Software Design 2

Software Engineering
Prof. Maged Elaasar

Learning Objectives

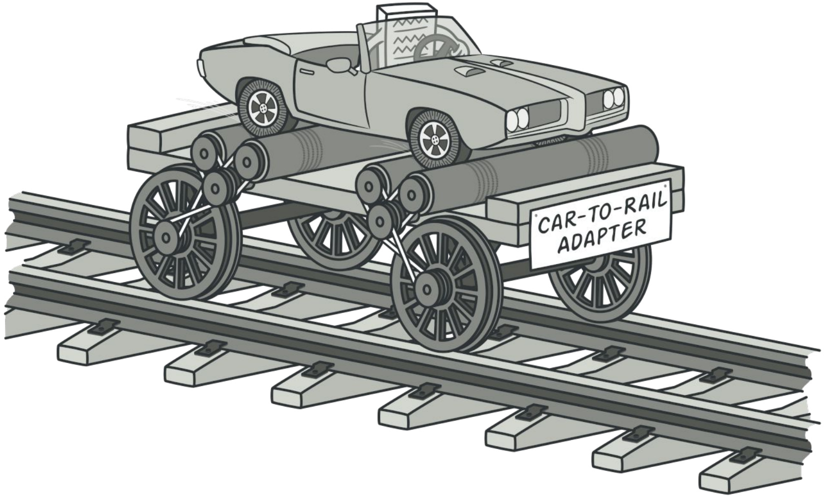
- GoF Structural Patterns
    - Adapter pattern
    - Facade pattern
    - Proxy pattern
    - Composite pattern

Adapter Pattern

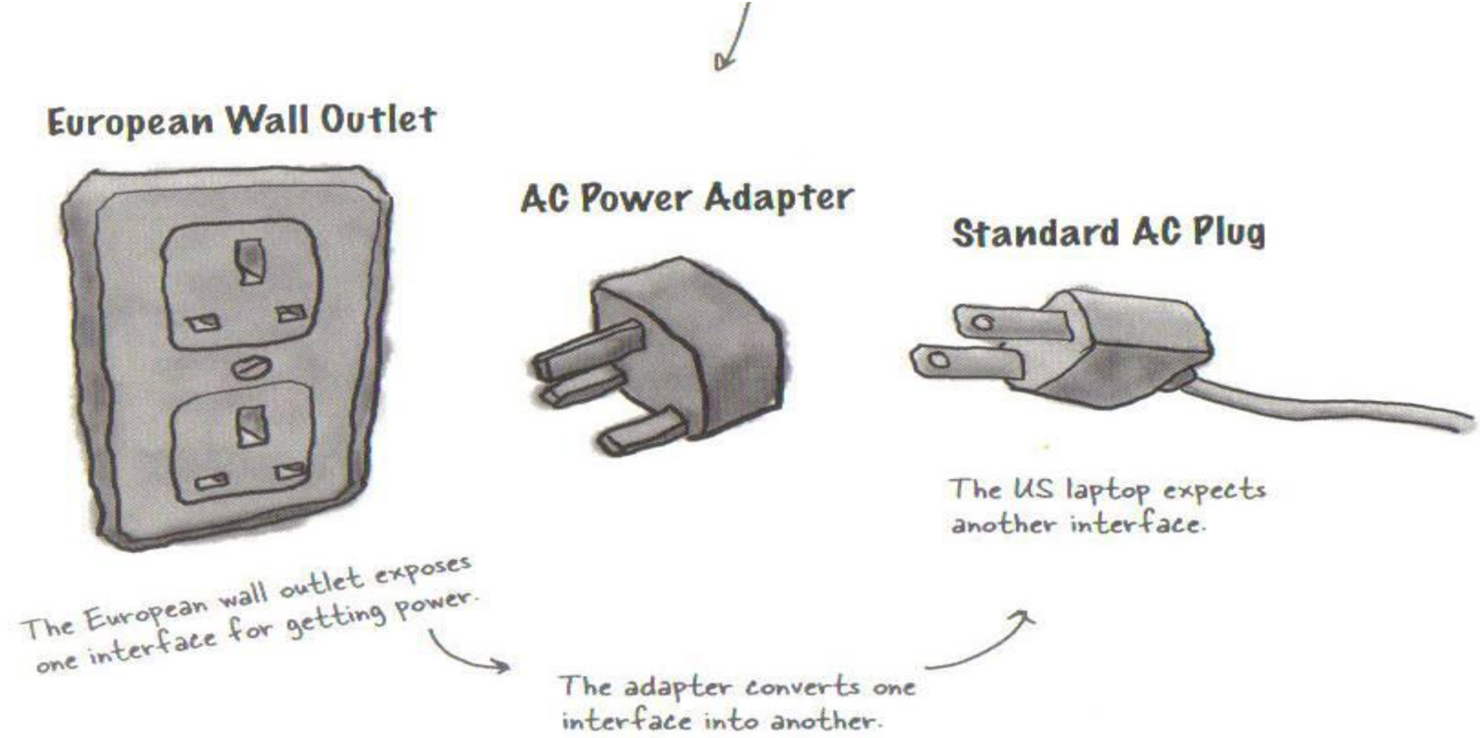
Adapter Pattern

Problem

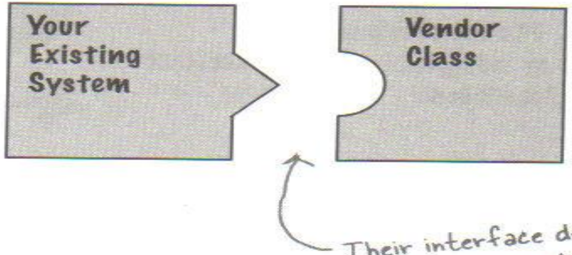
When an existing component is reused, but its interface is not compatible with the system that uses it.



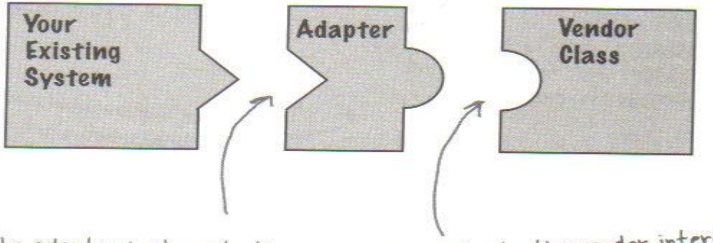
Motivation: Adapters in real life


European Wall Outlet
AC Power Adapter
Standard AC Plug
The US laptop expects another interface.
The European wall outlet exposes one interface for getting power.
The adapter converts one interface into another.

Motivation: Object-Oriented Adapters

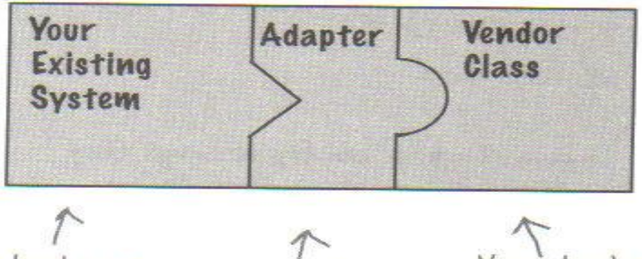


Their interface doesn't match the one yo
written your code against. This isn't goin



The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.

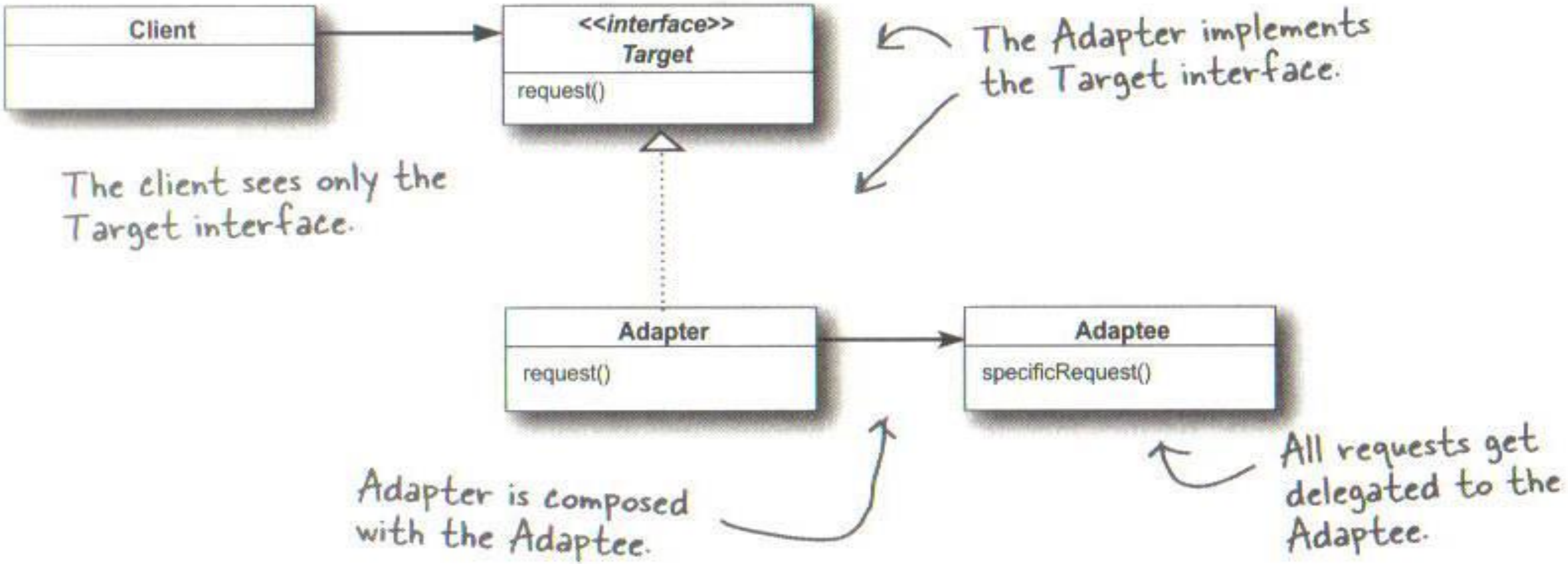


No code changes.

New code.

No code changes.

Adapter Pattern


Client
<<interface>>
Target
request()
Adapter
request()
Adaptee
specificRequest()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

Example: Duck and Turkey

```
interface Duck {
    public void quack();
    public void fly();
}

class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }
    public void fly() {
        System.out.println("I'm flying");
    }
}
```

```
interface Turkey {
    public void gobble();
    public void fly();
}

class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }
    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

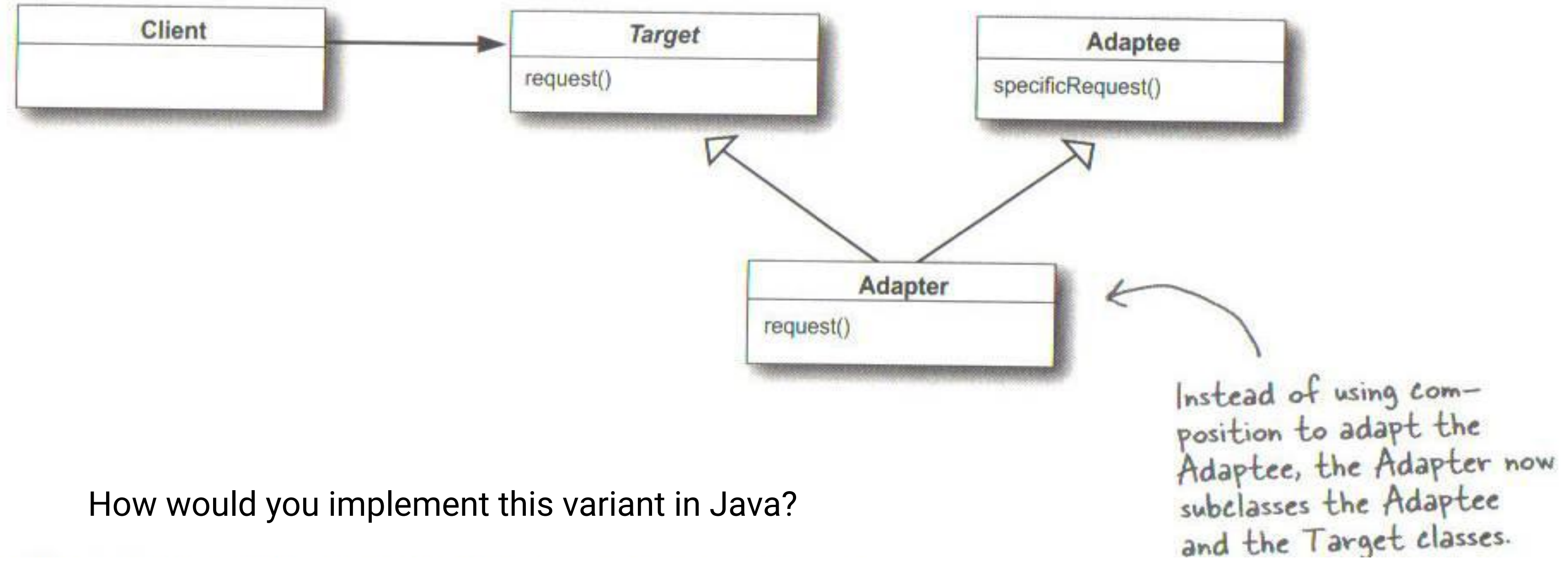
Example: Duck and Turkey

```
class TurkeyAdapter implements Duck {
    Turkey turkey;
    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }
    public void quack() {
        turkey.gobble();
    }
    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

```
class Farm { // Client
    public static void main (String args[]) {

        Turkey turkey = new Turkey();

        Duck duck = new TurkeyAdapter (turkey);

        duck.quack();

        duck.fly();

    }

}
```

Turkey adapter makes a turkey looks like a duck

Adapter variant



Instead of using composition to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.

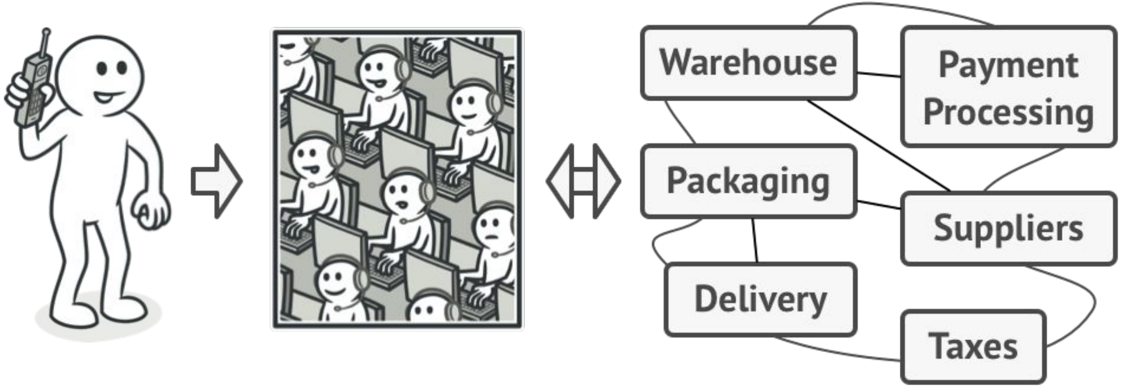
How would you implement this variant in Java?

Facade Pattern

Façade Pattern

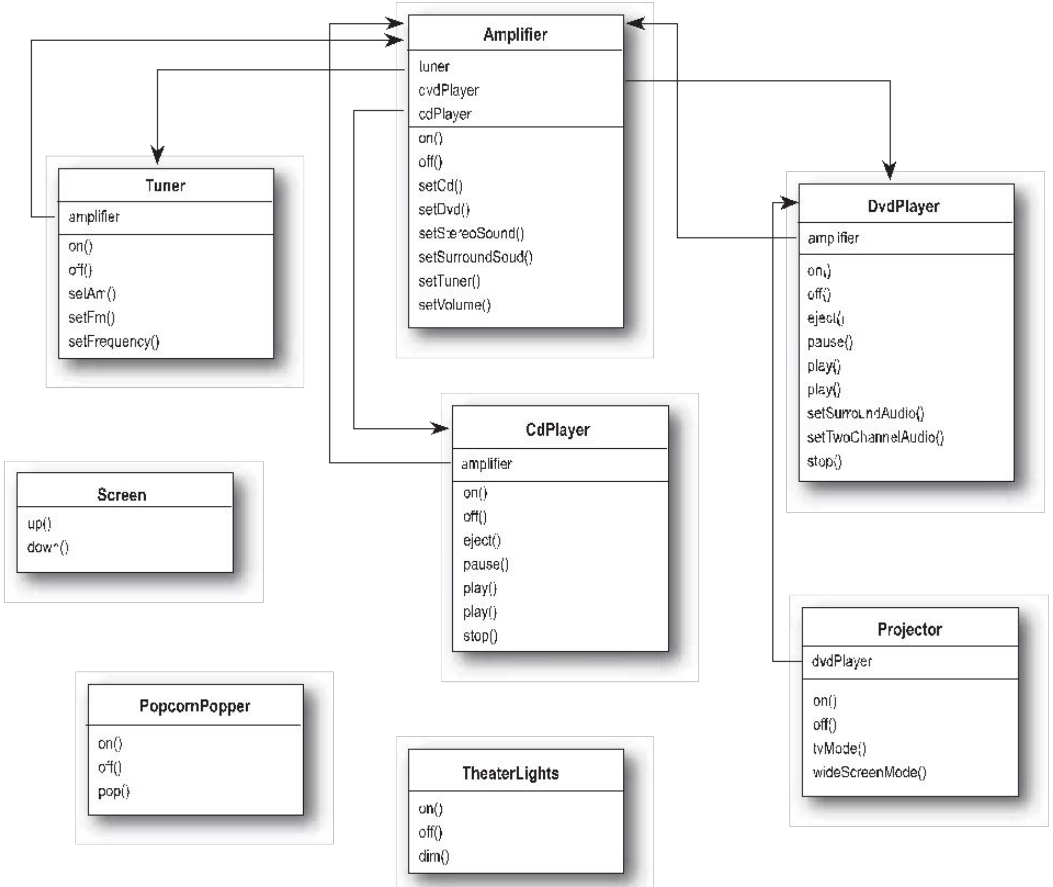
Problem

When a client needs a simplified interface to the overall functionality of a complex system.



Placing an order by phone

Motivation: Home Theater System



That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

Motivation: Using a Home Theatre System

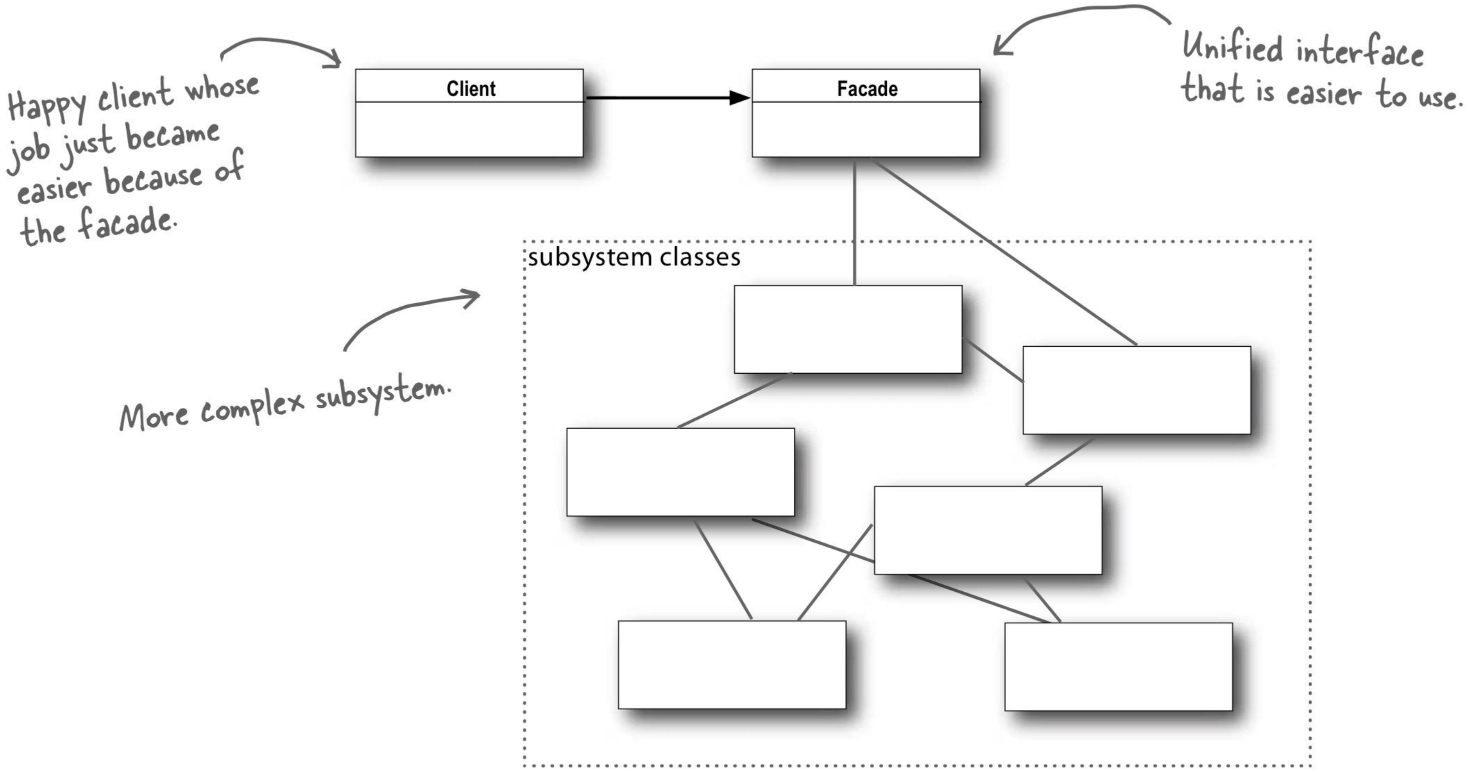
Sit back, relax, and…

1. Turn on the popcorn popper
2. Start the popper
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to DVD
7. Put the projector in wide-screen mode
8. Turn the sound amplifier on
9. Set the amplifier to DVD input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the DVD player on
13. Start the DVD player playing

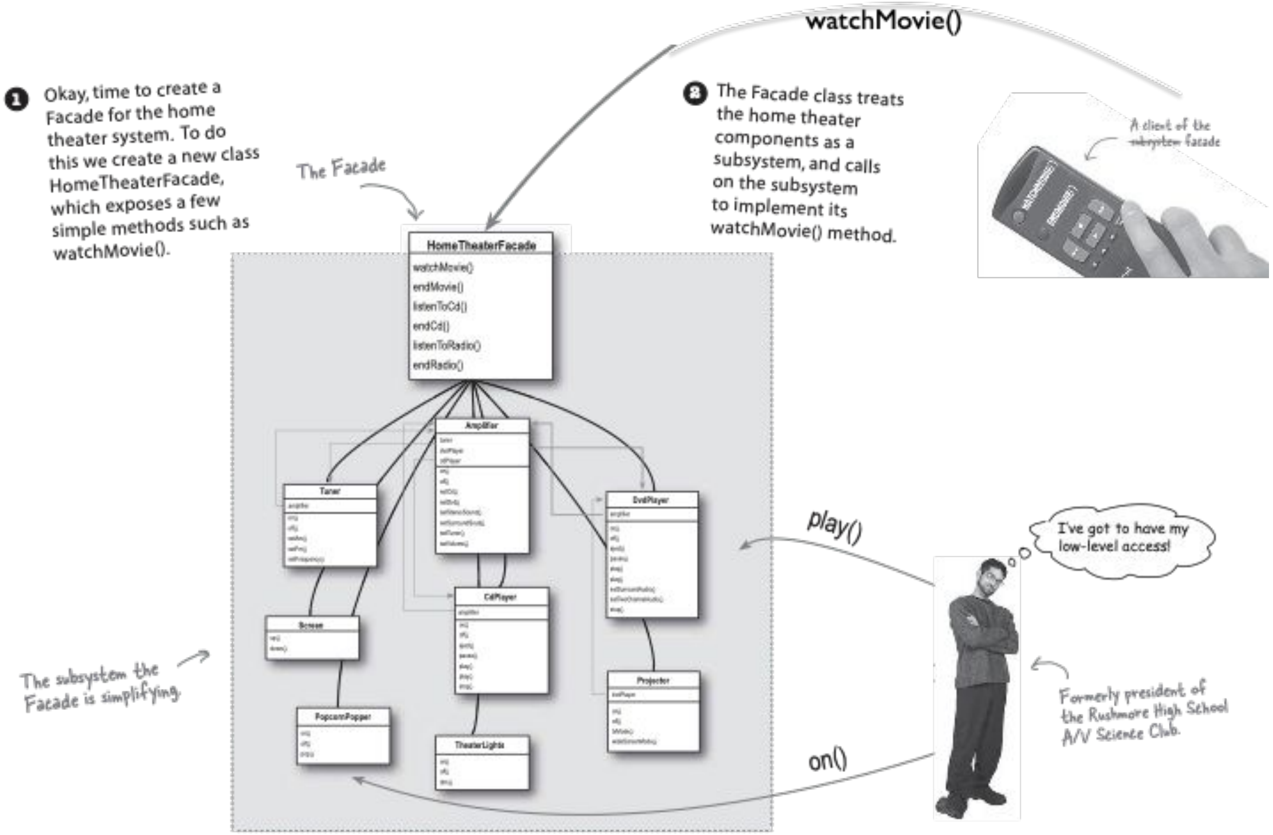
Motivation: Further Complications …

- When the movie finishes, you have to do it all in reverse!
- Doing a slightly different task (e.g., listen to streaming audio) is equally complex
- When you upgrade your system, you have to learn a slightly different procedure

Façade Pattern


Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

Example: Home Theatre Façade



❶ Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

❷ The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.

Example: Home Theatre Façade

```
interface DVDPlayer {
   public powerOn();
   public powerOff();
   public void play(int i);
   public void stop();
   public void pause();
   public void skip();
}

interface Project {
   public powerOn();
   public powerOff();
}

interface Amplifier {
   public powerOn();
   public powerOff();
   public increaseVolume(int n);
   public decreaseVolume(int n);
}
```

```
class HomeTheaterFacade  {
    private DVDPlayer player = new DVDPlayerImpl();
    private Projector projector = new ProjectorImpl();
    private Amplifier amplifier = new AmplifierImpl();

   public void start() {
      player.powerOn();
      projector.powerOn();
      amplifier.powerOn();
      player.play();
      amplifier.increaseSound(3);
   }

   public void end() {
      amplifier.powerOff();
      projector.powerOff();
      player.powerOff();
   }
   ...
}
```

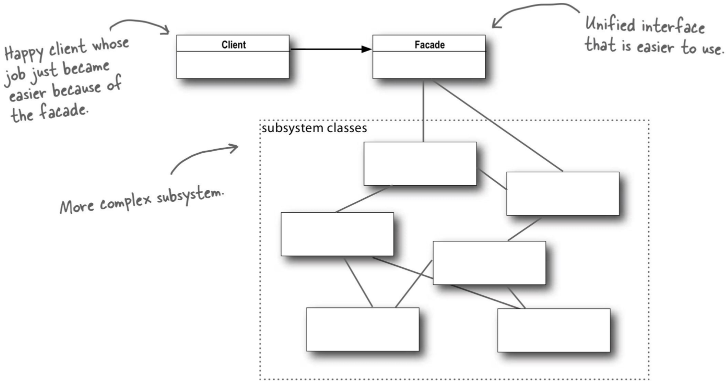
```
class Client {
   public static void main (String[] args) {
      HomeTheaterFacade facade =
              New HomeTheaterFacade();
      facade.start();
      ....
      facade.end();
   }
}
```

Façade vs. Adapter

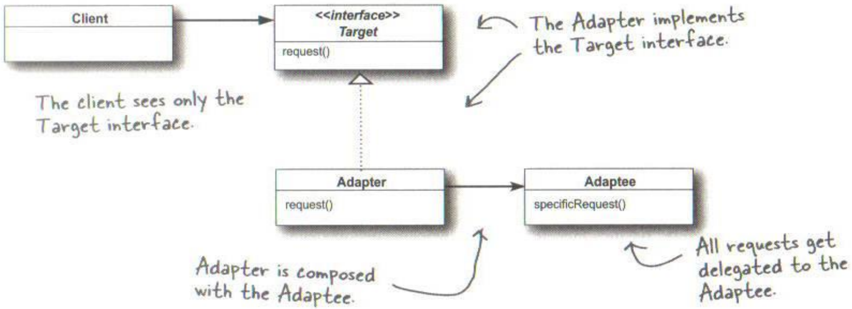
- Both adapter and façade delegate to other interfaces

- Façade decouples a client from a subsystem of components and **simplifies** the interface to those components

- Adapter **converts** one interface to another


Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

Facade


Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

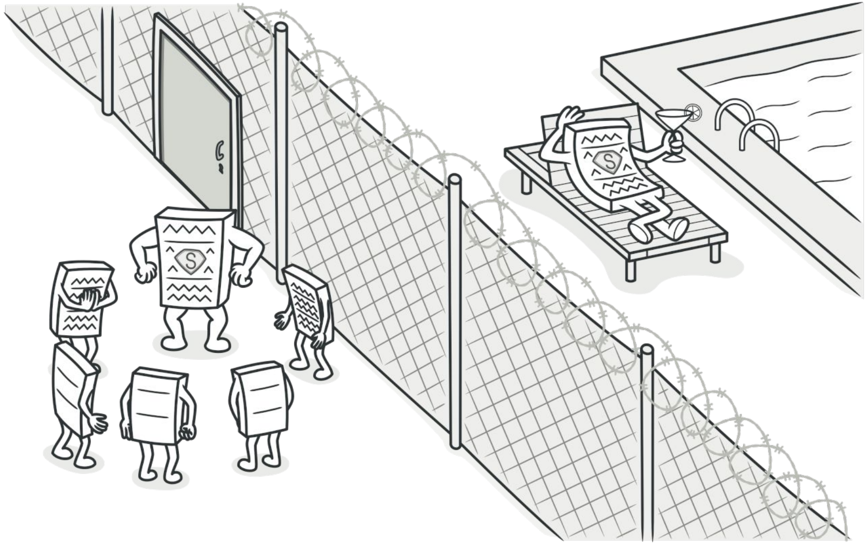
Adapter

Proxy Pattern

Proxy Pattern

Problem

When access to an object needs to be controlled or needs to hide complexity.



Motivation: Proxy Kinds

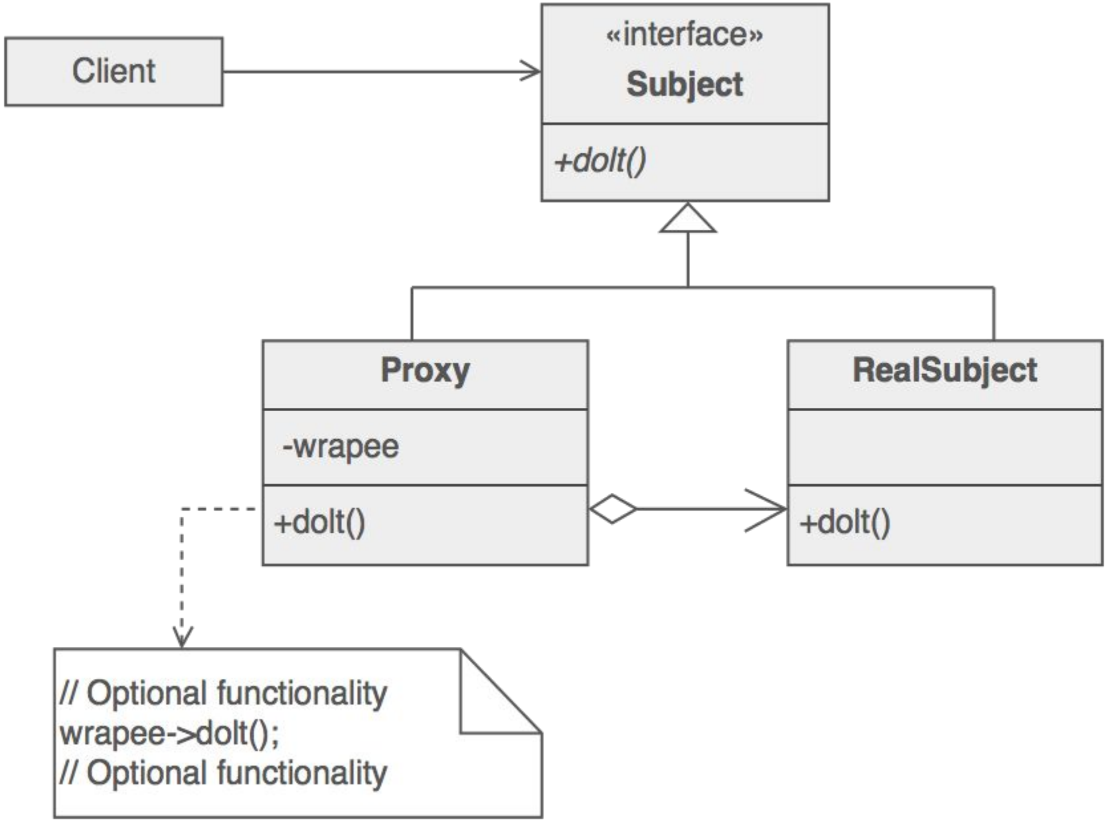
1. **Virtual proxy** can be a placeholder for expensive to create objects. The real object is only created when a client first requests/accesses the object.

2. **Remote proxy** provides a local representative for an object that resides in a different address space. This is what "stub" code in remote-procedure-calls (RPC) provides.

3. **Protective proxy** controls access to a sensitive master object. The surrogate object checks that the caller has the access permissions required prior to forwarding the request.

4. **Smart proxy** adds additional actions when an object is accessed. Typical uses include:
   a. Counting the number of references to the real object so that it can be freed automatically
   b. Checking that the real object is locked before it is accessed to ensure immutability

Proxy Pattern



// Optional functionality
wrapee->doIt();
// Optional functionality

Example: Internet Interface

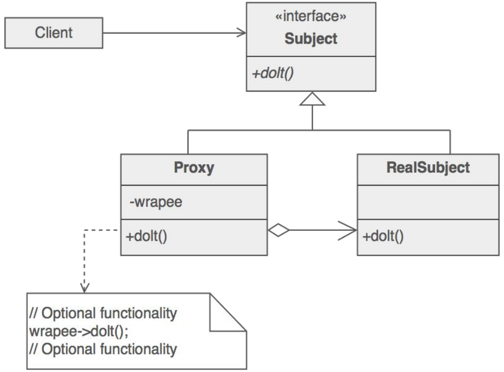
```
interface Internet { // Subject
    public void connectTo(String host) throws Exception;
}

class RealInternet implements Internet { // RealSubject
    public void connectTo(String host) throws Exception {
        System.out.println("Connecting to "+ host);
    }
}

class Client {
    public static void main (String[] args) {
        Internet internet = new ProxyInternet();
        try {
              internet.connectTo("geeks.org");
            internet.connectTo("abc.com");
        }
        catch (Exception e)  {
            System.out.println(e.getMessage());
        }
    }
}
```

```
class ProxyInternet implements Internet { // Proxy
    private Internet internet = new RealInternet();
    private static List<String> bannedHosts;
    static {
        bannedHosts= new ArrayList<String>();
        bannedHosts.add("abc.com");
        bannedHosts.add("def.com");
        bannedHosts.add("ijk.com");
    }

    public void connectTo(String host) throws Exception {
        If (bannedHosts.contains(host)) {
            throw new Exception("Access Denied");
        }
        internet.connectTo(serverhost);
    }
}
```

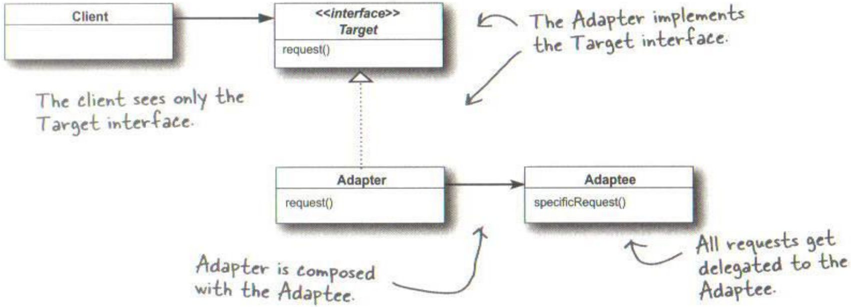
Proxy vs. Adapter

- **Proxy** implements the same interface as its aggregated object
- The intent of **Proxy** is to wrap an object to perform something before and/or after the calls to its interface

- **Adapter** implements a different interface from the aggregated object
- The intent of **Adapter** is to convert an interface to work in a new situation



Proxy



Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

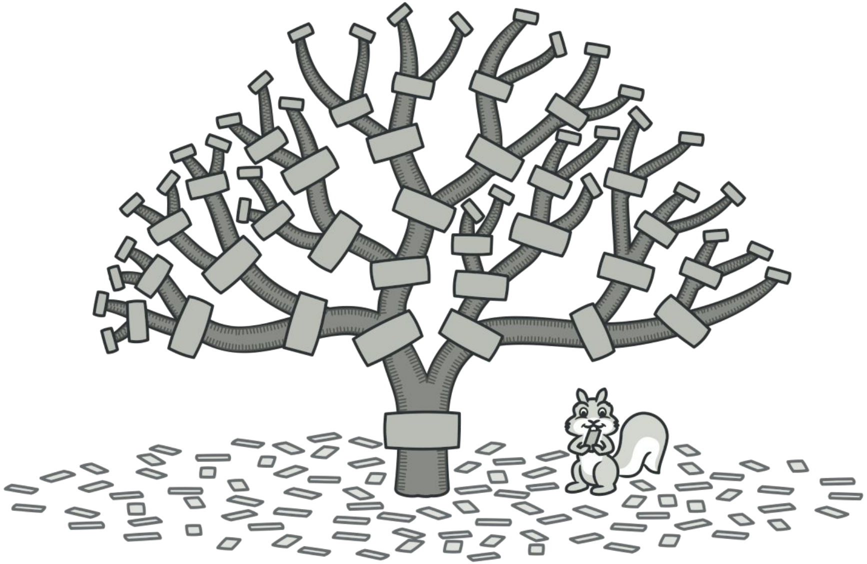
Adapter

Composite Pattern

Composite Pattern

Problem

When an application wants to compose objects into a tree structure then work with the structure as if it was an individual object.

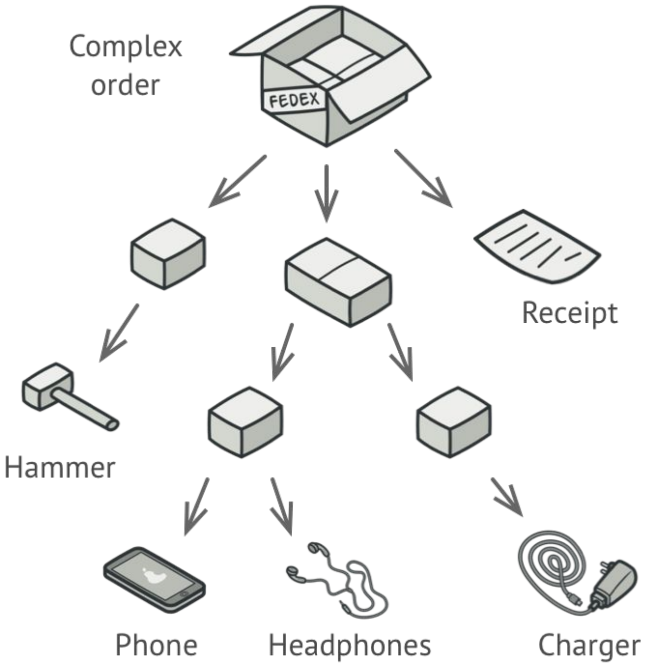


Motivation

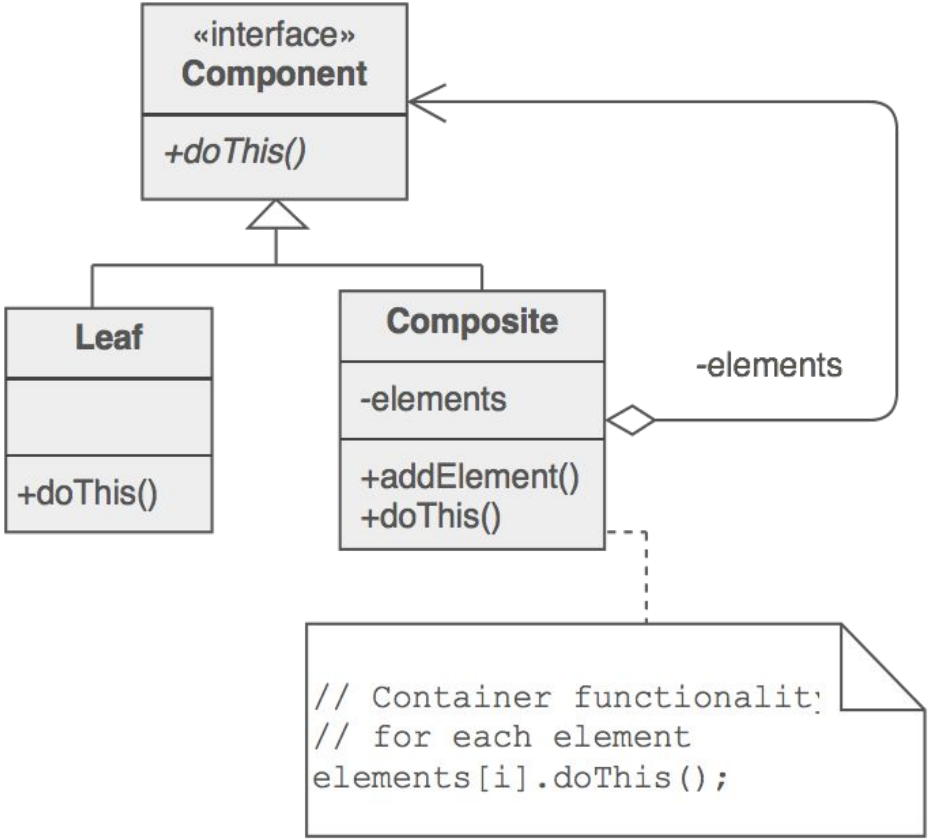
- You have two types of objects: **Products** and **Boxes**.
  - A Box can contain several Products as well as a number of smaller Boxes.
  - These little Boxes can also hold some Products or even smaller Boxes, and so on.

- You decide to create an ordering system, where orders could contain:
  - Simple products without any wrapping
  - Boxes stuffed with products and other boxes.

- How would you determine the total price of such an order?

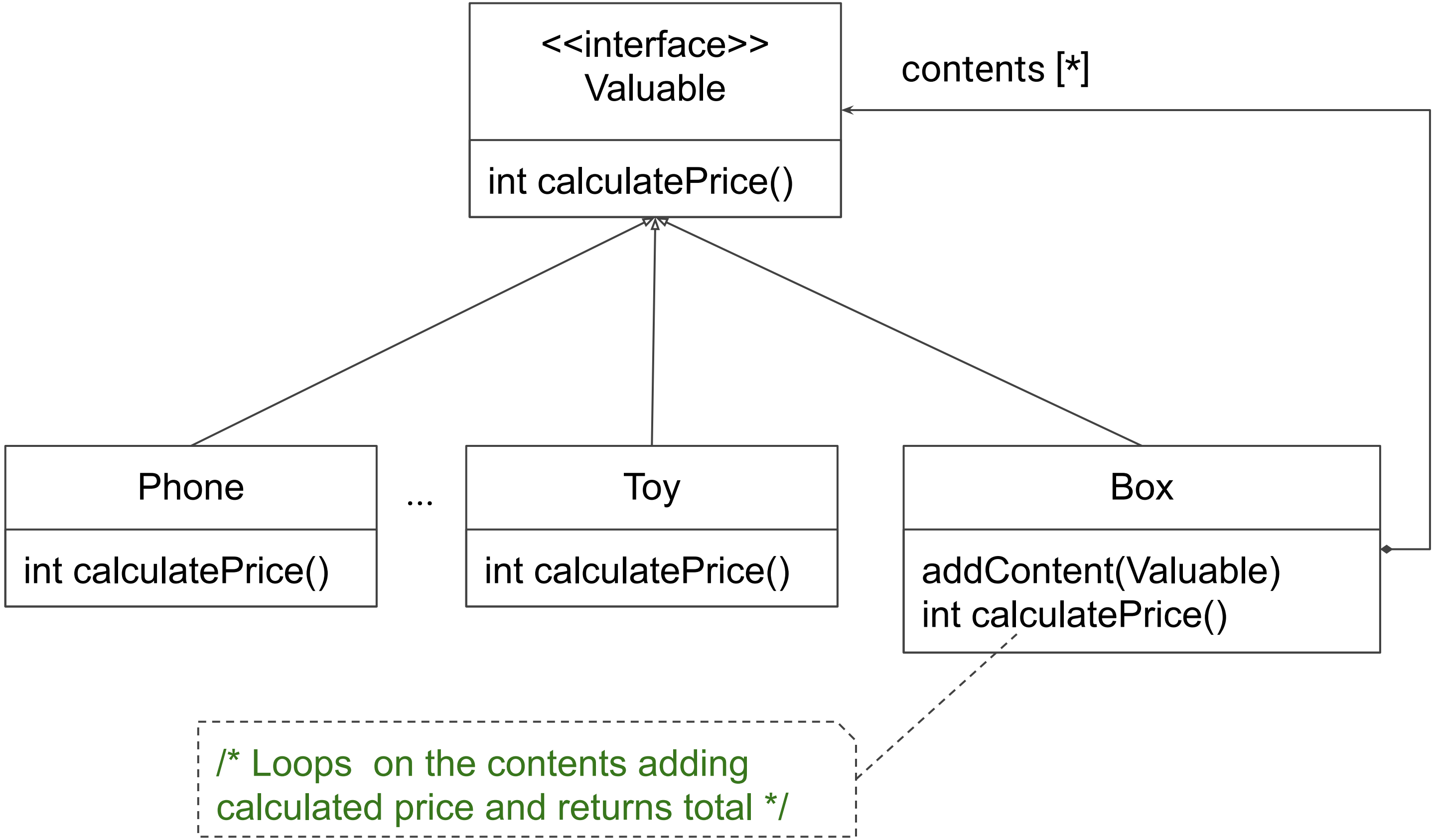


Composite Pattern



```
// Container functionalit
// for each element
elements[i].doThis();
```

Example: Box as a Composite



Example: Box as a Composite

```
interface Valuable { // the component
        int calculatePrice ();
}
class Phone implements Valuable { // the leaf 1
        public int calculatePrice()  { return 10; }
}
class Toy implements Valuable { // the leaf 2
        public int calculatePrice()  { return 2; }
}
class Box implements Valuable { // the composite
        private List<Valuable> contents = new ArrayList<>();
        public void addContent(Valuable v) {
                contents.add(v);
        }
        public void removeContent(Valuable v) {
                contents.remove(v);
        }
        public int calculatePrice()  {
                int total = 0;
                for (c : contents) {total += c.calculatePrice();}
                return total;
        }
}
```

```
class BoxingCompany { // the client
   public static void main(String args[]) {
      Phone phone1 = new Phone();
      Phone phone2 = new Phone();
      Box smallBox = new Box();
      smallBox.addContent(phone1);
      smallBox.addContent(phone2);
      Toy toy = new Toy();
      Box bigBox  = new Box();
      bigBox.addContent(smallBox);
      bigBox.addContent(toy);
      System.out.println (“Price = “+bigBox.calculatePrice());
   }
}
```

Recap

- **Adaptor** adapts legacy code to a target interface.
- **Façade** simplifies complex interfaces of a subsystem.
- **Proxy** acts as a wrapper to an object with the same interface
- **Composite** makes a set of objects implement the same interface as a single object

Structural Patterns Quiz

References

- Freeman, E., Freenman, E., “Head First Design Pattern.” O’Rielly, 2004.
- Software Design Patterns