Bridge Pattern

What is Bridge Pattern

- In the previous two behavioural patterns, Strategy and Adaptor, the focus was on the cases that behaviour of an object was subject to frequent changes. But in other cases it is desired to extend the abstraction and implementation independently.
- Bridge Pattern is a structural pattern. And in other words, it is useful to decouple an abstraction from its implementation to be able to change each of them independently.
- The key is that an interface acts as a bridge to make the functionality of concrete classes independent from interface implementers.
  - The structure of both classes can be altered without affecting the other class.

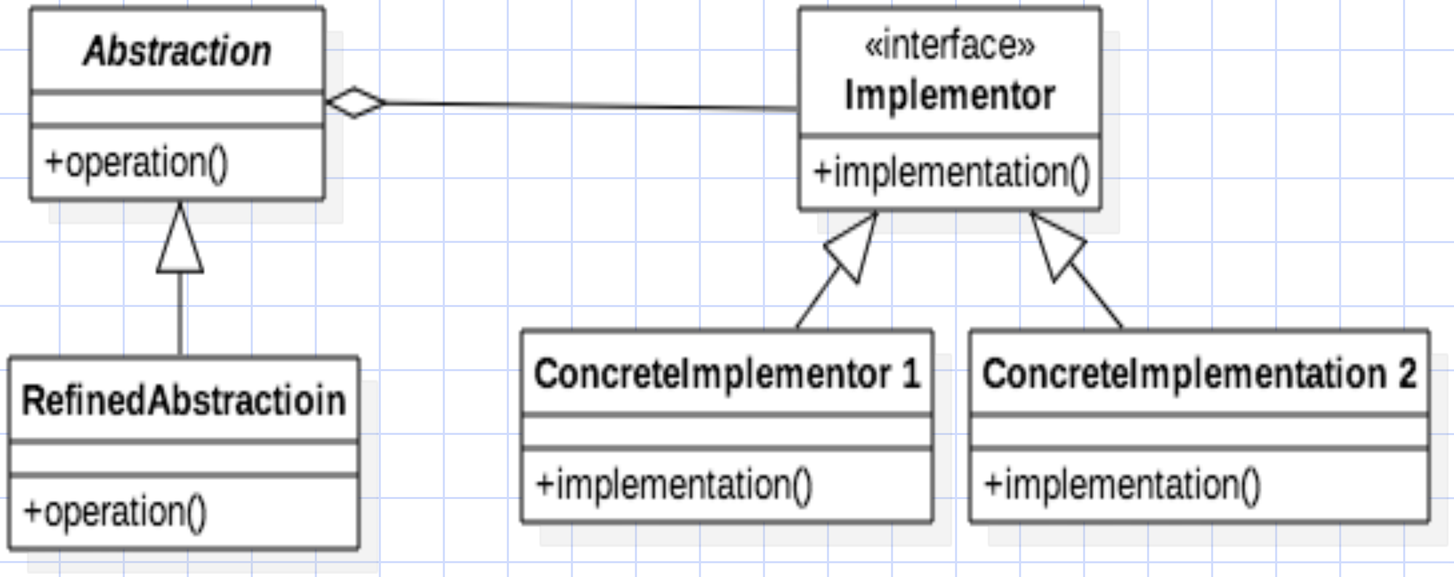
Key Elements of Bridge Pattern

- Two major parts of Bridge Pattern are: **Abstraction** and **Implementor** parts
  - **Abstraction:** An abstract class, which is the core element, and contains a reference to the implementor.
  - **Refined Abstraction:** Is a derivation of Abstraction, that provides finer details of an abstraction and hides them from implementor.
  - **Implementor:** An interface for implementation classes.
  - **Concrete Implementation:** implements implementor.

Advantages

- Decouple an abstraction from its implementation and you can change both parts independently.
- It is used mainly for implementing platform independence feature.

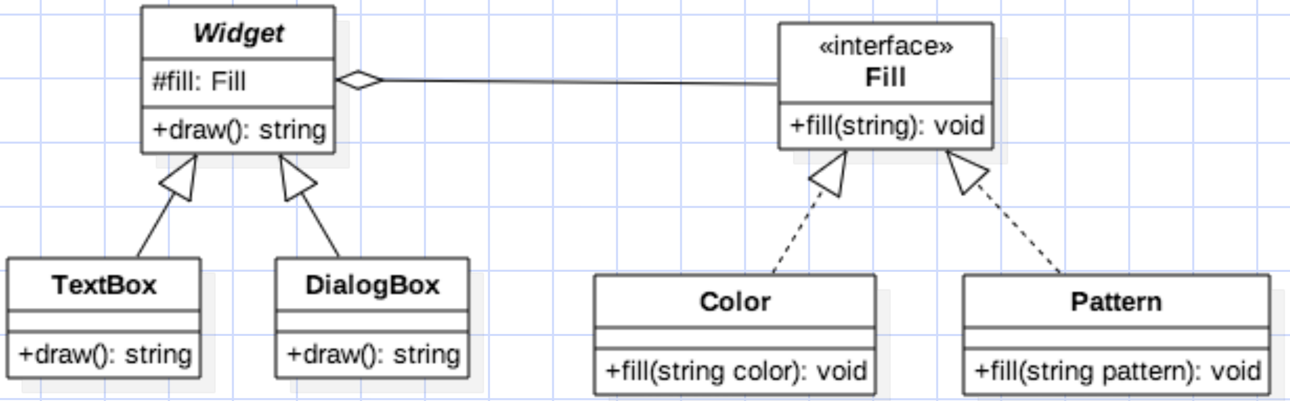
The UML Model



Bridge Design Pattern Example

```
interface Fill{
  public void fill (String f);
}

class Color implements Fill{
  public void fill(String f){
      System.out.println(”Fills with color ” + f);
  }
}

class Pattern implements Fill{
  public void fill(String p){
      System.out.println(”Fills with pattern” + p);
  }
}
```

```
abstract class Widget{
    protected f: Fill;
    abstract public string draw();
  }

class TextBox extends Widget{
  public string draw (){
      return (”Fill textbox with ” + f.fill());
  }
}

class DialogBox extends Widget{
  public string draw (){
      return (”Fill textbox with ” + f.fill());
  }
}
```



Abstract Factory

What is Abstract Factory Pattern

- The Abstract Factory is known as a **creational** pattern.
- it's used to construct objects such that they can be decoupled from the implementing system.
- The definition of Abstract Factory provided by GoF Gang states:
  - ***This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.***

Abstract Factor Pattern

- **Intent:** provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - **(From: Design Patterns. Elements of Object-Oriented Software, Erich Gamma, Richard Helm, Ralf Johnson, John Vlissides)**

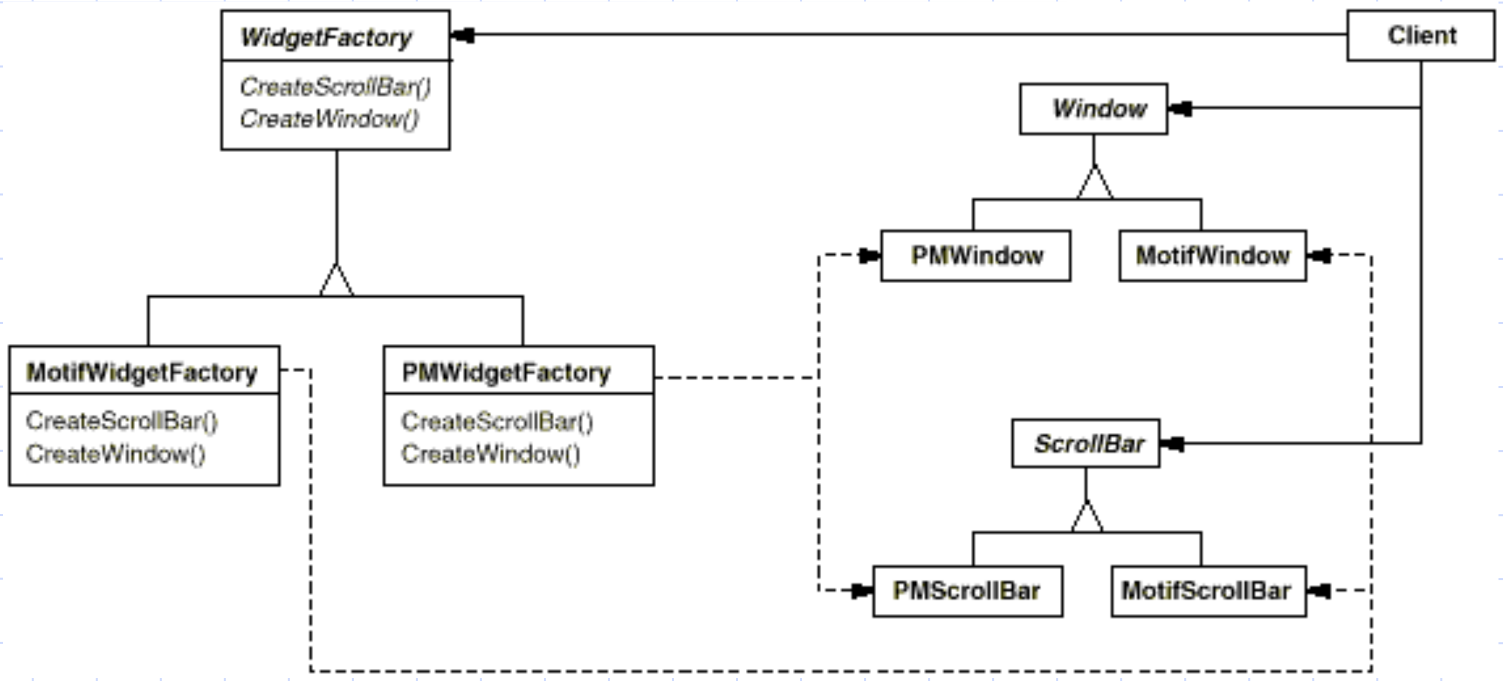
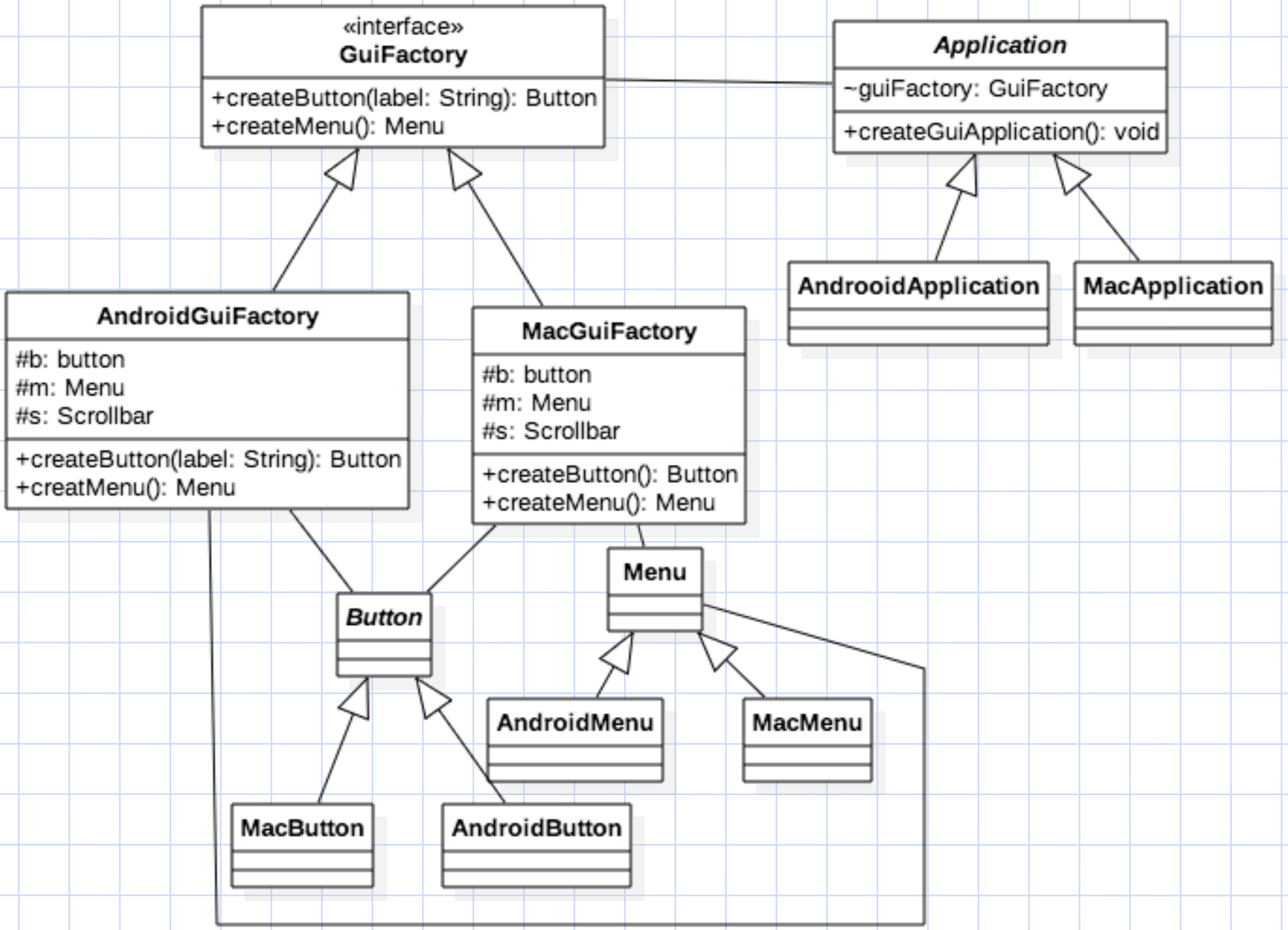


Figure from: Wikipedia

Where should you use this pattern?

- when your system must create multiple families of products or you want to provide a library of products without exposing the implementation details.
- A key characteristic is that the pattern will decouple the concrete classes from the client.
- An example of an Abstract Factory in use could be UI toolkits.
  - Across Windows, Mac and Linux, UI composites such as windows, buttons and textfields that their implementation vary across platforms.

Abstract Factory Pattern Learning By Example



```
interface GuiFactory   {
        public Button createButton(String label);
        public Menu createMenu();
}
```

```
class AndroidGuiFactory implements GuiFactory{
        Button b;
        Menu m;
        Scrollbar s;

        public Button createButton(String label){
                b = new AndroidButton(label);
                return b;
        }

        public Menu createMenu(){
                m = new AndroidMenu();
                return m;
        }
}
```

```
class MacGuiFactory implements GuiFactory{
        Button b;
        Menu m;
        Scrollbar s;

        public Button createButton(String label){
                b = new MacButton(label);
                return b;
        }

        public Menu createMenu(){
                m = new MacMenu();
                return m;
        }


}
```

```
class Button {
  abstract public void paint(String s);
}
```

```
class Menu {
        abstract  public void paint();
}
```

```
class AndroidButton extends Button{
        public AndroidButton(String type){
                paint(type);
        }
        public void paint(String type){
                System.out.println( type + " created.");
        }
}

class MacButton implements Button{
        public MacButton(String type){
                paint(type);
        }
        public void paint(String type){
                System.out.println(type + " created.");
        }
}
```

```
class AndroidMenu extends Menu{
        public AndroidMenu(){
                paint();
        }
        public void paint(){
                System.out.println("Android menu");
        }
}

class MacMenu implements Menu{
        public MacMenu(){
                paint();
        }
        public void paint(){
                System.out.println("Mac menu c");
        }
}
```

```
abstract class Application {
        GuiFactory guiFactory;
        abstract void createGuiApplication();
}
```

```
class MacApplication extends Application{
        public MacApplication () {
                guiFactory = new MacGuiFactory();
        }

        public void createGuiApplication(){
                guiFactory.createButton("Mac Button");
                guiFactory.createMenu();
        }
}
```

```
class AndroidApplication extends Application{
        public AndroidApplication () {
                guiFactory = new AndroidGuiFactory();
        }

        public void createGuiApplication(){
                guiFactory.createButton("Android Button");
                guiFactory.createMenu();
        }
}
```

```
public class DemoAbstractFactory {

    public static void main(String[] args) {
        Application application;
        application = new AndroidApplication();
        application.createGuiApplication();
        application = new MacApplication();
        application.createGuiApplication();
    }

}
```