;
  }
  ...
```

Potentially inconsistent state; mutable

A better solution: Use a `Builder` to hold build instructions.

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

  private User(UserBuilder builder) {
    this.firstName = builder.firstName;
    this.lastName = builder.lastName;
    …
  }

  public String getFirstName() { … }
  public String getLastName() { … }
  …
}
```

```
new User.Builder(“Fred”, “Rogers”)
    .age(30)
    .phone(“1234567”)
    .address(...)
    .build();
```

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

  private 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;
  }
  …
}
```

2. 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
- Key types: *Builder,* ConcreteBuilders, Director, Products
- JDK: java.lang.StringBuilder, java.lang.StringBuffer

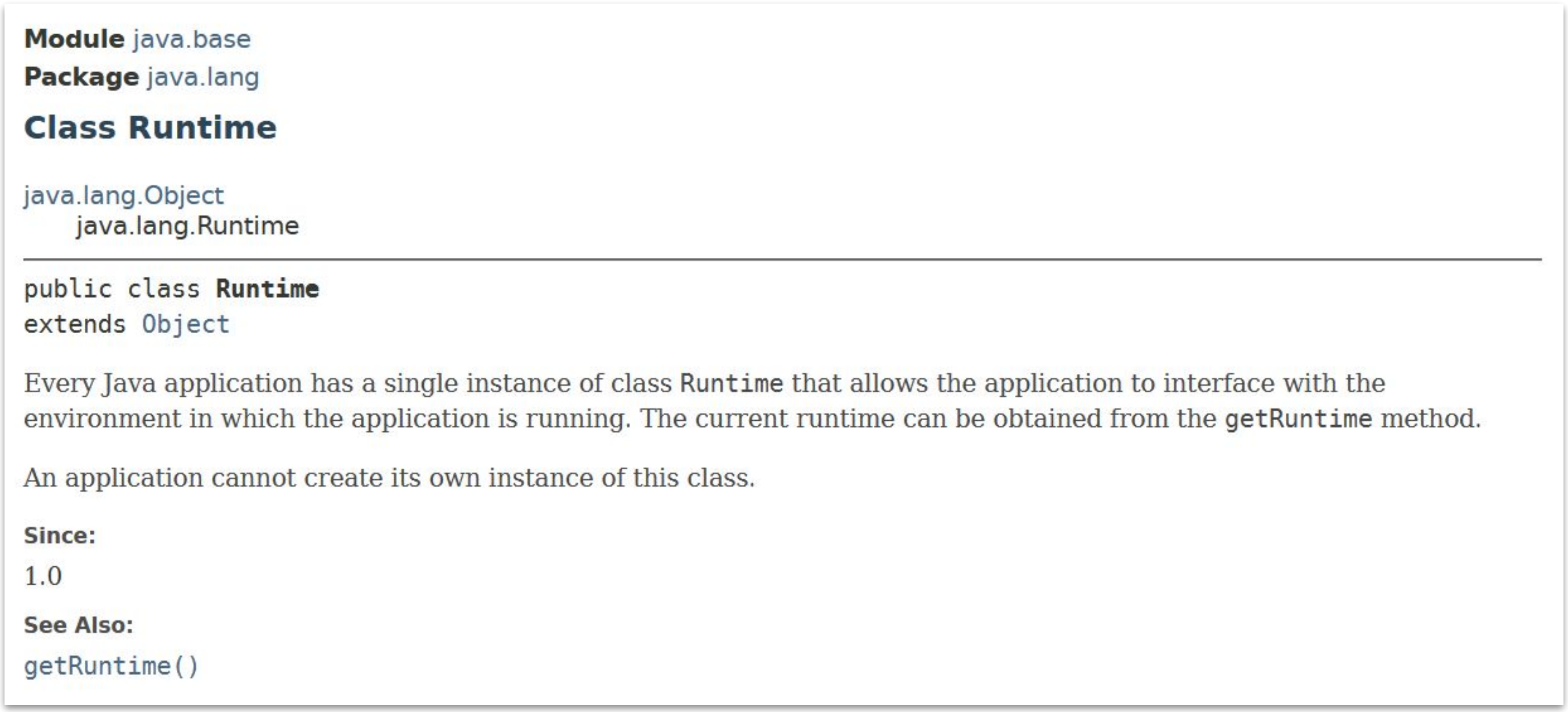
3. Factory Method

- 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()`

Illustration: Factory Method

```
public interface Iterable<E> {
    public 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: There should only be one instance of a class, and that class should be globally accessible.



Module java.base
Package java.lang

Class Runtime

java.lang.Object
    java.lang.Runtime

```
public class Runtime
extends Object
```

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `getRuntime` method.

An application cannot create its own instance of this class.

Since:
1.0

See Also:
`getRuntime()`

Prevent client from creating instances by using a private default constructor

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

Better: Use a static method to create and retrieve the single instance

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

Can you spot the bug in this code?

Better: Use a static method to create and retrieve the single instance

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

Even better: Use an Enum for guaranteed serialization and thread safety!

```
public enum Elvis {
    ELVIS;

    void sing(Song song) { … }
    void eat(Food food) { … }
    void playGuitar(Guitar guitar) { … }
    ...
}
```

5. 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 (e.g., stateless function objects, EmptySet, etc.)
- Key types: Singleton
- JDK: `java.lang.Runtime.getRuntime()`, `java.util.Collections.emptyList()`

Aside: Singleton is an *instance-controlled class*

- **Static utility class:** non-instantiable
  - difficult to mock; violates information hiding
- **Enum:** one instance per value, all values known at compile time
- **Interned class:** one canonical instance per value; new values created at runtime
  - Used for values -128 to 127 by `Integer.valueOf`

Generally prefer singletons over static utility classes

- If you ever might need more than one instance in the future.
- If you need an object that implements other interfaces.
  - Allows mocking and dependency injection

II. Structural Patterns

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

1. Adapter

- Intent: convert interface of a class into one that another class requires, allowing interoperability
  - make things work together after they're designed
- Use case: numerous, e.g., arrays vs. collections
- Key types: Target, Adaptee, Adapter
- JDK: `Arrays.asList(T[])`

Illustration: Adapter

Have this

Adaptee

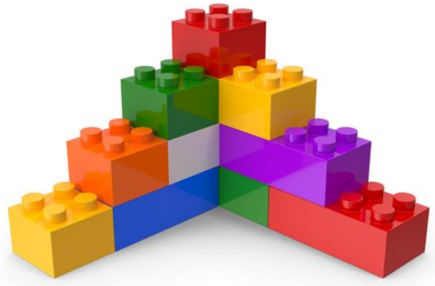
and this?

Target

Use this!

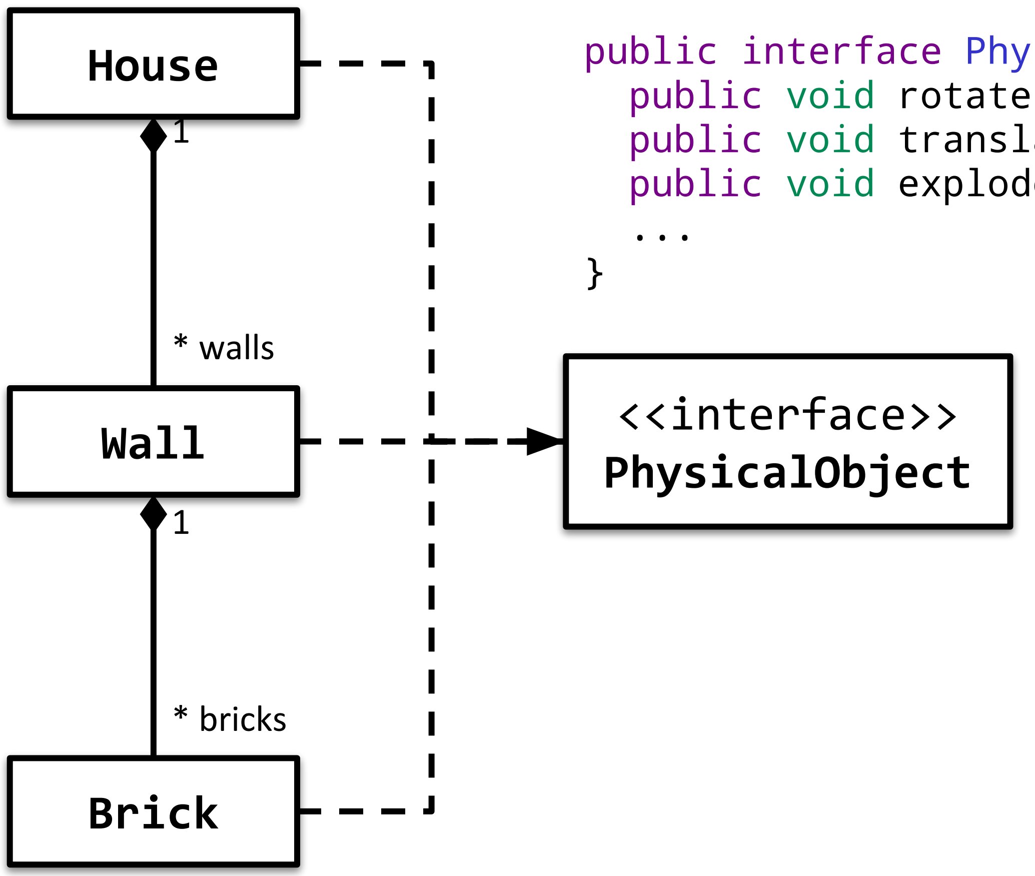
Adapter

Problem: You want to treat structures of objects as if they were an individual object.



Common operations: `rotate`, `translate`, `explode`, ...

Compose objects into tree structures, and implement a common interface.



```
public interface PhysicalObject {
  public void rotate(Orientation orientation);
  public void translate(Vector translation);
  public void explode(int explosiveness);
  ...
}
```

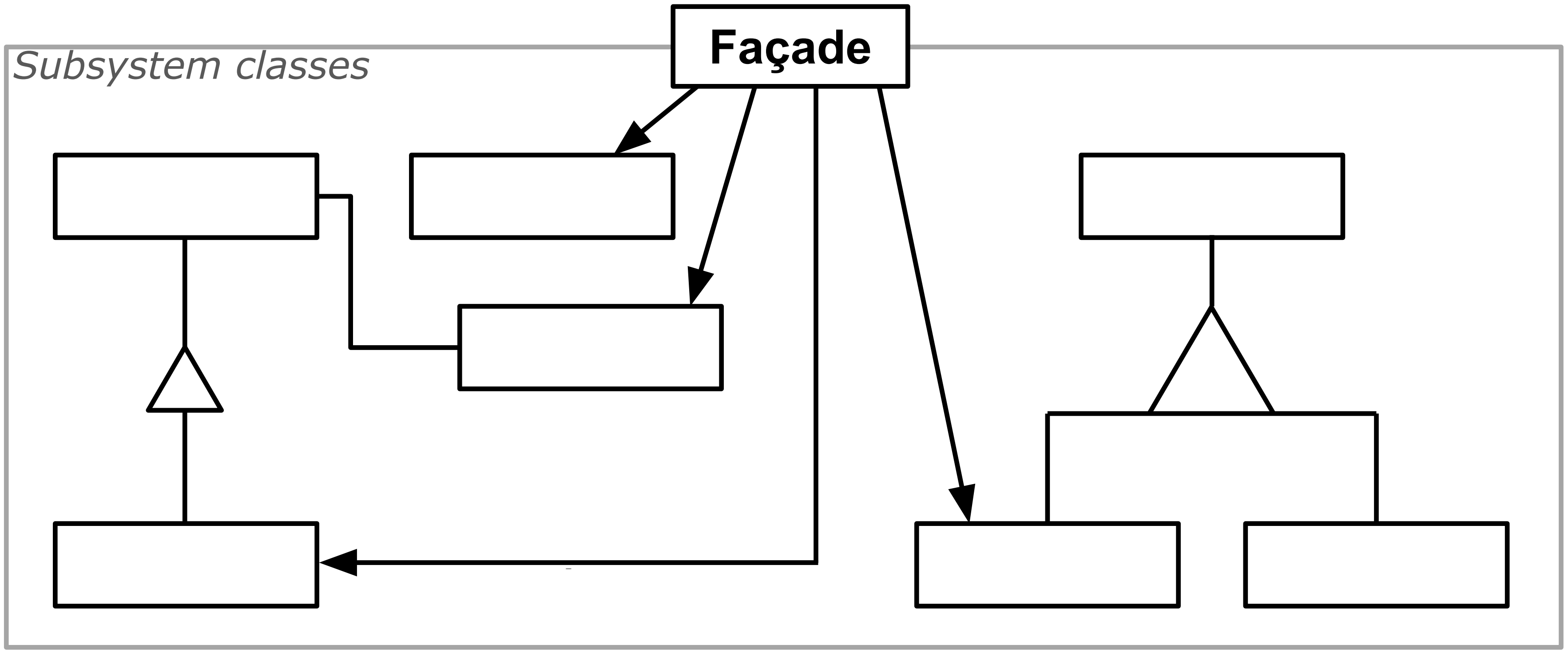
3. Composite

- 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`

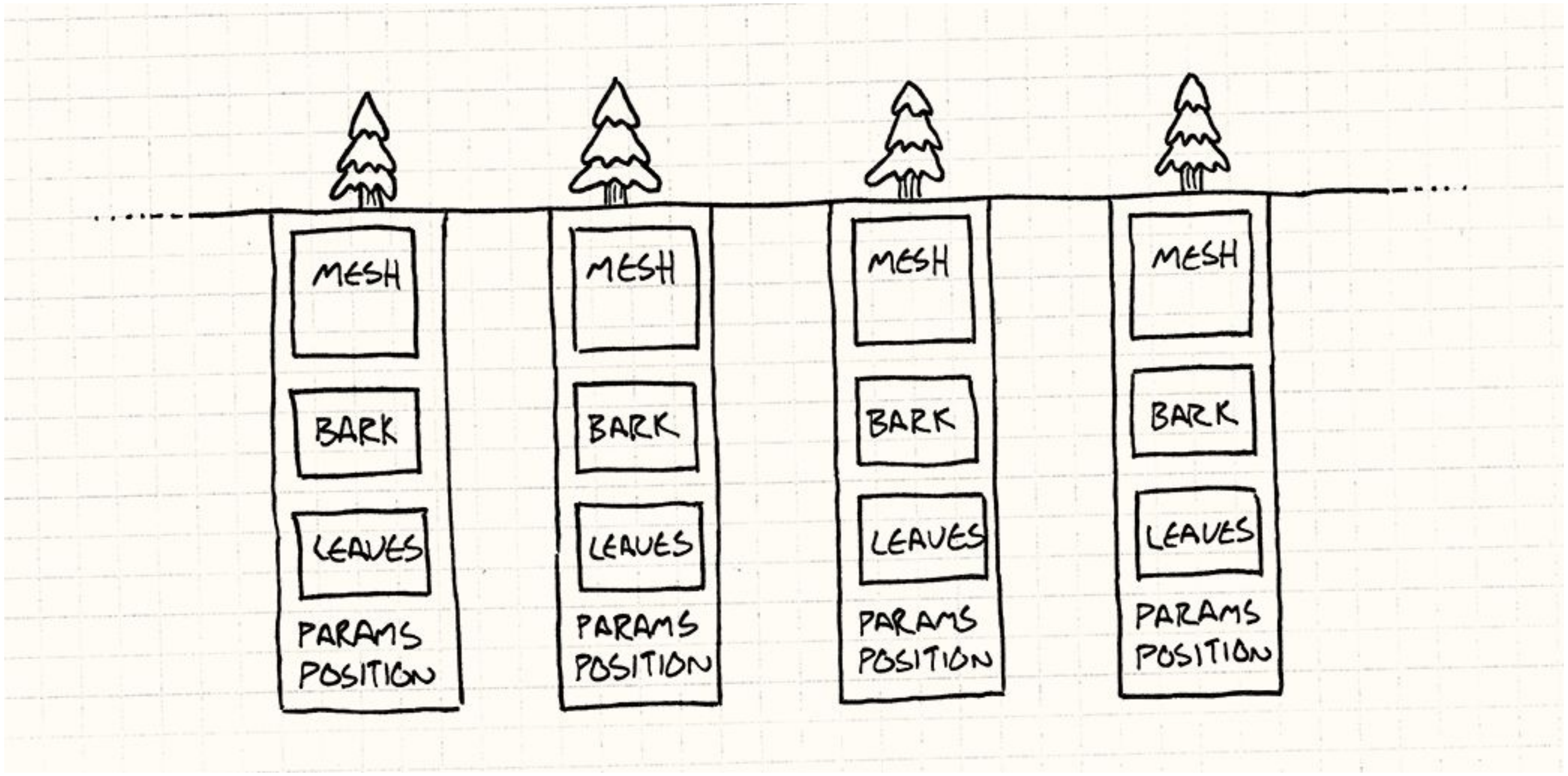
5. Façade

- 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`

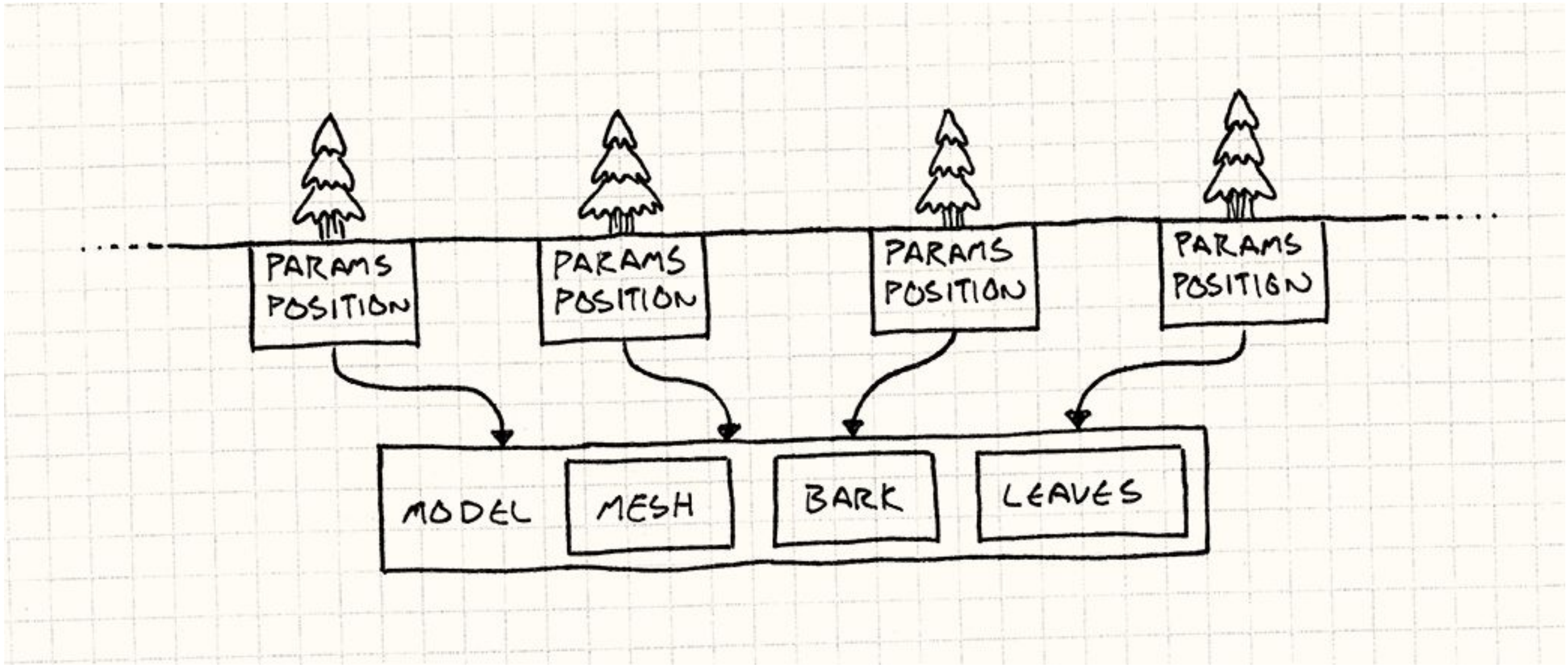
Illustration: Façade



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



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



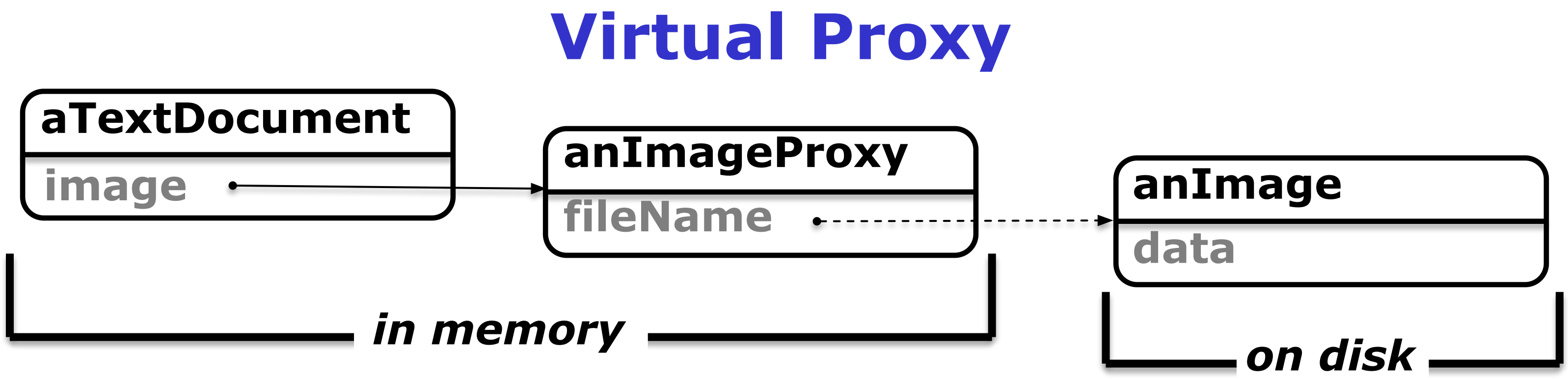
6. Flyweight

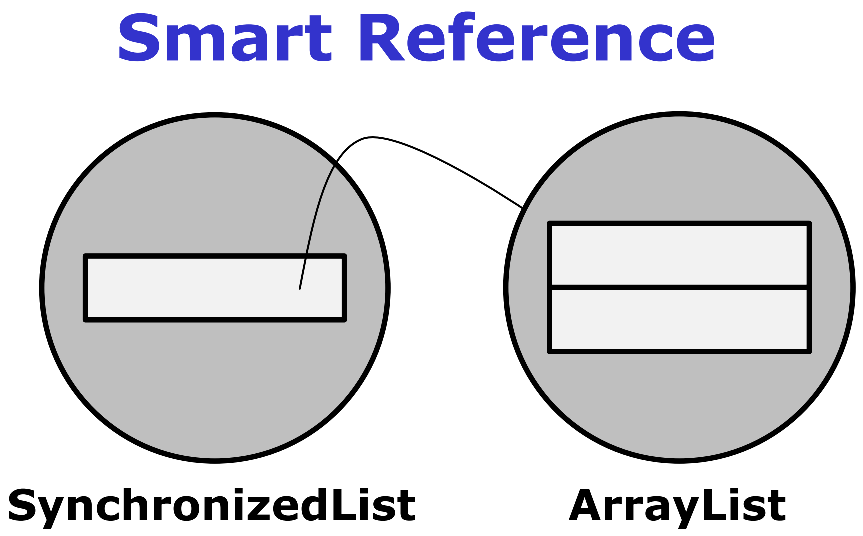
- Intent: use sharing to support large numbers of fine-grained objects efficiently
- Use case: characters in a document
- Key types: Flyweight (instance-controlled!)
  - Some state can be *extrinsic* to reduce number of instances
- JDK: Common! All enums, many others
  - `j.u.c.TimeUnit` has number of units as extrinsic state

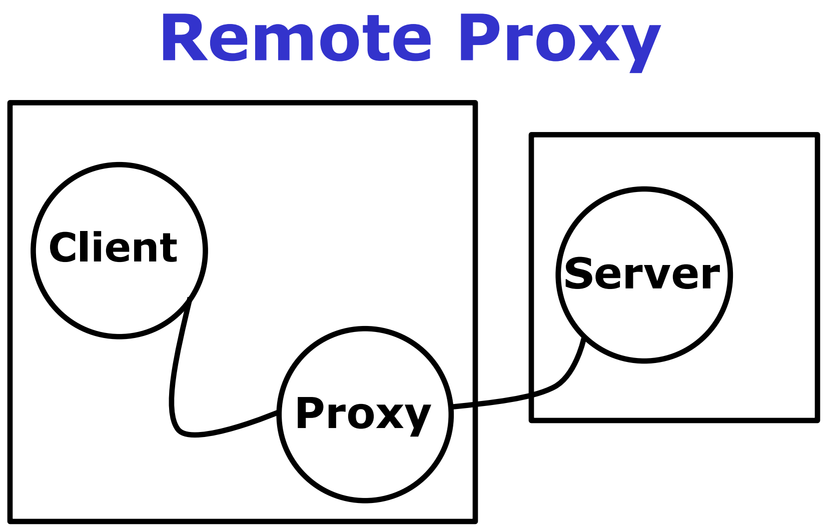
7. Proxy

- 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: remote method invocation, collections wrappers

Illustration: Proxy



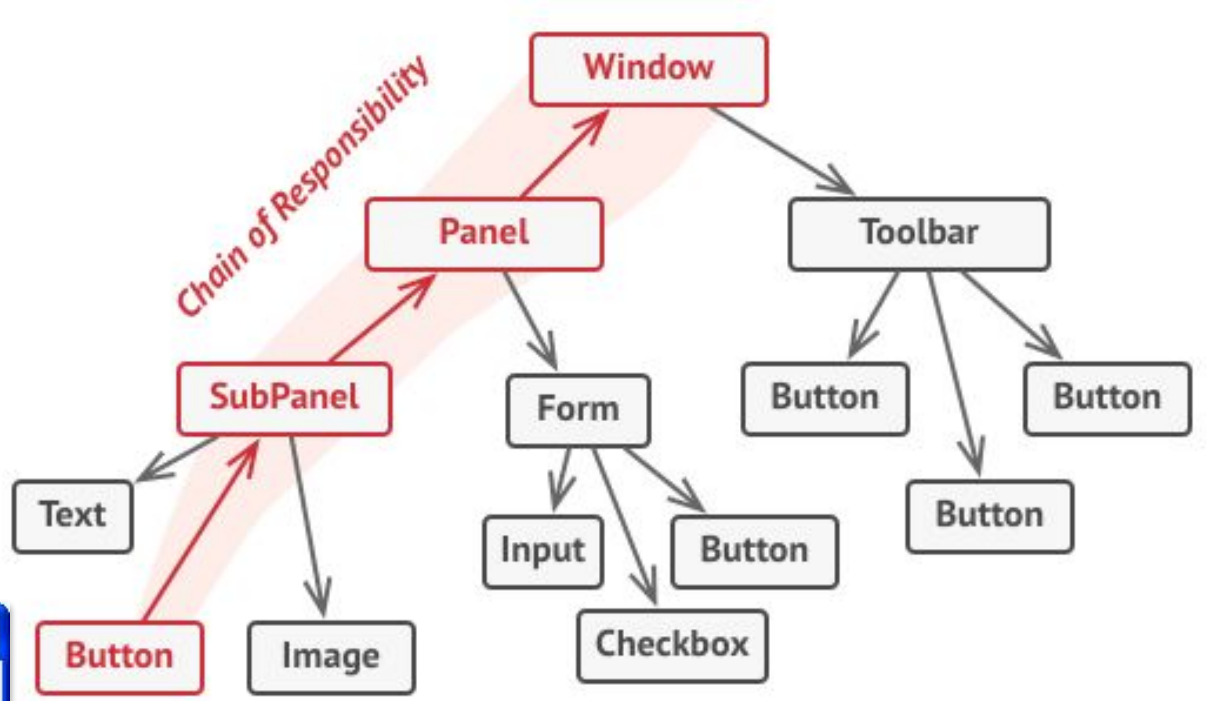


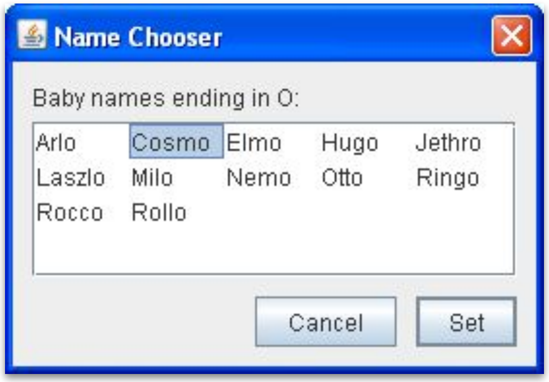


III. Behavioral Patterns

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~~

Problem: An *event or request* should be *handled* by one of its ancestral objects





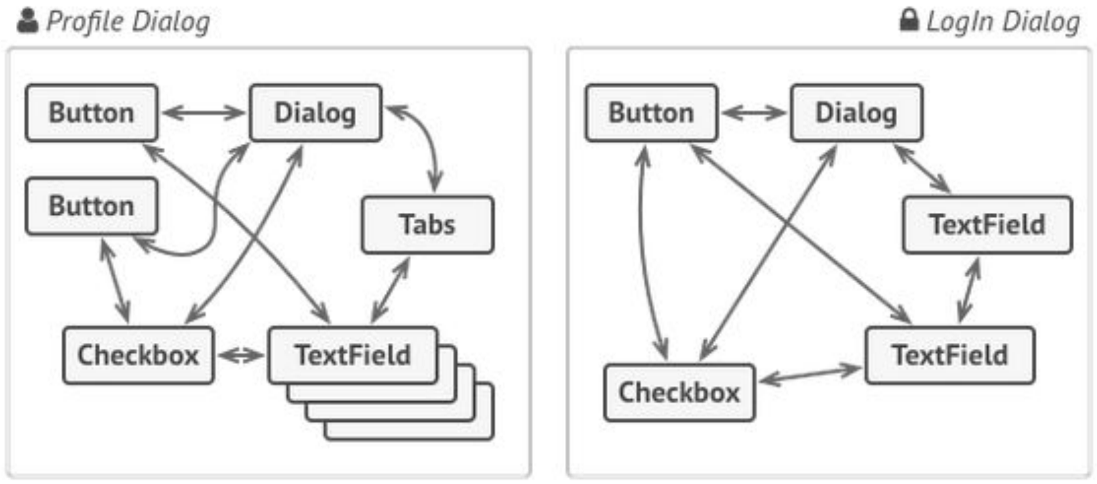
Baby names ending in O:

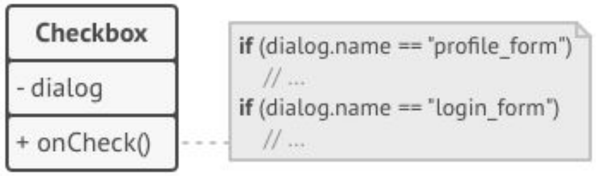
Arlo	Cosmo	Elmo	Hugo	Jethro
Laszlo	Milo	Nemo	Otto	Ringo
Rocco	Rollo			

1. 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

Problem: Components are tightly coupled.

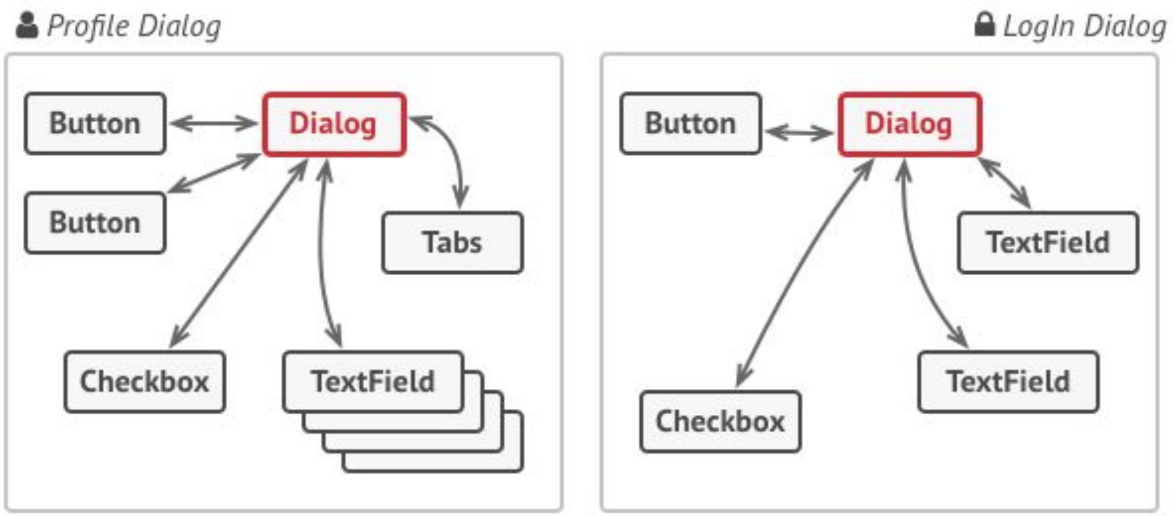




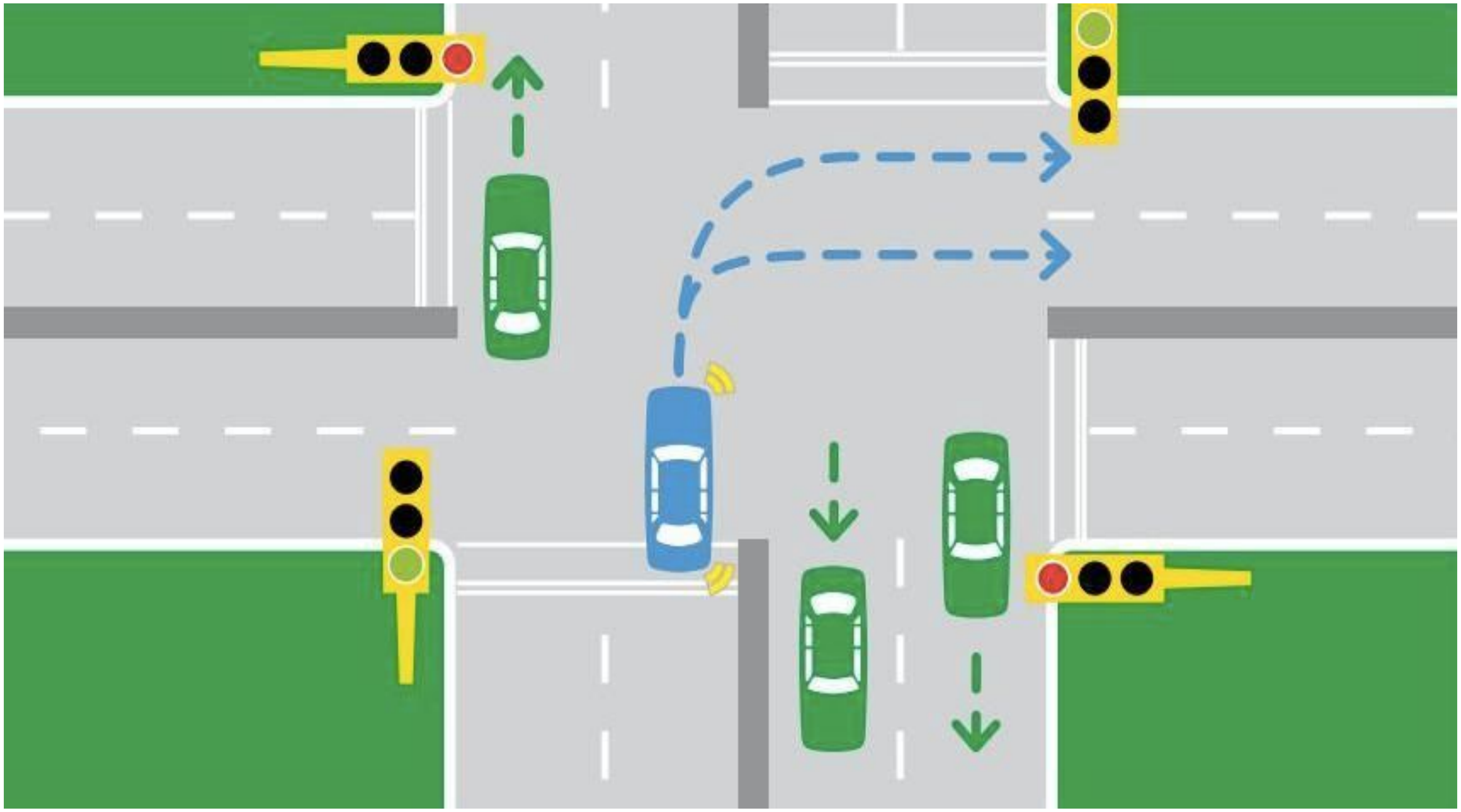
```
if (dialog.name == "profile_form")
  // ...
if (dialog.name == "login_form")
  // ...
```

Makes it difficult to understand, test, and reuse code :-(

Solution: Introduce a mediator to allow components to communicate indirectly



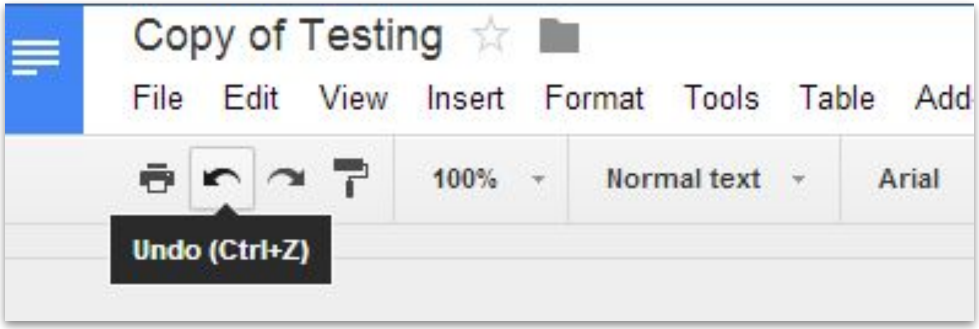
Analogy: Can you spot the mediator?



5. 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



I.e., add an "undo" ability to a text editor.

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

```
public class Editor {
    private String text;
    private Font font;
    private Position cursorPosition;
    …

    // a poor solution :-(
    public void setState(String text, Font font, ...) {
        this.text = text;
        ...
    }
}
```

Why is it a bad idea for a client to use setState to restore state?

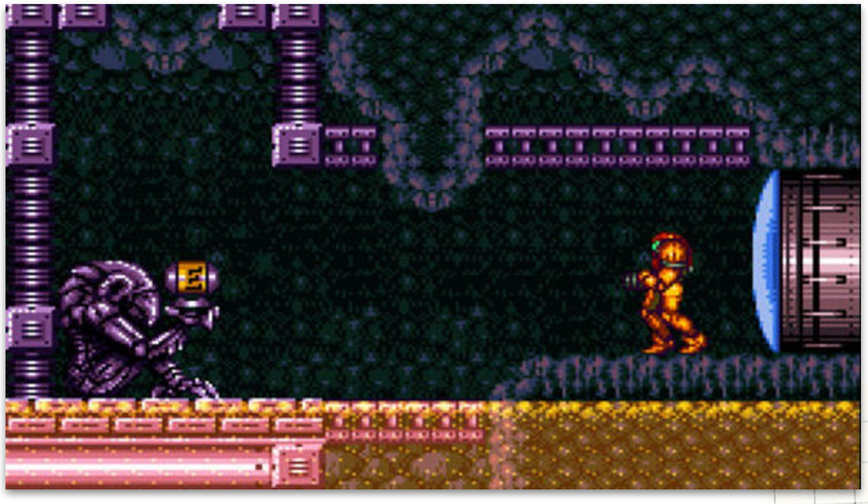
Problem: Without violating encapsulation, allow client of Editor to capture the object's state and restore later

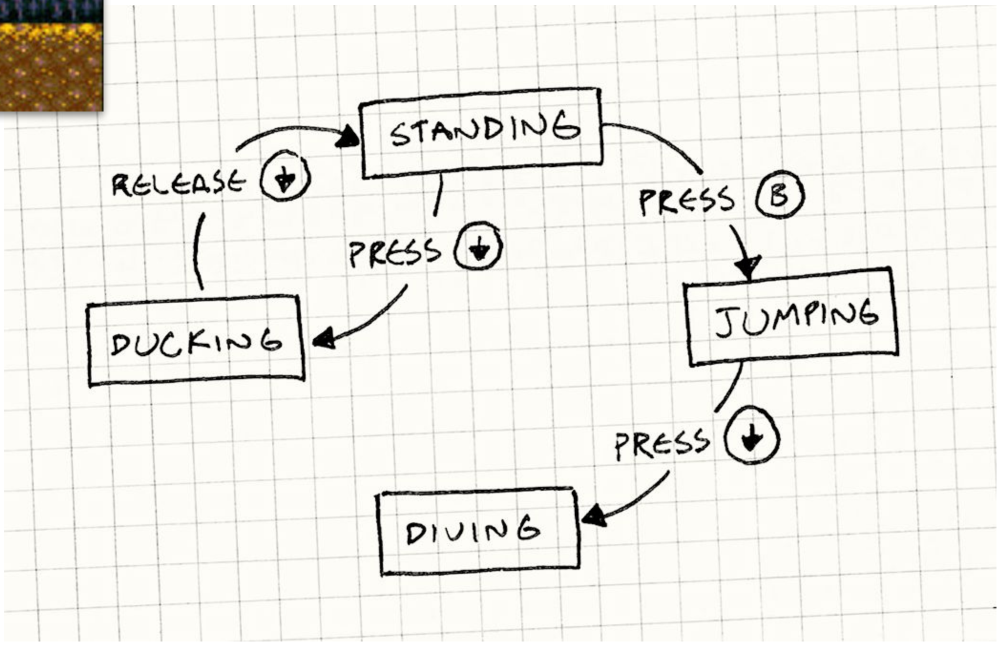
```
public class Editor {
    private String text;
    private Font font;
    private Position cursorPosition;
    …

    // a better solution :-)
    private class Memento {
        private String text;
        ...
    }
    public Memento save() {
        return new Memento(text, font, ...);
    }
    public void restore(Memento memento) {
        text = memento.text;
        ...
```

6. 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

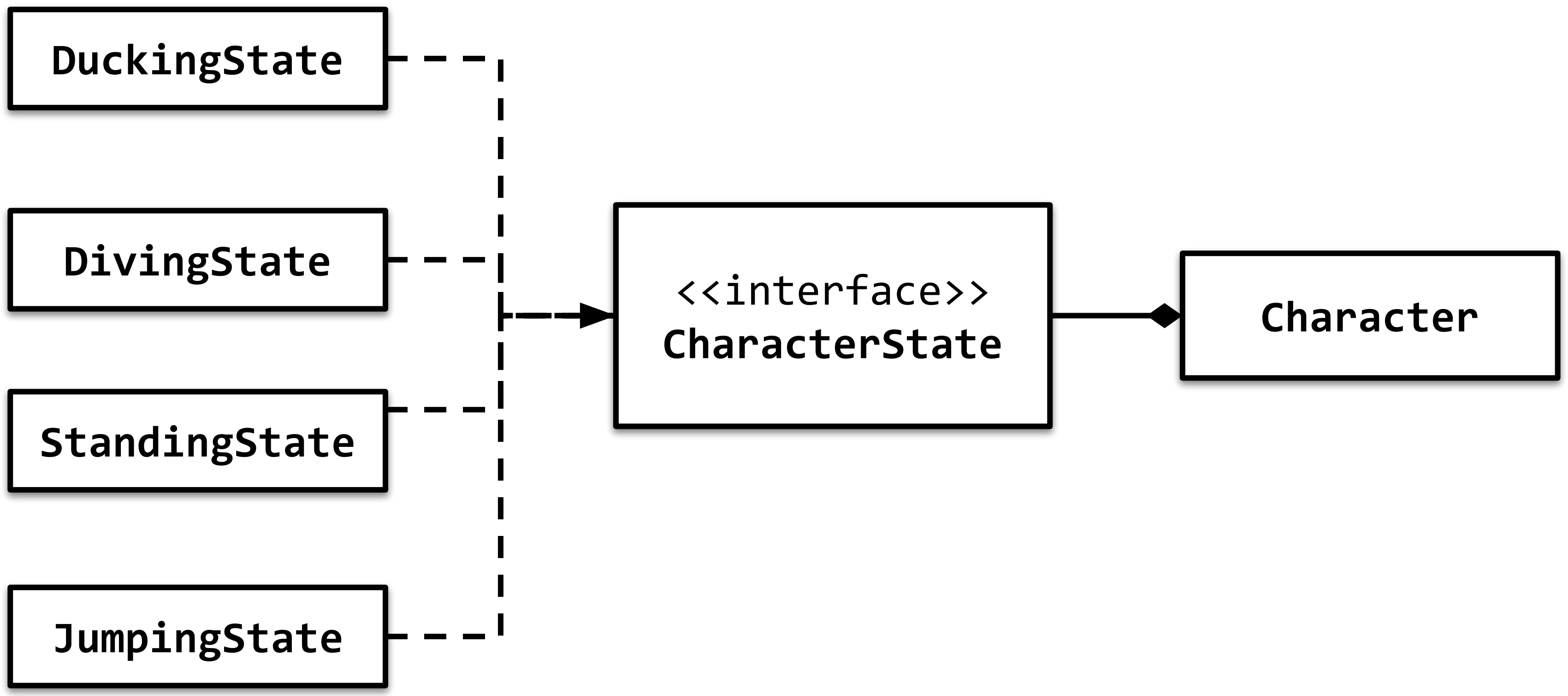
Problem: An object should behave differently based upon its internal state.





```
public class GameCharacter {
  …
  public void handleInput(Input input) {
    ...
  }
  …
}
```

Solution: Delegate behavior to a `State` object!



8. State

- Intent: allow an object to alter its behavior when internal state changes. "Object will appear to change class."
- Use case: TCP Connection, Game AI
- 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

Conclusion

- Now you know *most* of 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