

Programming Engineering

Course 8
18 April

Revision

- ▶ Design Patterns: Definitions, Elements
- ▶ GOF: Creational Patterns, Structural Patterns, Behavioral Patterns

OOD principles for classes

- ▶ The Single Responsibility Principle
- ▶ The Open Closed Principle
- ▶ The Liskov Substitution Principle
- ▶ The Interface Segregation Principle
- ▶ The Dependency Inversion Principle

- ▶ SOLID

Creational Patterns

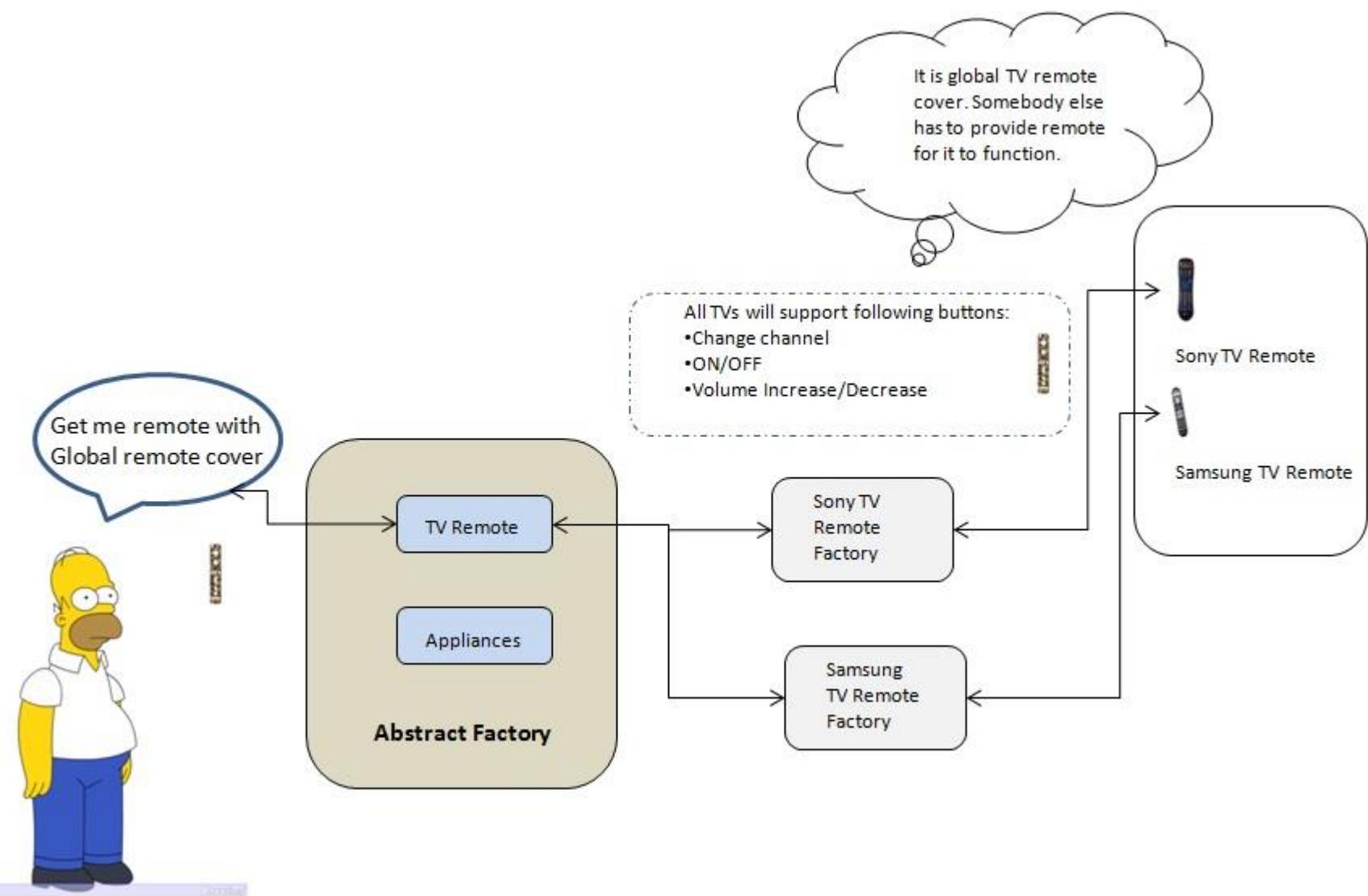
- ▶ Abstract Factory
- ▶ Builder
- ▶ Factory Method
- ▶ Prototype
- ▶ Singleton
- ▶ Lazy initialization
- ▶ Object pool
- ▶ Multiton
- ▶ Resource acquisition

Creational Patterns – Introduction

- ▶ Abstract the instantiation process
- ▶ Help make a system independent of how its objects are created, composed, and represented
- ▶ There are two recurring themes in these patterns:
 - First, they all encapsulate knowledge about which concrete classes the system uses
 - Second, they hide how instances of these classes are created and put together
- ▶ The creational patterns give a lot of flexibility in *what gets created, who creates it, how it gets created, and when*

Creational Patterns – Abstract Factory

- ▶ **Intent** – Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- ▶ **Also Known As** – Kit
- ▶ **Motivation**
 - Consider a user interface toolkit that supports multiple look-and-feel standards, defines different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons
 - To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel



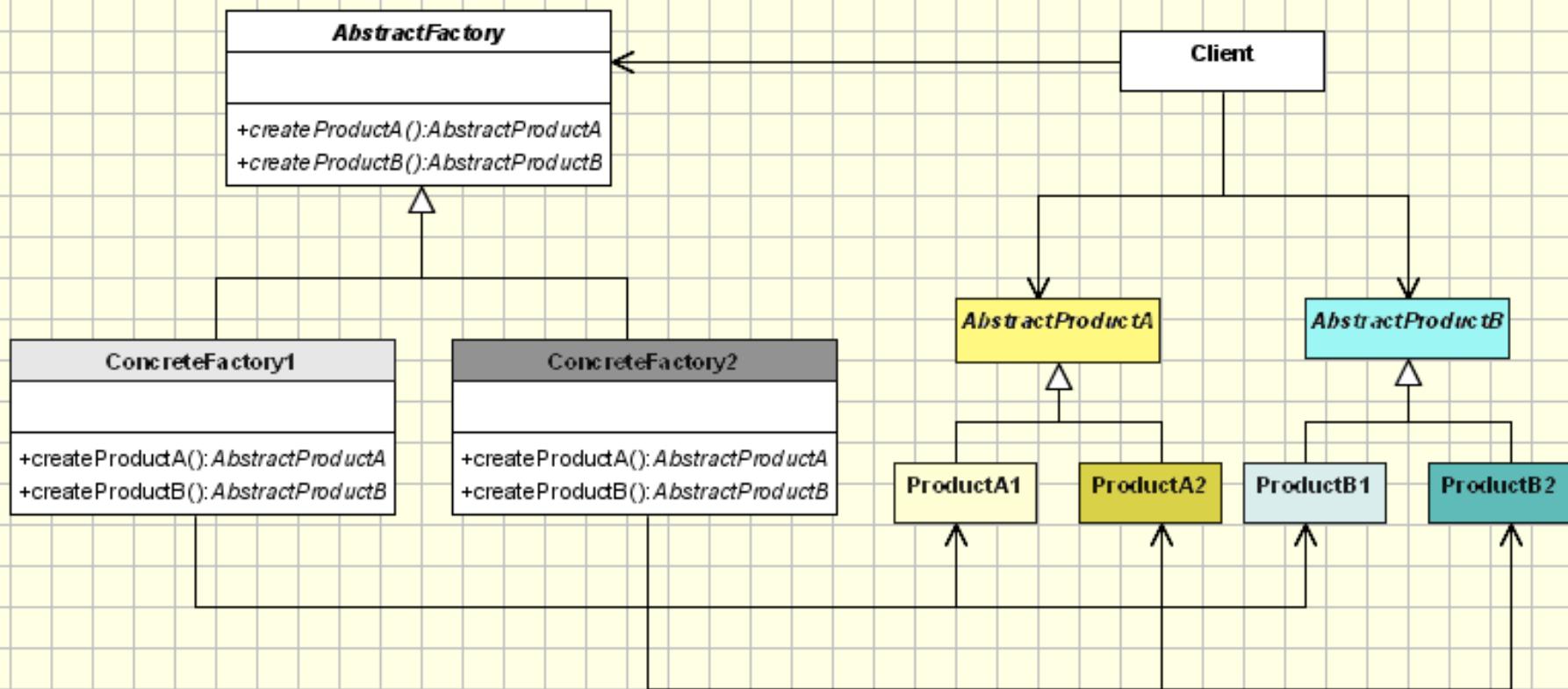
Abstract Factory

- ▶ **Applicability** – Use this pattern when:
 - a system should be independent of how its products are created, composed, and represented
 - a system should be configured with one of multiple families of products
 - a family of related product objects is designed to be used together, and you need to enforce this constraint
 - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

CP - Abstract Factory

► Abstract Factory – maze, computer components

cd: Abstract Factory Implementation - UML Class Diagram



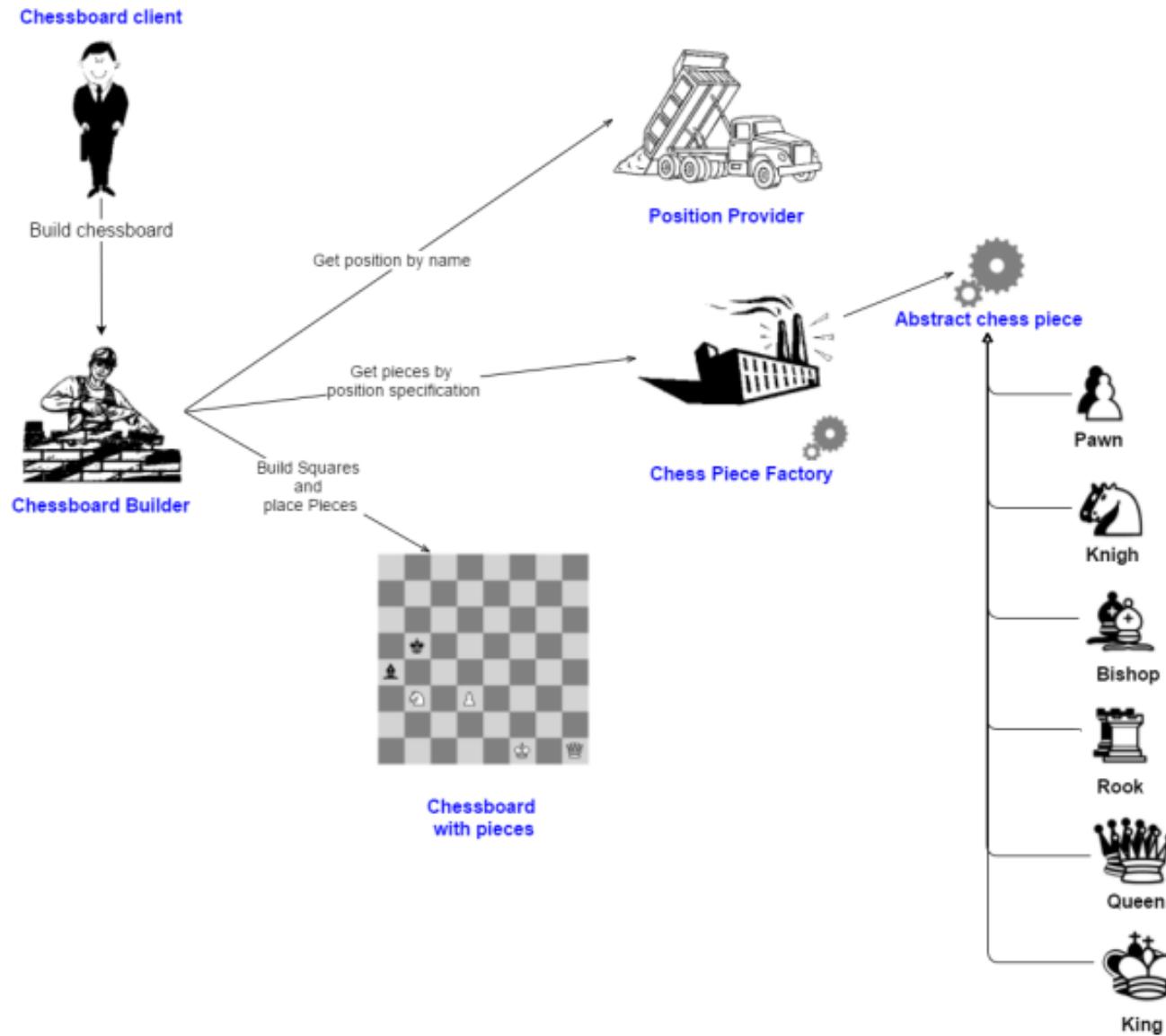
Abstract Factory- The Good, The Bad ...

- ▶ Detaches creators from created objects
- ▶ Guarantees compatibility of created objects
- ▶ Preserves S and O (from SOLID)
- ▶ Code becomes complicated because of lots of extra classes

Creational Patterns – Builder

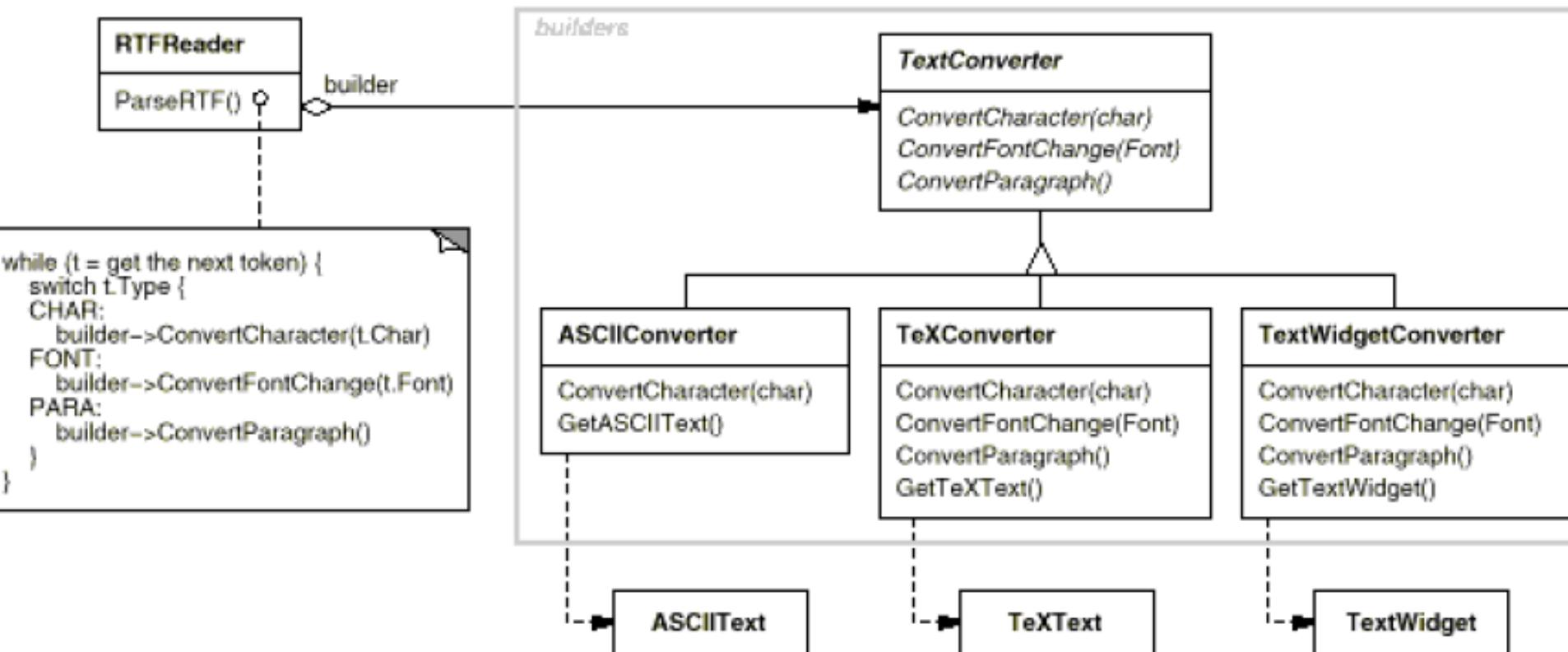
- ▶ Intent – Separate the construction of a complex object from its representation so that the same construction process can create different representations
- ▶ Motivation
 - A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats.
 - The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

Builder and Factory Design Pattern - Building chessboard with pieces



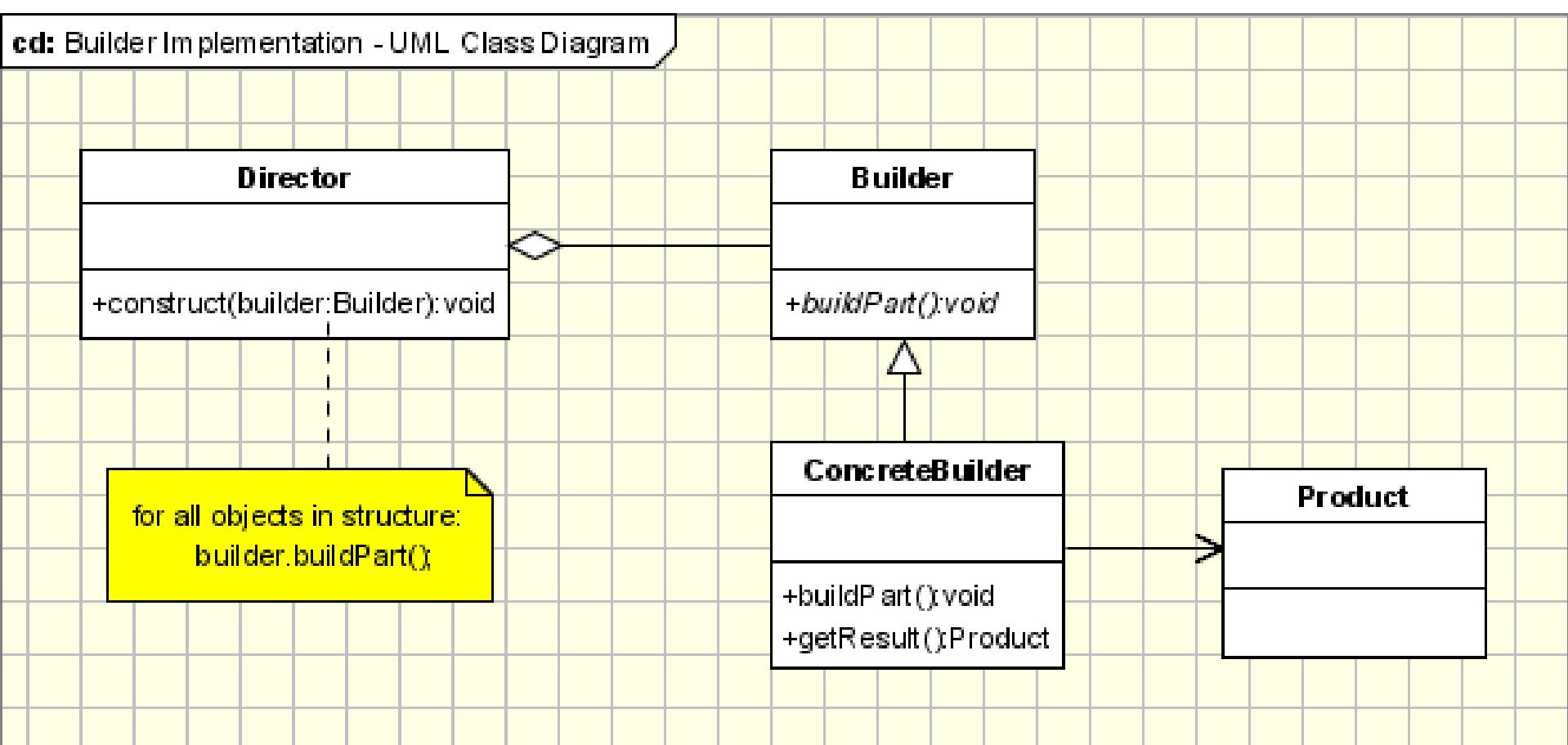
Builder 2

- ▶ Each kind of converter class takes the mechanism for creating and assembling
- ▶ The converter is separate from the reader



CP – Builder

► Builder – RTF Reader, Happy Meal



Builder 3

- ▶ **Applicability** – Use when the parameters required for construction are numerous, but only some are required for each specific object (different kinds of pizza)
- ▶ When you want to build different representations of similar objects (rail bridge and road bridge)

Builder - The Good, The Bad ...

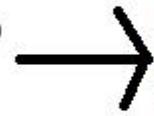
- ▶ Allows for step by step construction of objects
- ▶ Steps can be skipped and run later, or can be run recursively
- ▶ Allows for reusing code when building similar (but different) objects
- ▶ Preserves S (from SOLID) – decouples complex building processes from the business logic
- ▶ Code becomes complicated because of many new classes

Creational Patterns – Factory Method

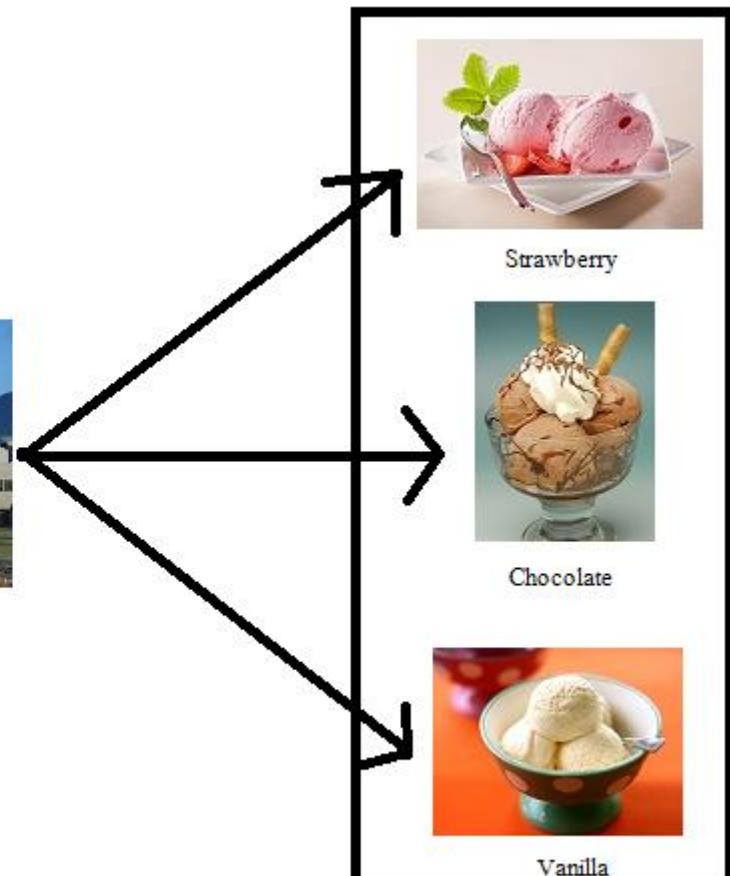
- ▶ **Intent** – Define an interface for creating an object, but let subclasses decide which class to instantiate
- ▶ **Also Known As** – Virtual Constructor
- ▶ **Motivation** – To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required (at Open or New from a menu, for example)



Client



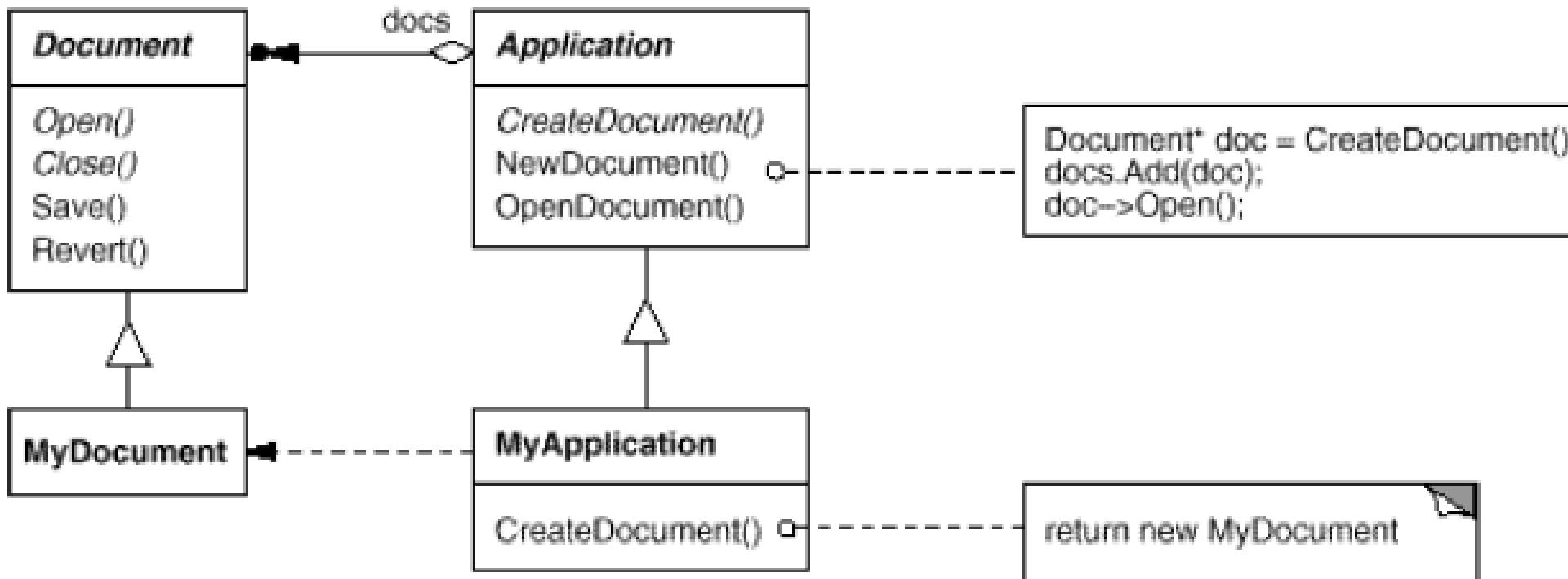
Factory



Product

Factory Method 2

- ▶ Subclasses redefine an abstract CreateDocument
- ▶ It can then instantiate application-specific Documents without knowing their class
- ▶ CreateDocument is a **factory method because it's responsible for "manufacturing" an object**



Factory Method 3

- ▶ **Applicability** – Use this pattern when
 - a class can't anticipate the class of objects it must create
 - a class wants its subclasses to specify the objects it creates
 - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

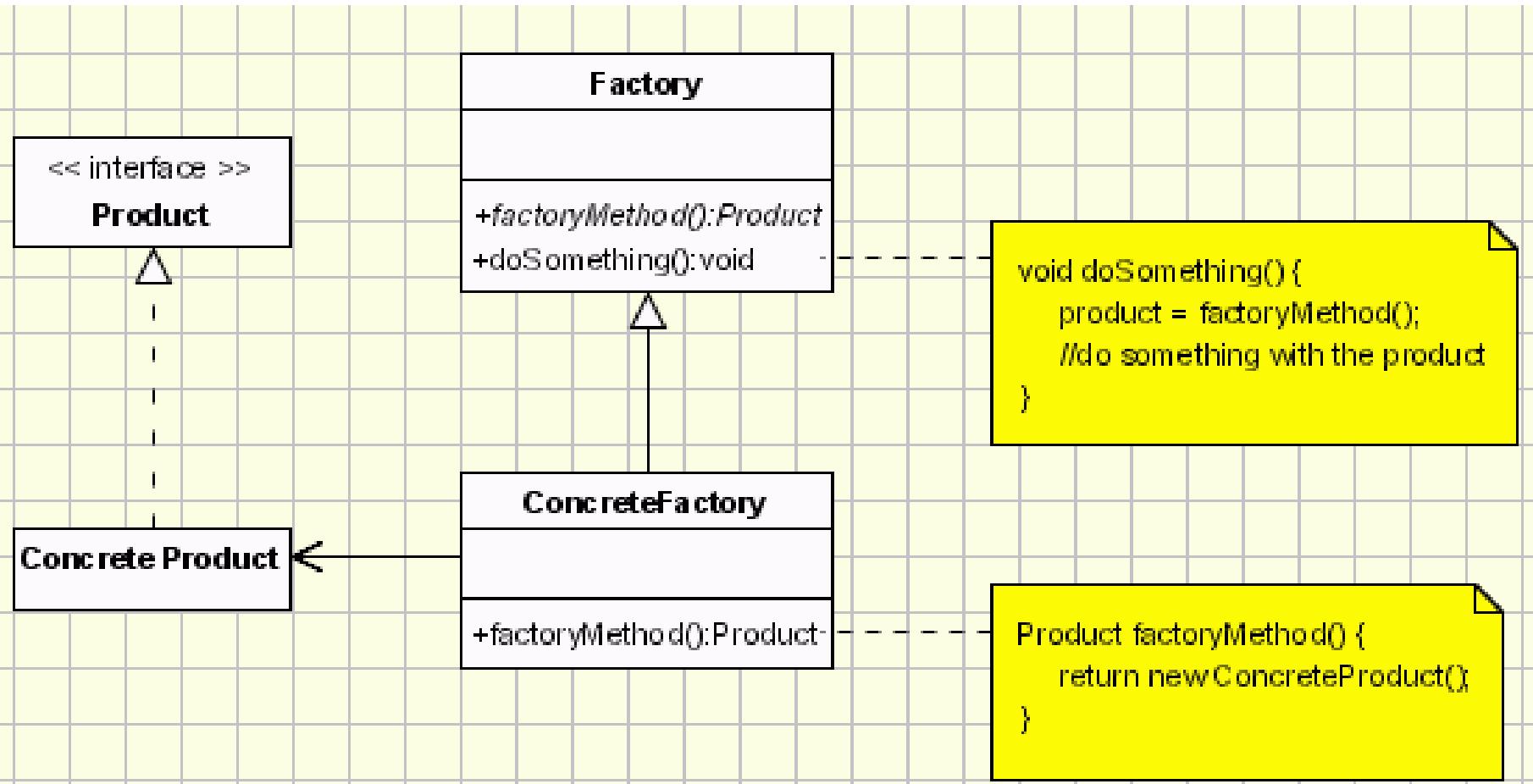
Factory Method 4

▶ Consequences

1. *Provides interconnections for subclasses*
2. *Connects parallel class hierarchies*

CP - Factory Method

▶ Factory Method – Open/New Project, Hello Mr/Ms



Factory Method- The Good, The Bad ...

- ▶ Detaches creators from created objects
- ▶ Preserves S and O (from SOLID)
- ▶ Code becomes complicated because of extra (derived) classes

Creational Patterns – Prototype

- ▶ **Intent** – Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- ▶ **Motivation** – You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves
- ▶ The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects

Dog Object



Cloned Dog Object



Clone

Leg object
Ear object
Eye object
Tail object

Name = Tommy
Age = 3 yrs
Color = white

Name = Tommy
Age = 3 yrs
Color = white

Modify

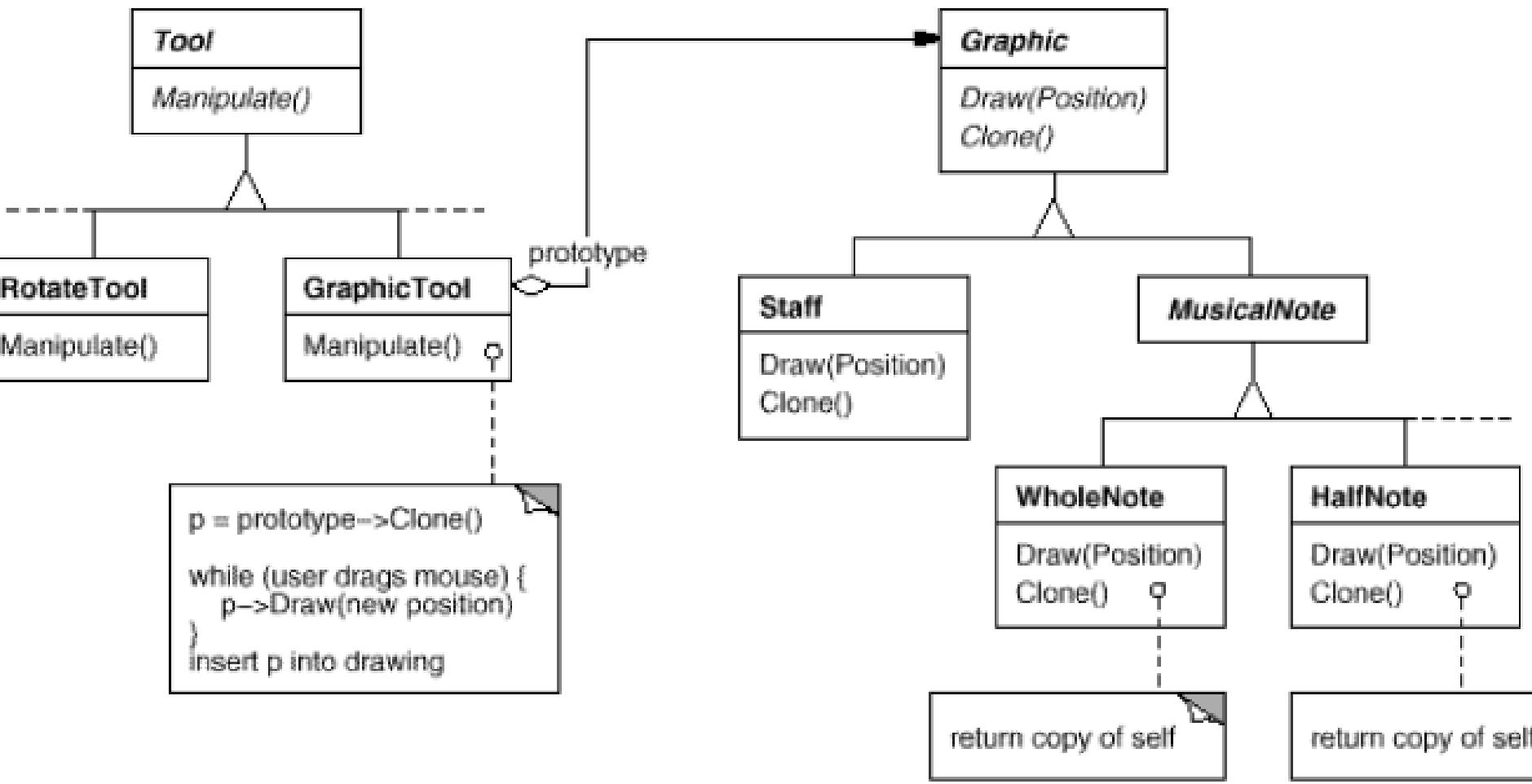
Modified Dog Object



Name = Jimmy
Age = 6 yrs
Color = Brown

Prototype 2

- ▶ We can use the Prototype pattern to reduce the number of classes

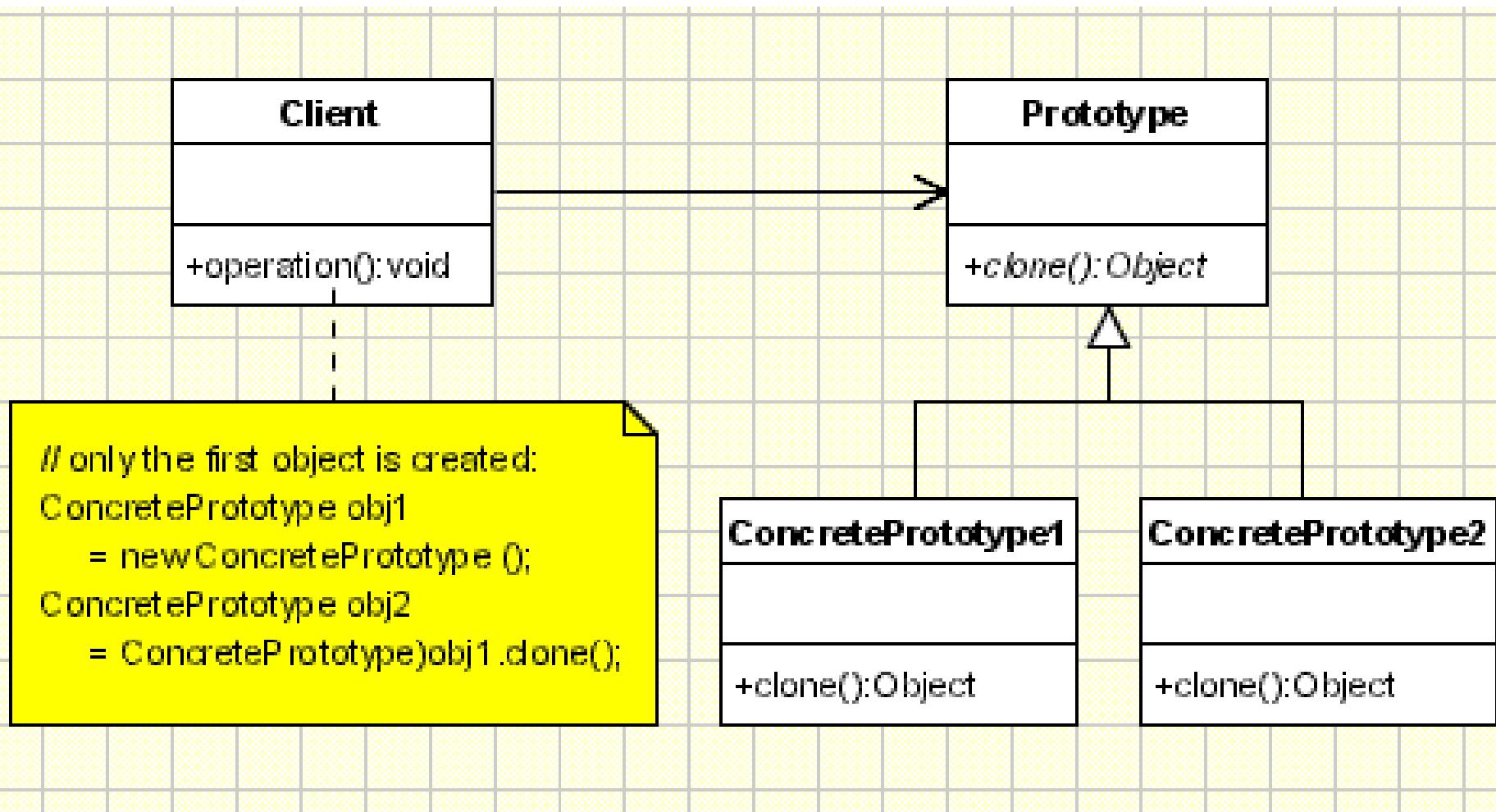


Prototype 3

- ▶ **Applicability** – Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; *and*
 - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
 - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
 - when instances of a class can have one of only a few different combinations of state

CP – Prototype

▶ Prototype – Music editor, Clonable

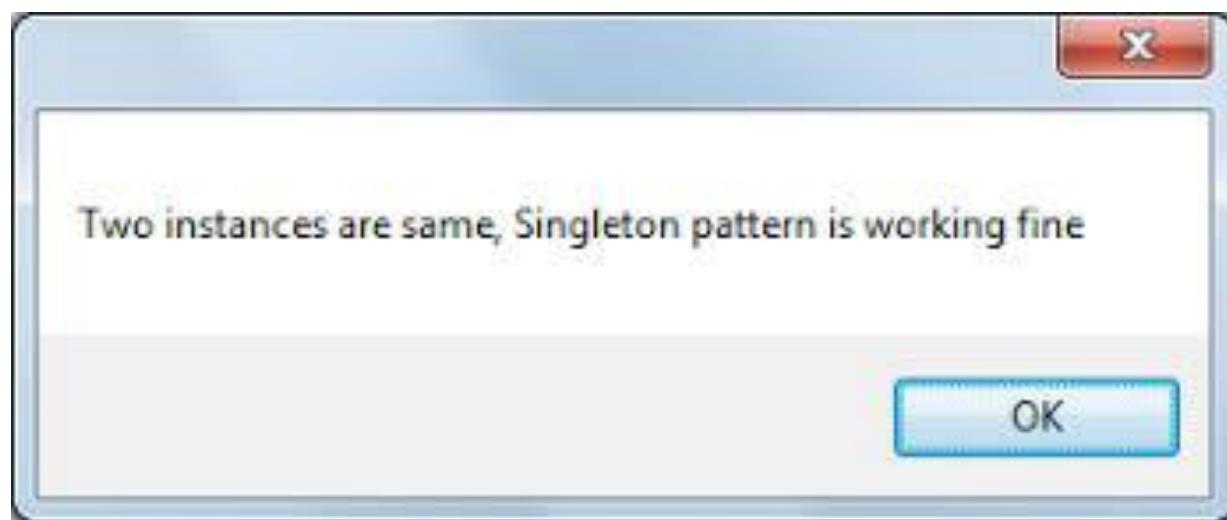


Prototype - The Good, The Bad ...

- ▶ Copy objects without using the concrete class
- ▶ Skip repeated initializations
- ▶ Quickly create complex objects
- ▶ It is very hard to clone complex classes with many references

Creational Patterns – Singleton

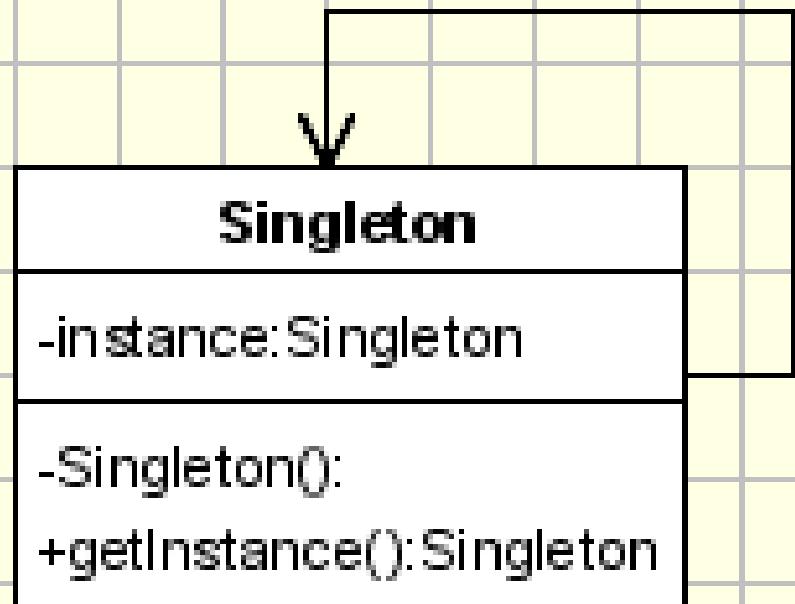
- ▶ **Intent** – Ensure a class only has one instance, and provide a global point of access to it
- ▶ **Motivation** – It's important for some classes to have exactly one instance. There should be only one file system and one window manager. An accounting system will be dedicated to serving one company.
- ▶ How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.



CP – Singleton

▶ Singleton – Logger

cd: Singleton Implementation- UML Class diagram



Singleton - The Good, The Bad ...

- ▶ Guarantees a single instance
- ▶ Global access point to an instance
- ▶ Breaks S (from SOLID) – the pattern has TWO responsibilities
- ▶ Can mask high coupling or other bad design
- ▶ Is difficult to use in multi thread systems

CP - Lazy initialization

- ▶ **Lazy initialization** is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed
- ▶ How? With a **flag**
- ▶ Each time when the object is called, the **flag** is tested:
 - If it is **ready**, it is **returned**
 - If not, it is **initialized**

Lazy initialization – Java 1

```
public class Fruit {  
    private static final Map<String,Fruit> types = new  
        HashMap<String, Fruit>();  
    private final String type;  
  
    //using a private constructor to force use of the factory method  
    private Fruit(String type) { this.type = type; }  
  
    public static synchronized Fruit getFruit(String type) {  
        if(!types.containsKey(type)) {  
            types.put(type, new Fruit(type)); // Lazy initialization  
        }  
        return types.get(type);  
    }  
}
```

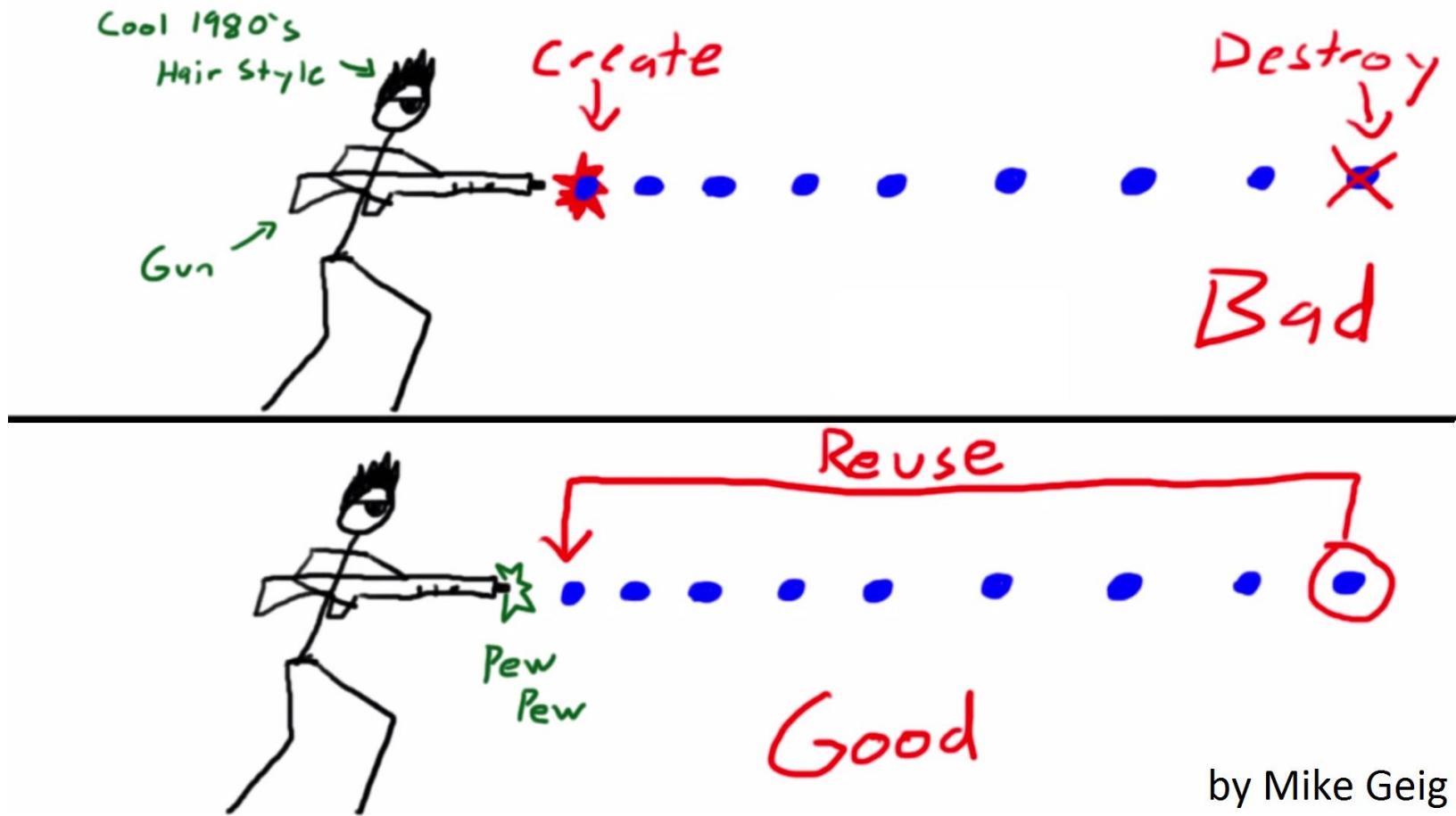
Lazy Initialization- The Good, The Bad ...

- ▶ Saves on memory, since it instantiates only as needed
- ▶ More time consuming in case you need many new instances

CP - Object Pool

- ▶ **Intent:** reuse and share objects that are expensive to create.
- ▶ **Motivation:** Performance can be sometimes the key issue during the software development and **the object creation (class instantiation) is a costly step.** The Object Pool pattern offer a mechanism to reuse objects that are expensive to create
- ▶ **Why use it?** Basically, we'll use an object pool whenever there are several clients who needs **the same stateless resource which is expensive to create**

Visual Example of Object Pooling

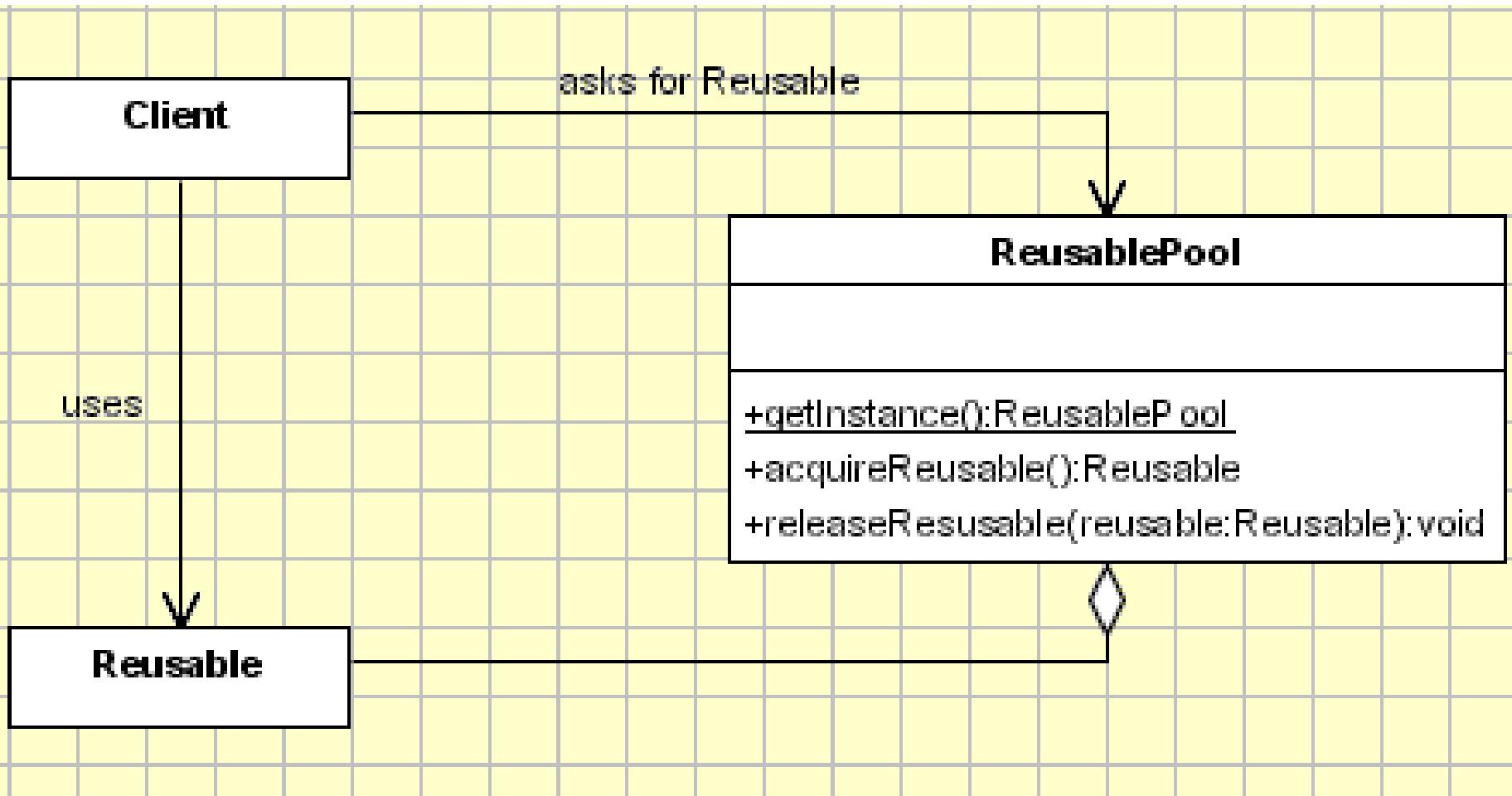


by Mike Geig

Object Pool – Implementation (1)

- ▶ When a client asks for a Reusable object, the pool performs the following actions:
 - Search for an available Reusable object and if it was found it will be returned to the client.
 - If no Reusable object was found then it tries to create a new one. If this actions succeeds the new Reusable object will be returned to the client.
 - If the pool was unable to create a new Reusable, the pool will wait until a reusable object will be released.

Object Pool - Implementation (2)



Object Pool – Applicability

- ▶ Lets' take the example of the **database connections**. It's obviously that opening too many connections might affect the performance:
 - **Creating a connection is an expensive operation**
 - When there are **too many connections opened it takes longer to create a new one** and the database server will become overloaded
- ▶ Here the **object pool comes to manage the connections** and provide a way to reuse and share them. It can also limit the maximum number of objects that can be created

Object Pool- The Good, The Bad ...

- ▶ Saves on computing resources, since it only allows for a set number of instances
- ▶ Increased overhead in terms of managing the pool
- ▶ Slow for the objects in the waiting queue

Structural Patterns 1

- ▶ Structural patterns are concerned with how classes and objects are composed to form larger structures
- ▶ Example: consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Structural Patterns 2

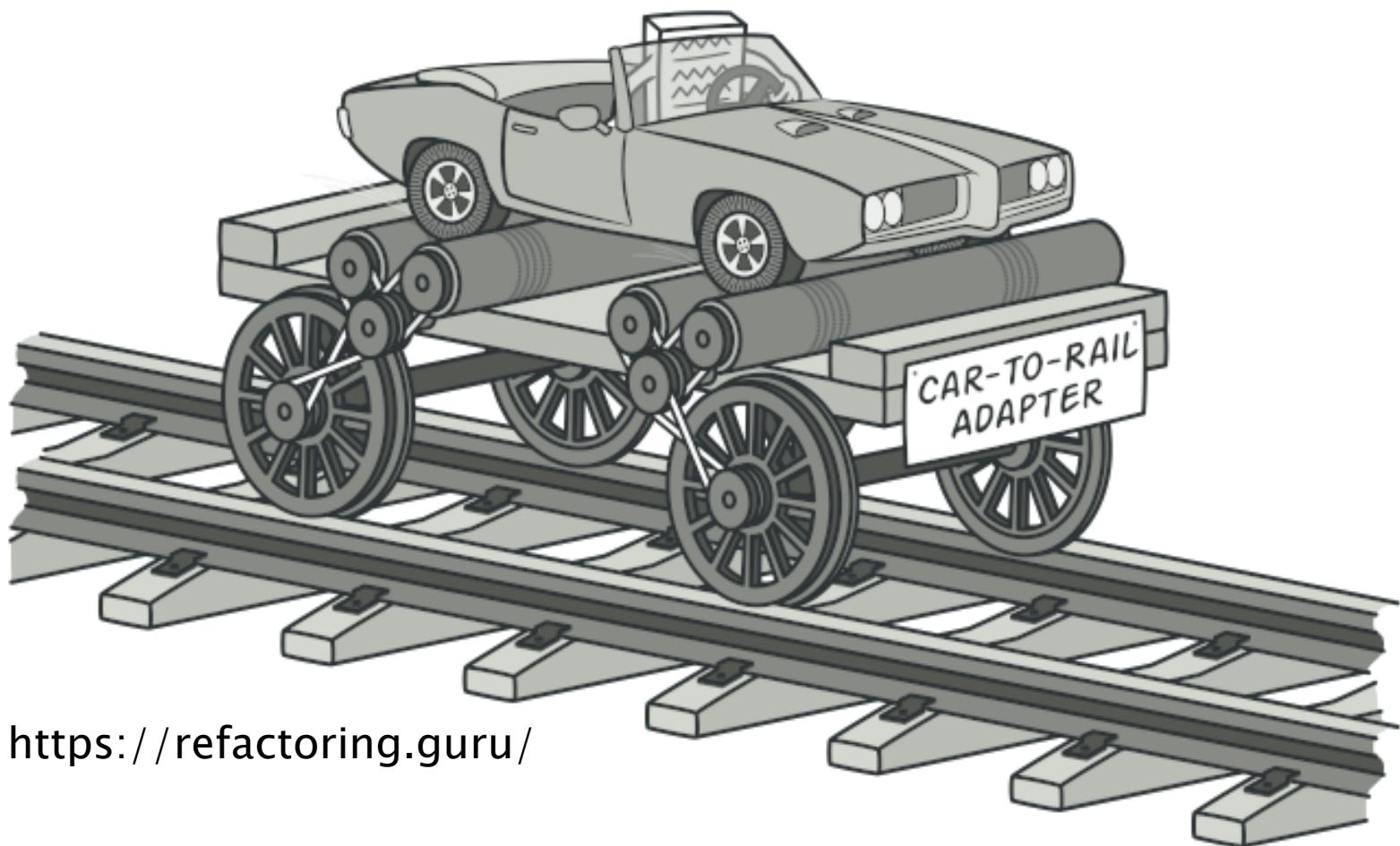
- ▶ Rather than composing interfaces or implementations, structural *object patterns* describe ways to compose objects to realize new functionality
- ▶ The added **flexibility of object composition** comes from the ability to change the composition at run-time

Structural Patterns

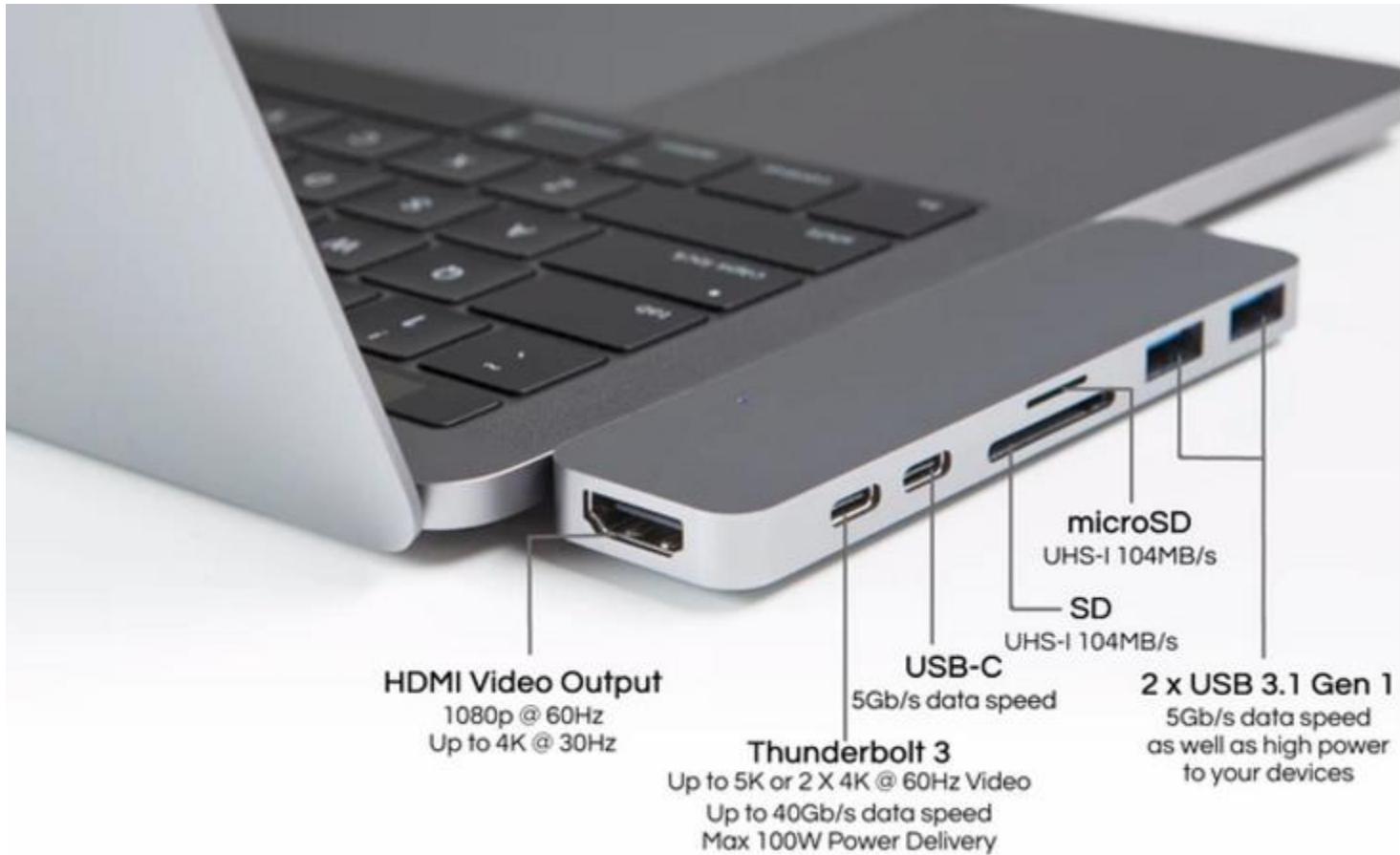
- ▶ Adapter
- ▶ Bridge
- ▶ Composite
- ▶ Decorator
- ▶ Façade
- ▶ Flyweight
- ▶ Proxy

Structural Patterns – Adapter

- ▶ **Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- ▶ **Also Known As:** Wrapper
- ▶ **Motivation:** Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

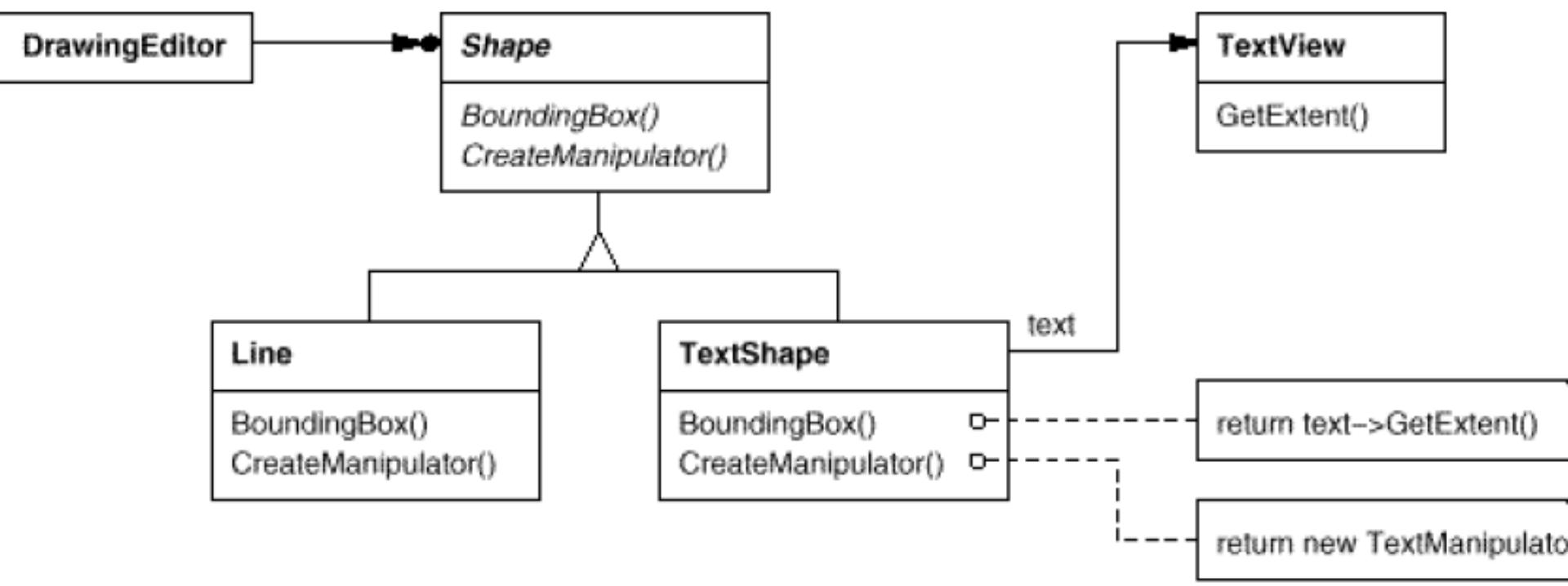


<https://refactoring.guru/>



Adapter 1

- ▶ A TextShape subclass that can display and edit text is considerably more difficult to implement
- ▶ We suppose the existence of a TextView class for displaying and editing text. So we can consider TextShape derived from these classes

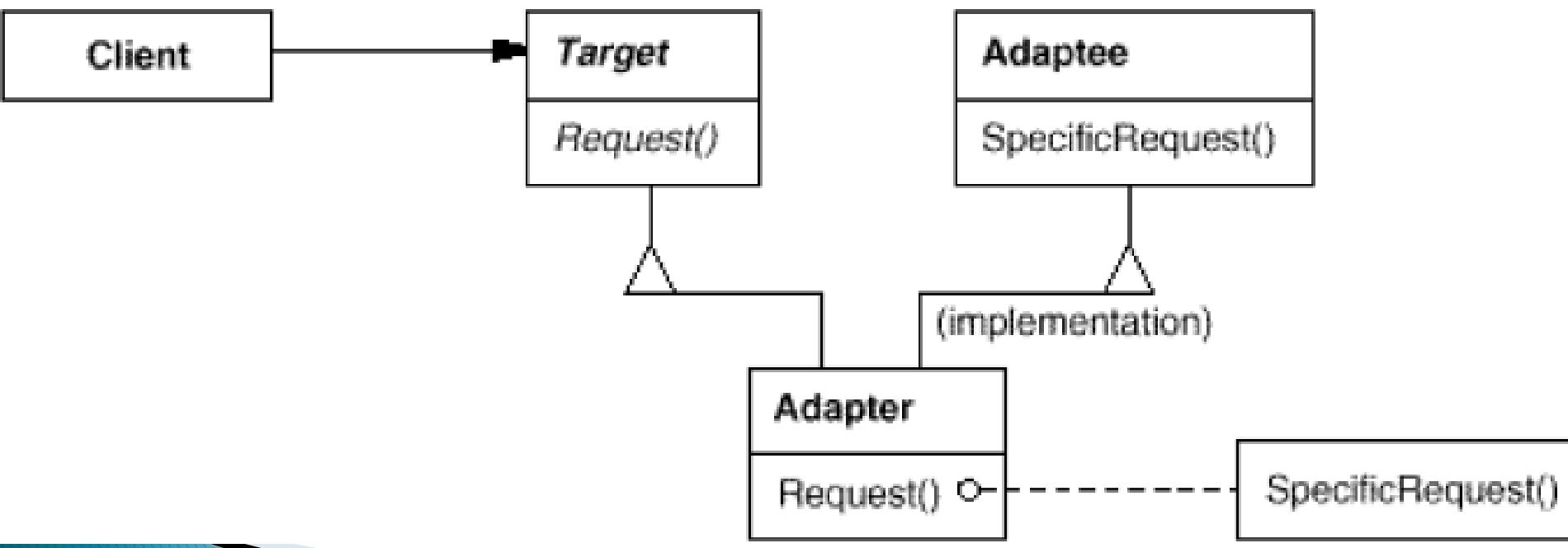


Adapter 2

- ▶ **Applicability** – Use the Adapter pattern when
 - you want to use an existing class, and its interface does not match the one you need
 - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
 - (*object adapter only*) *you need to use several existing subclasses, but it's impractical to adapt their interface by sub classing every one.*

Adapter 3

- Structure – The Adapter pattern is used so that two unrelated interfaces can work together. This is something like we convert interface of one class into interface expected by the client.



Adapter – Example

- ▶ We all have electric sockets in our houses of different sizes and shapes. I will take an example of a socket of 15 Ampere. This is a bigger socket and the other one which is smaller is of 5 Ampere. A 15 Amp plug cannot fit into a 5 Amp socket => we will use an Adapter
- ▶ We have a 5 Amp plug and want a 5 Amp socket so that it can work. We DO NOT have a 5 Amp socket, what we have is a 15 Amp socket in which the 5 Amp plug cannot fit. The problem is how to cater to the client without changing the plug or socket.

Adapter - Java 1

```
/** The socket class has a specs for 15 AMP.*/
public interface Socket {
    public String getOutput();
}
```

```
public class Plug {
    private String specification = "5 AMP";
    public String getInput() {
        return specification;
}
}
```

Adapter - Java 2

```
public class ConnectorAdapter implements Socket
{
    public String getOutput() {
        Plug plug = new Plug();
        String output = plug.getInput();
        return output;
    }
}
```

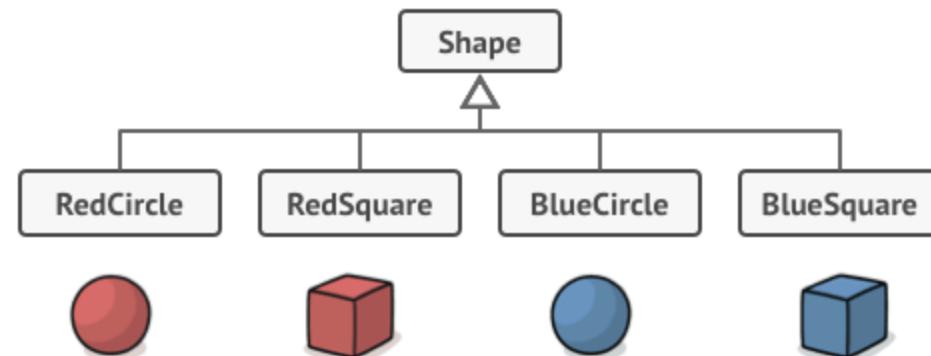
Adapter - The Good, The Bad ...

- ▶ Preserves S and O (from SOLID)
- ▶ Code becomes complicated because of extra classes

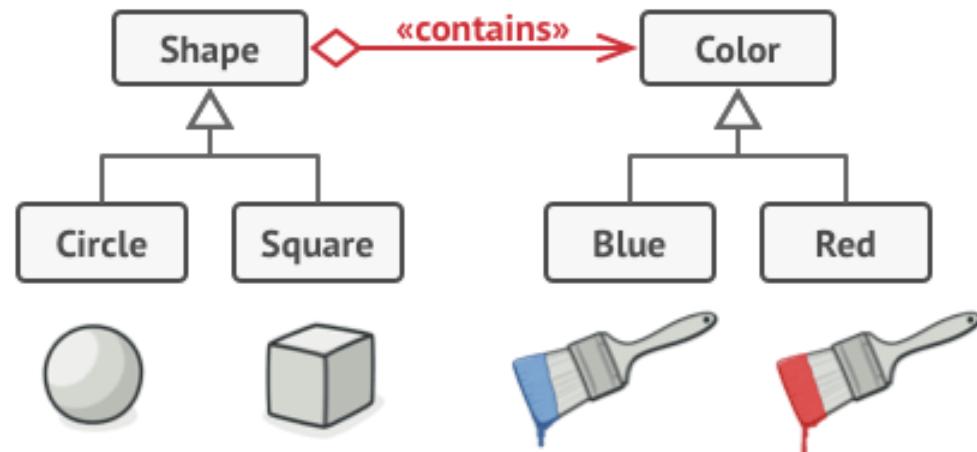
Structural Patterns – Bridge

- ▶ **Intent** – Decouple an abstraction from its implementation so that the two can vary independently
- ▶ **Also Known As** – Handle/Body
- ▶ **Motivation** – Consider the abstraction of shapes, each with its own properties. One thing all shapes can do is draw themselves. Drawing graphics to a screen can be dependent on different graphics implementations or operating systems. Shapes have to be able to be drawn on many types of operating systems

Shape + +



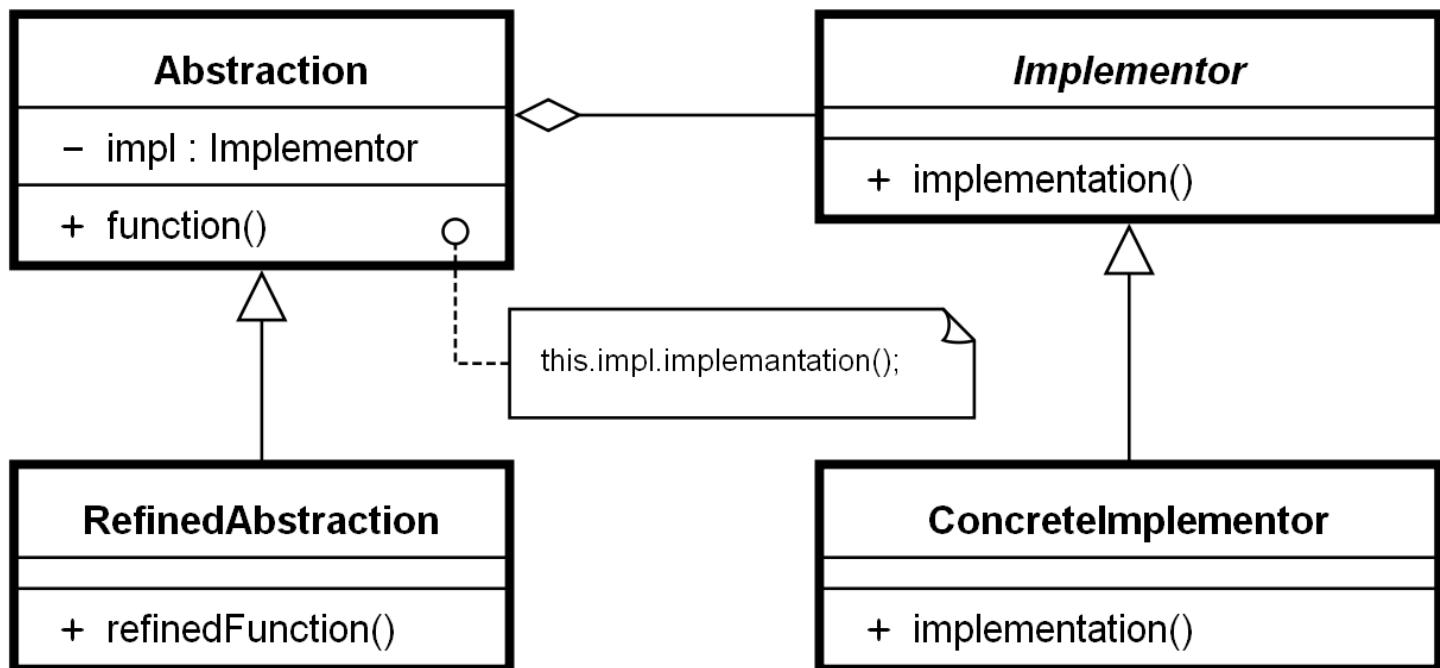
Vs.



<https://refactoring.guru/>

Bridge 1

- The bridge helps by allowing the creation of new implementation classes that provide the drawing implementation. Shape class provides methods for getting the size or properties of a shape. Drawing class provides an interface for drawing graphics



Bridge 2

- ▶ **Applicability** – When dividing large classes with many versions of the same feature (e.g. multiple types of network connections)
- ▶ When developing and extending classes in multiple and independent directions

Bridge - Java 1

```
interface DrawingAPI {/** "Implementor" */
    public void drawCircle(double x, double y, double
radius);
}
/** "ConcreteImplementor" 1,2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double
radius) { System.out.printf("API1" + x + y + radius);}
}
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double
radius) { System.out.printf("API2" + x + y + radius);}
}
```

Bridge - Java 2

```
interface Shape {/** "Abstraction" */
    public void draw();
    public void resizeByPercentage(double pct); }

/** "Refined Abstraction" */
class CircleShape implements Shape {
    private double x, y, radius;
    private DrawingAPI drawingAPI;
    public CircleShape(double x, double y, double radius,
        DrawingAPI drawingAPI) {
        this.x = x; this.y = y; this.radius = radius;
        this.drawingAPI = drawingAPI;
    }
    public void draw() {drawingAPI.drawCircle(x, y, radius); }
    public void resizeByPercentage(double pct) { radius *= pct; }
}
```

Bridge - Java 3

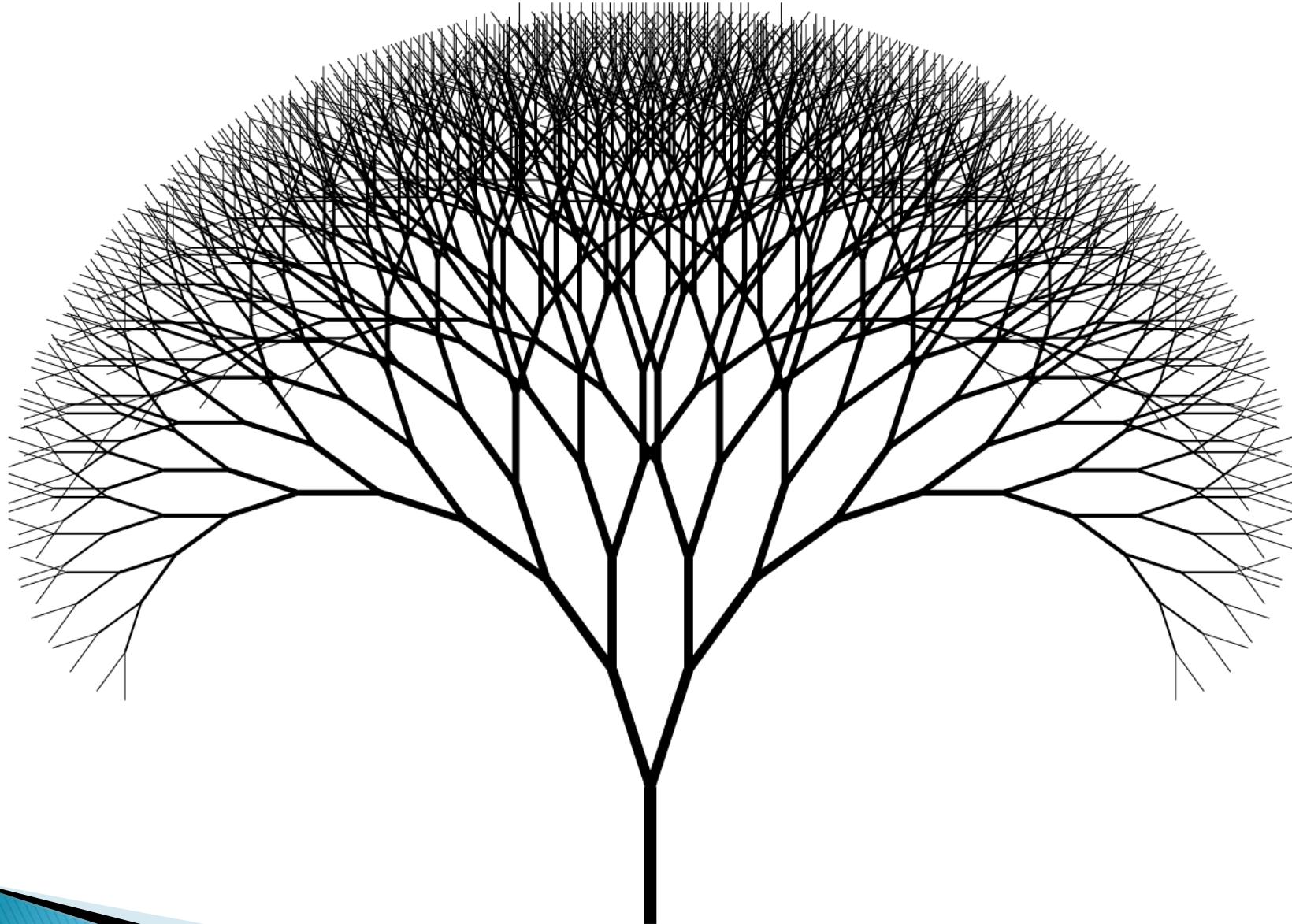
```
/** "Client" */
class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[2];
        shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
        shapes[1] = new CircleShape(5, 7, 8, new DrawingAPI2());
        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}
```

Bridge - The Good, The Bad ...

- ▶ Preserves S and O (from SOLID)
- ▶ Allows for platform independent systems
- ▶ High level connections
- ▶ Code becomes complicated and splits cohesive classes

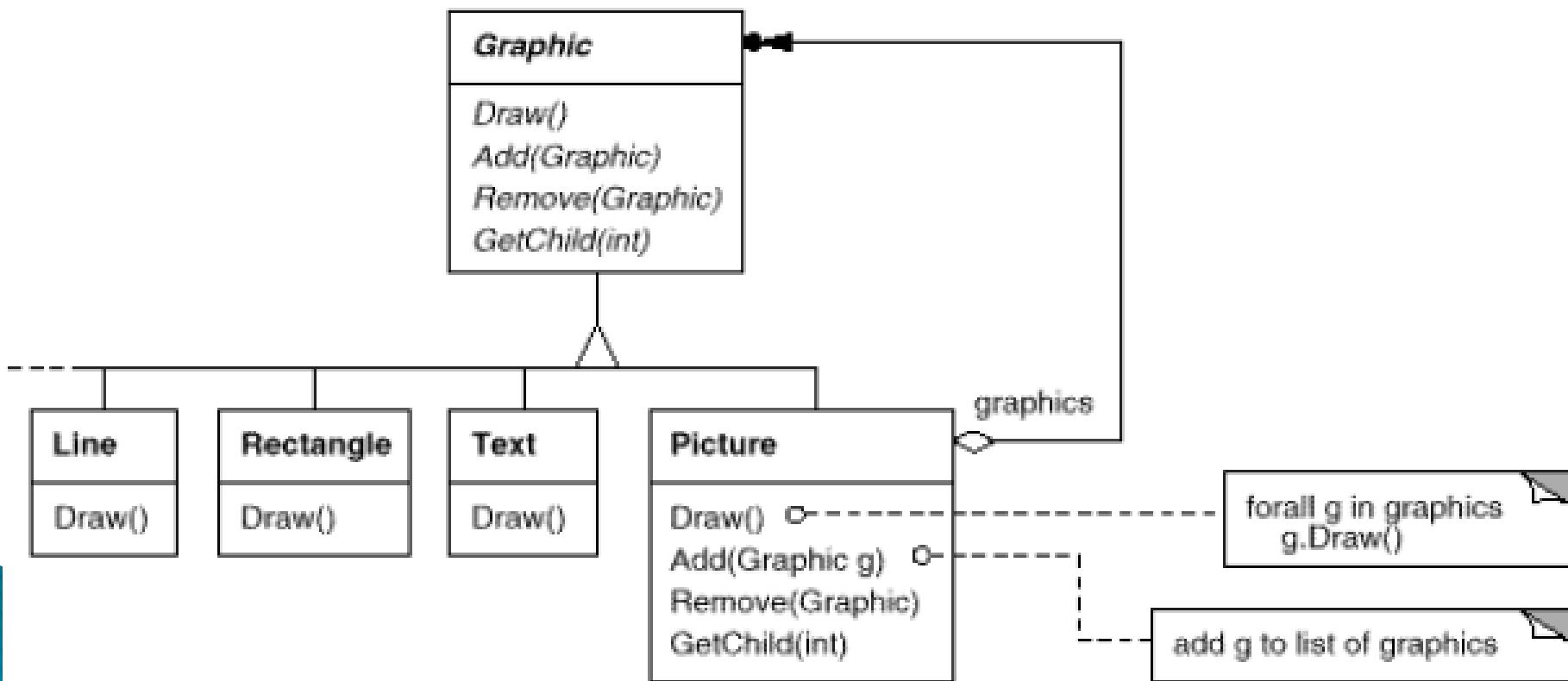
Structural Patterns – Composite

- ▶ **Intent** – Compose objects into tree structures to represent part–whole hierarchies. **Composite** lets clients treat individual objects and compositions of objects uniformly
- ▶ **Motivation** – Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically.



Composite 1

- ▶ Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components

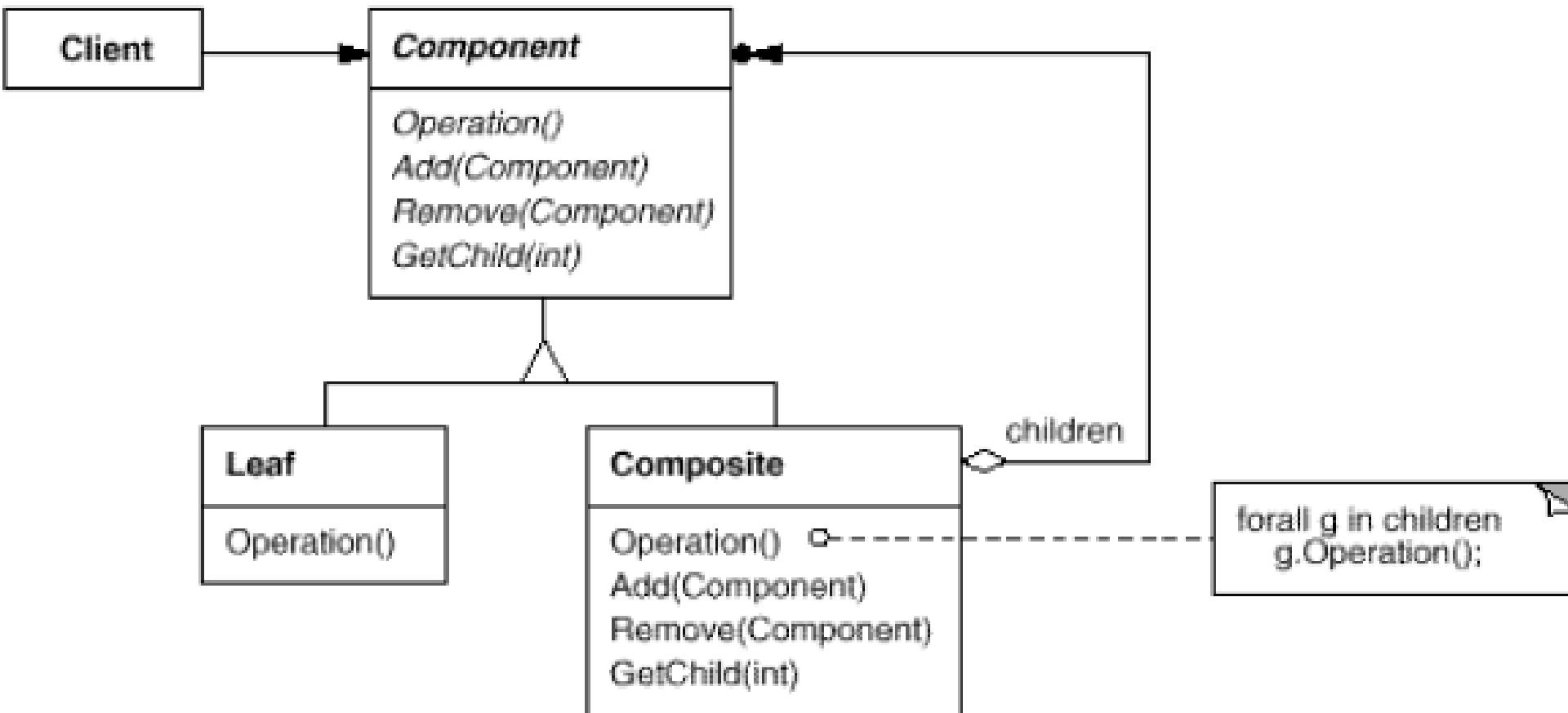


Composite 2

- ▶ **Applicability** – Use this pattern when
 - you want to represent part–whole hierarchies of objects
 - you want clients to be able to ignore the difference between compositions of objects and individual objects
 - Clients will treat all objects in the composite structure uniformly

Composite 3

▶ Structure



Composite – Example

- ▶ The most common example in this pattern is of a company's employee hierarchy
- ▶ The employees of a company are at various positions. Now, say in a hierarchy, the manager has subordinates; also the Project Leader has subordinates, i.e. employees reporting to him/her. The developer has no subordinates

Composite - Java 1

```
public class Employee {  
    private String name; private double salary;  
    private Vector subordinates;  
  
    public Vector getSubordinates() {return subordinates;}  
    public void setSubordinates(Vector subordinates) {  
        this.subordinates = subordinates;}  
  
    public Employee(String name, double sal) {  
        setName(name);setSalary(sal);  
        subordinates = new Vector();  
    }  
    public void add(Employee e) {subordinates.addElement(e);}  
    public void remove(Employee e) {subordinates.remove(e);}  
}
```

Composite - Java 2

```
private void addEmployeesToTree() {  
    Employee CFO = new Employee("CFO", 3000);  
    Employee headFinance1 = new Employee("HF. North", 2000);  
    Employee headFinance2 = new Employee("HF. West", 2200);  
    Employee accountant1 = new Employee("Accountant1", 1000);  
    Employee accountant2 = new Employee("Accountant2", 900);  
    Employee accountant3 = new Employee("Accountant3", 1100);  
    Employee accountant4 = new Employee("Accountant4", 1200);  
  
    CFO.add(headFinance1); CFO.add(headFinance2);  
    headFinance1.add(accountant1);headFinance1.add(accountant4);  
    headFinance2.add(accountant2);headFinance2.add(accountant3);  
}
```

CFO = chief financial officer

Composite - Java 3

- Once we have filled the tree up, now we can get the tree for any employee and find out whether that employee has subordinates with the following condition.

```
Vector subordinates = emp.getSubordinates();
if (subordinates.size() != 0)
    getTree(subordinates);
else
    System.out.println("No Subordinates for the
Employee: "+emp.getName());
```

Composite - The Good, The Bad ...

- ▶ Preserves O (from SOLID), because new classes can easily be added to the hierarchy
- ▶ Using polymorphism and inheritance works well for complex tree structures
- ▶ Prone to overgeneralization
- ▶ Difficult to provide a common interface

Structural Patterns – Decorator

- ▶ **Intent** – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- ▶ **Also Known As** – Wrapper (*similar Adapter*)
- ▶ **Motivation** – Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component



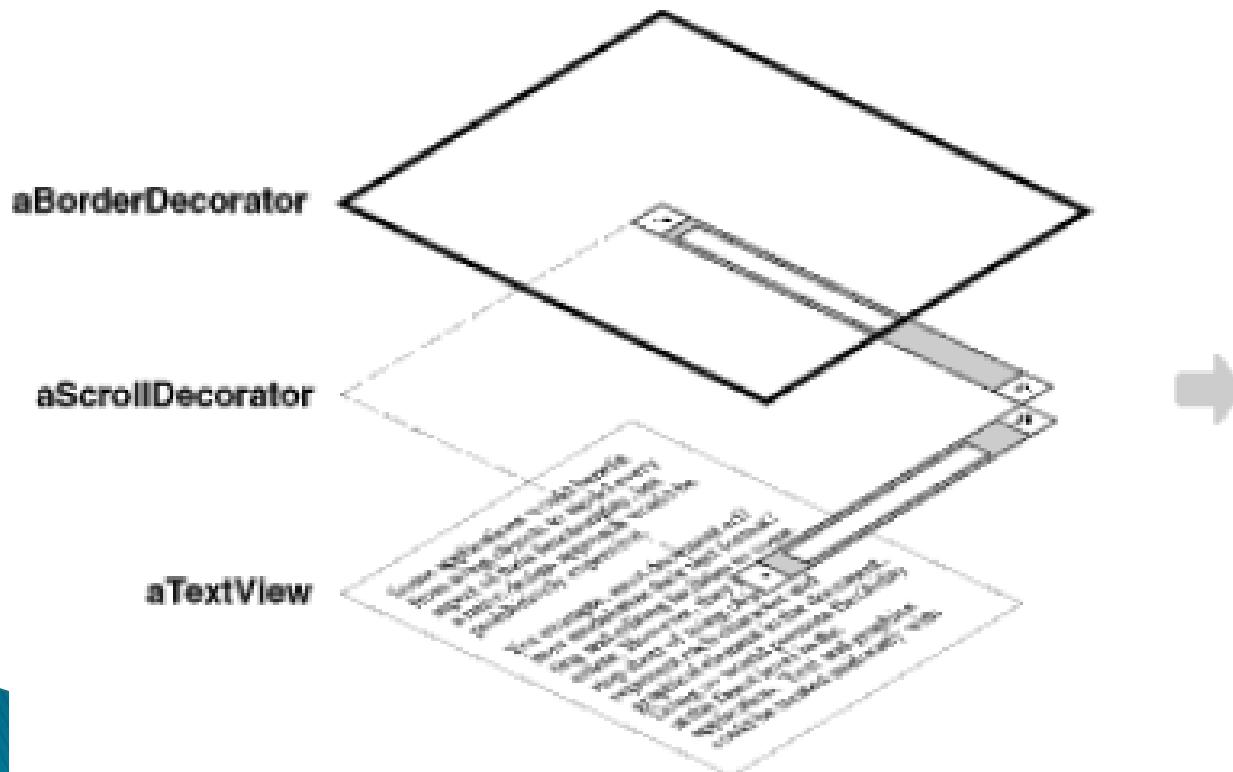
Component

Decorators



Decorator 1

- ▶ A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**

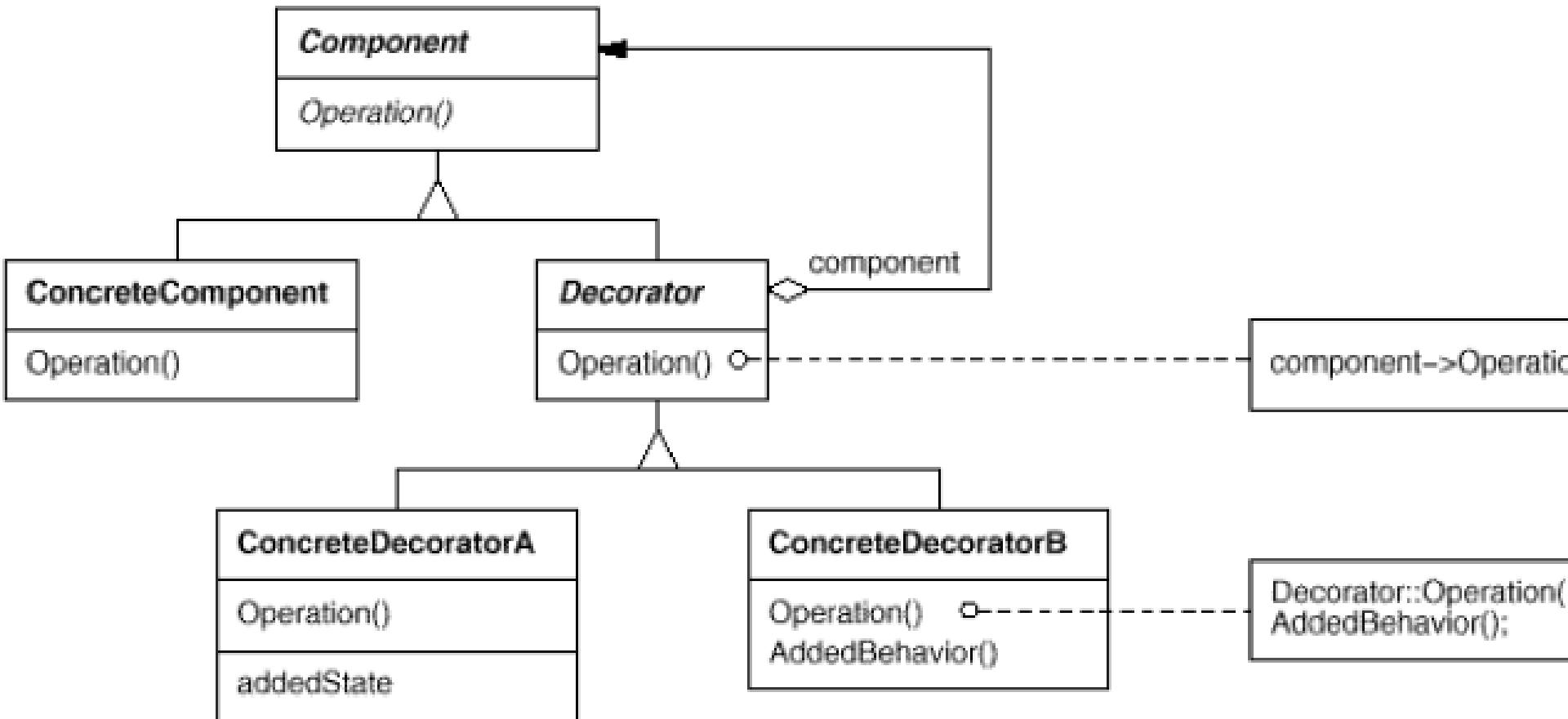


Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Decorator 2

▶ Structure



Decorator 3

▶ Applicability – Use Decorator

- to add responsibilities to individual objects dynamically and transparently
- for responsibilities that can be withdrawn
- when extension by subclassing is impractical

Decorator – Example

- ▶ Suppose we have some 6 objects and 2 of them need a special behavior, we can do this with the help of a decorator
- ▶ Let's take an example of a **Christmas tree**. There is a need to decorate a Christmas tree. Now we have many branches which need to be decorated in different ways

Decorator - Java 1

```
public abstract class Decorator {  
    /** The method places each decorative item on  
     * the tree. */  
    public abstract void place(Branch branch);  
}  
  
public class ChristmasTree {  
    private Branch branch;  
    public Branch getBranch() {  
        return branch;  
    }  
}
```

Decorator - Java 2

```
public class BallDecorator extends Decorator {  
    public BallDecorator(ChristmasTree tree) {  
        Branch branch = tree.getBranch();  
        place(branch);  
    }  
  
    public void place(Branch branch) {  
        branch.put("ball");  
    }  
}
```

Decorator – Java 3

- ▶ Similarly, we can make StarDecorator and RufflesDecorator

```
StarDecorator decorator = new StarDecorator(new  
ChristmasTree());
```

- ▶ This way the decorator will be instantiated and a branch of the Christmas tree will be decorated.

Decorator - The Good, The Bad ...

- ▶ Change object behavior without inheritance
- ▶ Preserves S (from SOLID) – a large class can be split in smaller ones
- ▶ Can add multiple behaviors by adding multiple decorations
- ▶ Change an object's responsibilities at runtime
- ▶ The sequence of applying decorators matters
- ▶ Difficult to remove a decorator from the middle of the pile
- ▶ The code is hard to manage

Bibliografie

- ▶ Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)

Links

- ▶ Design Patterns: <http://www.odesign.com/>
- ▶ Gang-Of-Four: <http://c2.com/cgi/wiki?GangOfFour>,
<http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf>
- ▶ Design Patterns Book: <http://c2.com/cgi/wiki?DesignPatternsBook>
- ▶ About Design Patterns: <http://www.javacamp.org/designPattern/>
- ▶ Design Patterns – Java companion:
<http://www.patterndepot.com/put/8/JavaPatterns.htm>
- ▶ Java Design patterns:
http://www.allapplabs.com/java_design_patterns/java_design_patterns.htm
- ▶ Overview of Design Patterns:
http://www.mindspring.com/~mgrand/pattern_synopses.htm
- ▶ Lazy initialization: http://en.wikipedia.org/wiki/Lazy_initialization
- ▶ Use Lazy Initialization to Conserve Resources:
<http://www.devx.com/tips/Tip/18007>
- ▶ Python design patterns <https://refactoring.guru/design-patterns>