Bridge

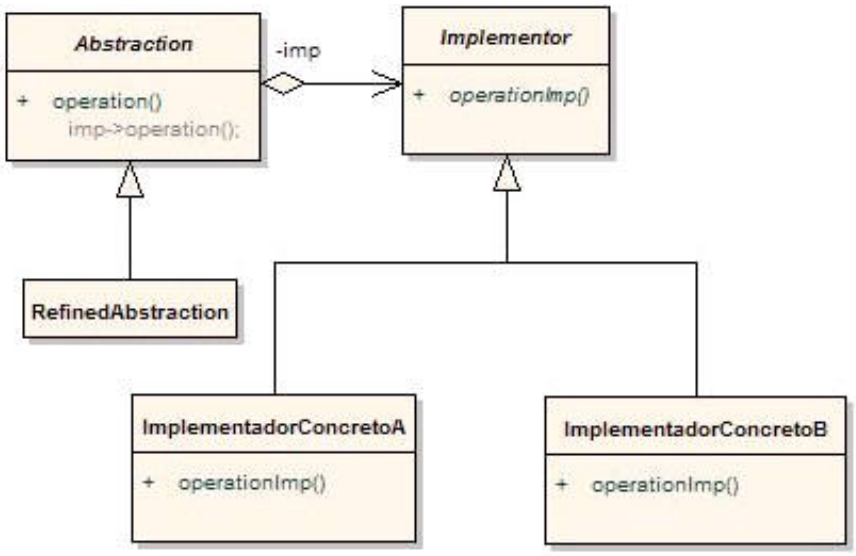
Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente. ¿Que quiere decir exactamente esto? Una abstracción se refiere a un comportamiento que una clase debería implementar, ya sea porque esta obligada por una interface o una clase abstracta. Por otro lado, con implementación se refiere a colocarle lógica a dicha obligación.
Con lo cual, este patrón permite modificar las implementaciones de una abstracción en tiempo de ejecución. Básicamente es una técnica usada en programación para desacoplar la interface de una clase de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra.

Cuando un objeto tiene unas implementaciones posibles, la manera habitual de implementación es el uso de herencias. Muchas veces la herencia se puede tornar inmanejable y, por otro lado, acopla el código cliente con una implementación concreta. Este patrón busca eliminar la inconveniencia de esta solución.

Este patrón debe ser utilizado cuando:

- Se desea evitar un enlace permanente entre la abstracción y su implementación. Esto puede ser debido a que la implementación debe ser seleccionada o cambiada en tiempo de ejecución.
- Tanto las abstracciones como sus implementaciones deben ser extensibles por medio de subclases. En este caso, el patrón Bridge permite combinar abstracciones e implementaciones diferentes y extenderlas independientemente.
- Cambios en la implementación de una abstracción no deben impactar en los clientes, es decir, su código no debe tener que ser recompilado.
- Se desea compartir una implementación entre múltiples y este hecho debe ser escondido a los clientes.
- Permite simplificar jerarquías demasiado pobladas.

Diagrama UML



Abstraction: define una interface abstracta. Mantiene una referencia a un objeto de tipo Implementor.
RefinedAbstraction: extiende la interface definida por Abstraction
Implementor: define la interface para la implementación de clases. Esta interface no se tiene que corresponder exactamente con la interface de Abstraction; de hecho, las dos interfaces pueden ser bastante diferentes entre sí. Típicamente la interface Implementor provee sólo operaciones primitivas, y Abstraction define operaciones de alto nivel basadas en estas primitivas.
ImplementadorConcreto: implementa la interface de Implementor y define su implementación concreta.

Abstraction emite los pedidos de los clientes a su objeto Implementor. El cliente no tiene que conocer los detalles de la implementación.

Ejemplo

Imaginemos que necesitamos dibujar distintas figuras geométricas (círculo, rectángulo, etc). Cada figura geométrica puede ser dibujada con diferentes tipos de líneas (normal, punteada, etc).

¿Que alternativas tenemos?

Por ejemplo, podría hacer una clase dibujo con todos los tipos de dibujos posibles. Pero a medida que me agreguen dibujos y formas, se va a parecer a un antipatrón (the God Class). Con esto quiere decir, que sería inmanejable.
Entonces podría hacer la clase Círculo y Rectángulo y que ellas dibujen su propia forma. Pero que pasa si me modifican el Dibujo en sí; por ejemplo, el espesor de la línea punteada tiene que se más grueso...¿porque un círculo debería tener conocimientos tan finos de dibujo?
Se podrían pensar más alternativas pero ninguna terminará de satisfacer en un 100% nuestro problema. Es por ello que existe el Bridge.

Vamos a combinar las clases de tal forma que tanto un Círculo como un Rectángulo sepan como dibujarse de manera abstracta, pero le dejaremos la implementación a un especialista en dibujo.

```
public abstract class Dibujo {

    public abstract void dibujaRectangulo(double x1, double y1, double x2,
            double y2);

    public abstract void dibujaCirculo(double x, double y, double r);

}
```

```
public class DibujandoNormal extends Dibujo {

    public void dibujaRectangulo(double x1, double y1, double x2, double y2) {
        System.out.println("Dibujando un rectangulo normal...");
    }

    public void dibujaCirculo(double x, double y, double r) {
        System.out.println("Dibujando un circulo...");
    }

}
```

Obviamente estas clases no debería realizar una simple salida por consola sino que debería poseer el algoritmo del dibujo, pero nos sirve a modo de ejemplo.

```
public class DibujandoPunteado extends Dibujo {

    public void dibujaRectangulo(double x1, double y1, double x2, double y2) {
        System.out.println("Dibujando un rectangulo punteado...");
    }

    public void dibujaCirculo(double x, double y, double r) {
        System.out.println("Dibujando un circulo punteado...");

    }
}
```

Bien, ya tenemos las implementaciones de ambos dibujos. Ahora veamos de utilizarlas en las clases correspondientes.

```
public abstract class Forma {
    private Dibujo dibujo;

    public Forma(Dibujo d) {
        dibujo = d;
    }

    public abstract void dibuja();

    public void dibujaRectangulo(double x1, double y1, double x2, double y2) {
        dibujo.dibujaRectangulo(x1, y1, x2, y2);
    }

    public void dibujaCirculo(double x, double y, double r) {
        dibujo.dibujaCirculo(x, y, r);
    }

}
```

```
    public class Circulo extends Forma {
        private double coordenadaX, coordenadaY, coordenadaR;

        public Circulo(Dibujo d, double x, double y, double r) {
            super(d);
            coordenadaX = x;
            coordenadaY = y;
            coordenadaR = r;
        }

        public void dibuja() {
            dibujaCirculo(coordenadaX, coordenadaY, coordenadaR);
        }

    }
```

```
public class Rectangulo extends Forma {
    private double x1, x2, y1, y2;

    public Rectangulo(Dibujo dibujo, double x1, double y1, double x2, double y2) {
        super(dibujo);
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }

    public void dibuja() {
        dibujaRectangulo(x1, y1, x2, y2);
    }

}
```

Veamos como se llama desde el cliente:

```
public class Main {

    public static void main(String[] args) {

        Rectangulo rectangulo = new Rectangulo(new DibujandoPunteado(), 1, 1,2, 2);
        rectangulo.dibuja();

        Circulo circulo = new Circulo(new DibujandoNormal(), 2, 2, 3);
        circulo.dibuja();
    }

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Main (1) [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (03/06/2011 10:05:43)
Dibujando un rectangulo punteado...
Dibujando un circulo...
```

Consecuencias

- Desacopla interface e implementación: una implementación no es limitada permanentemente a una interface. Le es posible a un objeto cambiar su implementación en tiempo de ejecución. Este desacoplamiento fomenta las capas, que pueden conducir a un sistema mejor estructurado.
- La parte de alto nivel de un sistema sólo tiene que conocer Abstraction e

Implementor.

- Mejora la extensibilidad: se puede extender las jerarquías de Abstraction e Implementor independientemente.
- Esconde los detalles de la implementación a los clientes