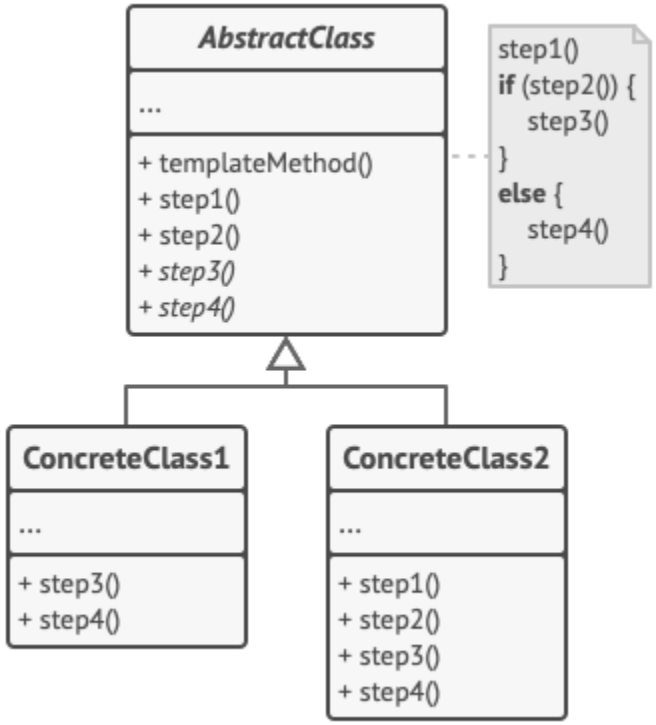
# Diagrama de clases del patrón:




Este patrón define el funcionamiento general de un algoritmo en una operación de una clase, delegando en otras clases, a través de herencia o dependencia entre clases.
Permite implementar en una clase abstracta el código común que será usado por las clases que heredan de ella, permitiéndoles que implementen el comportamiento que varía mediante la reescritura (total o parcial) de determinados método

# Codigo generico

Persona.java

```java
public abstract class Persona {

    private String nombre;
    private int dni;

    public String getNombre() {
        return nombre;
    }

    public Persona(String nombre, int dni) {
        super();
        this.nombre = nombre;
        this.dni = dni;
    }

    public void identificar() {
        if (estipo()) {
            accion();
        }
    }

    public abstract boolean estipo();

    public abstract void accion();
}
```

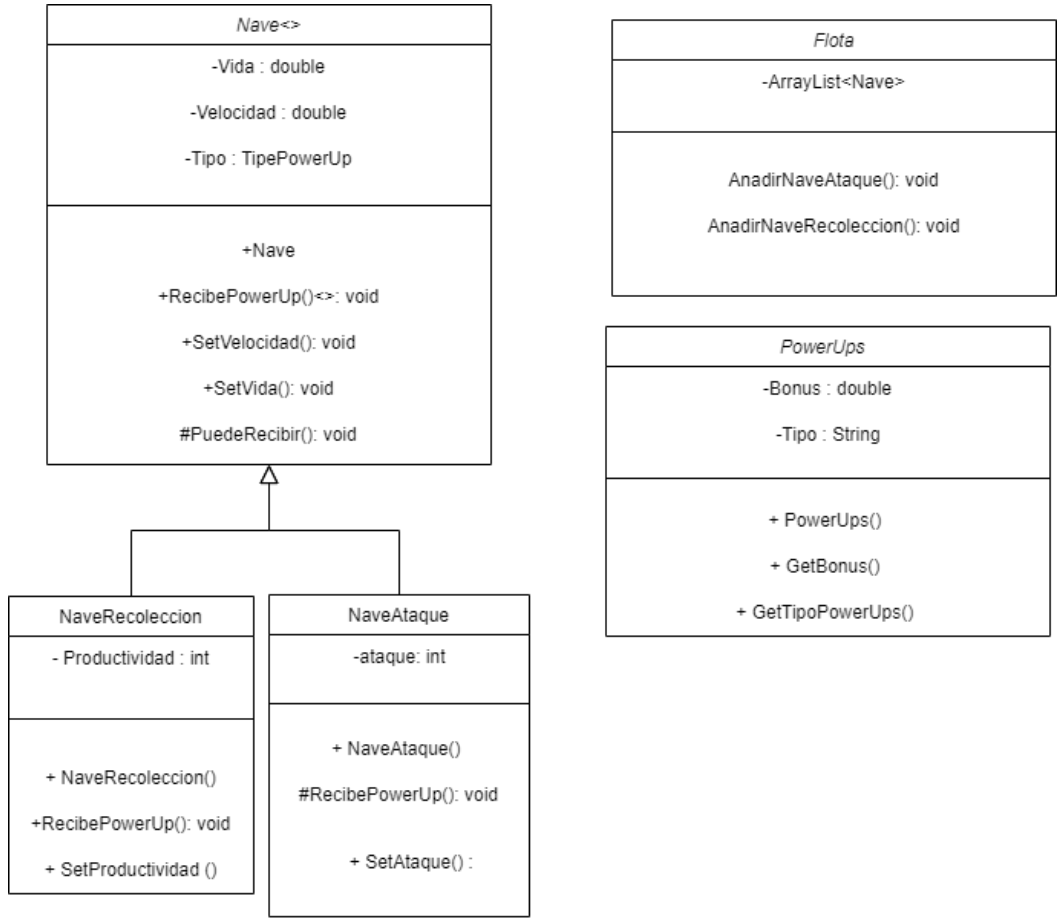
EmpleadoA.java

```java
kage paradigmas.template;

lic class EmpleadoA extends Persona {

public EmpleadoA(String nombre, int dni) {
    super(nombre, dni);
    // TODO Auto-generated constructor stub
}

public boolean estipo() {
    return super.getNombre().equals("EmpleadoA");
}

public void accion() {
    System.out.println("Soy EmpleadoA");
}
```

EmpleadoB.java

```java
package paradigmas.template;

public class EmpleadoB extends Persona {

    public EmpleadoB(String nombre, int dni) {
        super(nombre, dni);
        // TODO Auto-generated constructor stub
    }

    public boolean estipo() {
        return super.getNombre().equals("EmpleadoB");
    }

    public void accion() {
        System.out.println("EmpleadoB");
    }
}
```

# Diagrama de clases con contexto:



# Fragmento de código:

Nave.java

```java
public abstract class Nave {

    protected double velocidad;
    protected double vida;
    protected TipoPowerUp tipo;

    public abstract boolean puedeRecibir(PowerUp power);
    public abstract void respuesta(PowerUp power);

    public final void recibePowerUp(PowerUp power){
        if (this.puedeRecibir(power)) {
            respuesta(power);
        }
    }

```

Desventaja: Es poco útil mientras menos cantidad de subclases se utilicen

Ventaja: Poder tener diferentes implementaciones para el mismo método con un formato ya definido

NaveAtaque.java

```java
            this.tipo = TipoPowerUp.DISPAROS;
        }

        public void setAtaque(double ataque) {
            this.ataque = ataque;
        }

        @Override
        public boolean puedeRecibir(PowerUp power) {
            // return power.getTipoPowerUp() == this.tipo;
            return power.isTipoAtaque();
        }

        @Override
        public void respuesta(PowerUp power) {
            ataque += power.getBonus();
            System.out.println("Recibio PowerUp ataque");
        }
}

```

NaveRecoleccion.java

```java
    }

    public void setProductividad(double productividad) {
        this.productividad = productividad;
    }

    @Override
    public boolean puedeRecibir(PowerUp power) {
        //return power.getTipoPowerUp() == this.tipo;
        return power.isTipoRecolecion();
    }

    @Override
    public void respuesta(PowerUp power) {
        productividad += power.getBonus();
        System.out.println("Recibio PowerUp produccion");
    }

}
```