Bridge Design Pattern :

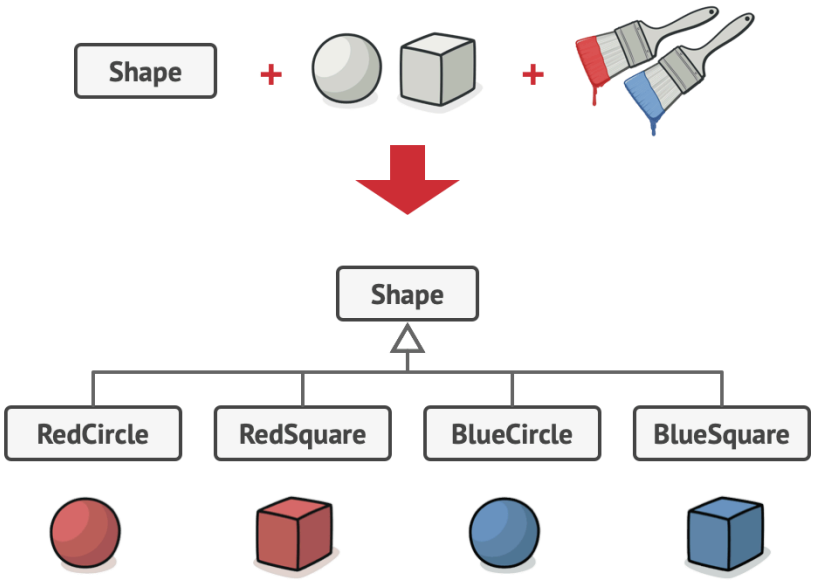
Abstraction has a implementation.

(Birbirine yakın olan iki sınıfı -> Abstraction-Implementation olarak 2 parça özelinde değerlendir. Bridge mean is "has -a")

Shape is a abstraction level. Color is a implementation level.

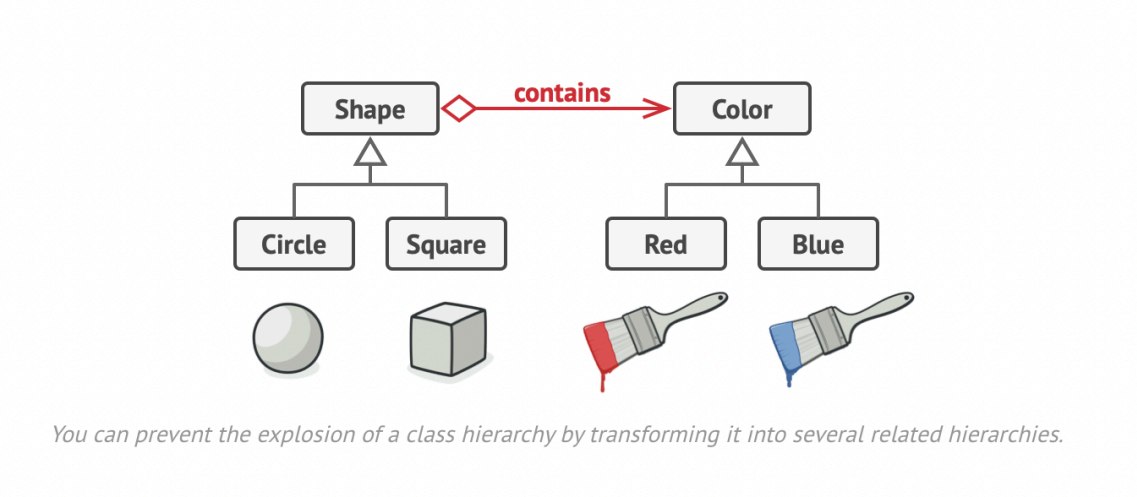
- Intent:
  - Bridge, büyük bir sınıfı veya birbirine yakın ilişkili sınıf kümesini iki ayrı hiyerarşiye—soyutlama ve gerçekleştirme—bölmenizi sağlayan yapısal bir tasarım desenidir. Bu iki hiyerarşi, birbirinden bağımsız olarak geliştirilebilir.

- Problem
  - *Abstraction? Implementation?* Sound scary? Stay calm and let's consider a simple example.
  - Diyelim ki bir **Geometrik Şekil (Shape)** sınıfınız var ve bunun **Daire (Circle)** ve **Kare (Square)** gibi iki alt sınıfı bulunuyor.
  - Bu sınıf hiyerarşisine renkleri eklemek istiyorsunuz; bu yüzden **Kırmızı (Red)** ve **Mavi (Blue)** şekil alt sınıfları oluşturmayı planlıyorsunuz.
  - Ancak, elinizde zaten iki alt sınıf olduğundan, **MaviDaire (BlueCircle)** ve **KırmızıKare (RedSquare)** gibi dört farklı sınıf kombinasyonu oluşturmanız gerekecek.

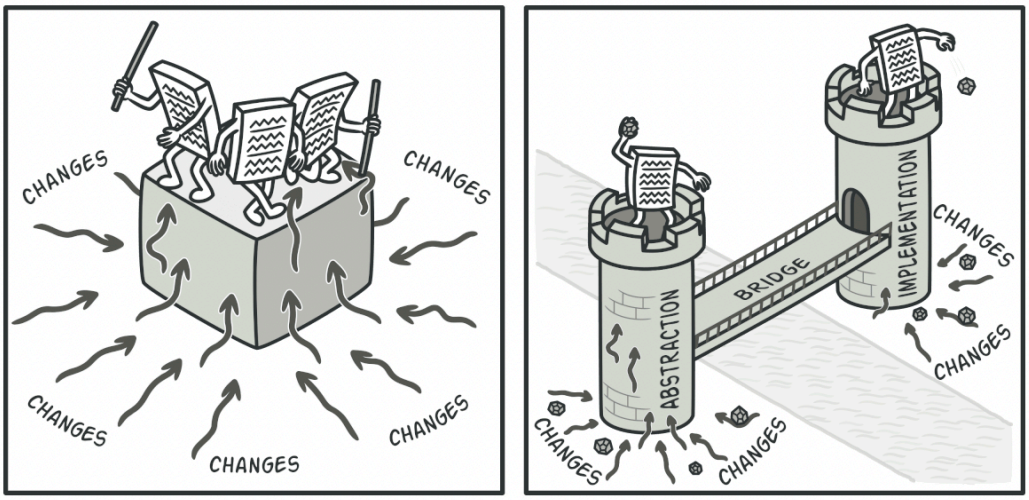


Number of class combinations grows in geometric progression.

Hiyerarşiye yeni şekil türleri ve renkler eklemek hiyerarşiyi katlanarak büyütecektir.
Örneğin, bir üçgen şekli eklemek için her renk için bir tane olmak üzere iki alt sınıf oluşturmanız gerekir.
Ve bundan sonra, yeni bir renk eklemek, her şekil türü için bir tane olmak üzere üç alt sınıf oluşturmayı gerektirecektir.
Ne kadar ileri gidersek, durum o kadar kötüleşiyor.

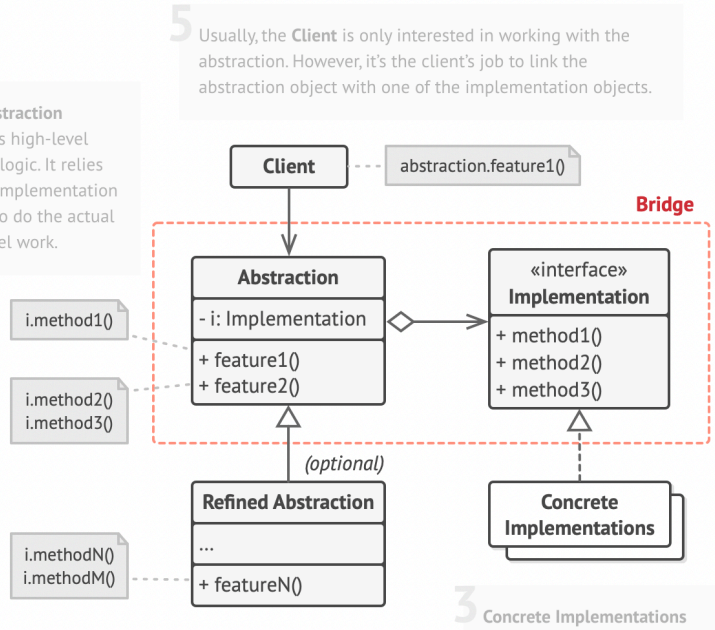
- Solution
  - Bu sorun, şekil sınıflarını iki bağımsız boyutta genişletmeye çalıştığımız için ortaya çıkıyor: şekle ve renge göre. Bu, sınıf kalıtımıyla ilgili çok yaygın bir sorundur.
  - The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
  - 

You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.
  -



Making even a simple change to a monolithic codebase is pretty hard because you must understand the entire thing very well. Making changes to smaller, well-defined modules is much easier.

Structure



1 The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

2 The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

3 **Concrete Implementations** contain platform-specific code.

4 **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

5 Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

```
public interface Color {
void fill();
}

public class RedColor implements Color {
@Override
public void fill() {
System.out.println("Filling red color");
}
}

public abstract class Shape {
protected final Color color;

protected Shape(Color color) {
this.color = color;
}

abstract void draw();
}

public class Circle extends Shape {
public Circle(Color color) {
super(color);
}

public void draw() {
color.fill();
}
}

public class BridgePatternExample {
public static void main(String[] args) {
// Create instances with different combinations of shapes and colors
Shape redCircle = new Circle(new RedColor());

redCircle.draw(); // Output: Drawing a circle. Filling with red color
}
}
```

Applicability:

Use the Bridge if you need to be able to switch implementations at runtime.