7b. Design Patterns – Structural Patterns

Three types of design patterns

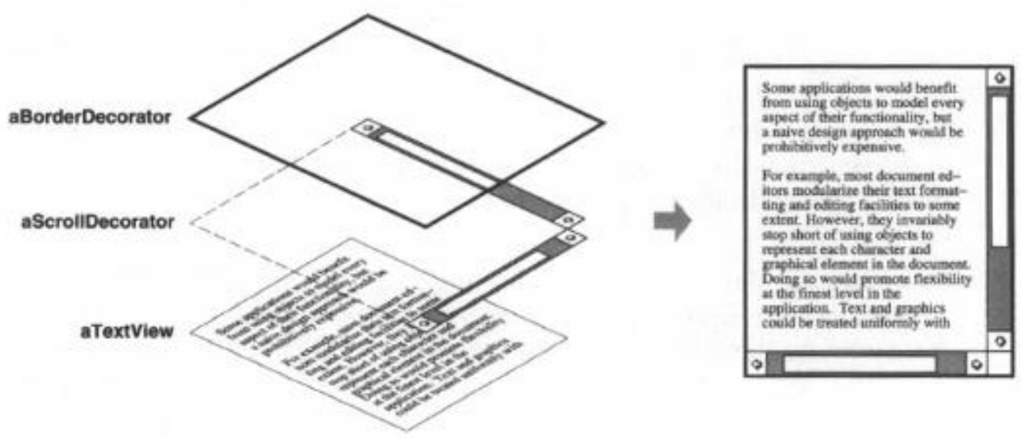
- **Creational** patterns: How do we create objects?
- **Structural** patterns: How do we compose large objects out of small objects?
- **Behavioral** patterns:

Structural patterns

- Decorator – Flexibly add functionality to objects during runtime
- Flyweight – Use sharing to reduce cost of using lots of small objects
- Composite – Use a Compound class to recursively compose objects
- Adapter – Composing two objects using inheritance
- Bridge – Split a monolithic class into several hierarchies
- Proxy – Use a proxy object to control access to an existing object

Decorator

- In some cases we may want to add potential responsibilities to a class
- Example: text view



GoF textbook

- Whether or not a specific object will have such responsibilities is determined during runtime

Decorator

- Some solutions?
- Add all potential functionalities to the TextView class
  - This bloats the TextView code and weakens the single responsibility principle
- Use Inheritance
  - TextViewWithHorizontalScrollbar, TextViewWithVerticalScrollbar, TextViewWithHorizontalandVerticalScrollbar, TextViewWithHorizontalScrollbarAndBorder…
  - Some classes cannot be inherited…
- Decorator is a pattern that allows us to do so without modifying the TextView class at all (or the client code that uses TextView)

Decorator: Implementation

- First, define an interface for the class we want to add functionalities to
  - TextView -> TextComponent

Component

```
interface TextComponent {
   public void Draw();
   public void Resize();
}
```

Concrete Component

```
public class TextView implements TextComponent {
   public void Draw() {//…}
   public void Resize() {//…}
}
```

Decorator: Implementation

- A Decorator implements **and is also** composed of an object of that interface

```
public class TextDecorator implements TextComponent {
   private TextComponent wrappee;
   TextDecorator(TextComponent wrappee) {
      this.wrappee = wrappee;
   }
   public void Draw() {wrappee.Draw();}
   public void Resize() {wrappee.Resize();}
}
```

Decorator

Decorator: Implementation

- We then subclass TextDecorator to implement specific decorations

```
public class BorderDecorator extends TextDecorator {
   public void Draw() {
      wrappee.Draw();
      DrawBorder();
   }
   private void DrawBorder() {//…}
}
```

Decorator: Terminology

- Component: Interface for object to be compounded. *(TextComponent)*
- Concrete Component: Implementation of that object, will get additional responsibilities. *(TextView)*
- Decorator: Base class that implements and contains the Component to support decorating. *(TextDecorator)*
- Concrete Decorator: Subclass of Decorator to add a functionality. *(BorderDecorator)*

Decorator: Usage

- Client code needs to wrap object properly

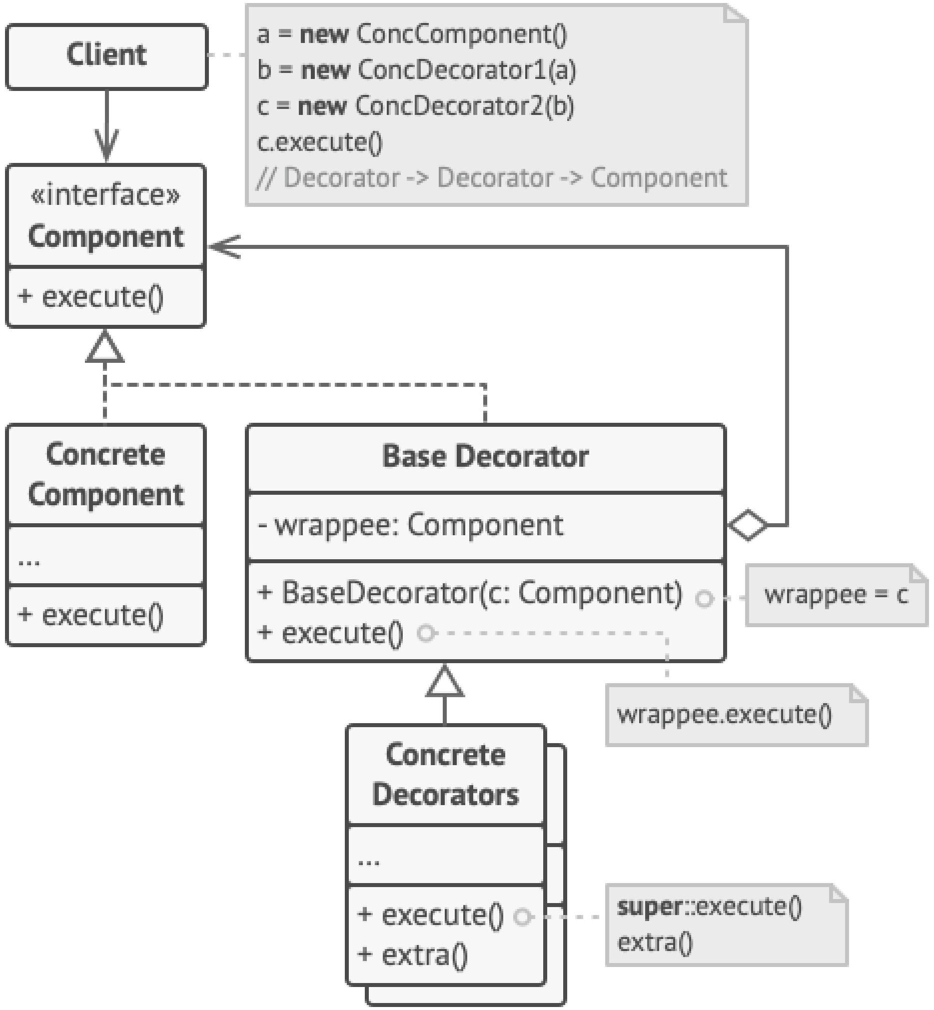
```
TextComponent myTextView = new TextView();
//set up potential decorators
if (max_line_size > 70) myTextView = new XScrollDecorator(myTextView);
if (line.num > 100) myTextView = new YScrollDecorator(myTextView);
if (Settings.borders_enabled) myTextView = new BorderDecorator(myTextView);

window.setTextComponent(myTextView);
```

Decorator: Usage

- Rest of client code can completely ignore the details of what decorators we have
  - They only care about the TextView interface
  - e.g. Manipulating a blinking text cursor does not care if the TextView has a border, scroll bar or not
- What happens when we call myTextView.Draw()?
  - Recursive function calling: outermost wrapper draws, then calls next wrapper

Decorator: Usage



refactoring.guru

Decorator: Example #2

- Data may be encrypted, compressed, both or neither

```
interface DataIO {
    public void Read(File f);
    public void Write(File f, String s);
}
```

```
public class FileDataIO implements DataIO {
    public void Read(File f) {//…}
    public void Write(File f, String s) {//…}
}
```

Decorator: Example #2

- What does FileDataDecorator contain?

```
public class FileDataDecorator implements DataIO {
   private DataIO wrappee;
   FileDataDecorator(DataIO wrappee) {
      this.wrappee = wrappee;
   }
   public void Read(File f) {wrappee.Read();}
   public void Write(File f, String s) {wrappee.Write(f);}
}
```

Decorator: Implementation

- Subclass FileDataDecorator to implement EncryptDecorator

```
public class EncryptDecorator extends FileDataDecorator {
   public void Write(File f, String input) {
      String encInput = EncryptString(input);
      wrappee.Write(f, encInput);
   }
   private String EncryptString(String input) {
      //deal with encryption on input, return output
   }
}
```

Decorator: Downsides

- Client code is fully responsible for correctly calling wrappers
  - This means the client code needs to understand all wrappers
  - Decryption and De-compression: Which one is first?
- Wrapper code must not affect each other
  - Border clips into the scroll bar…
  - Horizontal scroll bar is drawn, then vertical scroll bar is drawn and covers all of it. Woops…
- Is myTextView instanceof TextView?
- One class for each functionality may be too much

Flyweight

- When you have lots of instances of the same class, each object normally contains its own (possibly expensive) resource
  - e.g. each letter is its own image
  - e.g. each bullet is its own 3D model
- Motivation: expensive resources, such as images and models, should be **shared**
  - Reduce loadtime, reduce memory consumption
- How can we do so?

Flyweight: Design

- The first step is to distinguish between each object’s **extrinsic** and **intrinsic** states
  - Extrinsic: Outside. Other objects interact with and change extrinsic state
  - Intrinsic: Inside. Internal to the object; cannot be changed by other objects.
- Example: a raindrop particle
- Flyweight: Only intrinsic states should belong to an object
  - Extrinsic states should be passed around by client code

Flyweight: Example

- Let's imagine a big enough Cat Game that there may be hundreds of different Rat objects being allocated
- Each rat's graphic depends on several *intrinsic* states:
  - How healthy the rat is (4 possibilities)
  - How big the rat is (3 possibilities)
- Maximum of 12 Flyweight objects needed
- Extrinsic states:
  - Rat's position
  - Rat's attack/defense stats

Flyweight: Example

```
class RatFlyweight {
   int healthState;
   int sizeState;
   Image myGraphic;
   RatFlyweight(healthState, sizeState) {//set them}
   void paint(Graphics g, Point position) {…}
}
```

- RatFlyweight contains only intrinsic state, to paint it would require passing it extrinsic state
- paint can lazy initiate and store the bitmap (check if null)

Flyweight: Example

```
class Rat {
   RatFlyweight myFlyweight;
   Rat() {
      myFlyweight = RatFlyweightFactory.get(curHP, maxHP, size);
   }
   void onHit {
      //reduce HP, then…
      myFlyweight = RatFlyweightFactory.get(curHP, maxHP, size);
      repaint();
   }
}
```

- Rat contains a reference to a RatFlyweight
- Initiation is done using a Flyweight Factory

Flyweight: Example

```
class RatFlyweightFactory {
    ArrayList<RatFlyweight> flyweights;
    RatFlyweight get(curHP, maxHP, size) {
        int healthState = (curHP*3)/maxHP;
        int sizeState = min(size/20, 2);
        RatFlyweight flyweight = flyweights.find(healthState, sizeState);
        if (flyweight == null) {
            flyweight = new RatFlyweight(healthState, sizeState);
        }
        return flyweight;
    }
}
```

- RatFlyweight lazy initiates Flyweights whenever necessary

Flyweight: Implementation details

- You do not want a Flyweight's state to ever be changed

```
class RatFlyweight {
   private int healthState;
   private int sizeState;
   private Image myGraphic;
   RatFlyweight(healthState, sizeState) {//set them}
   void paint(Graphics g, Point position) {…}
}
```

- Don't create setters for Flyweight variables

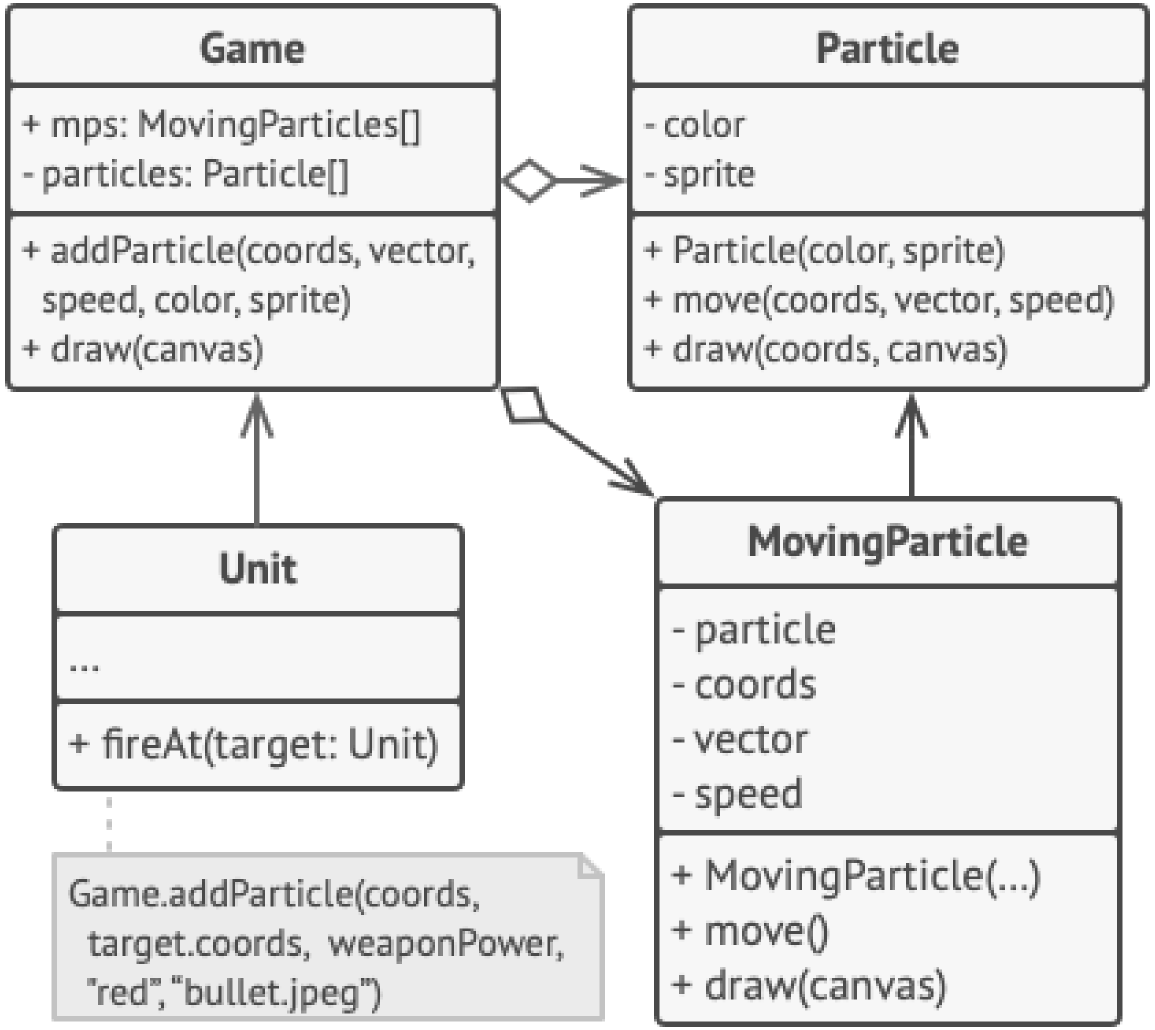
Flyweight: when to use?

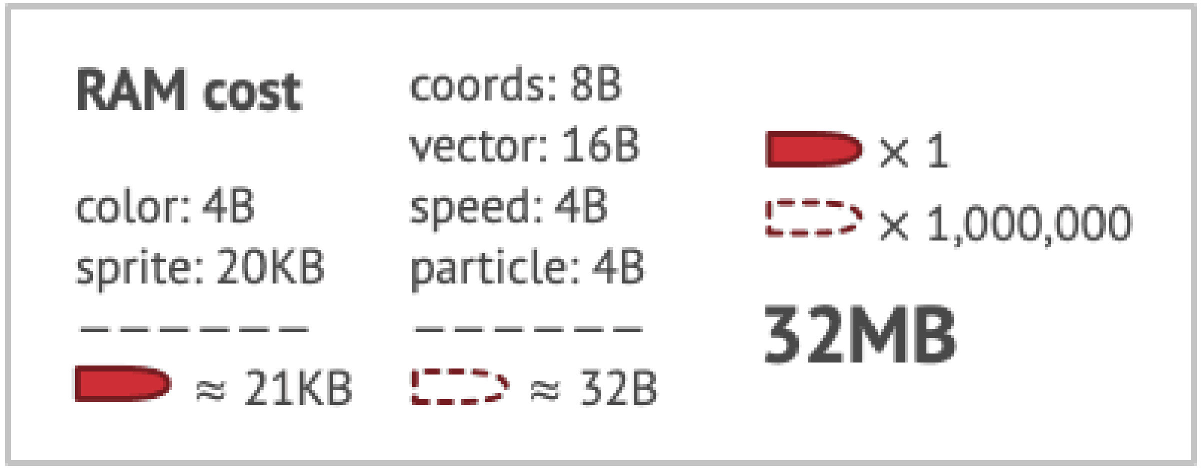
- Flyweight is a specific design pattern and you should only use it when:
  - The app creates a very large number of objects
  - These objects are expensive
  - The expensive part is their shared state
- The advantage is reduction of storage
  - Sometimes this can be traded off for computational cost

Flyweight: Downsides

- It can be inherently unintuitive to use Flyweight because a single object’s state is being separated into two classes
  - Who will explain the reasoning to a code reader?
- Object identity tests: Is Rat1 == Rat2?
- Possible downside: computational cost
  - This is why we stored the Image directly in the Flyweight

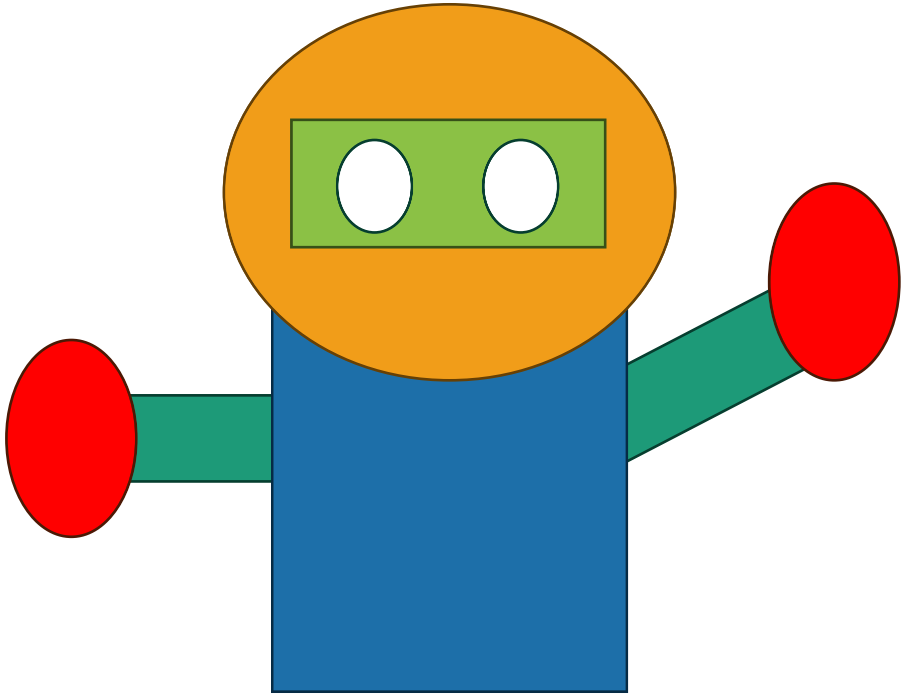
Flyweight: Example #2 (refactoring guru)

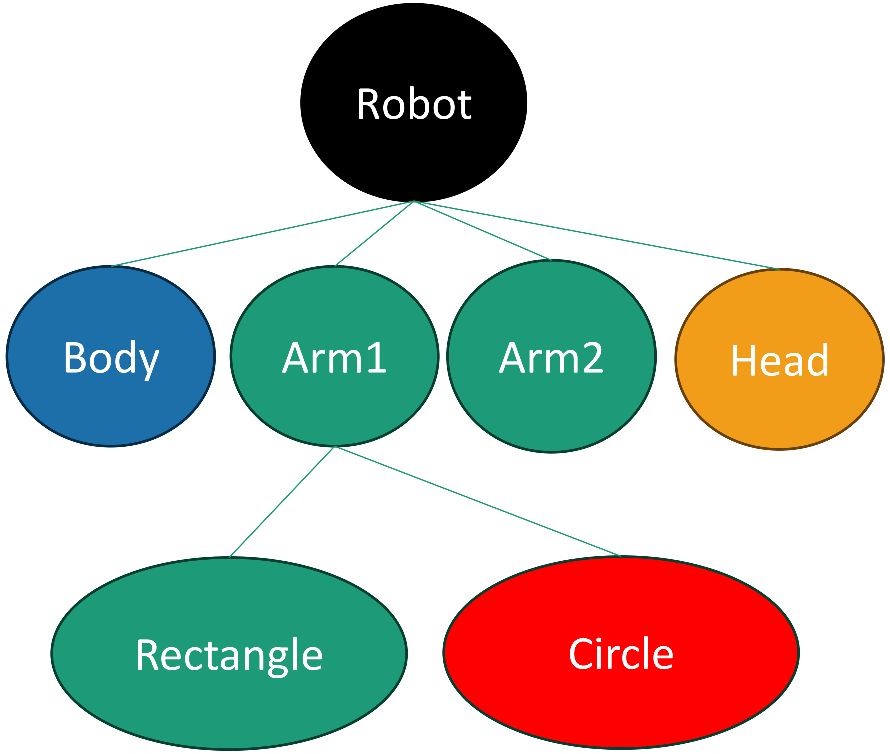




Composite

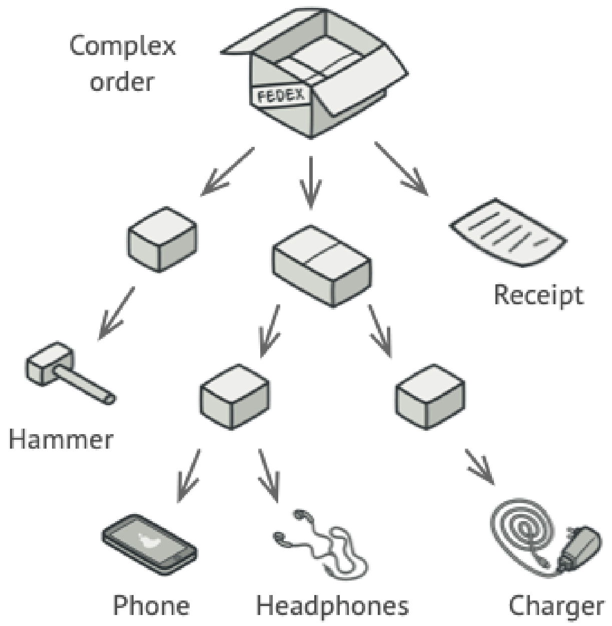
- Composite is useful when we want to manipulate an object that has a **<u>tree</u>** structure
- Example:





Composite

- Composite is useful when we want to manipulate an object that has a **tree** structure
- Example #2: What is the total price?



refactoring.guru

Composite: Implementation

- Define an interface for the object to be composed

```
interface Graphic {
   void draw();
   void resize(Point anchor, int xscale, int yscale);
}
```

```
public class Circle implements Graphic {
   int x; int y; int radius;
   //implement draw and resize…
}
```

Composite: Implementation

- Create the CompoundGraphic composite:
  - CompoundGraphic performs delegation to its children
  - It has basic tree operations

```
public class CompoundGraphic implements Graphic {
   List<Graphic> children = new List();
   void add(Graphic graphic) {children.add(graphic);}
   void remove(Graphic graphic) {children.remove(graphic);}
   void draw() {
      for (Graphic child: children) {
         child.draw();
      }
   }
}
```

Composite: Terminology

- Component: Interface for object to be compounded. *(Graphic)*
- Composite: Object that supports tree traversal. *(CompoundGraphic)*
- Leaf: End object that has no children. *(Circle, Rectangle)*

Composite: Downsides

- It can be difficult to define the Component interface correctly
  - ApplyDiscount() to all items in a complex order. But it has a receipt…
  - Should ChangeBorderColor() apply to an arrow in a CompoundGraphic?
- Specific application

Adapter

- A simple design pattern to allow two incompatible interfaces to work together
- Client Code has been written for Interface A (*Target*) and we want it to also function for Interface B (*Adaptee*)
- Two versions: Object Adapter and Class Adapter

Adapter: Object Adapter

- Uses object composition: Adapter contains the Adaptee, converts the necessary functions to the Adaptee
- Example: a RoundPeg object fits with a RoundHole object, but how about a SquarePeg object?

```
public class RoundHole {
    int radius;
    boolean fits(RoundPeg peg) {
        return (this.radius >= peg.getRadius());
    }
}
```

```
public class RoundPeg {
    int radius;
    int getRadius() {//…};
}
```

```
public class SquarePeg {
    int width;
    int getWidth() {//…};
}
```

Adapter: Object Adapter

- SquarePegAdapter converts SquarePegs into RoundPegs:

```
public class SquarePegAdapter extends RoundPeg {
   private SquarePeg squarePeg; //set by constructor, omitted
   int getRadius() {
      return squarePeg.getWidth() * Math.sqrt(2) / 2;
   }
}
```

Client code:

```
RoundHole hole = new RoundHole(5);
SquarePeg squarePeg = new SquarePeg(5);
RoundPeg squarePegAdapter = new SquarePegAdapter(squarePeg);
hole.fits(squarePegAdapter); //returns true
```

Adapter: Class Adapter

- Inherits from both Target and Adaptee to implement
- Only possible for languages with multiple inheritance!

```
public class SquarePegAdapter extends RoundPeg, SquarePeg {
    private SquarePeg squarePeg;
    private RoundPeg roundPeg; //one of those set by constructor, omitted
    int getRadius() {
        if (roundPeg == null) {
            return squarePeg.getWidth() * Math.sqrt(2) / 2;
        }
        else return roundPeg.getRadius();
    }
    int getWidth() {//…}
}
```

Adapter: Downsides

- What is the main alternative to using Adapter?
  - Changing the Adaptee's code
- So why use Adapter instead of changing the Adaptee's code?
  - Open/Closed Principle: Changing the Adaptee's code might break some other code
  - Adaptee's code is possibly not changeable
- Which one: Object Adapter or Class Adapter?
  - A class adapter that subclasses a lot of classes may have significant duplicate code

Bridge

- Separate a large monolithic class into two parts so it can be better extended in two different directions
- Example: I have three types of Rats based on how strong they are
  - WeakRat, NormalRat, BossRat
  - Stronger rats can break through walls and bully weaker rats
- I have three types of Rats based on their aggressiveness
  - CowardRat, TacticalRat, FierceRat
  - Fiercer rats will attack more
- Now I have 9 classes with a lot of code duplication…

Bridge (fake example)

- Using object composition, we can allow reasonable inheritance

```
public class WeakRat extends Rat {
   RatAggression ratAggression;
   boolean isEscaping() {
      return ratAggression.isEscaping();
   }
   @Override
   boolean isWallEater() {
      return false; //
   }
}
```

Bridge: Advanced

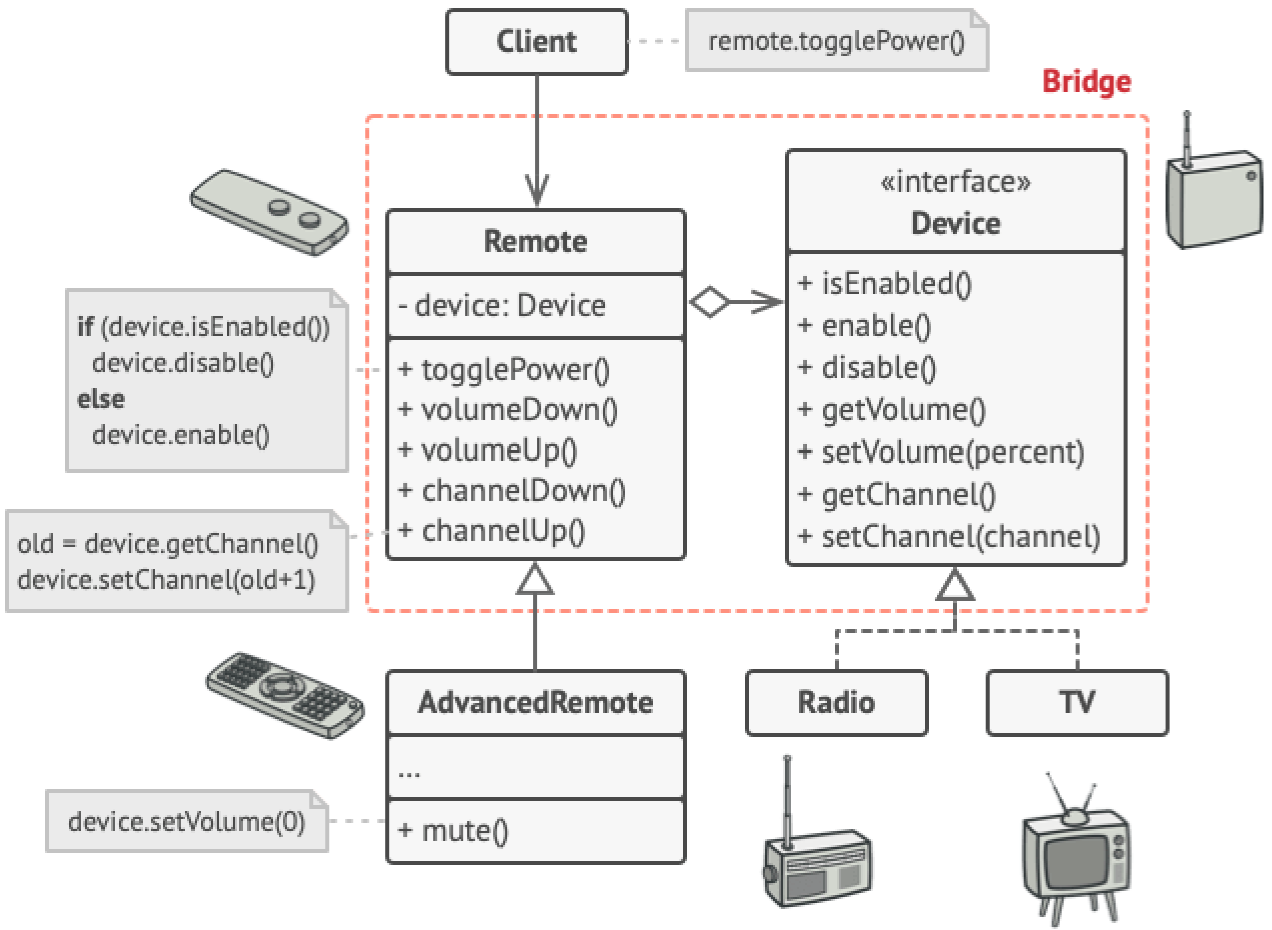
- What should we really be separating?
- Bridge pattern: Separate the **abstraction** and the **implementation** of a large monolithic class for better readability and flexibility
- Abstraction: Client-facing code. High-level control layer.
- Implementation: Internal code. Actual function calls.
  - Note that here implementation doesn’t mean “implementing an interface”
- Example: Window GUI for different OS’s
  - Make different API calls for different OS’s
  - Present different types of windows for different customer preferences

Bridge: Example

```
public class WindowGUI {
    WindowImpl impl; //setters omitted
    void Open() {//…}
    void Close() {//…}
    void DrawLine(Point a) {impl.DeviceLine(a);}
    void DrawRect(Point a, Point b) {impl.DeviceRect(a, b);}
}
```

```
public class LinuxWindowImpl extends WindowImpl {
    void DeviceLine(Point a) {//…}
    void DeviceRect(Point a, Point b) {//…}
}
```

Bridge: Example #2



refactoring.guru

Bridge: Terminology

- Bridge separates Abstraction and Implementation
- Subclasses of Abstraction are Refined Abstractions
- Implementations of Implementation (☹) are called Concrete Implementations

Bridge: Benefits

- Large monolithic class is broken down to avoid code bloat
- Client code is exposed only to abstraction
- Implementation can be switched at runtime
- Natural solution for cross-platform programming

Bridge: Caveats

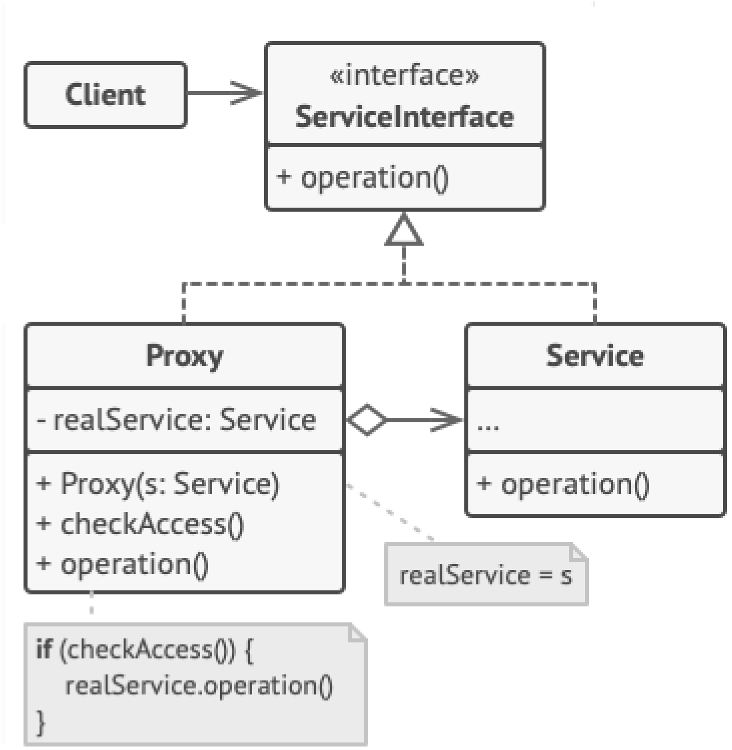
- Implementation has to be an interface if we want more than one concrete implementation (why?)
- Abstraction could also be an interface
- Maximize cohesion, minimize coupling

Proxy

- A Proxy can add functionality to a class without changing the class
- Examples:
  - A video player that just loads data from a server and plays it
    - Add a caching service proxy so it can replay without loading data again
  - A game interface for local play, but you want multiplayer
    - Add a networking service proxy to pass and receive information
  - Adding Logging to a class

Proxy: Implementation

- Similar to Bridge:



refactoring.guru

Proxy: Example (Virtual Proxy)

- Lazy (on demand) initiation of an image

```
interface Graphic {
   void Draw();
   void Load(String filename);
}
```

```
public class Image implements Graphic {
   Image image;
   String filename;
   void Draw() {
      if (image == null) {
         image = new Image();
         image.Load(fiename);
      }
      image.Draw();
   }
   void Load(String filename) {
      this.filename = filename;
   }
}
```

Proxy: Uses

- Protection Proxy: Restrict access to service object
- Remote Proxy: Allows use of remote service object
- Caching Proxy: Cache results to save time/bandwidth, intelligently destroy objects when unnecessary ("smart reference")
  - Can also perform object locking

Relationships between design patterns

- A Composite object can have its functionality extended by a Decorator
  - e.g. CompoundGraphic: Decorator adds the ability to add lighting/shading
  - Note that the Decorator would also need to support add, remove, child reference

Relationships between design patterns

- A Builder is effective for building a complex Composite object
  - Example: A Composite for representing a file system
  - The Builder should:
    1. Read files in the directory, add them as child leaves to the Filesystem tree
    2. Read folders, add them as child nodes to the Filesystem tree
    3. Traverse these folders, and repeat until done
  - Different OS's can use different concrete builders

Relationships between design patterns

- Three design patterns are implemented as ‘wrappers’ around a target object (“wrappee”):
  - Adapter produces a **different** interface than the wrappee, allowing client code on a different interface to work with the wrappee
  - Proxy produces the **same** interface, modifying existing functionality
  - Decorator **adds** new functionality onto the interface
- Decorators and Composite can both result in recursive object composition and function calls
  - Decorators require client code to wrap things in the right order; composites should not require additional client code