THE ‘BRIDGE’ DESIGN PATTERN

Daniel Cuevas

CS 445 – Spring 2015

WHAT IS THE 'BRIDGE' DESIGN PATTERN?

- A structural design pattern meant to simplify the relationships between identities.
- Decouple an abstraction from its implementation, allowing the two to vary independently.
  - Both the abstraction and the implementation develop separate inheritance structures.
  - The abstraction can be either an abstract or interface class.
  - The implementer is also either an abstract or interface class.

THE ABSTRACTION

- The abstraction contains a reference to the implementer.
- Children of an abstraction are referred to as refined abstractions.
- Possible to change the reference to the implementer during run-time.
  - Can be thought of as the device being controlled by the implementer.
  - The implementer would affect the abstraction through the ‘bridge’ that is created via the abstraction

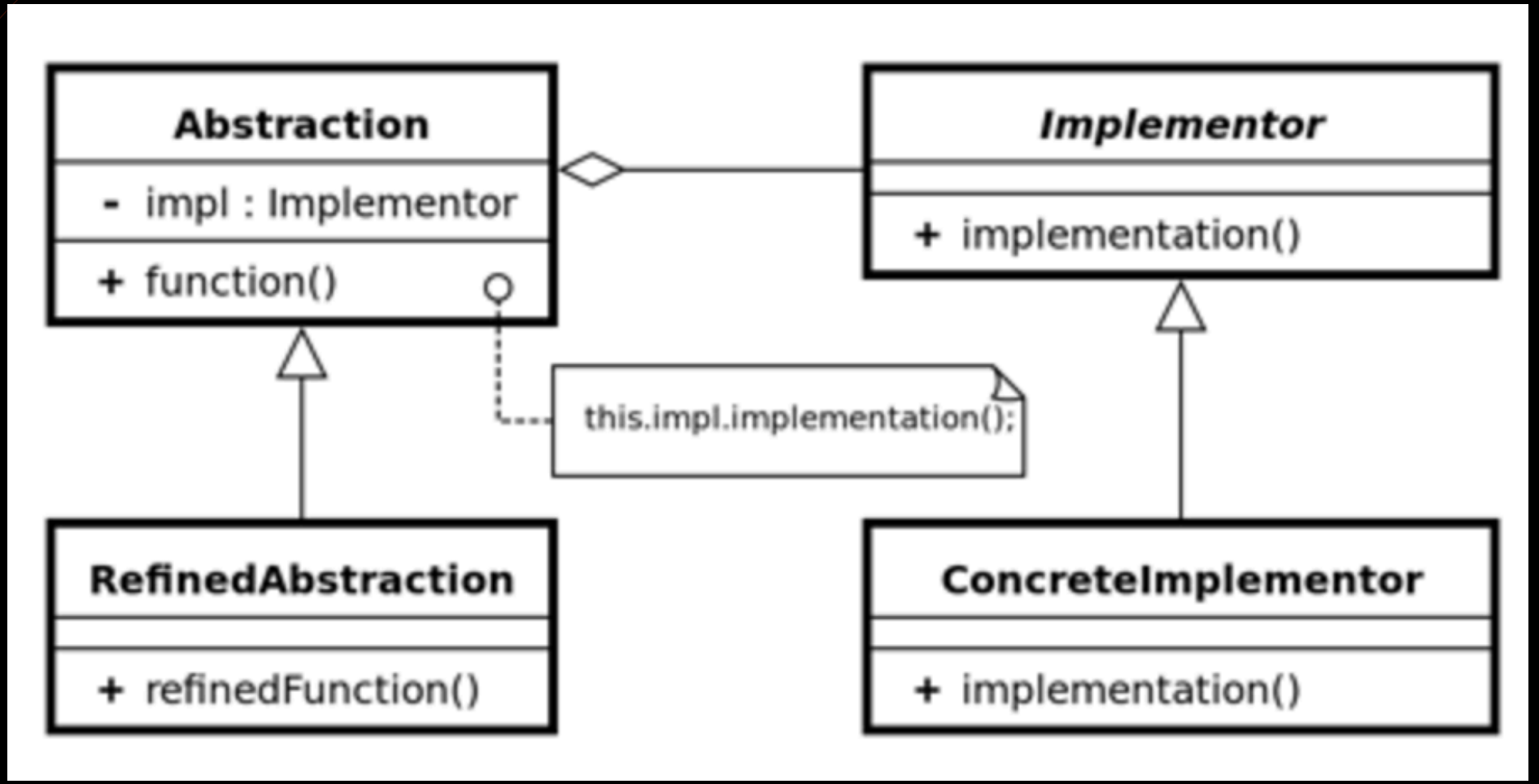
THE IMPLEMENTER

- Children of the implementer are referred to as concrete implementers.
- Changes can be done to the implementer classes during run-time.
- Changes do not affect the client (abstractions).
- Controls the actions that would affect the abstraction.
  - Can be thought of as a remote to the TV
  - Multiple different remotes with different functionality for multiple TVs

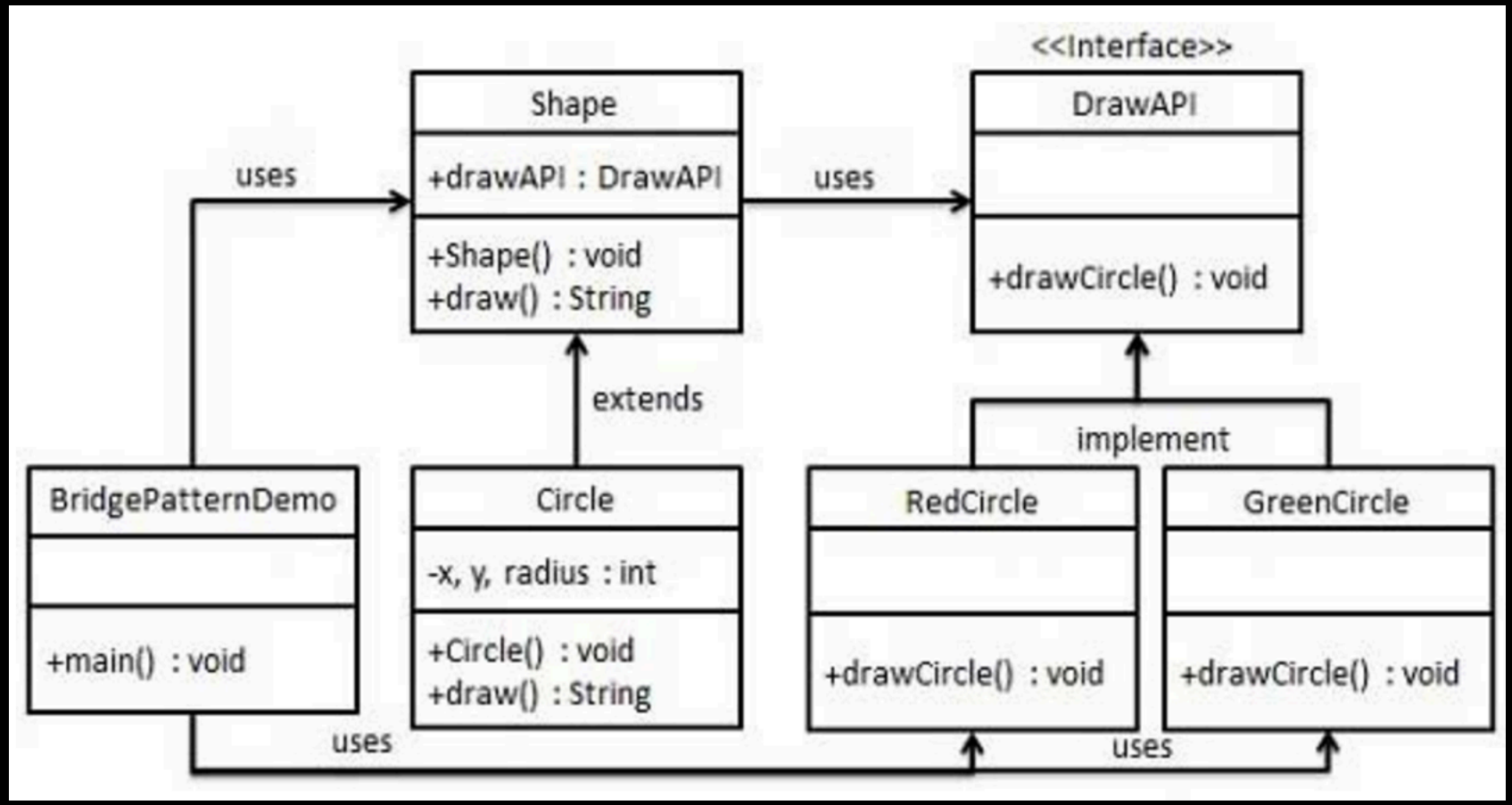
WHEN TO USE THE DESIGN PATTERN?

- When you need to change the device and the controller independently by decoupling them.
  - The device being the abstraction and the controller being the implementer.
- When you want to add functionality without affecting the client (device) code.
- If there's different user interfaces in different systems that do different functionality on a similar client (device)

BASIC EXAMPLE:



A SIMPLE EXAMPLE:



REFINED ABSTRACTIONS **CONCRETE IMPLEMENTERS**

THE IMPLEMENTERS.

IMPLEMENTS

DrawAPI.java

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

RedCircle.java

```
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");
   }
}
```

GreenCircle.java

```
public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");
   }
}
```

THE ABSTRACTIONS

Shape.java

```
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

EXTENDS

Circle.java

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

USES

BridgePatternDemo.java

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

RESOURCES:

- http://www.avajava.com/tutorials/lessons/bridge-pattern.html
- http://simpleprogrammer.com/2015/06/08/design-patterns-simplified-the-bridge-pattern/
- https://sourcemaking.com/design_patterns/bridge
- http://www.oodesign.com/bridge-pattern.html
- http://programmers.stackexchange.com/questions/121326/understanding-the-bridge-design-pattern
- https://www.youtube.com/watch?v=9jIgSsIfh_8