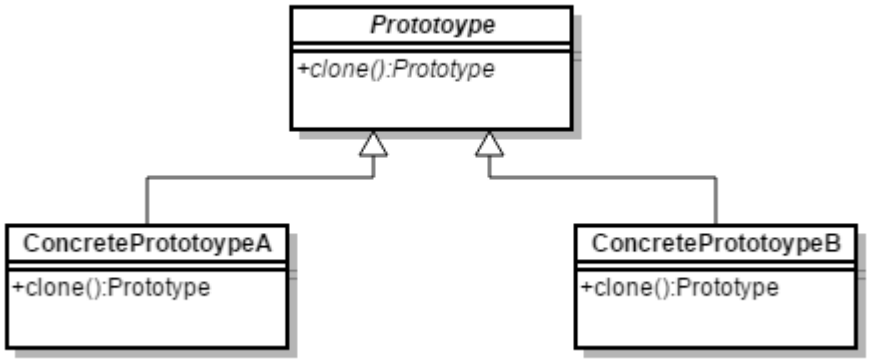
Design Patterns Part 2

by Vlad Ungureanu

Agenda

Design Patterns

- Prototype
- Adapter
- Bridge
- Composite


Prototoype
+clone():Prototype
ConcretePrototoypeA
+clone():Prototype
ConcretePrototoypeB
+clone():Prototype

Prototype - Implementation

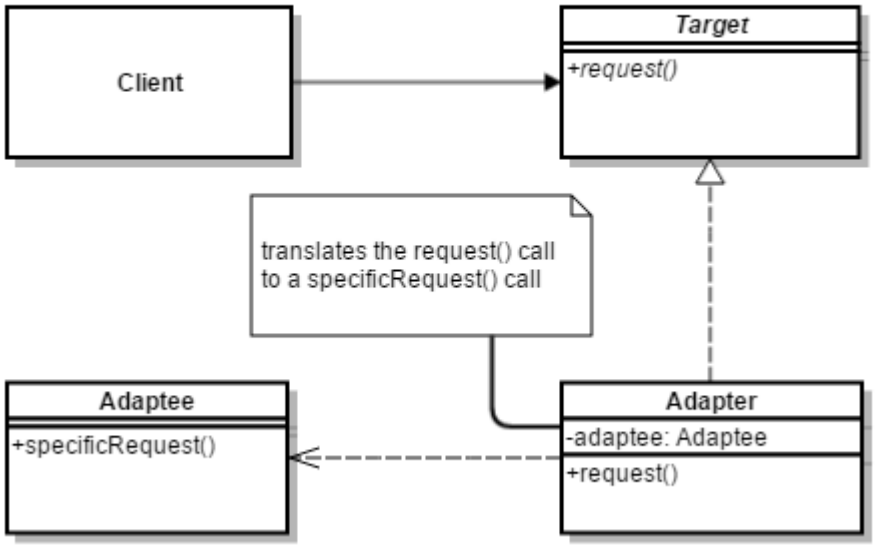
```
public abstract class Prototype implements Cloneable {
        public abstract Prototype instance();
}
public class ComplexPrototype extends Prototype {

      @Override
      public Prototype instance() {
            try {
                  return (Prototype) super.clone();
            } catch (CloneNotSupportedException e) {
                  return null;
            }
      }
      public void complexProcess() {
        System.out.println("Complex Process!");
      }

}
```

Prototype – Why ?

- Avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
- Avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.
- Eliminates the (potentially expensive) overhead of initializing an object
- Simplifies and can optimize the use case where multiple objects of the same type will have mostly the same data

Prototype – Why not ?

- Requires a Prototype Manager for multiple prototypes in the same application
- Clones can be shallow or deep (includes instance variables); deep clones might require cloning prototype complex objects.
- Prototypes might require extra initialization after they were created; if this step is omitted we could get shallow clones


Client
Target
+request()
translates the request() call
to a specificRequest() call
Adaptee
+specificRequest()
Adapter
-adaptee: Adaptee
+request()

Adapter – Implementation

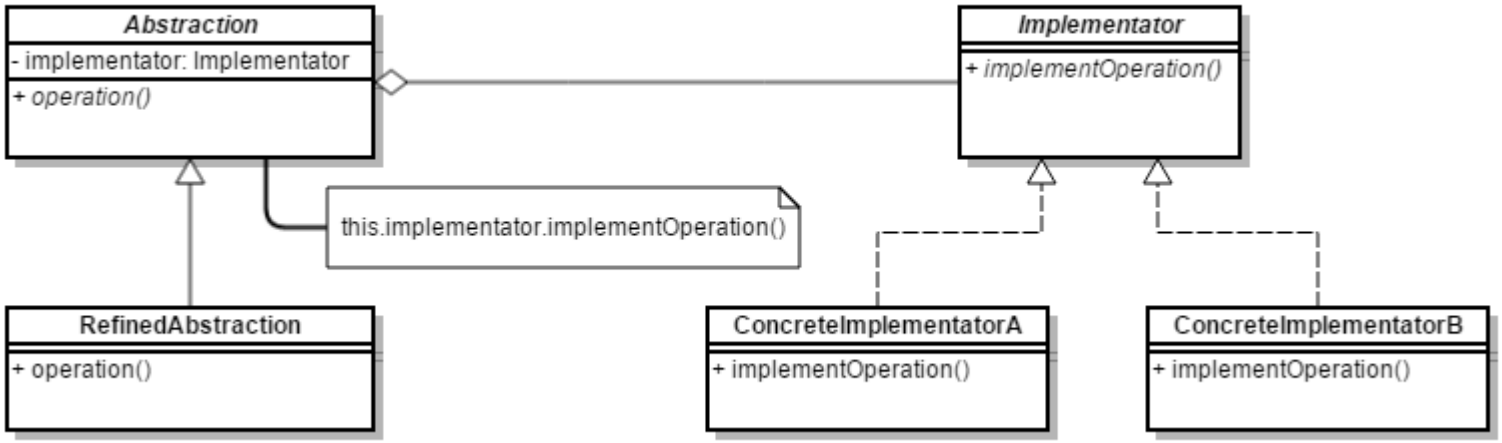
```
public interface ListPrinter {
        public void printList(List<String> stringList);
}
public class Printer {
      public void print(String value){
              System.out.println(value);
      }
}
public class Adapter implements ListPrinter {
      @Override
      public void printList(List<String> stringList) {
         String listString = "";
         for (String s : stringList) {
              listString += s + "\t";
         }
         Printer printer = new Printer();
         printer.print(listString);
      }
}
```

Adapter – Why ?

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system
- Reuse code instead of duplicating code or duplicating functionality

Adapter – Why not?

- Extra layer of abstraction as adapter usually makes two existing interfaces work together
- Adapters might encapsulate application logic or business logic
- If one of the elements that need to be adapted does not have an interface then the adaptation process might include any number of implementations making the adapter too generic
- Some implementations presume a lot of code duplication
- Sometimes several adapters might be required to link two functionalities


Abstraction
- implementator: Implementator
+ operation()
Implementator
+ implementOperation()
this.implementator.implementOperation()
RefinedAbstraction
+ operation()
ConcreteImplementatorA
+ implementOperation()
ConcreteImplementatorB
+ implementOperation()

Bridge – Implementation

```
public abstract class AbstractOperations {
        private Operations operations;

        public AbstractOperations(Operations operations){
          this.operations = operations;
        }

        public void turnOn(){
          this.operations.turnOn();
        }

        public void turnOff(){
          this.operations.turnOff();
        }
}

public interface Operations {
        void turnOn();
        void turnOff();
}
```

Bridge – Implementation

```
public class DeviceOneController  implements  Operations{
        @Override
        public void turnOn() {
        System.out.println("Type one - turn on!");
        }
        @Override
        public void turnOff() {
        System.out.println("Type one - turn off!");
        }
}
public class DeviceTwoController  implements  Operations{
        @Override
        public void turnOn() {
        System.out.println("Type two - turn on!");
        }
        @Override
        public void turnOff() {
        System.out.println("Type two - turn off!");
        }}
```

Bridge – Implementation

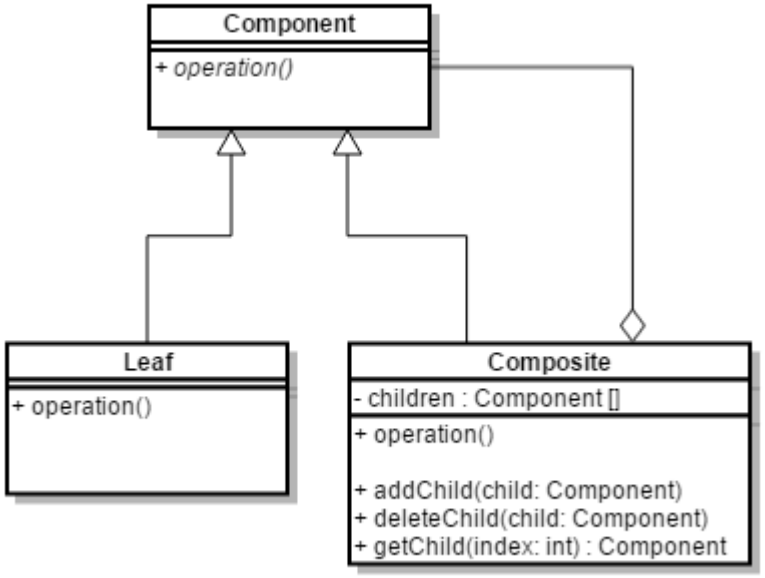
```
public static void main(String[] args) {
        // class with Device One operations
        Operations operationsOne = new DeviceOneController();
        Bridge bridgeOne = new Bridge(operationsOne);
        bridgeOne.turnOn();
        bridgeOne.turnOff();

        // same class with Device Two operations
        Operations operationsTwo = new DeviceTwoController();
        Bridge bridgeTwo = new Bridge(operationsTwo);
        bridgeTwo.turnOn();
        bridgeTwo.turnOff();
        }
}
```

Bridge – Why ?

- You want to hide implementation details (changes) from clients.
- Selection or switching of implementation is at run-time rather than design time.
- Abstraction and implementation should both be extensible by sub classing.
- You want to share an implementation among multiple objects and this should be hidden from the client.

Bridge – Why not ?

- Increases complexity.
- Double indirection – This will have a slight impact on performance. The abstraction needs to pass messages along to the implementer for the operation to get executed.
- Extra layer of abstraction
- Bridge needs to be included in the initial design


Component
+ operation()
Leaf
+ operation()
Composite
- children : Component []
+ operation()
+ addChild(child: Component)
+ deleteChild(child: Component)
+ getChild(index: int) : Component

Composite – Implementation

```
public interface Component {
        void work();
}

public class Composite {
      private List<Component> components = new ArrayList<>();
      public void add(Component component) {
        this.components.add(component);
      }
      public void remove(Component component) {
        this.components.remove(component);
      }
      public void work() {
        for (Component component : this.components) {
                component.work();
        }
      }
}
```

Composite – Implementation

```
public class ComponentOne implements Component{

    @Override
    public void work() {
      System.out.println("Work One!");
    }

}
public class ComponentTwo implements Component {

    @Override
    public void work() {
      System.out.println("Work Two!");
    }

}
```

Composite – Why ?

- Compose objects into tree structures to represent whole-part hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- Composite can be traversed with Iterator
- Composition over inheritance

Composite – Why not ?

- Composition over inheritance can lead to a general approach or to a too complex system of classes
- Once composition is defined adding new types of leaves is very difficult
- Specific behavior for leaf processing leads to all sorts of bad practices (if-else programming, spaghetti code, no clean code, code duplication)

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –