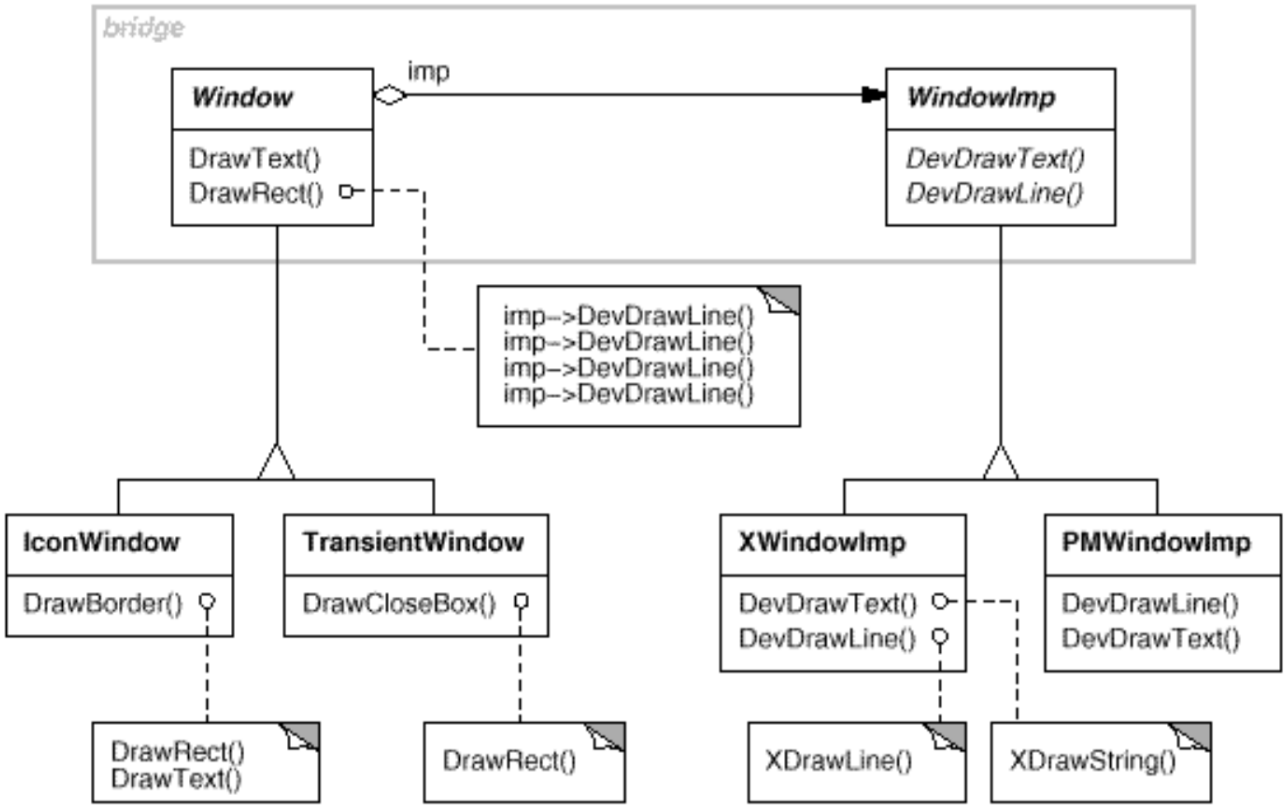
Bridge

↓ INTENT ↓

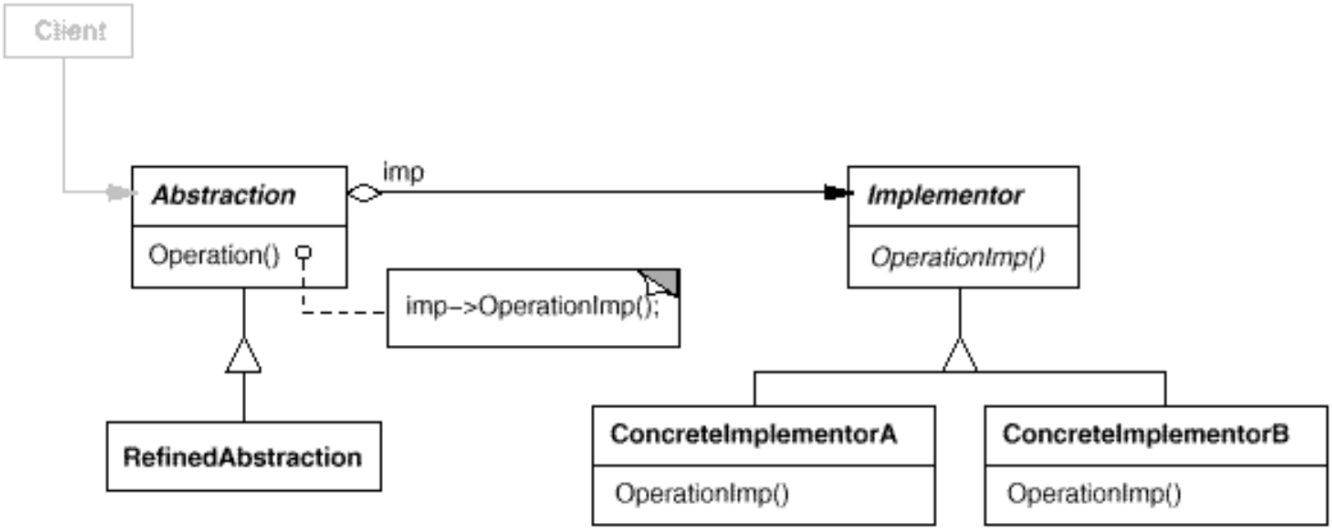
```
Disaccoppia un'astrazione dalla sua implementazione in modo che i due
possano variare indipendentemente. Per esempio nel caso di utilizzo di
librerie esterne.

(nel progetto lo usaiamo per isolare le librerie per inviare email e
creare pdf)
```

↓ MOTIVATION ↓



↓ STRUCTURE ↓



↓ IMPLEMENTATION ↓

---

1. Solo un implementazione.

Se abbiamo solo un'implementazione, non è necessario creare una classe Implementor astratta. In tal caso ci sarà una relazione uno a uno tra Abstraction e Implementor.

Questa separazione è ancora utile quando una modifica nell'implementazione di una classe non deve influenzare i suoi client esistenti.

2. Creazione dell'oggetto Implementor corretto.

Come, quando e dove decidi quale classe Implementor istanziare quando ce n'è più di una?

Un'implementazione di elenchi collegati può essere utilizzata per piccole raccolte e una tabella hash per quelle più grandi.

Un altro approccio consiste nello scegliere inizialmente un'implementazione predefinita e modificarla in seguito in base all'utilizzo.

Oppure si può delegare del tutto la decisione ad un altro oggetto. Nell'esempio Window / WindowImp, possiamo introdurre un oggetto Factory il cui unico compito è incapsulare le specifiche della piattaforma.

La factory sa che tipo di oggetto WindowImp creare per la piattaforma in uso; a Window gli chiede semplicemente un WindowImp e restituisce il tipo giusto.

↓ EXAMPLE ↓

DrawImplementation ↓

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

Shape ↓

```
public abstract class Shape {
    protected DrawImplementation drawImplementation;

    protected Shape(DrawImplementation drawAPI) {
        this.drawImplementation = drawAPI;
    }

    public abstract void draw();
}
```

Circle ↓

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

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

    @Override
    public void draw() {
        drawImplementation.drawCircle(x, y, radius); // specifico
dell'ambiente windowing usato a runtime
    }
}
```

RedCircle ↓

```
public class RedCircle implements DrawImplementation{

    @Override
    public void drawCircle(int x, int y, int radius) {
        System.out.println("cerchio rosso di: " + radius + x + y);
    }

}
```

GreenCircle ↓

```
public class GreenCircle implements DrawImplementation{

    @Override
    public void drawCircle(int x, int y, int radius) {
        System.out.println("cerchio verde di: " + radius + x + y);
    }

}
```

Main ↓

```
public static void main(String[] args) {
    Shape redC = new Circle(new RedCircle(), 100, 100, 10);
    Shape greC = new Circle(new GreenCircle(), 100, 100, 10);

    redC.draw();
    greC.draw();
}
```