Principles of Software Construction: Objects, Design, and Concurrency

23 Patterns in 80 Minutes: a Whirlwind Java-centric Tour of the Gang-of-Four Design Patterns

Josh Bloch Charlie Garrod

Carnegie Mellon University
School of Computer Science

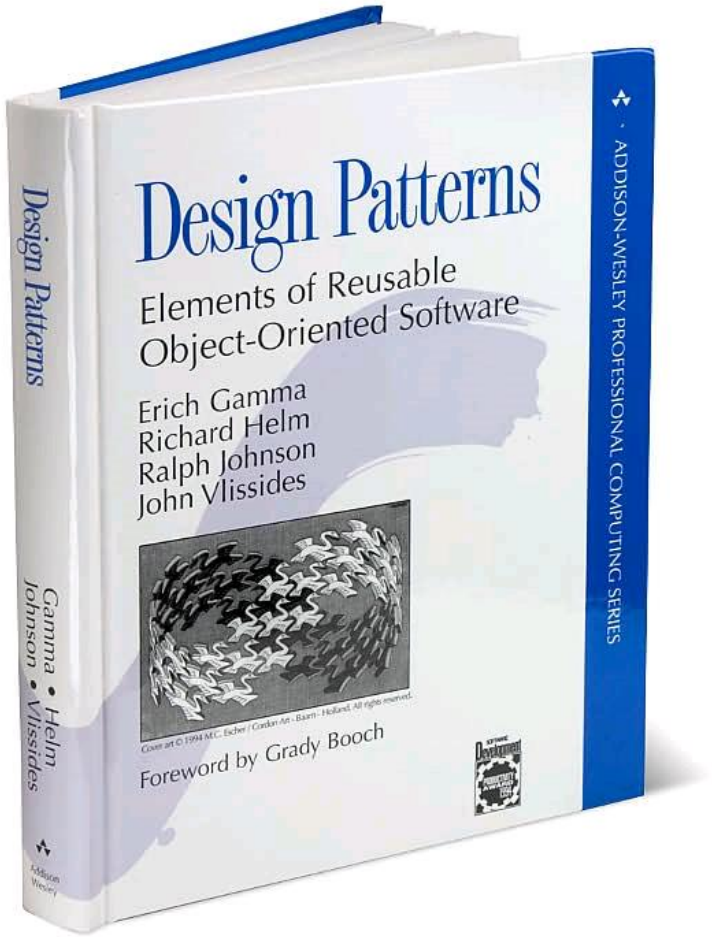
institute for SOFTWARE RESEARCH

Administrivia

- Homework 6 due tomorrow (Wednesday) 11:59 pm EST
- Final exam review session Wednesday, May 12th, 7-9 p.m. EDT
  - Practice problems coming out later this week
- Final exam
  - Released Thursday, May 13th (evening EDT)
  - Due Friday, May 14th, 11:59 p.m. EDT
  - Designed to take 3 hrs.
  - Open book, open notes, open Internet
  - Closed person, no interaction with others about the exam

Outline

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



Pattern Name

- **Intent** – the aim of this pattern
- **Use case** – a motivating example
- **Types** – the key types that define pattern
  - *Italic* type name indicates an abstract class; typically this is an interface type 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 impossible to include illustrations from some patterns

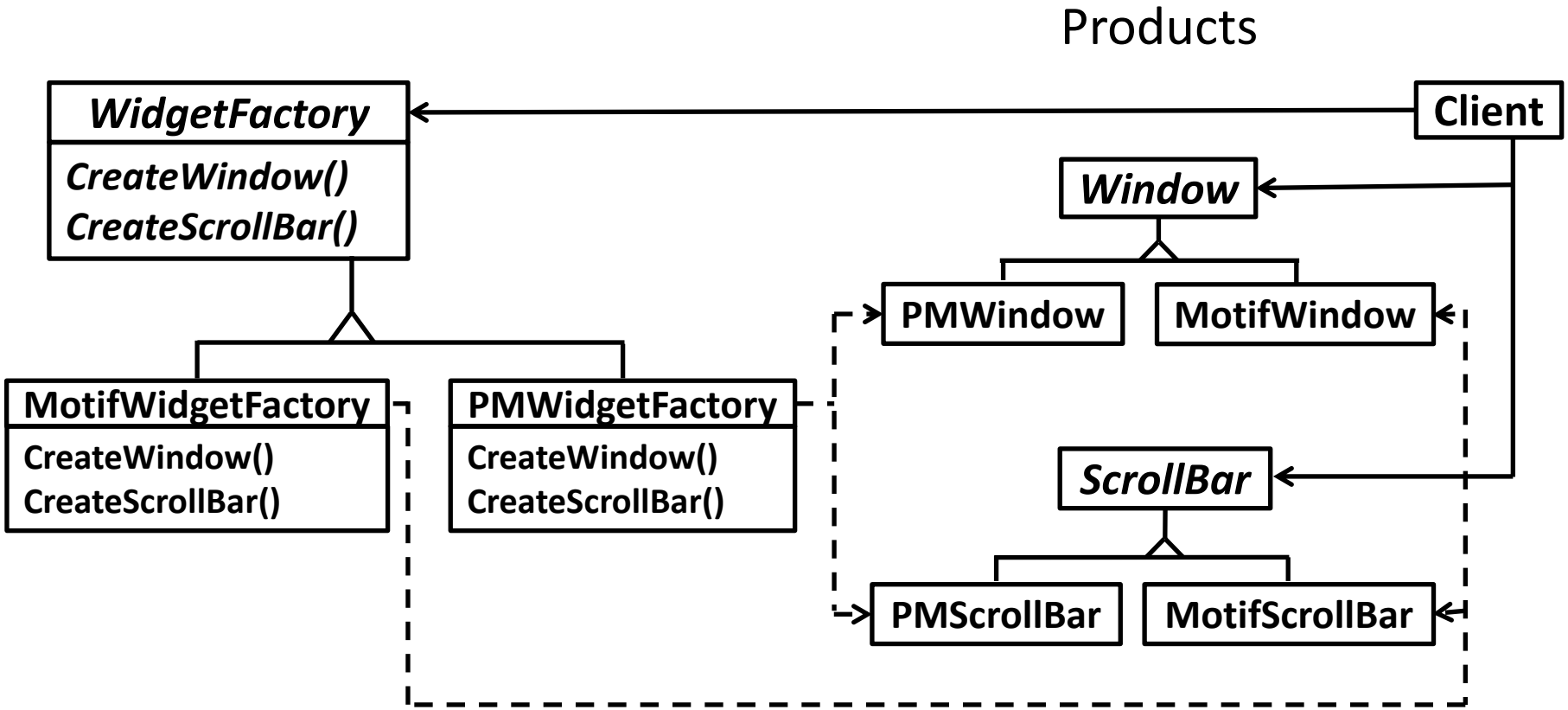
I. Creational Patterns

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 look-and-feel has its own windows, scrollbars, etc.
- Types – *AbstractFactory* with methods to create each family member; *AbstractProducts,* the family members themselves; (ConcreteFactories and ConcreteProducts)
- JDK – not common

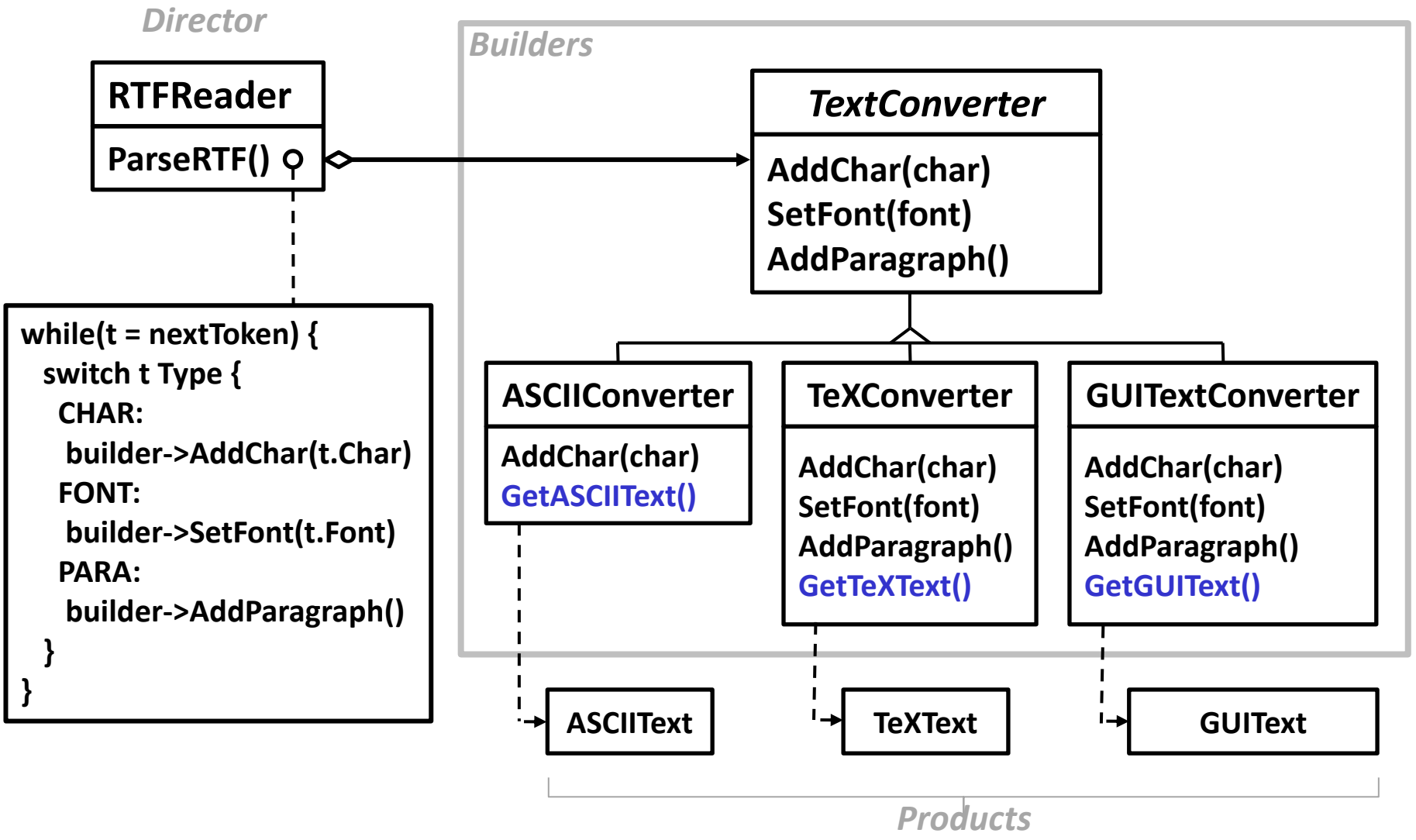
GoF Abstract Factory Illustration



2. Builder

- Intent – separate construction of a complex object from its representation so the same creation process can create different representations
- Use case – converting *rich text* to various formats
- Types – *Builder,* ConcreteBuilders, Director, Products
- JDK – `StringBuilder`, `StringBuffer` (sorta)
  - But there is no (visible) abstract supertype…
  - And both generate same product class (`String`)

GoF Builder Illustration



My take on Builder [*Effective Java* Item 2]

- Emulates *named optional parameters* in languages that don't support them
- Emulates 2^n constructors or factories with only n builder methods
- *Emulates variable arity parameter lists* (varargs) in languages that don't support them, and provides better type-safety in languages that do
- Cost is an intermediate (Builder) object
- Not the same as GoF pattern, but related

```
Pizza largeHawaiian =
    new Pizza.Builder(LARGE).add(HAM).add(PINEAPPLE).build();
```

3. Factory Method

- Intent – abstract creational method that lets subclasses decide which class to instantiate
- Use case – creating documents in a framework
- Types – *Creator*, contains abstract method to create an instance
- JDK – `SortedMap.subMap(K fromKey, K toKey)`
  - `TreeMap`, `SkipListConcurrentMap` return different implementations
- Related *Static Factory pattern* is *very* common in Java
  - Technically *not* a GoF pattern, but close enough

Factory Method Illustration

```
public interface SortedMap<K,V> {
    ...
    SortedMap<K,V> subMap(K fromKey, K toKey);
}

public class TreeMap<K,V> implements SortedMap<K,V> {
     ...
     SortedMap<K,V> subMap(K fromKey, K toKey) { ... }
     ...
}

public class ConcurrentSkipListMap<K,V> implements SortedMap<K,V> {
     SortedMap<K,V> subMap(K fromKey, K toKey) { ... }
     ...
}

SortedMap<K,V> dictionary = ...;

SortedMap<K,V> firstHalf = dictionary.submap("A", "M");
```

4. Prototype

- Intent – create an object by cloning another and tweaking
- Use case – writing music score editor in graphical editor framework
- Types – *Prototype*
- JDK – Cloneable, but **avoid** (except on arrays)
  - Java and Prototype pattern are a poor fit
  - Or maybe I just don't like the pattern, because it only works for mutable types

5. Singleton

- Intent – ensure that 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, the EmptySet
- 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 is required by another class, allowing interoperability
- Use case – numerous, e.g., arrays vs. collections
- Types – Target (what you need), Adaptee (what you have), Adapter (class that implements Target atop Adaptee)
- JDK – `Arrays.asList(T[])`

Adapter Illustration

Have this

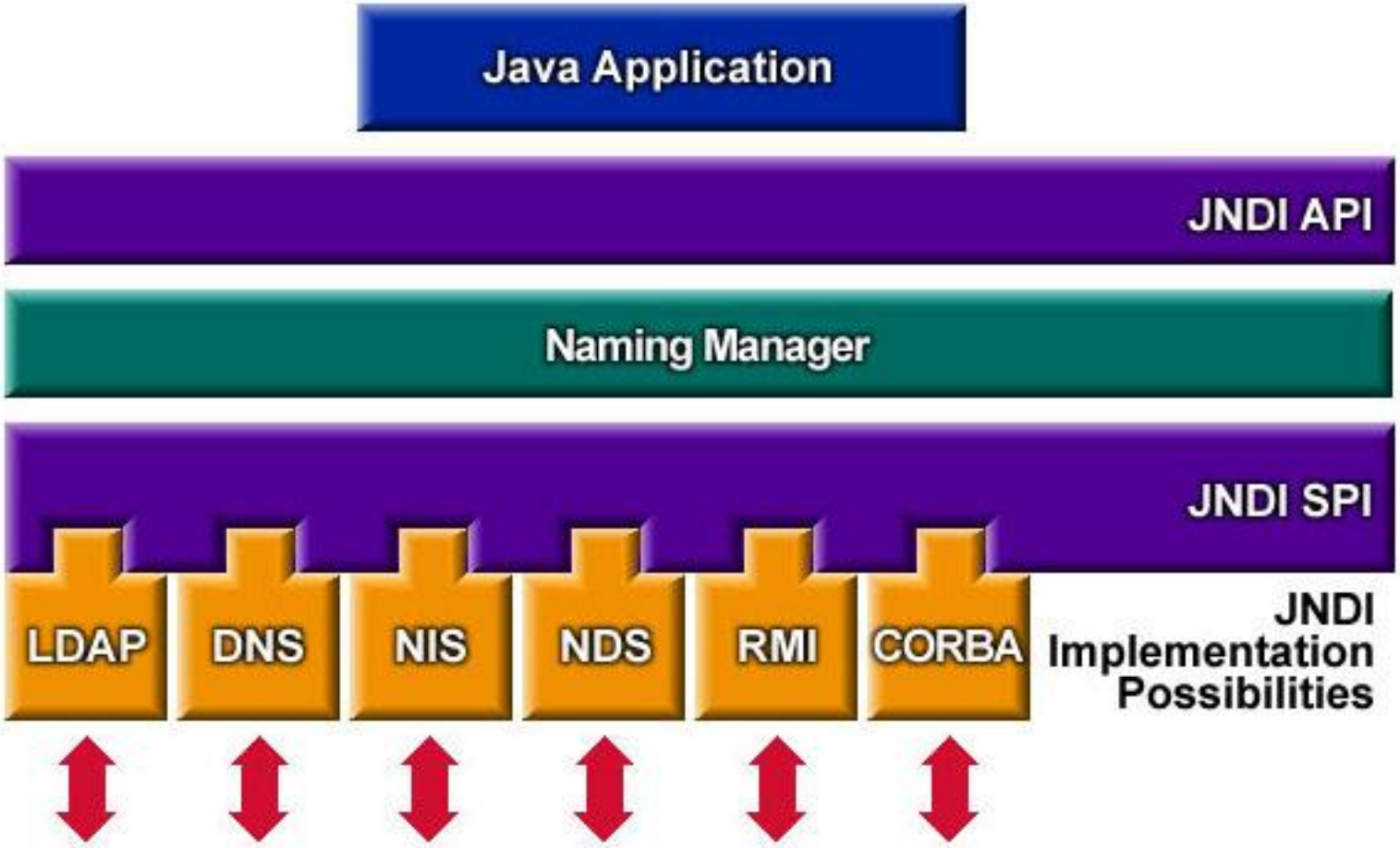
and this?

Use this!

2. Bridge

- Intent – decouple an abstraction from its implementation so they can vary independently
- Use case – portable windowing toolkit
- Types – Abstraction, *Implementor*
- JDK – Java Database Connectivity (JDBC), Java Cryptography Extension (JCE), Java Naming & Directory Interface (JNDI)
- Bridge pattern *very* similar to *Service Provider* (not a GoF pattern)
  - Abstraction ~ API, *Implementer* ~ SPI

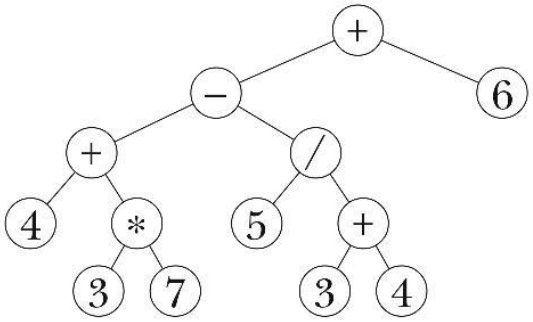
Bridge Illustration



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

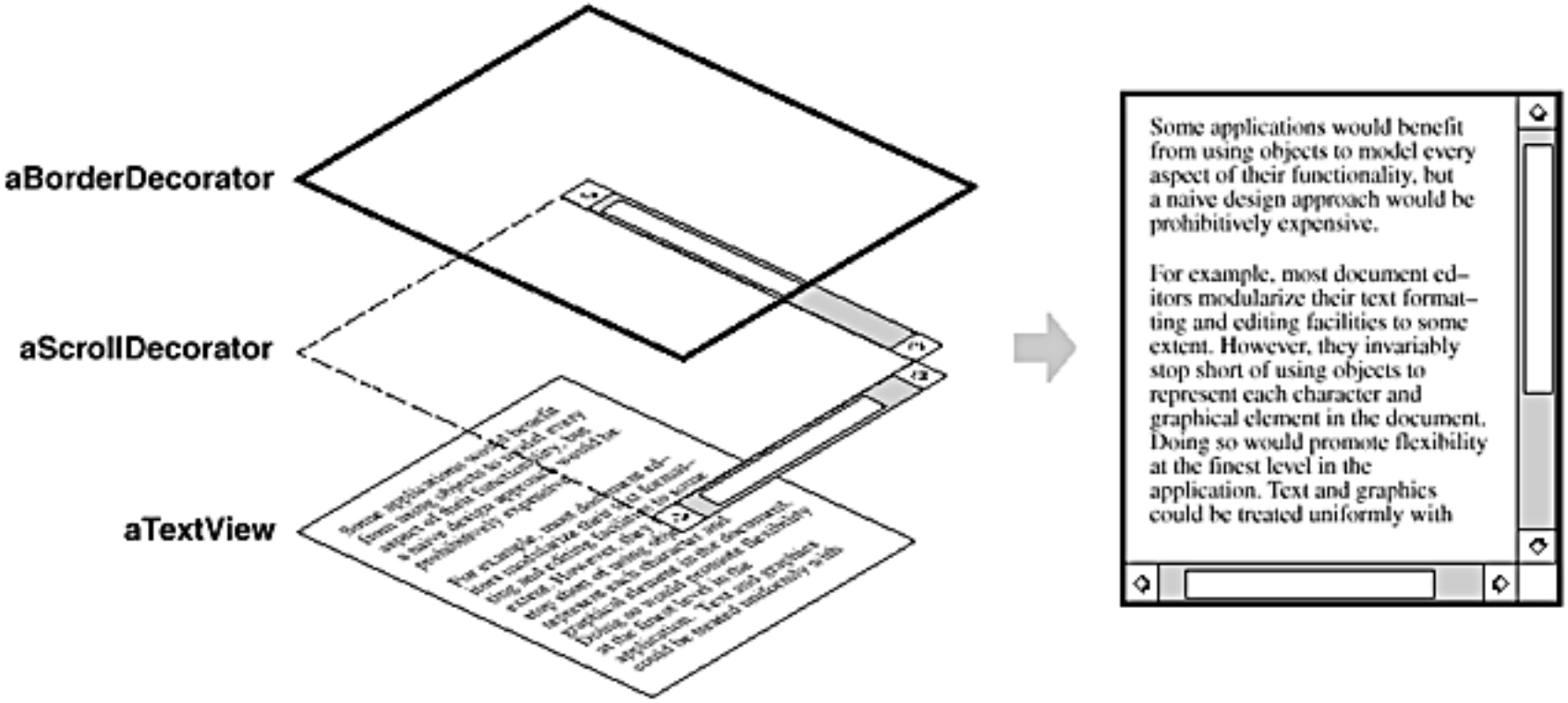



```
public interface Expression {
    double eval();      // Returns value
    String toString(); // Returns infix expression string
}

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
- Types – *Component*, implemented by decorator and decorated
- JDK – Collections (e.g., `Unmodifiable` wrappers), `java.io` streams, Swing components

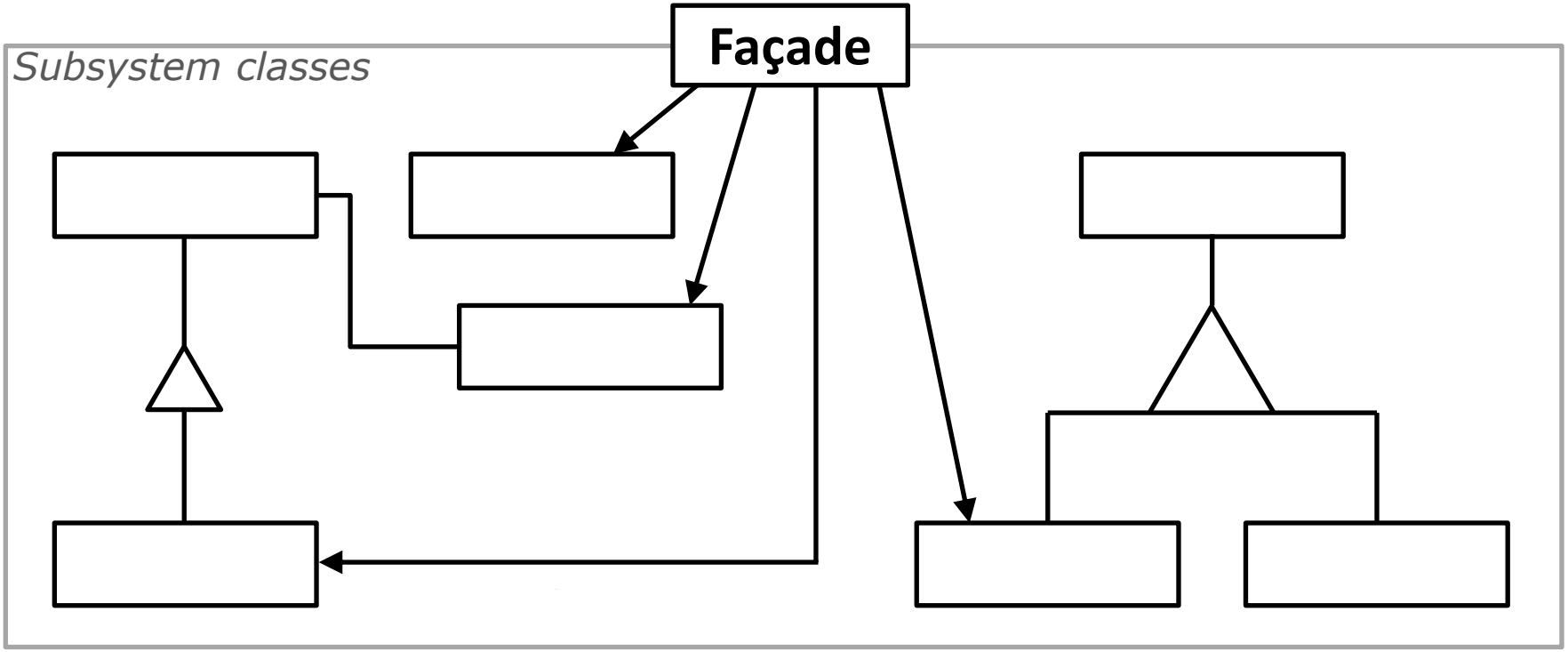
GoF Decorator Illustration



5. Façade

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

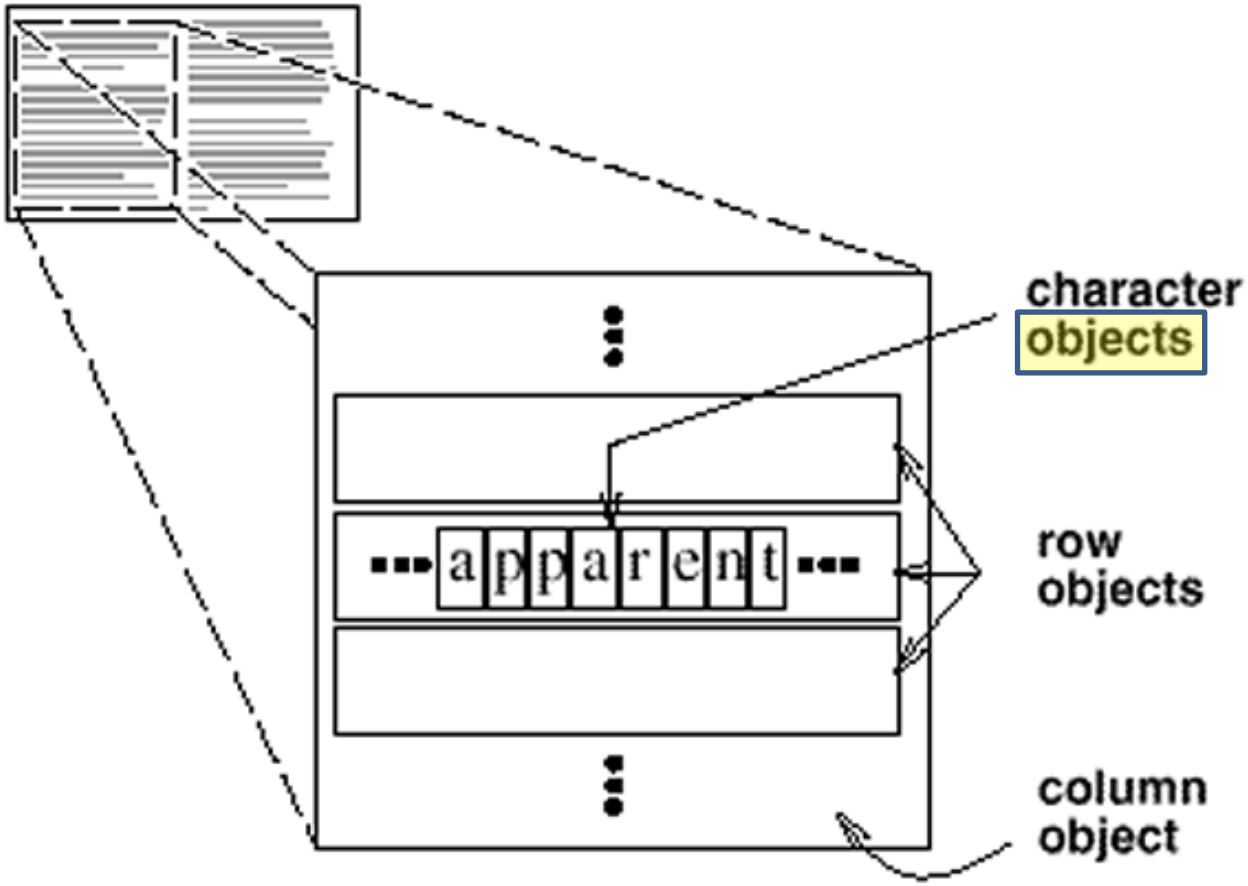
Façade Illustration



6. Flyweight

- Intent – use sharing to support large numbers of fine-grained immutable objects efficiently
- Use case – characters in a document
- 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 `long` number-of-units as extrinsic state

GoF Flyweight Illustration

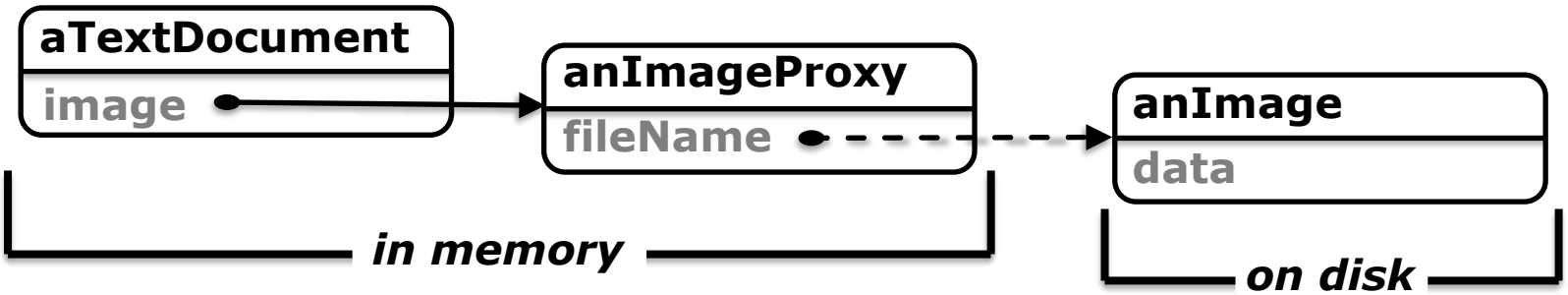


7. Proxy

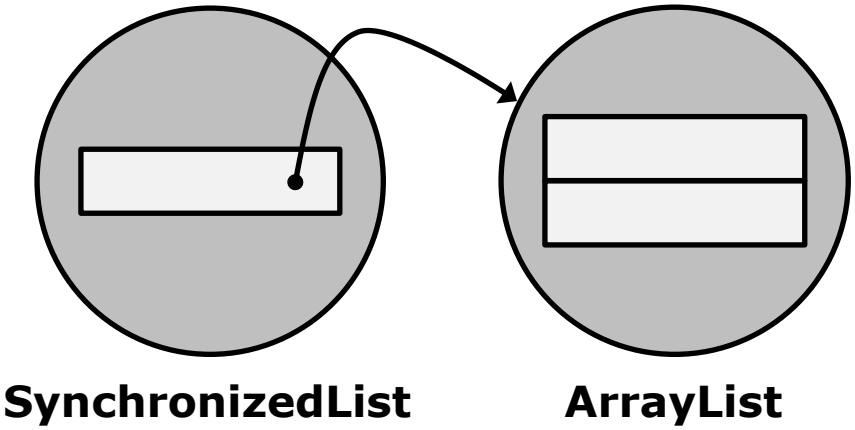
- Intent – Use one object as a surrogate for another object
- Use case – delay loading of images till needed
- Types – *Subject*, Proxy, RealSubject
- Gof mention several flavors
  - virtual proxy – stand-in that instantiates lazily
  - remote proxy – local representative for remote obj
  - protection proxy – denies some operations to some users
  - smart reference – does locking or reference counting atop real subject
- JDK – RMI (remote proxy), collection wrappers (`Unmodifiable` is protection proxy, `Synchronized` is smart reference)

Proxy Illustrations

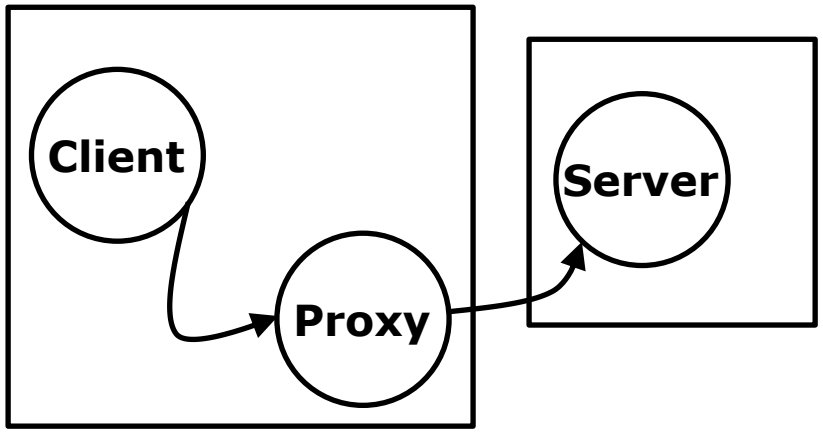
Virtual Proxy



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
- Types – *RequestHandler*
- JDK – `ClassLoader`, `Properties` (FWIW)
- 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* (in Java, `Runnable`)
- JDK – Common! Executor framework, etc.
- Is it Command pattern if you run the command repeatedly? If it takes an argument? Returns a val? GoF are vague on this.

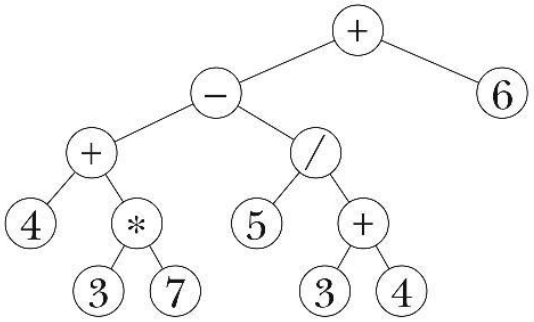
Command Illustration

```
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(s) to interpret it
- Use case – regular expression matching
- Types – *Expression*, *NonterminalExpression*, *TerminalExpression*
- JDK – no exported uses I'm aware of
  - Our cryptarithm expression evaluator (HW3) 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
- Types – *Iterable*, *Iterator*
  - But GoF discuss internal iteration, too
- JDK – collections, for-each statement, etc.
- Note that the Iterator pattern uses the Factory Method pattern: the `iterator()` method is a factory method

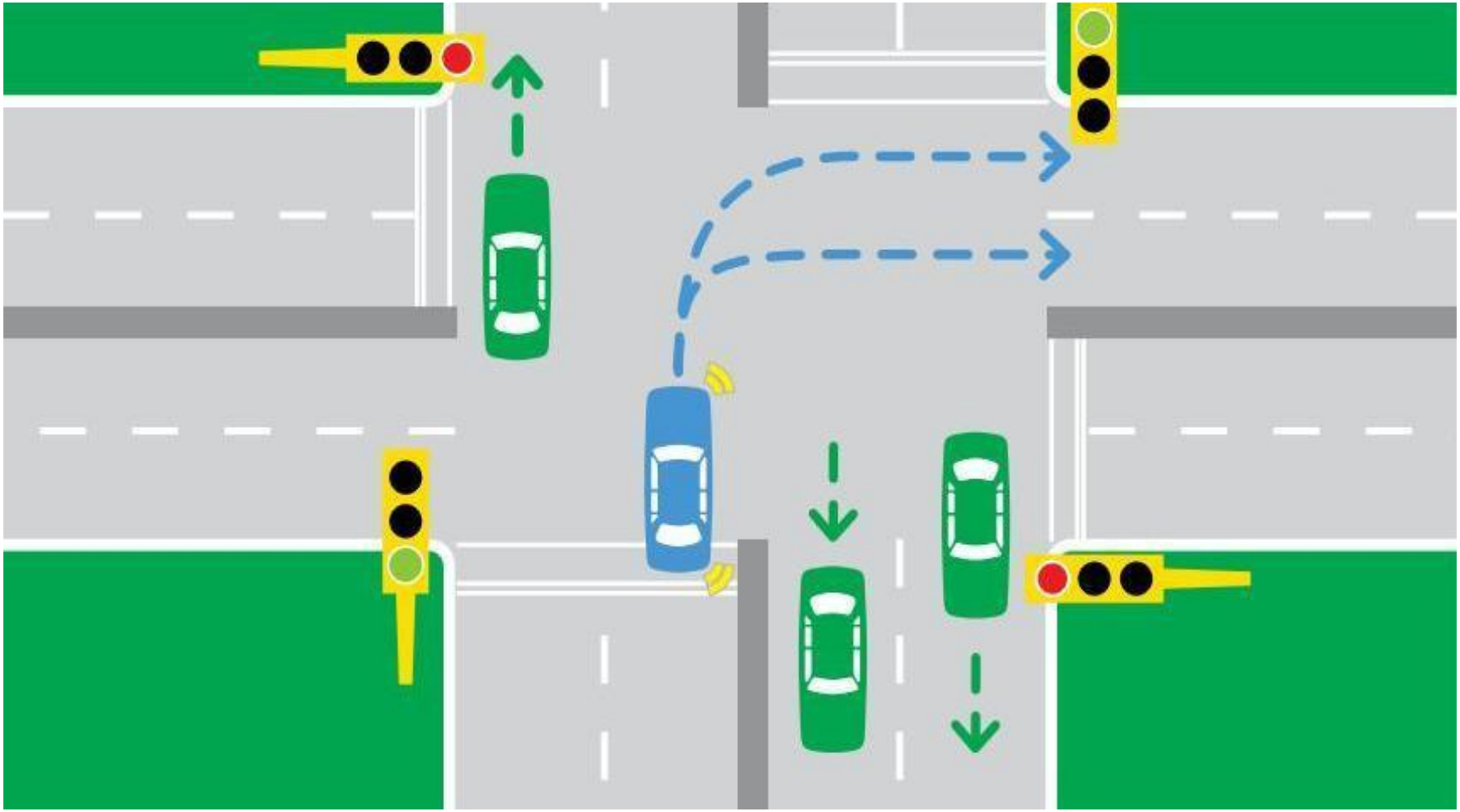
Iterator Illustration

```
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.
  - Instead of directly interacting with each other, objects interact indirectly through a mediator
  - $\mathcal{O}(n)$ couplings instead of $\mathcal{O}(n^2)$
- Use case – dialog box where change in one component affects behavior of others
- Types – Mediator, Components
- JDK – I'm sure there's one hiding somewhere

Mediator Illustration



6. Memento

- Intent – without violating encapsulation, allow client to capture an object's state, and restore it later if desired
- 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 with minimal coupling
- Use case – multiple views of a data object in a GUI
- Types – *Subject*, *Observer* (AKA event handler, AKA listener)
  - 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 (if you need thread-safety); resulting state machine is highly concurrent.

9. Strategy

- Intent – represent a behavior that parameterizes an algorithm for behavior or performance
- Use case – line-breaking for text compositing
- Types – *Strategy*
- JDK – `Comparator`

Strategy Illustration

`Comparator` is a strategy for ordering

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

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

java Test 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
- Types – *AbstractClass*, ConcreteClass
- JDK – skeletal collection implementations (e.g., `AbstractList`)

Template Method Illustration

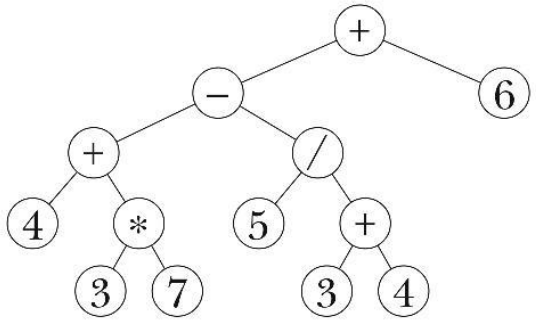
```
// List adapter for primitive int arrays
public static List<Integer> intArrayList(final int[] a) {
    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i];
        }

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

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

11. Visitor

Probably the trickiest GoF pattern

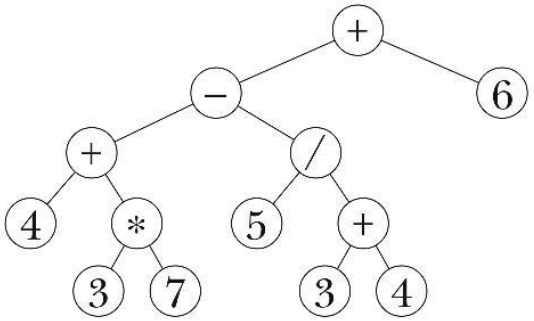
- Intent – represent an operation to be performed on a recursive object structure (e.g., a parse tree). **This pattern lets you add new operations on a tree without adding a method to the types used to represent the tree.**
- Use case – type-checking, pretty-printing, etc.
- Types – *Visitor*, ConcreteVisitors, all element types that get visited
- JDK – none that I'm aware of (but pattern is important)

Visitor Illustration (1/3: machinery)



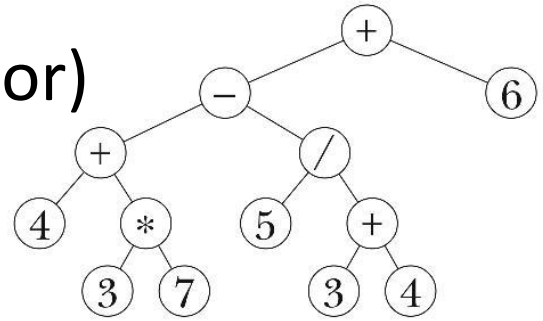

```
public interface Expression {
    public <T> T accept(Visitor<T> v);  // No eval or toString!
}
public interface Visitor<T>  { // T is result type
    public T visitUnaryExpr(Unary Expression expr);
    public T visitBinaryExpr(BinaryExpression expr);
    public T visitNumberExpr(NumberExpression expr);
}
public class UnaryOperationExpression implements Expression {
    public final UnaryOperator operator; public final Expression operand;
    public <T> T accept(Visitor<T> v) { return v.visitUnaryExpr(this); }
}
public class BinaryOperationExpression implements Expression {
    public final BinaryOperator operator; public final Expression op1, op2;
    public <T> T accept(Visitor<T> v) {  return v.visitBinaryExpr(this) ; }
}
public class NumberExpression implements Expression {
    public final double val;
    public <T> T accept(Visitor<T> v) { return v.visitNumberExpr(this); }
}
```

Visitor Illustration (2/3, `eval` visitor)




```
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.op1.accept(this),
                                 be.op2.accept(this));
    }

    public Double visitNumberExpr(NumberExpression ne) { return ne.val; }
}
```

Visitor Illustration (3/3, `toString` visitor)




```
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 and applying a method
  - That's Iterator!
  - Knowing this can prevent you from flunking a job interview 😉
- **The essence of Visitor is *double-dispatch of* `vist` *method***
  - First (dynamically) dispatch on the node type (visitee)
  - Then dispatch on the operation (visitor)
  - This gives you the flexibility to add a new operation to the tree without adding a method to the node types

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