Principles of Software Construction: Objects, Design, and Concurrency

A Java-centric tour of the Gang of Four design patterns

Josh Bloch **Charlie Garrod** Darya Melicher

Carnegie Mellon University
School of Computer Science

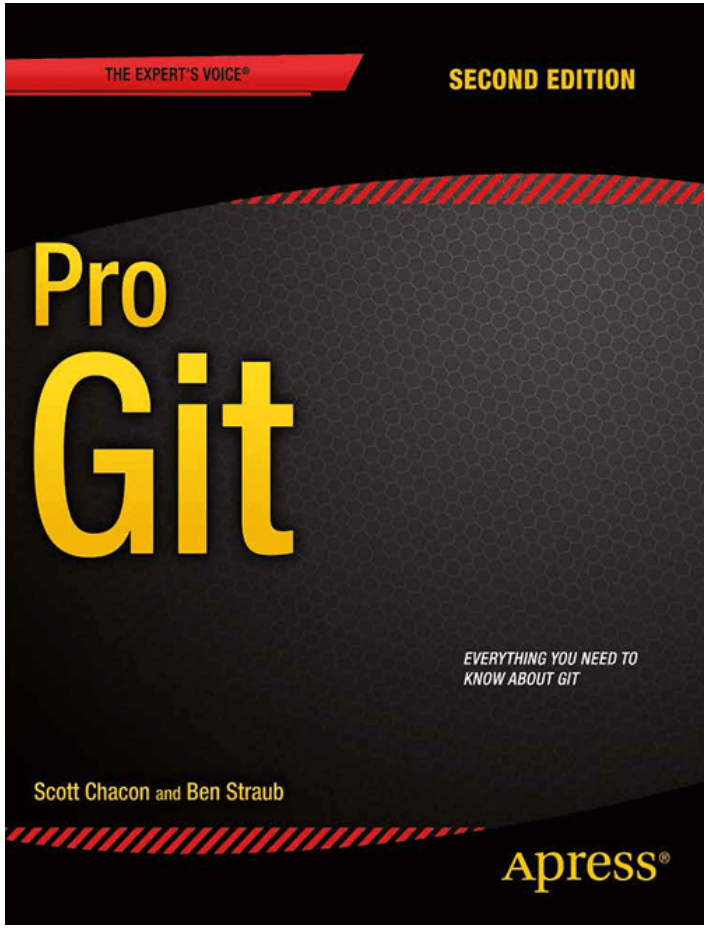
institute for SOFTWARE RESEARCH

Administrivia

- Homework 6 due tomorrow night
- Final exam review session:
  - Saturday, Dec. 15th noon – 2:30 p.m. Scaife 125
- Final exam:
  - Sunday, Dec 16th 5:30 – 8:30 p.m.: GHC 4401

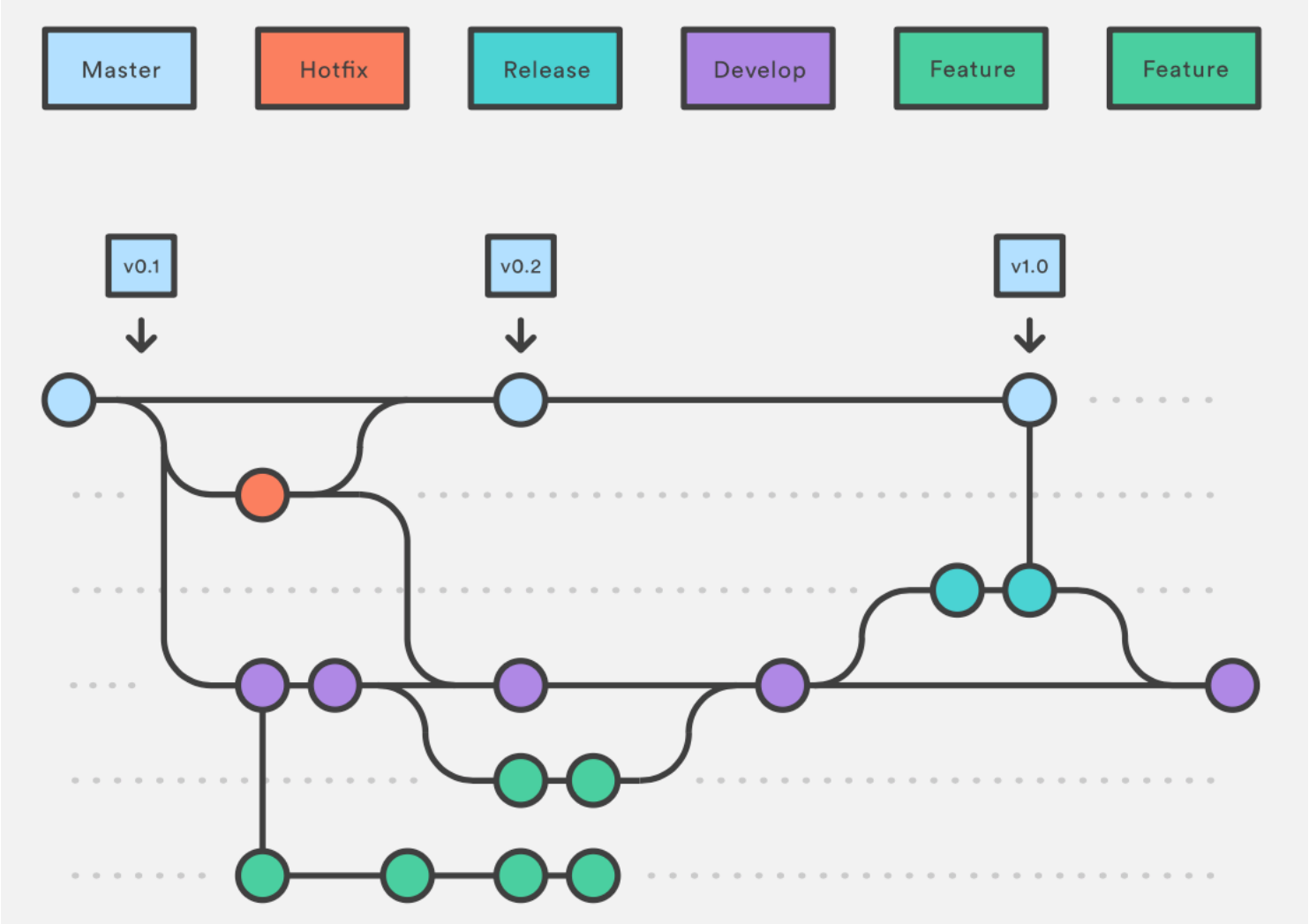
Key concepts from last Thursday

Highly recommended



https://git-scm.com/book/en/v2

Common workflows using Git

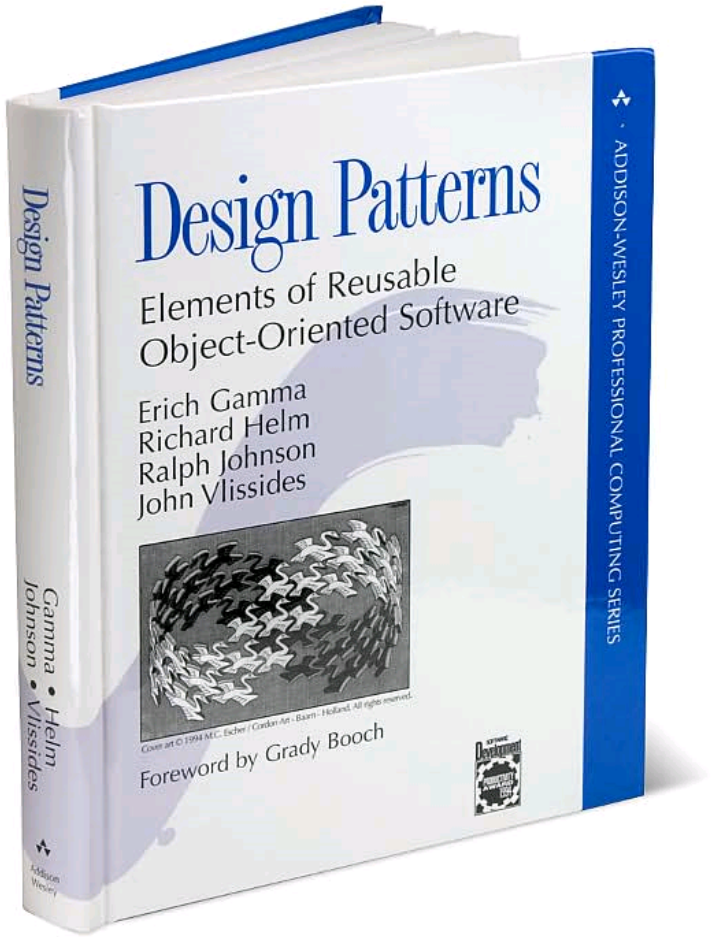


Managing a huge monorepo

- Automated testing…
- Lots of automation…
- Smart tooling...

Today: A Java-centric tour of the Gang of Four patterns

I. Creational Patterns
II. Structural Patterns
III. Behavioral Patterns



Common object-oriented design principles

- Program to an interface, not an implementation
- Favor composition over 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

Illustration

- **Code sample, diagram, or drawing**
  - Time constraints make it difficult to include illustrations for some patterns

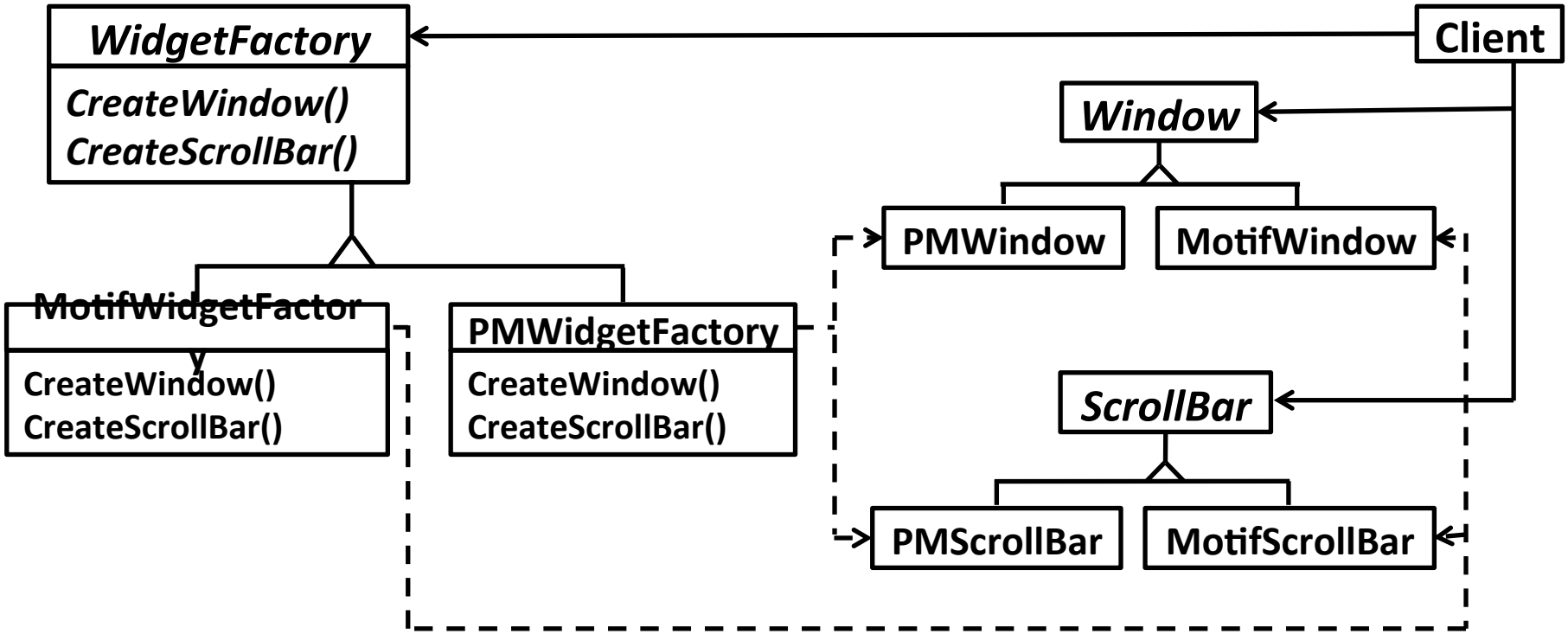
I. Creational Patterns

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

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

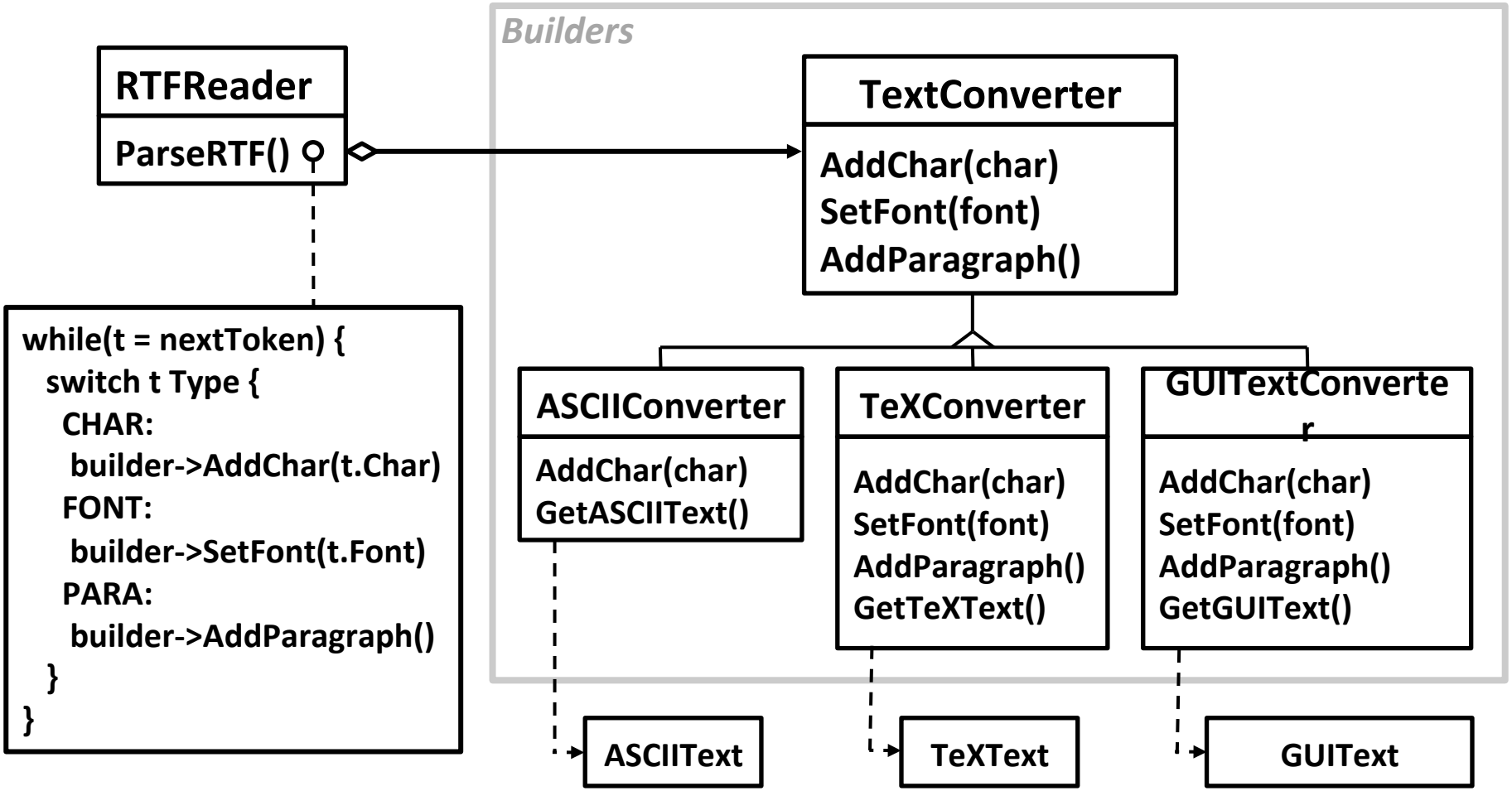
Abstract Factory Illustration



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
- types – *Builder,* ConcreteBuilders, Director, Products
- JDK – `StringBuilder`, `StringBuffer`*
  - But there is no (visible) abstract supertype…
  - And both generate same product class (`String`)

Gof4 Builder Illustration



My take on Builder [EJ Item 1]

- 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

EJ-style Builder Illustration

```
NutritionFacts twoLiterDietCoke = new NutritionFacts.Builder(
    "Diet Coke", 240, 8).sodium(1).build();

public class NutritionFacts {
    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) { ... }
}
```

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()`
- Related *Static Factory pattern* is very common
  - Technically not a GoF pattern, but close enough

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() { ... }
    ...
}

Collection<String> c = ...;

for (String s : c)  // Creates an Iterator appropriate to c
    System.out.println(s);
```

4. 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

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
- Key types – Singleton
- JDK – `java.lang.Runtime`

Singleton Illustration

```
public enum Elvis {
    ELVIS;

    sing(Song song) { ... }

    playGuitar(Riff riff) { ... }

    eat(Food food) { ... }

    take(Drug drug) { ... }
}

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

My take on Singleton

- It's an ***instance-controlled class***; others include
  - **Static utility class** – non-instantiable
  - **Enum** – one instance per value, all values known at compile time
  - **Interned class** – one canonical instance per value, new values created at runtime
- There is a duality between singleton and static utility class

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
- Use case – numerous, e.g., arrays vs. collections
- Key types – Target, Adaptee, Adapter
- JDK – `Arrays.asList(T[])`

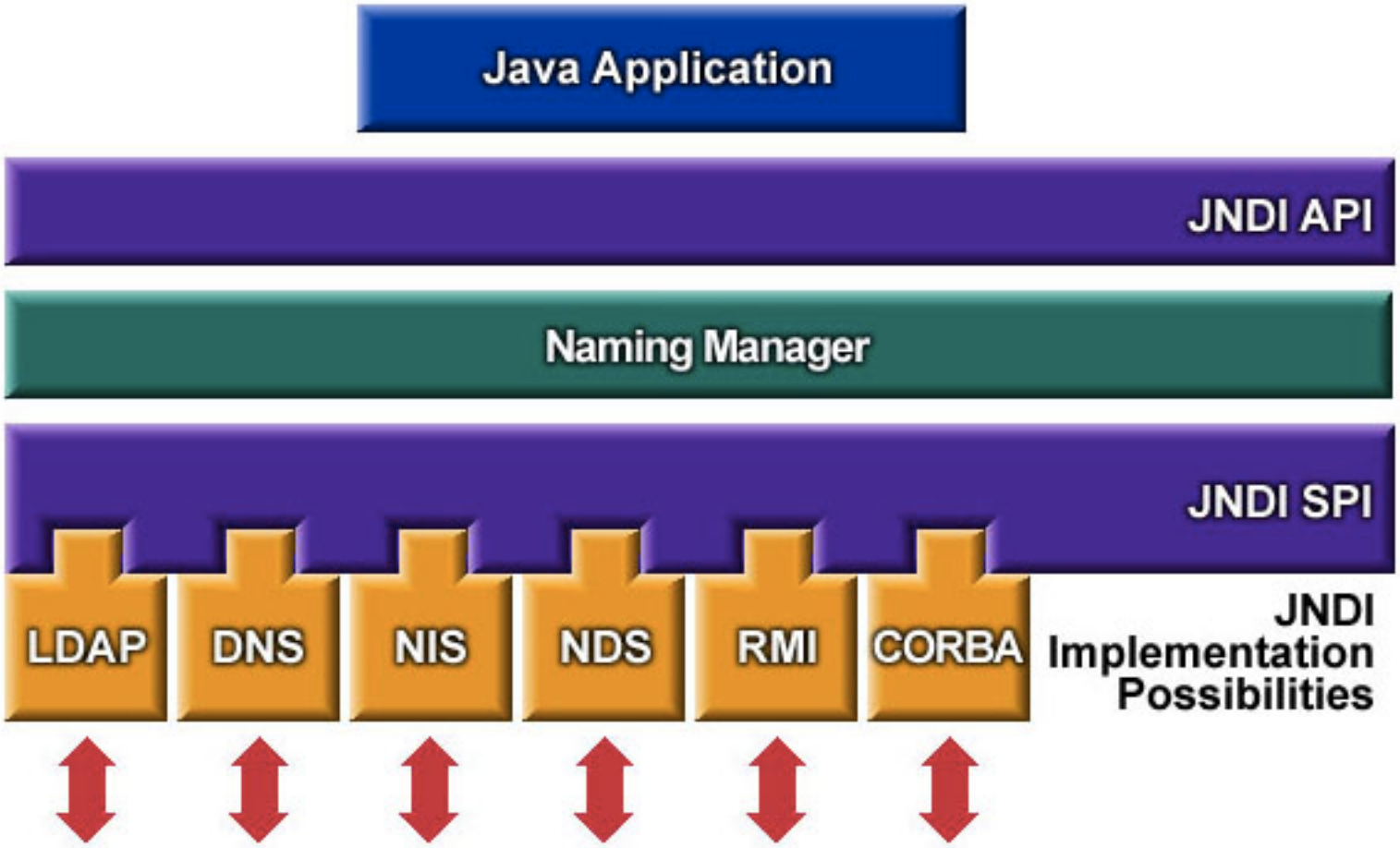
Adapter Illustration

Have this and this? Use this!

2. 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

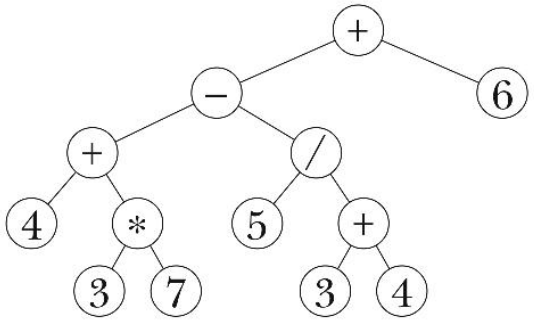
Bridge Illustration



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`

Composite Illustration

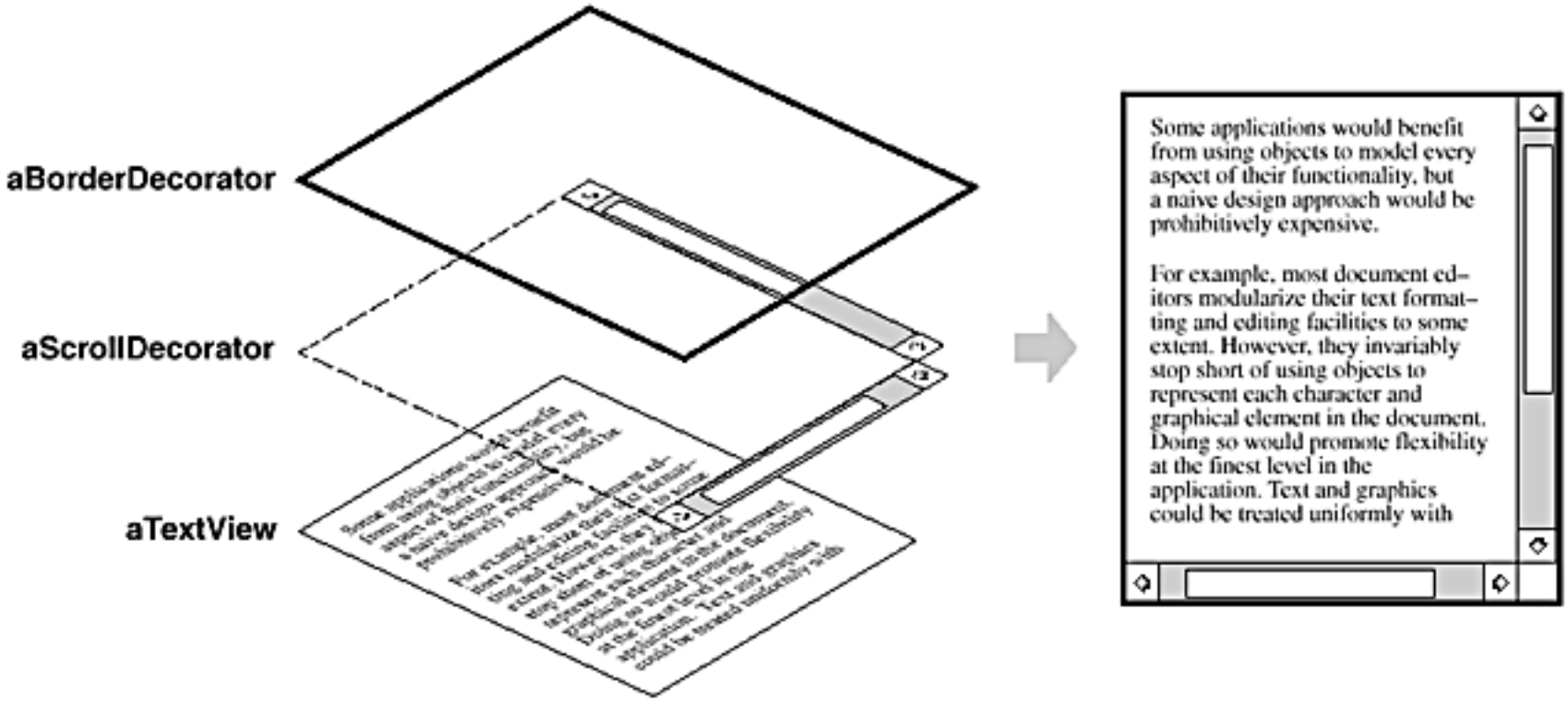



```
public interface Expression {
    double eval();     // Returns value
    String toString(); // Returns infix expression string
}

public class UnaryOperationExpression implements Expression {
    public UnaryOperationExpression(
            UnaryOperator operator, Expression operand);
}
public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
            Expression operand1, Expression operand2);
}
public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
```

4. 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

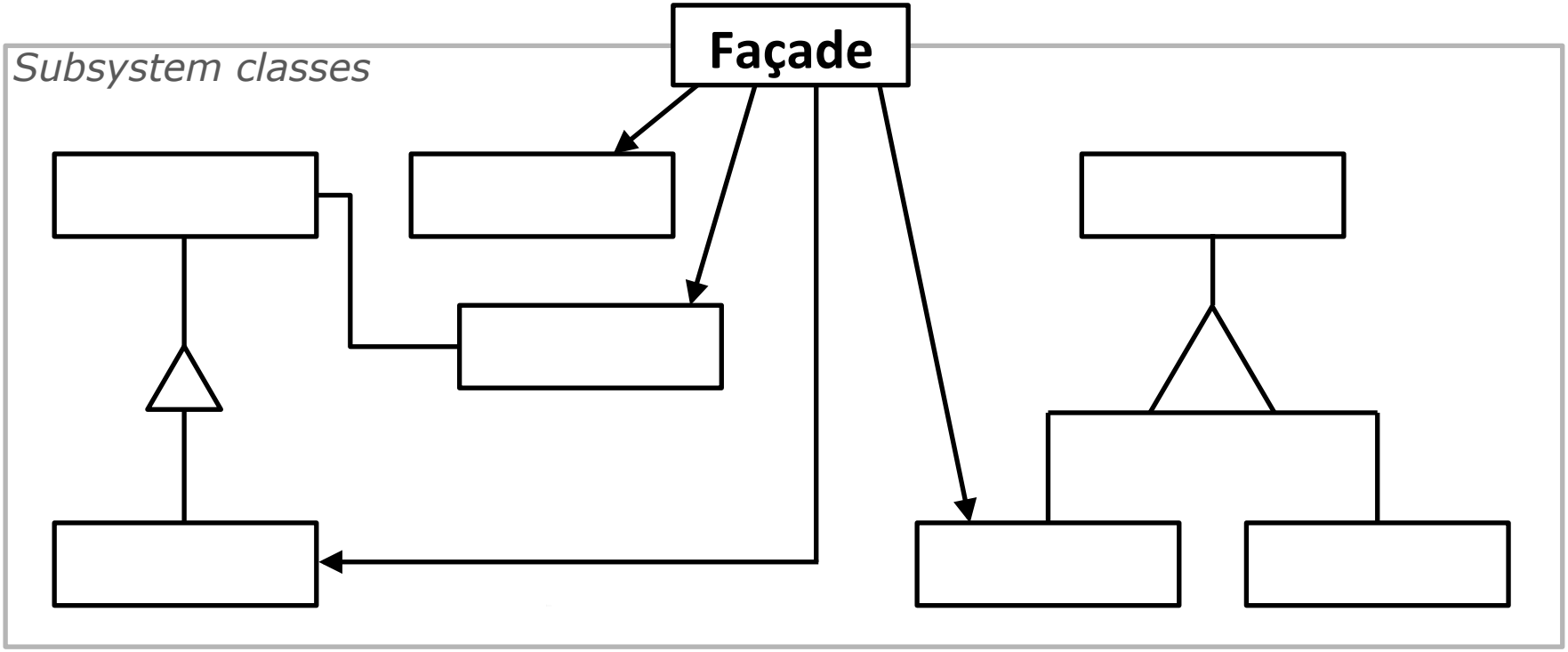
Decorator Illustration



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`

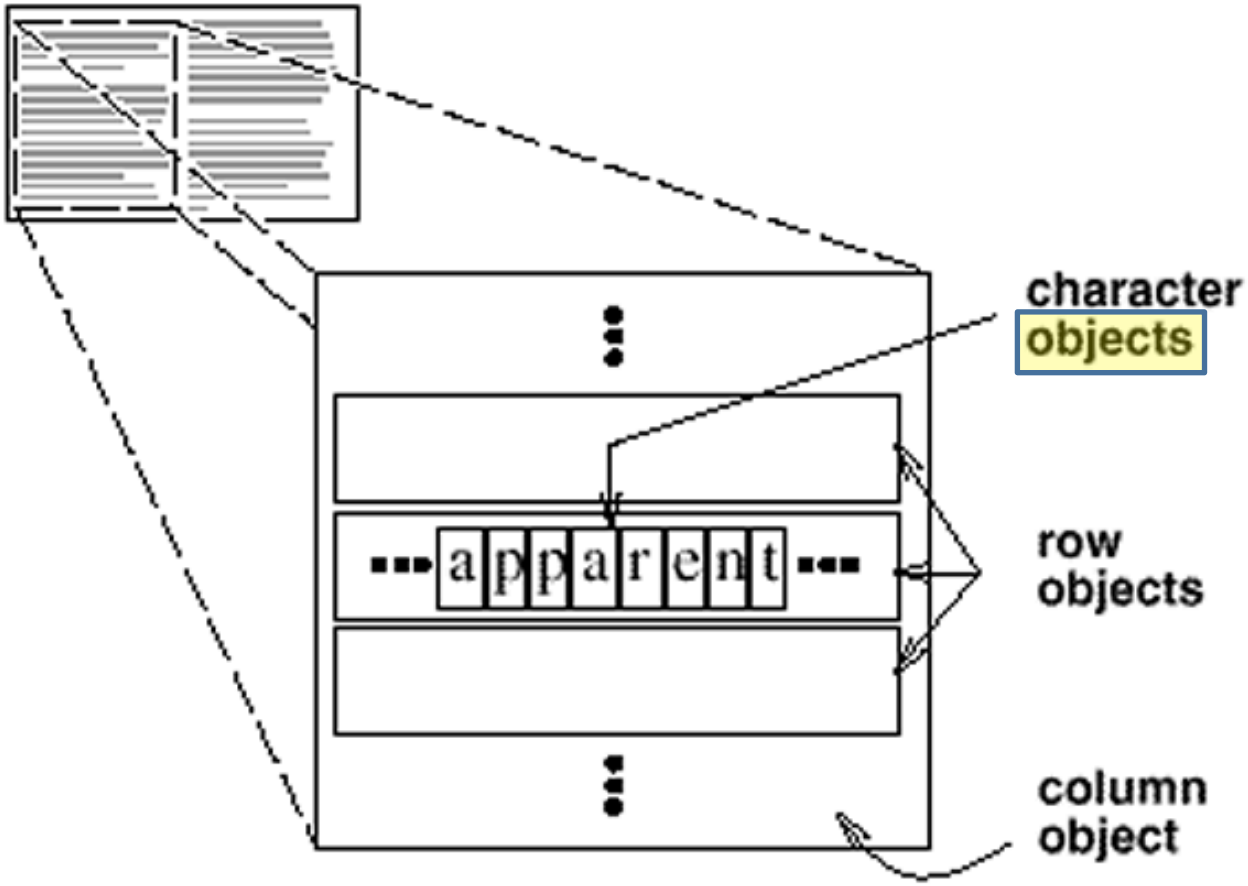
Façade Illustration



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

Flyweight Illustration

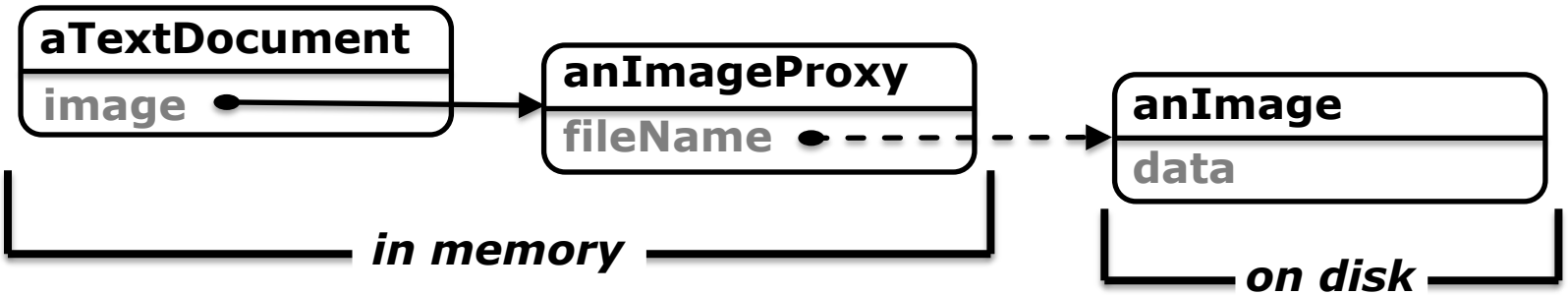


7. Proxy

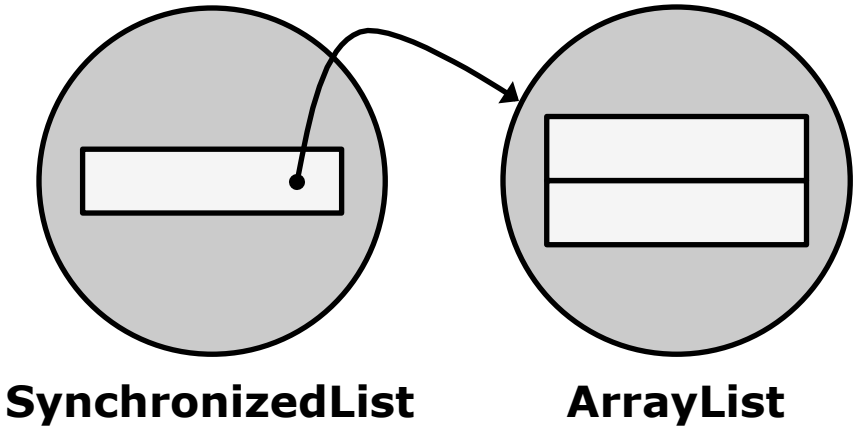
- Intent – surrogate for another object
- Use case – delay loading of images till needed
- Key types – *Subject*, Proxy, RealSubject
- Gof mention several flavors
  - virtual proxy – stand-in that instantiates lazily
  - remote proxy – local representative for remote obj
  - protection proxy – denies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

Proxy Illustrations

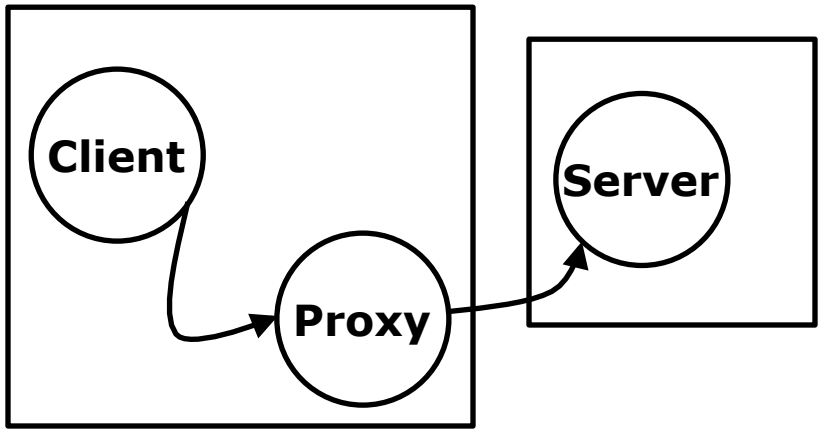
Virtual Proxy



Smart Reference



Remote 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

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

2. Command

- Intent – encapsulate a request as 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.
- Is it Command pattern if you run it repeatedly? If it takes an argument? Returns a val?

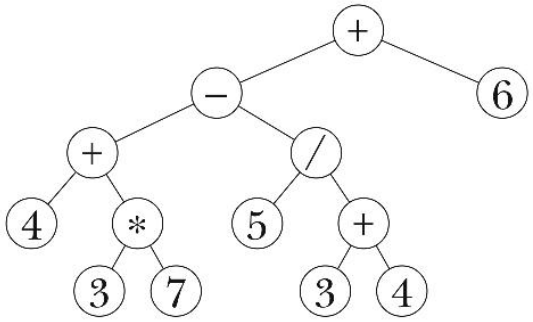
Command Illustration

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

3. Interpreter

- Intent – given a language, define class hierarchy for parse tree, recursive method to interpret it
- Use case – regular expression matching
- Key types – *Expression*, *NonterminalExpression*, *TerminalExpression*
- JDK – no uses I'm aware of
  - Our expression evaluator (midterm 2) is a classic example
- Necessarily uses Composite pattern!

Interpreter Illustration




```
public interface Expression {
    double eval();     // Returns value
    String toString(); // Returns infix expression string
}

public class UnaryOperationExpression implements Expression {
    public UnaryOperationExpression(
            UnaryOperator operator, Expression operand);
}
public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
            Expression operand1, Expression operand2);
}
public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
```

4. 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.

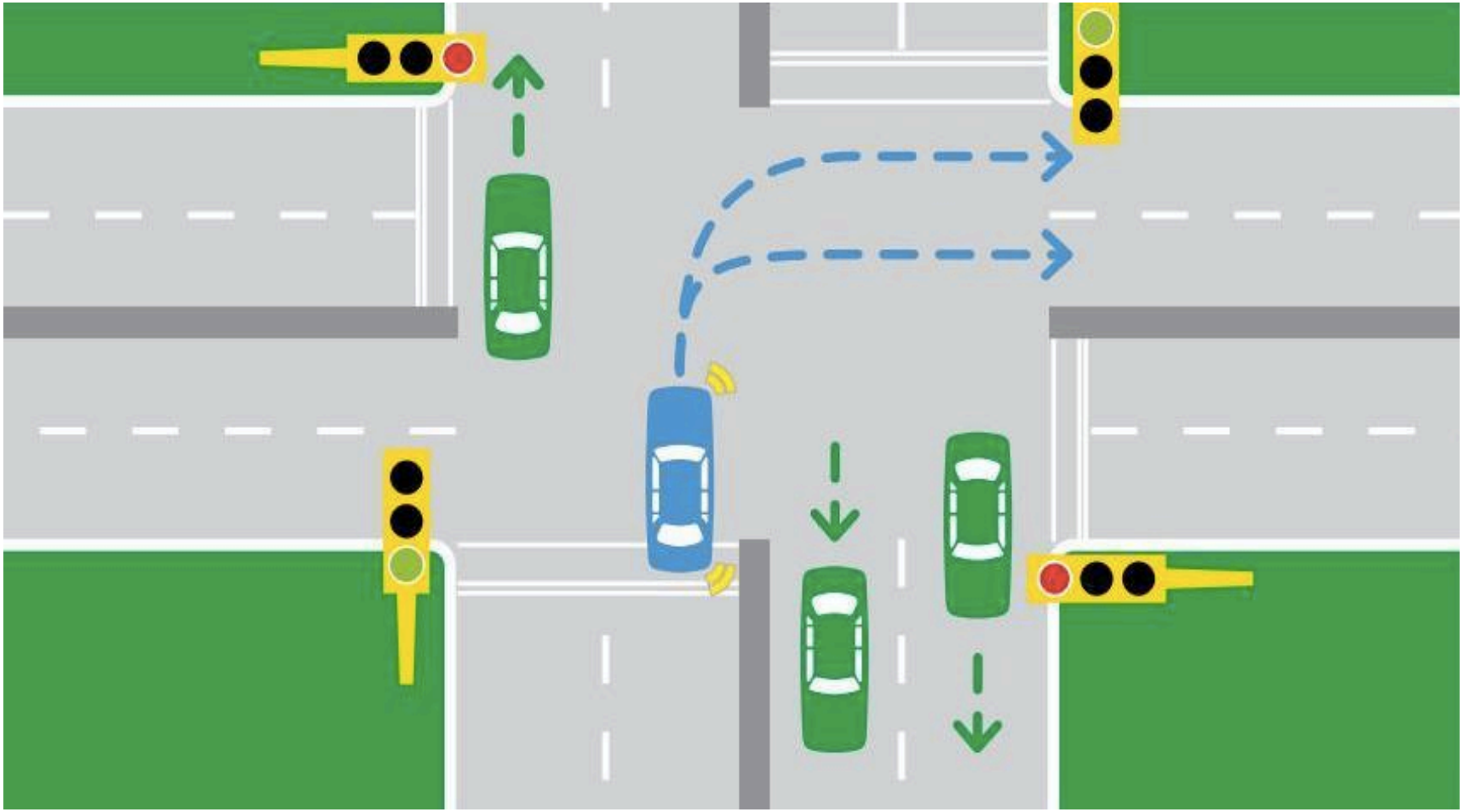
Iterator Illustration

```
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);
```

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

Mediator Illustration



6. Memento

- Intent – without violating encapsulation, allow client to capture an object's state, and restore
- Use case – undo stack for operations that aren't easily undone, e.g., line-art editor
- Key type – Memento (opaque state object)
- JDK – none that I'm aware of (*not* serialization)

7. 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

Observer Illustration

```
// Implement roll button and dice type field
JTextField diceSpecField = new JTextField(diceSpec, 5); // Field width
JButton rollButton = new JButton("Roll");
rollButton.addActionListener(event -> {
if (!diceSpecField.getText().equals(diceSpec)) {
      diceSpec = diceSpecField.getText();
      dice = Die.dice(diceSpec);
      jDice.resetDice(dice);
  }
  for (Die d : dice) d.roll();
  jDice.repaint();
});
```

8. State

- Intent – allow an object to alter its behavior when internal state changes. “Object will appear to change class.”
- Use case – TCP Connection (which is stateful)
- Key type – *State* (Object delegates to state!)
- JDK – none that I’m aware of, but…
  - Works *great* in Java
  - Use enums as states
  - Use `AtomicReference<State>` to store it

9. 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`

Strategy Illustration

`Comparator` is a strategy for ordering

```
public static synchronized void main(String[] args) {
    Arrays.sort(args, Comparator.reverseOrder());
    System.out.println(Arrays.toString(args));

    Arrays.sort(args, Comparator.comparingInt(String::length));
    System.out.println(Arrays.toString(args));
}

java Foo i eat wondrous spam
[wondrous, spam, i, eat]
[i, eat, spam, wondrous]
```

10. 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`)

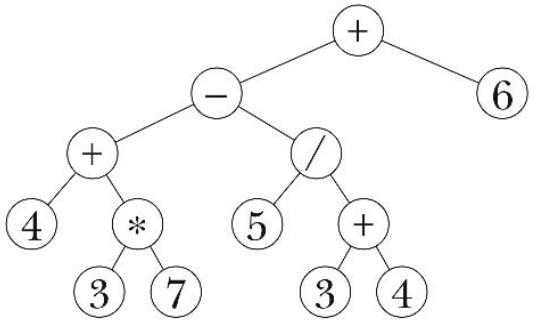
Template Method Illustration

```
// List adapter for primitive int arrays
public static List<Integer> intArrayList(final int[] a) {
    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i];
        }

        public Integer set(int i, Integer val) {
            Integer oldVal = a[i];
            a[i] = val;
            return oldVal;
        }

        public int size() {
            return a.length;
        }
    };
}
```

11. Visitor

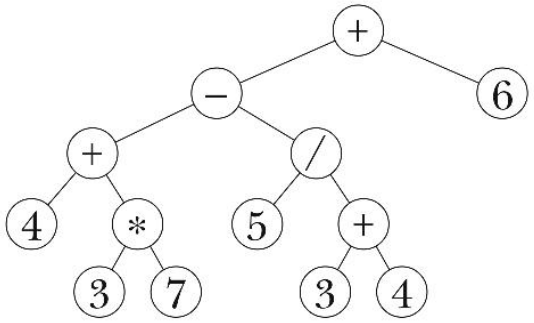
- Intent – represent an operation to be performed on elements of an object structure (e.g., a parse tree). Visitor lets you define a new operation without modifying the type hierarchy.
- Use case – type-checking, pretty-printing, etc.
- Key types – *Visitor*, ConcreteVisitors, all the element types that get visited
- JDK – none that I'm aware of

Visitor Illustration (1/3)



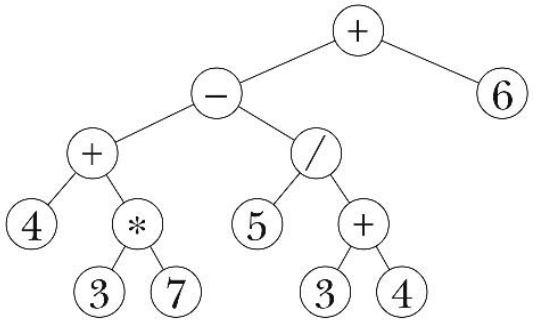

```
public interface Expression {
    public <R> R accept(Visitor<R> v);  // No eval or toString!
}

public class UnaryOperationExpression implements Expression {
    public UnaryOperationExpression(
            UnaryOperator operator, Expression operand);
    public <R> R accept(Visitor<R> v) { return v.visitUnaryExpr(this); }
}
public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
            Expression operand1, Expression operand2);
    public <R> R accept(Visitor<R> v) {  return v.visitBinaryExpr(this) ; }
}
public class NumberExpression implements Expression {
    public NumberExpression(double number);
    public <R> R accept(Visitor<R> v) { return v.visitNumberExpr(this); }
}
```

Visitor Illustration (2/3)




```
public interface Visitor<R>  { // R is result type
    public R visitUnaryExpr(UnaryExpression ue);
    public R visitBinaryExpr(BinaryExpression be);
    public R visitNumberExpr(NumberExpression ne);
}

public class EvalVisitor implements Visitor<Double> {
    public Double visitUnaryExpr(UnaryExpression ue) {
        return ue.operator.apply(ue.operand.accept(this));
    }
    public Double visitBinaryExpr(BinaryExpression be) {
        return be.operator.apply(be.operand1.accept(this),
                                 be.operand2.accept(this));
    }
    public Double visitNumberExpr(NumberExpression ne) { return ne.number; }
}
```

Visitor Illustration (3/3)




```
public class ToStringVisitor implements Visitor<String> {
    public String visitUnaryExpr(UnaryExpression ue) {
        return ue.operator + ue.operand.accept(this);
    }
    public String visitBinaryExpr(BinaryExpression be) {
        return String.format("(%s %s %s)", be.operand1.accept(this),
                             be.operator, be.operand2.accept(this));
    }
    public String visitNumberExpr(NumberExpression ne) {
        return Double.toString(ne.number);
    }
}

// Sample use of visitors
System.out.println(e.accept(new ToStringVisitor()) + " = " +
                   e.accept(new EvalVisitor()));
```

More on Visitor

- Visitor is NOT merely traversing a graph structure and applying a method
  - That's Iterator!
- The essence of visitor is *double-dispatch*
  - First dynamically dispatch on the Visitor
  - Then on the element being visited

Summary

- Now you know *almost 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