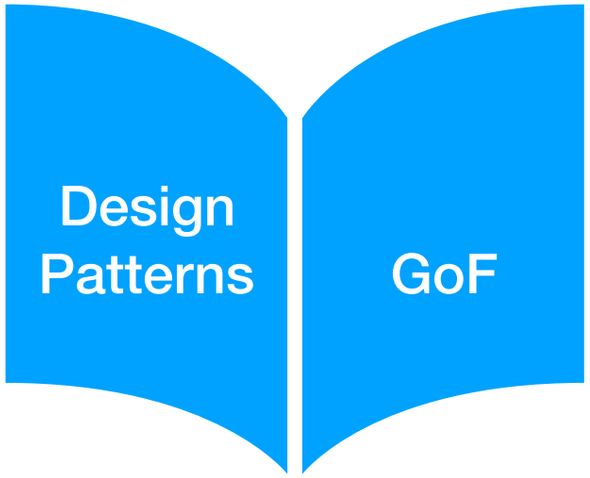
Design Patterns

Rational reconstruction of existing programming practice



Tushaar Gangarapu

What and Why?

"Design pattern is a proven solution for a general design problem. It consists of communicating classes and objects that are customised to solve the problem in a particular context."

***Ref**: Gamma, Erich, et al. "Design patterns: Abstraction and reuse of object-oriented design." European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 1993.*

"Design pattern is a general reusable solution to a commonly occurring problem in software development. Good OO designs are reusable, extensible and maintainable."

***Ref**: Jiang, Shuai, and Huaxin Mu. "Design patterns in object oriented analysis and design." Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on. IEEE, 2011.*

What and Why? (cont.)

- Encapsulation and Abstraction
- Openness and Variability
- Generativity and Composability
- Equilibrium

***Ref**: Shirin A. L. "CSC532 Term Paper- Design Patterns."* http://www2.latech.edu/~box/ase/tp_2003/Term%20Paper_Lakhani_Shirin.doc.

Pattern?

"A design pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, ..., and the rule which tells us how to create that thing, and when we must create it."

***Ref**: Alexander, Christopher. The timeless way of building. Vol. 1. New York: Oxford University Press, 1979.*

Schema of *four* parts-

- name – vocabulary for using the pattern
- problem – context in which the pattern is applicable
- solution – components of the pattern and how they interact
- consequences – trade-offs and implications that arise from adopting the solution to the problem in context

Classification (GoF)

Classification by 'Scope' and by 'Purpose'

Purpose / Scope	Creational	Structural	Behavioural
Class	1. Factory Method	1. Adapter (Class)	1. Interpreter 2. Template Method
Object	1. Abstract Factory 2. Builder 3. Prototype 4. Singleton	1. Adapter (Object) 2. Bridge 3. Composite 4. Decorator 5. Façade 6. Flyweight 7. Proxy	1. Chain of Responsibility 2. Command 3. Iterator 4. Mediator 5. Memento 6. Observer 7. State 8. Strategy 9. Visitor

Classification (cont.)

- **Object Patterns** (has-a: composition)
- **Class Patterns** (is-a: inheritance)

"Scope"

"Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static-fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labelled *class patterns* are those that focus on class relationships. Note that most patterns are in the object scope."

***Ref**: Gamma, Erich. Design patterns: elements of reusable object-oriented software. Pearson Education India, 1995.*

Classification (cont.)

We are restricting ourselves to GoF patterns!

- **Creational Patterns**
  - Abstraction of the instantiation process
  - Make a system independent of how its objects are created, composed, and represented
- **Structural Patterns**
  - How classes and objects are composed to form larger structures (use inheritance to compose interfaces or implementations)
  - Useful for making independently developed class libraries work together
- **Behavioural Patterns**
  - Most specifically concerned with communication between objects.

Pattern Format

Followed by GoF!

- **Context** – aim of the pattern
- **Use case** – a "motivating" example
- **Key types** – the interfaces that define the pattern
  - *Italic type*: abstract class; typically this is an interface when the pattern is used in Java
- **JDK** – example(s) of this pattern in JDK
- **Illustration** – code sample, diagram, or drawing

Creational Patterns

- Singleton Pattern *
- Abstract Factory Pattern
- Builder Pattern
- Factory Method Pattern *
- Prototype Pattern

Singleton Pattern

- **Context** – ensuring a class has only one instance per JVM

Multi-threading?

- **Use case** – printer spooler, file system, an accounting system dedicated for a company, A/D converter
    - Compelling uses are *rare* but they do exist
- **Key types** – Singleton

Unit Testing?

- **JDK** – `java.lang.Runtime`

Singleton Illustration

```
public enum Tushaar {
     TUSHAAR ;
     public eatCarrots(Carrot carrot) { ... }
     public exercise(Gym gym) { ... }
     public takeMedicine(Drug drug) { ... }
     public drinkWater(Water water) { ... }
}

// Alternate Implementation
public class Tushaar {
     public static final Tushaar TUSHAAR = new Tushaar() ;
     private Tushaar() { ... }
}
```

My personal favourite way!

My take on singleton

- It's an *instance-controlled* class; ways of implementation include –
  - **Static utility class**: non-instantiable
  - **Enum**: one instance per value, new values are created at runtime
  - **Interned class**: one canonical instance per value, new values created at runtime

Lazy vs. Eager

- "There is a duality between singleton and static utility class" - GoF

Abstract Factory

- **Context** – allow creation of families of related objects independent of implementation
- **Use case** – look-and-feel in a GUI toolkit
- **Key types** – *Factory* with methods to create each family member
- **JDK** – not common

Abstract Factory Illustration

```java
// Abstract Factory
public abstract class Fashion {
   abstract FashionShow getDetails(String show);
   abstract NextTopModel getJudges(String series) ;
}

public interface FashionShow {
     void details() ;
}

public class MilanFahionWeek implements
FashionShow {
     @override
     public void details() {
          // February 21 - 27, 2018
     }
}

public class NewYorkFahionWeek implements
FashionShow {
     @override
     public void details() {
          // February 8 - 16, 2018
     }
}
```

```java
public interface NextTopModel {
     void judges() ;
}

public class ANTM implements NextTopModel {
     @override
     public void judges() {
          // Tyra Banks
     }
}

public class AusNTM implements NextTopModel {
     @override
     public void judges() {
          // Alex Perry
     }
}
```

Builder Pattern

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

Builder Illustration

```
public class NutritionalInformation {
     public static class Builder {
          public Builder(String name, int servingSize, int numServings) { ... }
          public Builder totalFat(int val) { totalFat = val ; }
          public Builder saturatedFat(int val) { saturatedFat = val ; }
          public Builder transFat(int val) { transFat = val ; }
          public Builder cholesterol(int val) { cholesterol = val ; }
          ...  // many more setters (protein, calories, ...)

          public NutritionalInformation build() {
               return new NutritionalInformation(this) ;
          }
     }
     private NutritionalInformation(Builder builder) { ... }
}

// Usage
NutritionalInformation twoLiterMartini = new NutritionalInformation.Builder("Martini", 240,
8).calories(244).build() ;
```

My take on Builder Pattern

- Emulates named parameters in languages that don't support them
- Reduces exponential `O(2ⁿ)` creational methods to `O(n)` by allowing them to be combined freely
- Cost of having an intermediate (Builder) object

Prototype Pattern

The Life-Size Chess Match

Rupert Grint, who plays Harry Potter's best friend, Ron Weasley, rides one of the live chess pieces that guard the hidden Sorcerer's Stone at Hogwarts.

this boy. The following day I called Alan at nine o'clock in the morning. Being with Dan took me to another world. I was meeting Harry Potter."

For Dan, the whole experience was rather overwhelming. When he went for an audition, he was asked to read the scene where the children find out that Hagrid is hiding a contraband dragon's egg. "I was totally scared out of my wits," says Radcliffe, whose mother, Marcia Gresham, is a casting director. "It was so terrifying. You go in there with these really important people, and you just kind of feel really small. So then I went to three other auditions after that, and then they phoned me up and asked me if I wanted to play the part. It was probably *the* single most exciting thing that's ever happened to me."

And now all that's left is the judgment of millions of Rowling aficionados. For those who find themselves disappointed by the movie, there's always the next Harry Potter book to anticipate. Number five is entitled *Harry Potter and the Order of the Phoenix,* but although it has been widely reported that the book will come out next spring, Rowling's publishers caution that any such announcement is regrettably premature. "We don't have it," says Judy Corman, a senior vice president for Scholastic Inc., adding that Rowling is still writing the book and its publication date hasn't been scheduled yet. "Probably mid-2002," says Rowling's agent, Christopher Little.

But the wait—always agonizing for true Potter fanatics counting the days until the next installment—will surely be eased if the movie turns out to be a winner.

Steve Kloves, whose previous films have been critical but not commercial successes, resorts to self-deprecating humor when discussing the prospects for *Harry Potter and the Sorcerer's Stone.* "I have a pretty consistent track record of no one going to the movies I've made, so if people show up, I'll be happy," he says.

That much certainly seems assured. The larger question is whether the film will measure up to the standard set by the book—an exceptional test, to be sure.

"I hope people will feel that we have been true to the spirit of the books, and that they are able to enjoy the movie on its own terms," says David Heyman.

Then he adds what every Potter-lover will surely join him in feeling as the movie's release date approaches: "I hope it's going to be a classic." □

Prototype Pattern

- **Context** – create an object by cloning another and tweaking as necessary
- **Use case** – initial setup of a chess game
- **Key terms** – *Prototype* (AKA Cloneable)
- **JDK** – `clone`, usually not used other than on arrays
    - Java and Prototype pattern are a poor fit

Factory Method

- **Context** – 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()`

***Further Reading**: 'Static Factory' pattern is very common; technically not a GoF pattern, but close enough*

Factory Method Illustration

```
public interface FashionShow {
     void details() ;
}

public class MilanFahionWeek implements FashionShow {
     @override
     public void details() {
          // February 21 - 27, 2018
     }
}

public class NewYorkFahionWeek implements FashionShow {
     @override
     public void details() {
          // February 8 - 16, 2018
     }
}
```

Factory Method vs. Abstract Factory

Structural Patterns

- Adapter Pattern *
- Bridge Pattern
- Composite Pattern *
- Decorator Pattern *
- Façade Pattern
- Flyweight Pattern
- Proxy Pattern

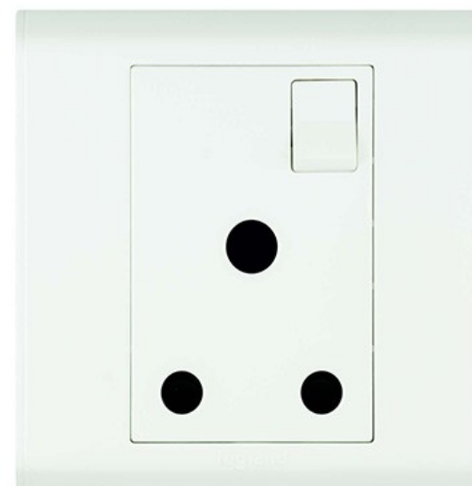
Adapter Pattern

- **Context** – convert interface of a class into one that another class requires, allowing interoperability
- **Use case** – arrays vs. collections
- **Key types** – Target, Adaptee, Adapter
- **JDK** – `Arrays.asList(T[])`

Adapter Illustration

Have this

and this?

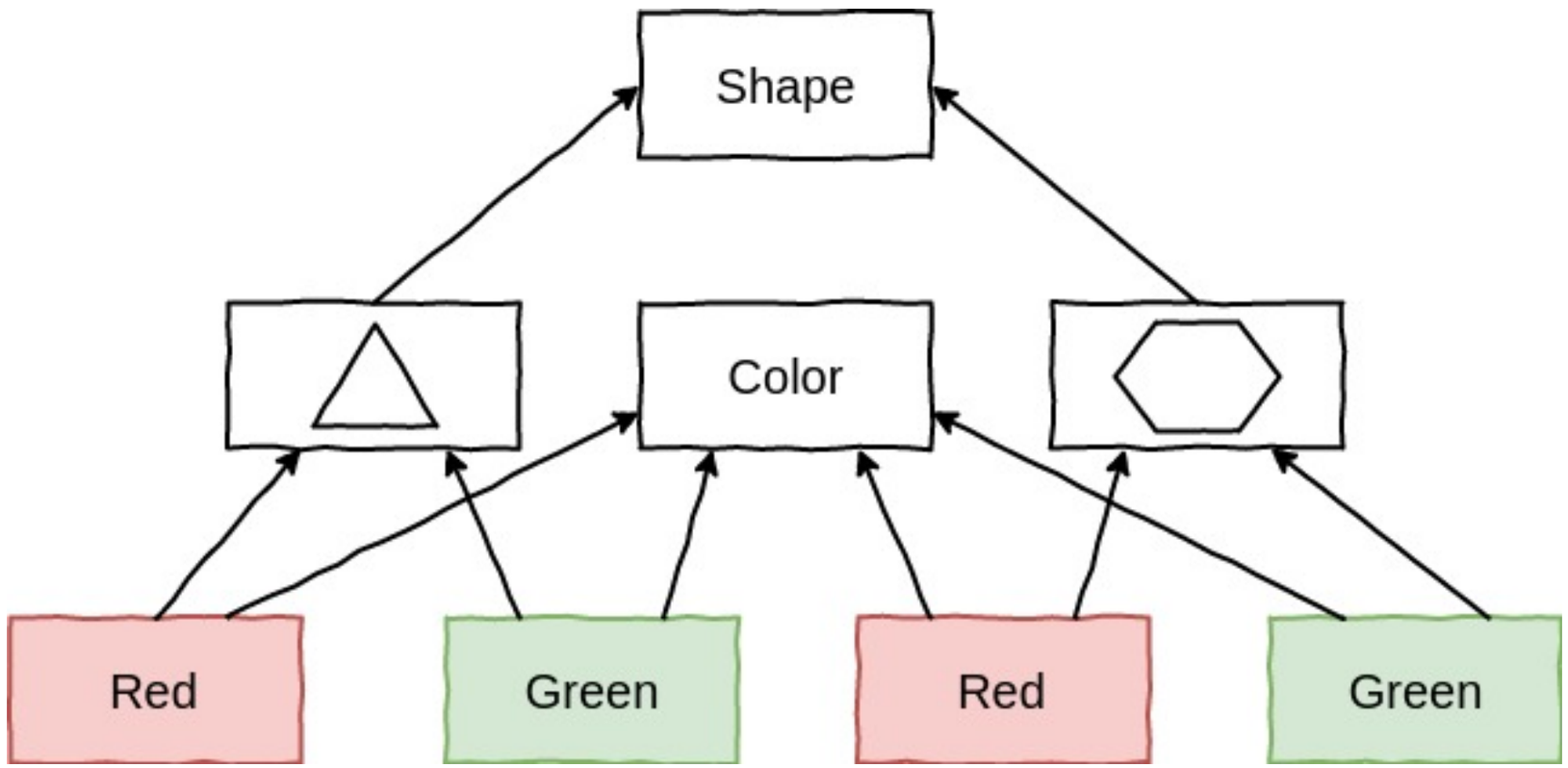


Use this!

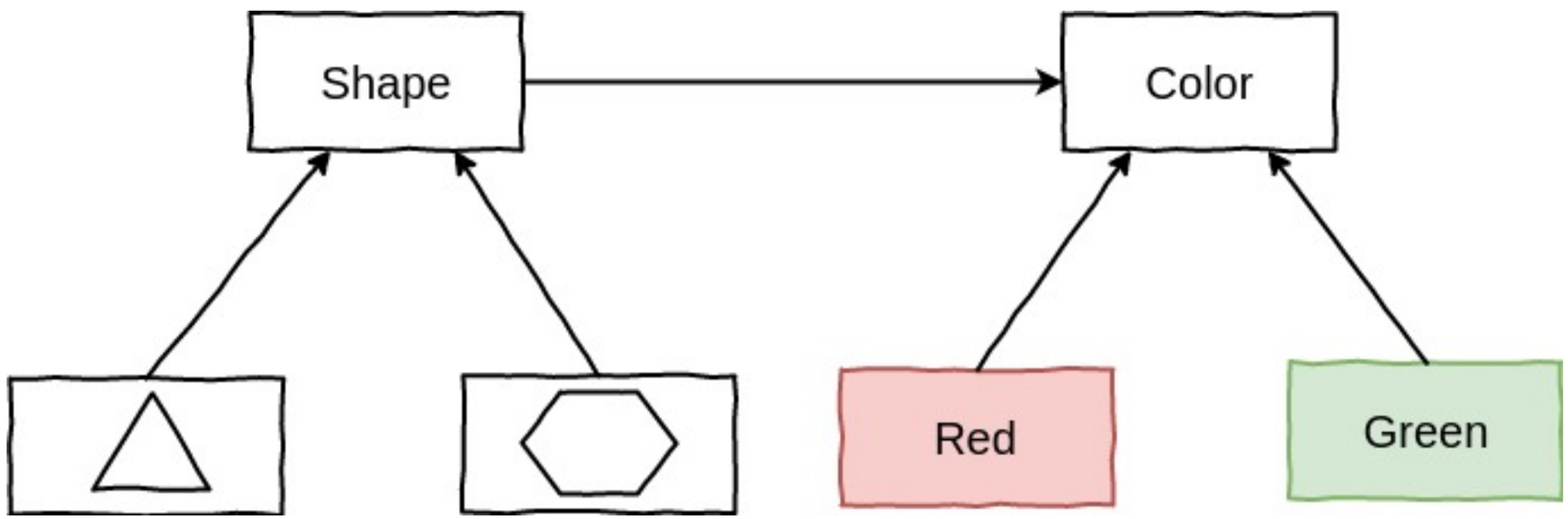
Bridge Pattern

- **Context** – decouple an abstraction from its implementation so that 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 Illustration



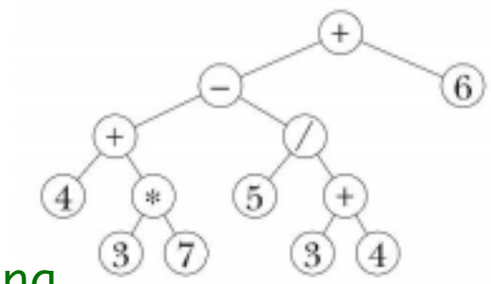
Bridge Pattern Illustration (cont.)



Composite Pattern

- **Context** – compose objects into tree structures; let clients treat primitives and compositions uniformly
- **Use case** – GUI toolkit (widgets and containers)
- **Key types** – *Component* that represents both primitives and their containers
- **JDK** – `javax.swing.JComponent`

Composite Illustration

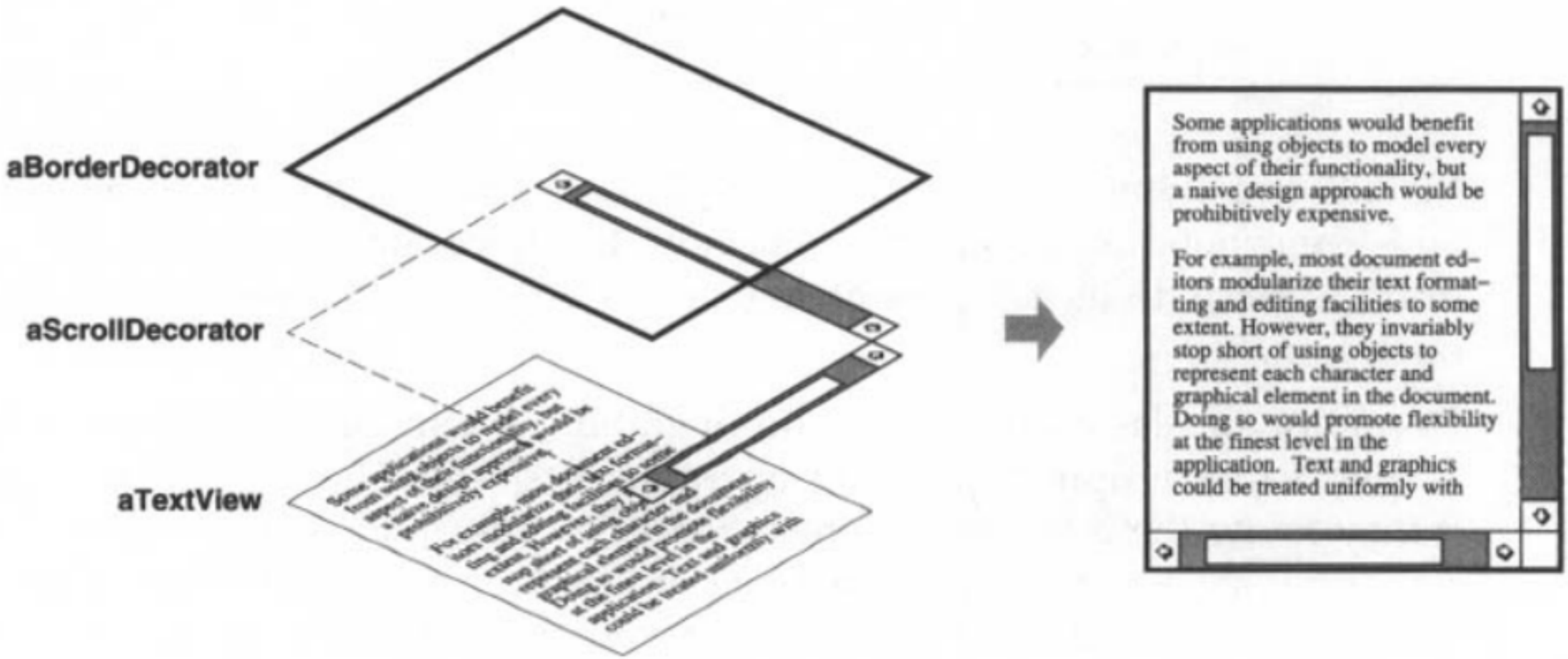



```
public interface Expression {
    double eval();            // Returns value
    String toString();        // Returns infix expression string
}

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) ;
}
```

Decorator Pattern

- **Context** – 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

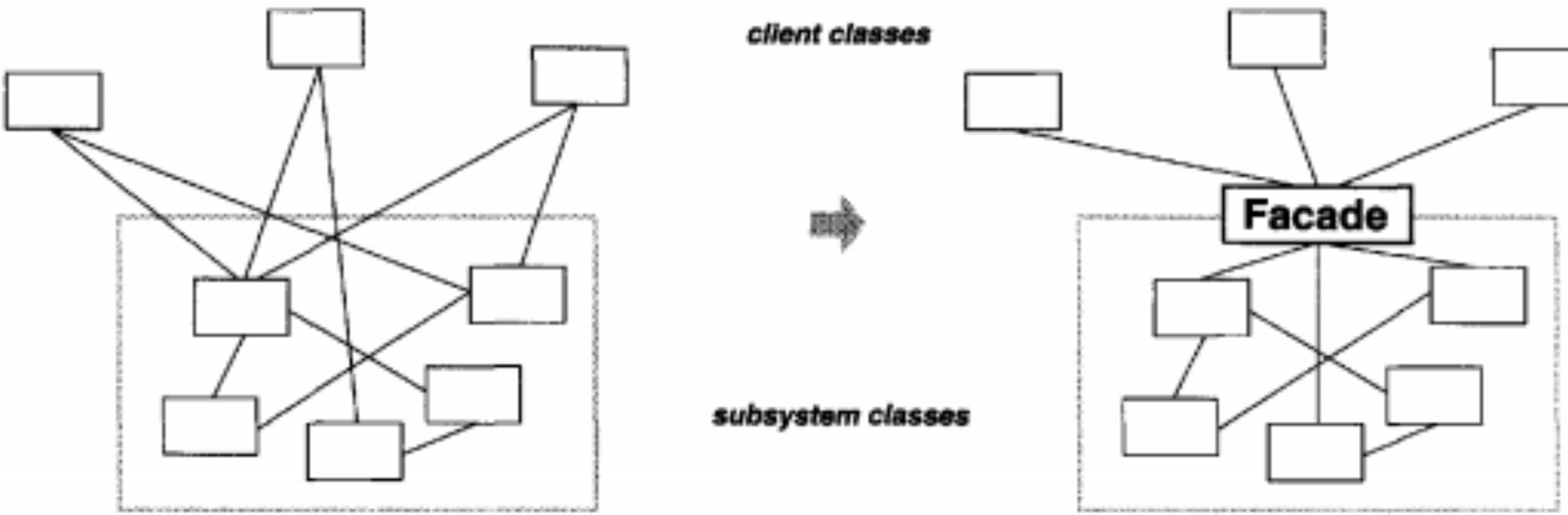


***Ref**: GoF*

Façade Pattern

- **Context** – 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: compiler
- **Key types** – Façade (the simple unified interface)
- **JDK** – `java.util.concurrent.Executors`

Façade Illustration

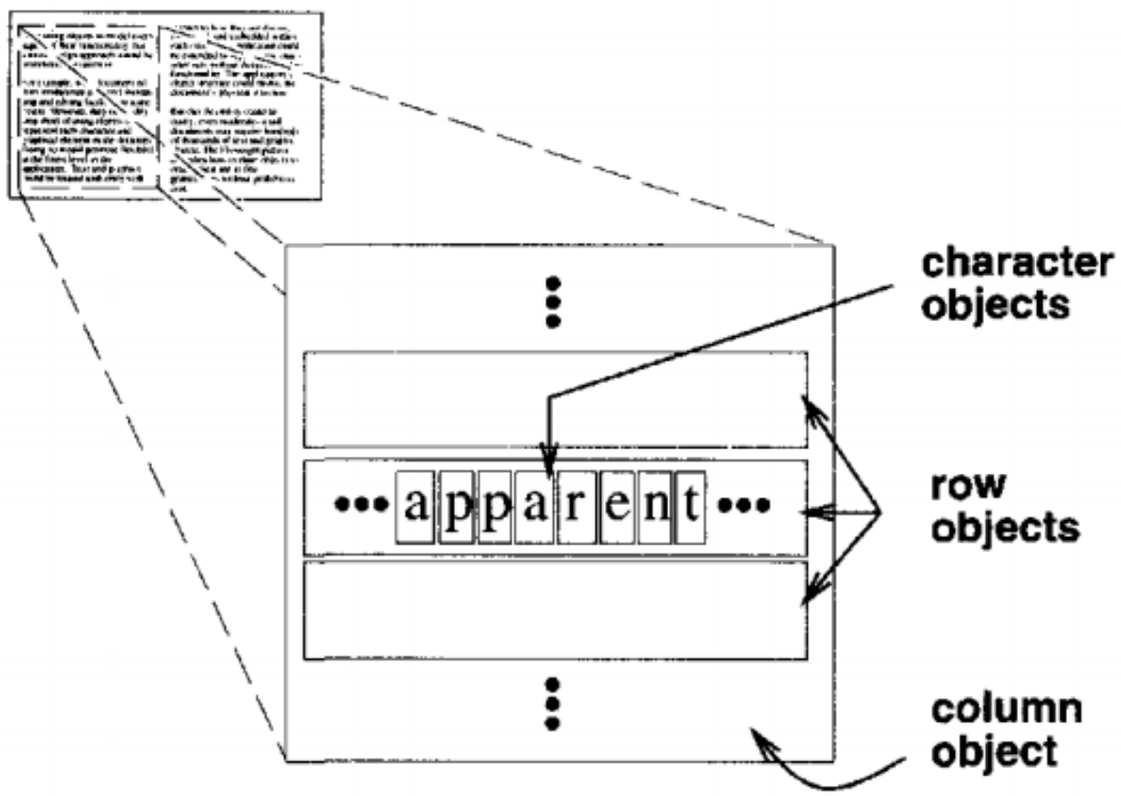


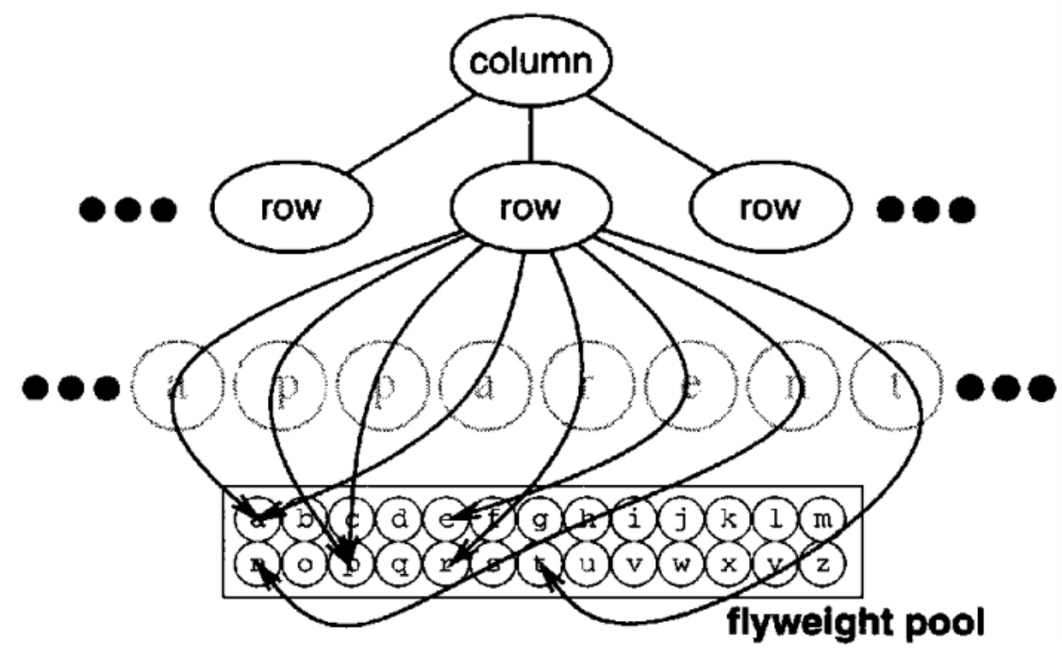
***Ref**: GoF*

Flyweight Pattern

- **Context** – use sharing to support large numbers of fine-grained objects efficiently
- **Use case** – characters in a document
- **Key types** – Flyweight (instance-controlled!)
  - State can be made *extrinsic* to keep Flyweight sharable
- **JDK** – Pervasive! All enums, many others.
  `j.u.c.TimeUnit` has number of units as *extrinsic* state

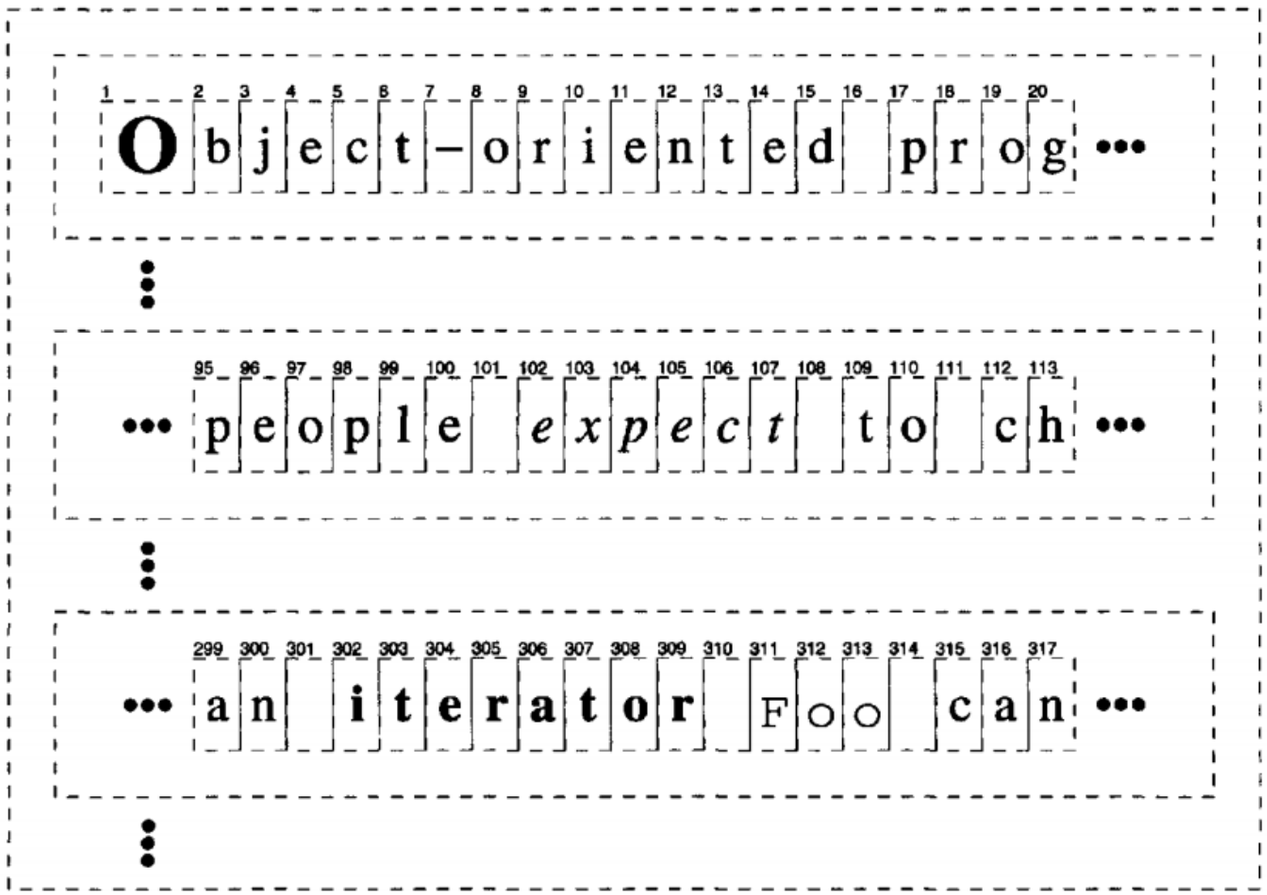
Flyweight Illustration

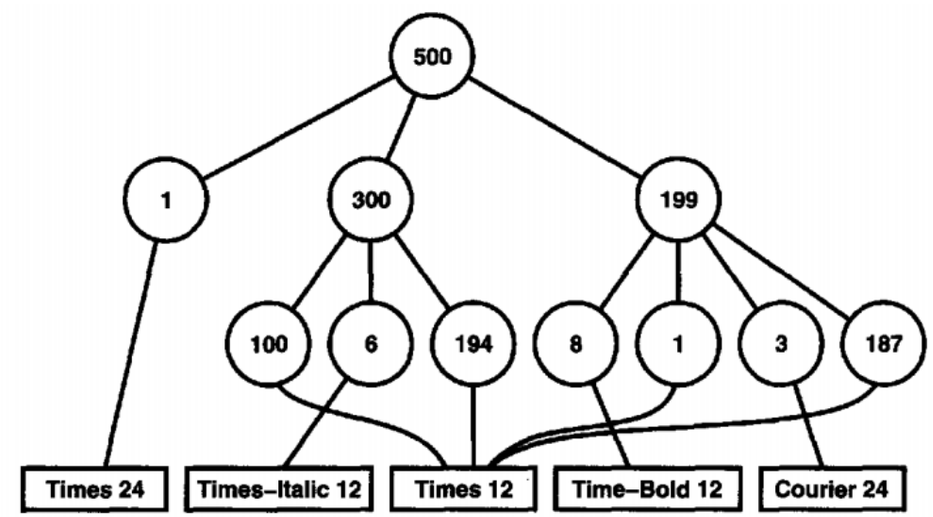




***Ref**: GoF*

Flyweight Illustration (cont.)



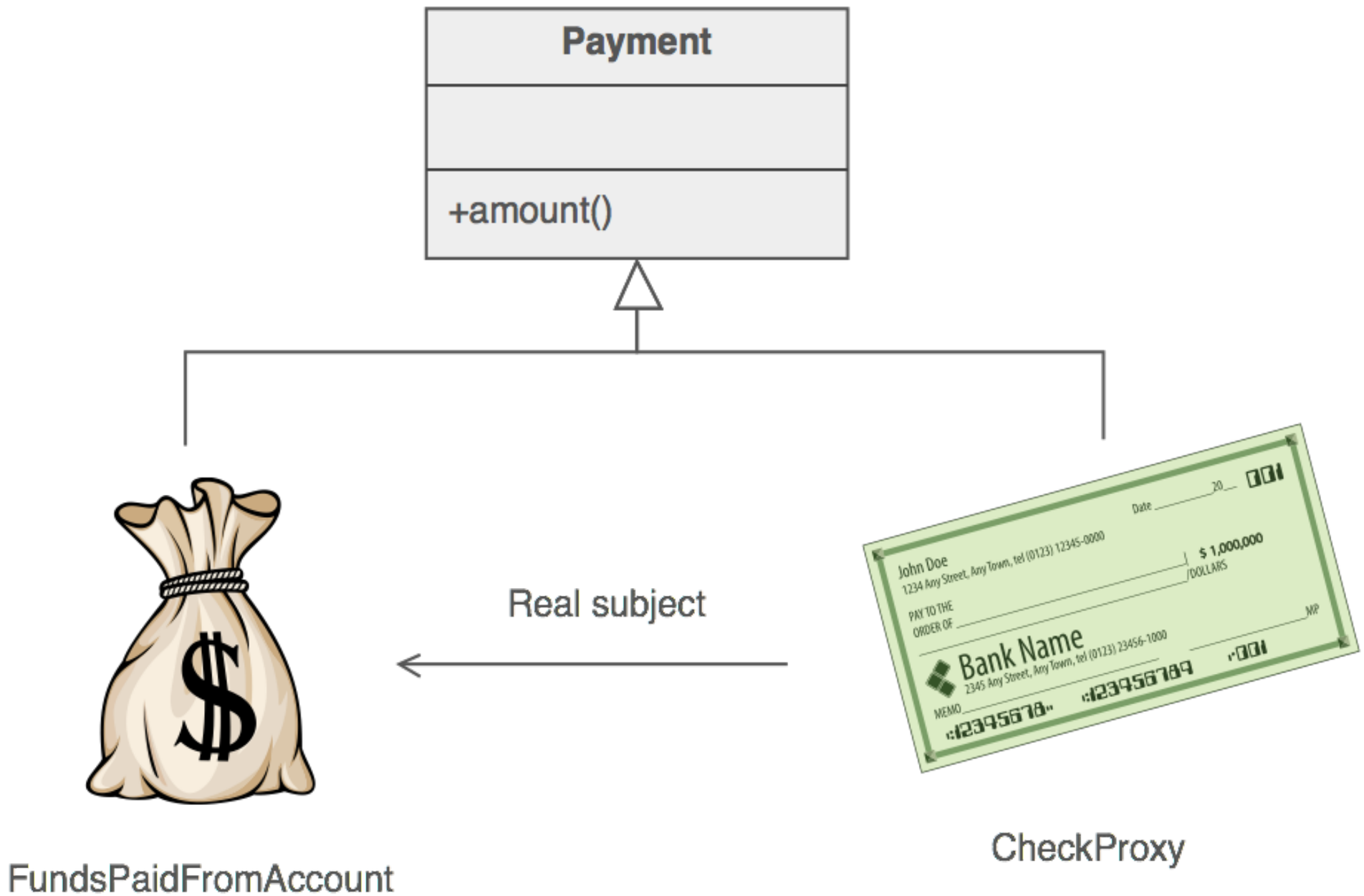


***Ref**: GoF*

Proxy Pattern

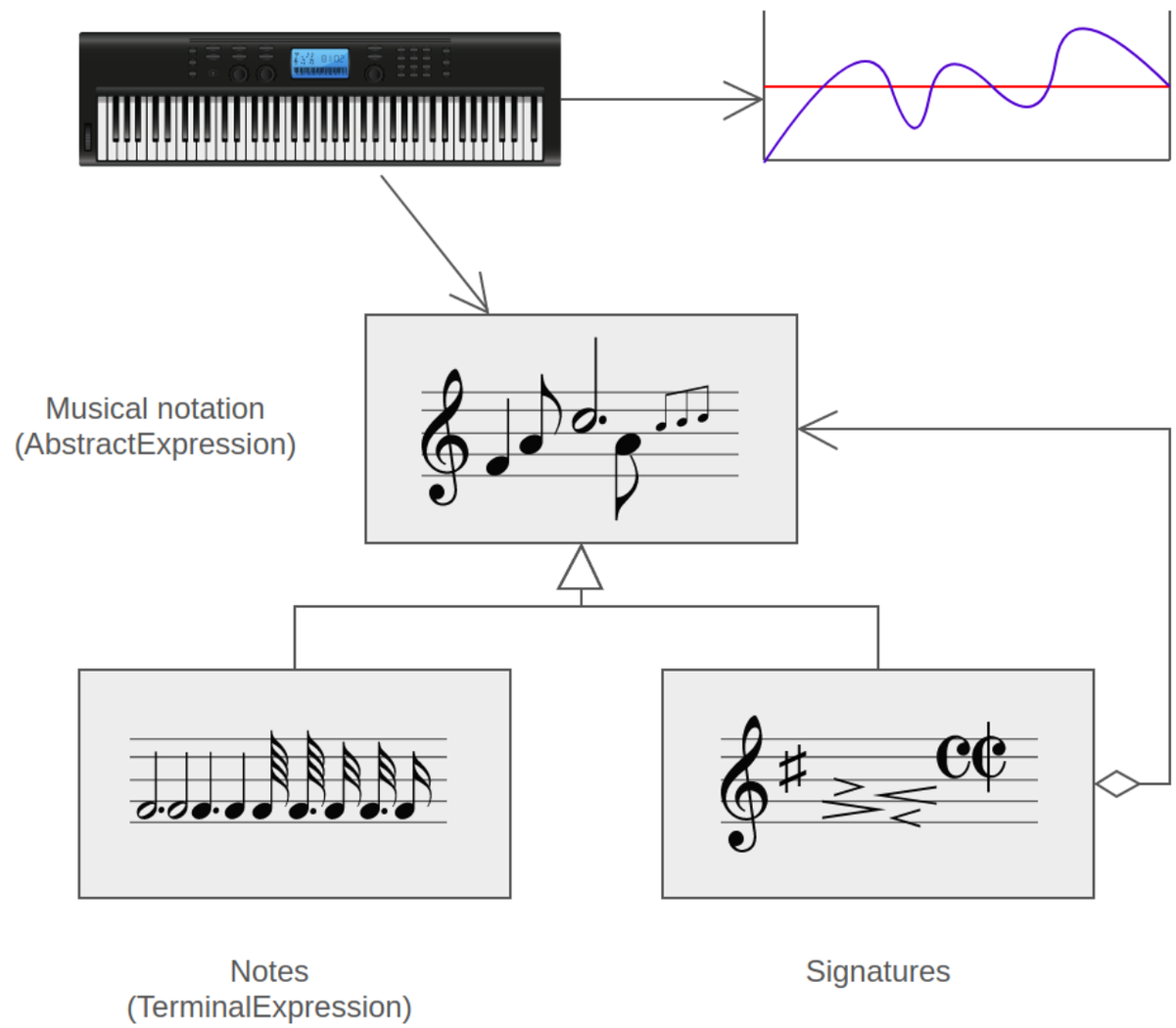
- **Context** – surrogate for another object
- **Use case** – delay loading of images till needed
- **Key types** – *Subject*, Proxy, RealSubject
- **GoF Flavors** –
  - **Virtual proxy**: stand-in that instantiates lazily
  - **Remote proxy**: local representative for remote object
  - **Protection proxy**: denies some operations to some users
  - **Smart reference**: does locking or reference counting
- **JDK** – `RMI`, `collections` wrappers

Proxy Pattern Illustration



Behavioral Patterns

- Chain of Responsibility
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template method
- Visitor Pattern



Further Reading

- MLA Heer, Jeffrey, and Maneesh Agrawala. "Software design patterns for information visualization." IEEE transactions on visualization and computer graphics 12.5 (2006): 853-860.
- Astrachan, Owen, et al. "Design patterns: an essential component of CS curricula." ACM SIGCSE Bulletin. Vol. 30. No. 1. ACM, 1998.
- Mu, Huaxin, and Shuai Jiang. "Design patterns in software development." Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on. IEEE, 2011.
- Jiang, Shuai, and Huaxin Mu. "Design patterns in object oriented analysis and design." Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on. IEEE, 2011.
- Subburaj, R., Gladman Jekese, and Chiedza Hwata. "Impact of object oriented design patterns on software development." (2015).
- http://www.oodesign.com/

For queries and material

tushaargvsg45@gmail.com