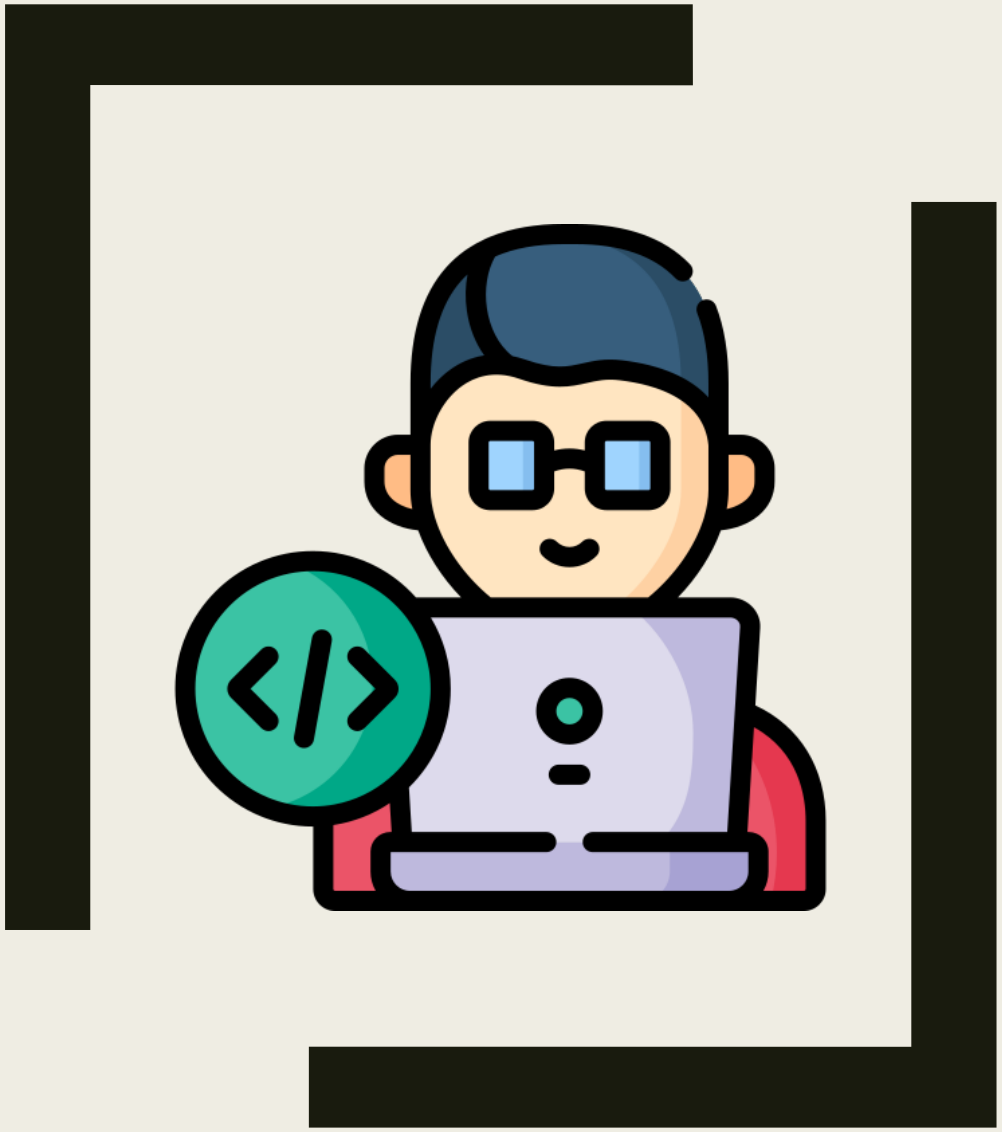
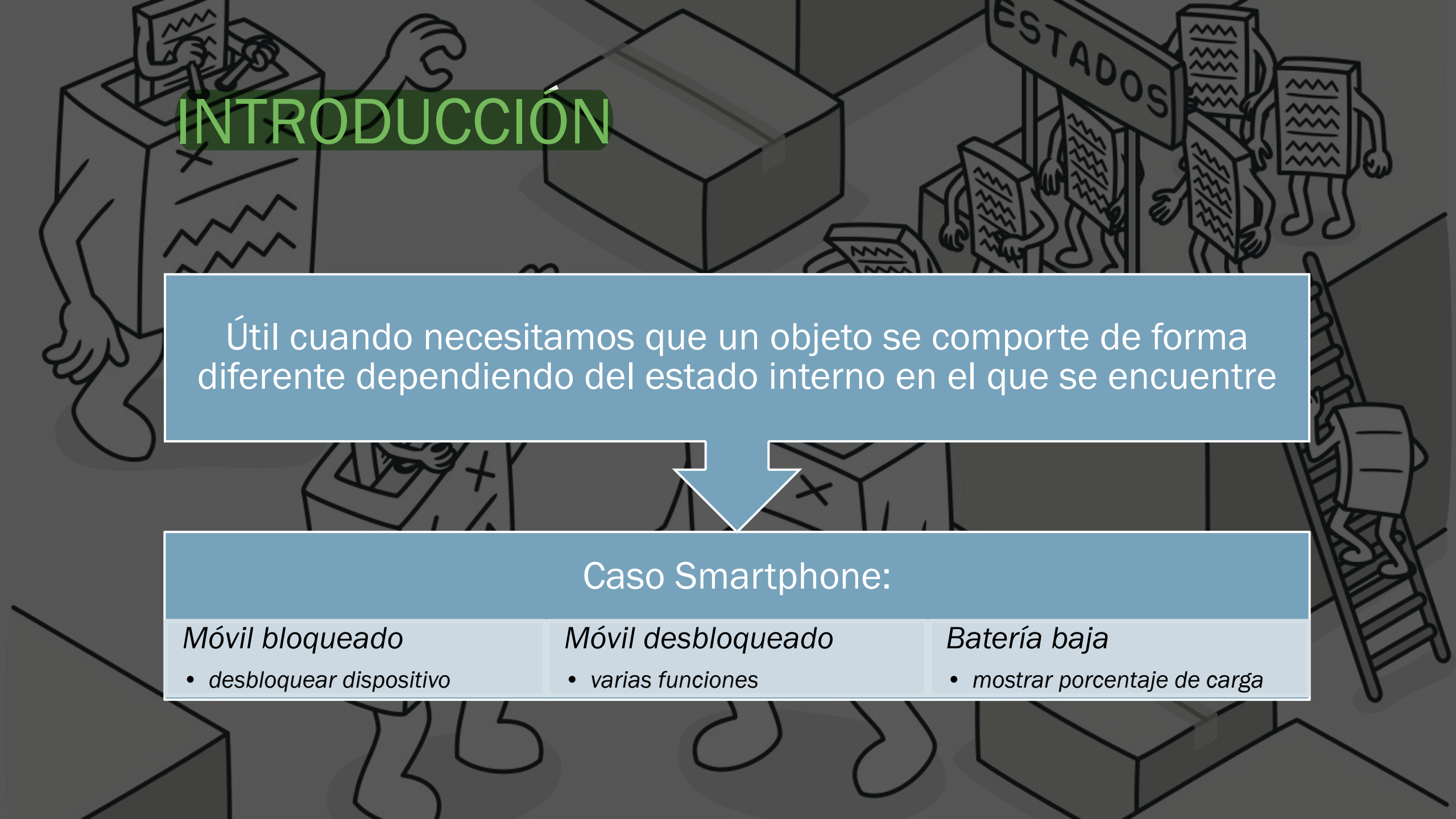


PATRÓN STATE

Juan García A. , Tomás Machín y Paloma Pérez de Madrid

INTRODUCCIÓN

- Permite al objeto cambiar su comportamiento cuando su estado interno cambia
- Relacionado con el concepto de Máquinas de Estado
- “Un Switch de clases”
- Sugiere que extraigas todo el código específico del estado y lo metas dentro de un grupo de clases específica



INTRODUCCIÓN

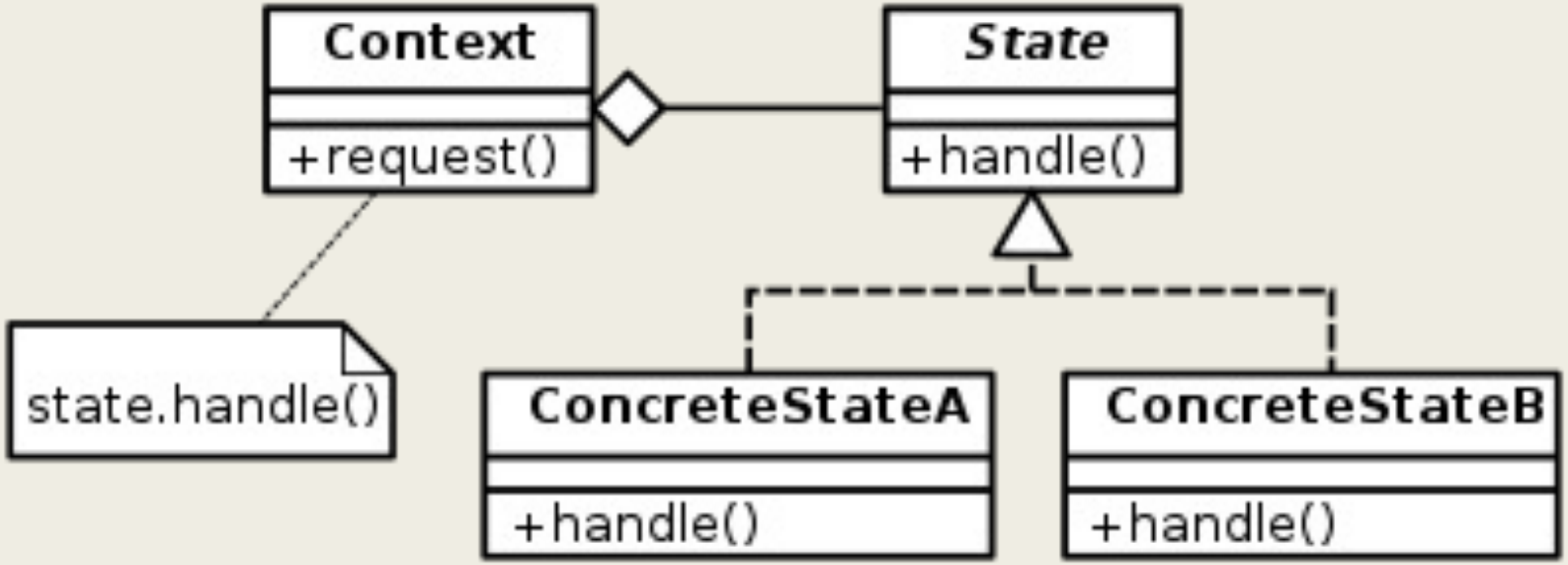
Útil cuando necesitamos que un objeto se comporte de forma diferente dependiendo del estado interno en el que se encuentre

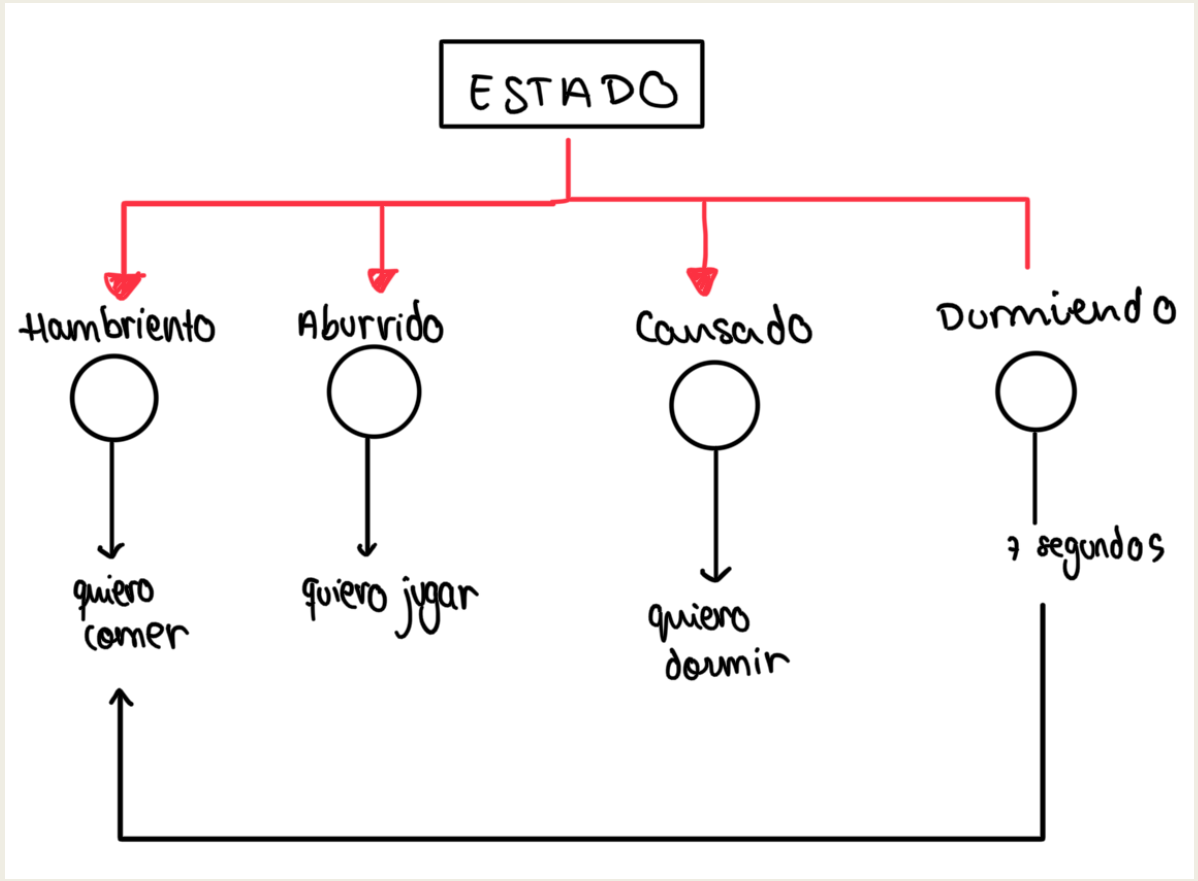
Caso Smartphone:

Móvil bloqueado	*Móvil desbloqueado*	*Batería baja*
• *desbloquear dispositivo*	• *varias funciones*	• *mostrar porcentaje de carga*

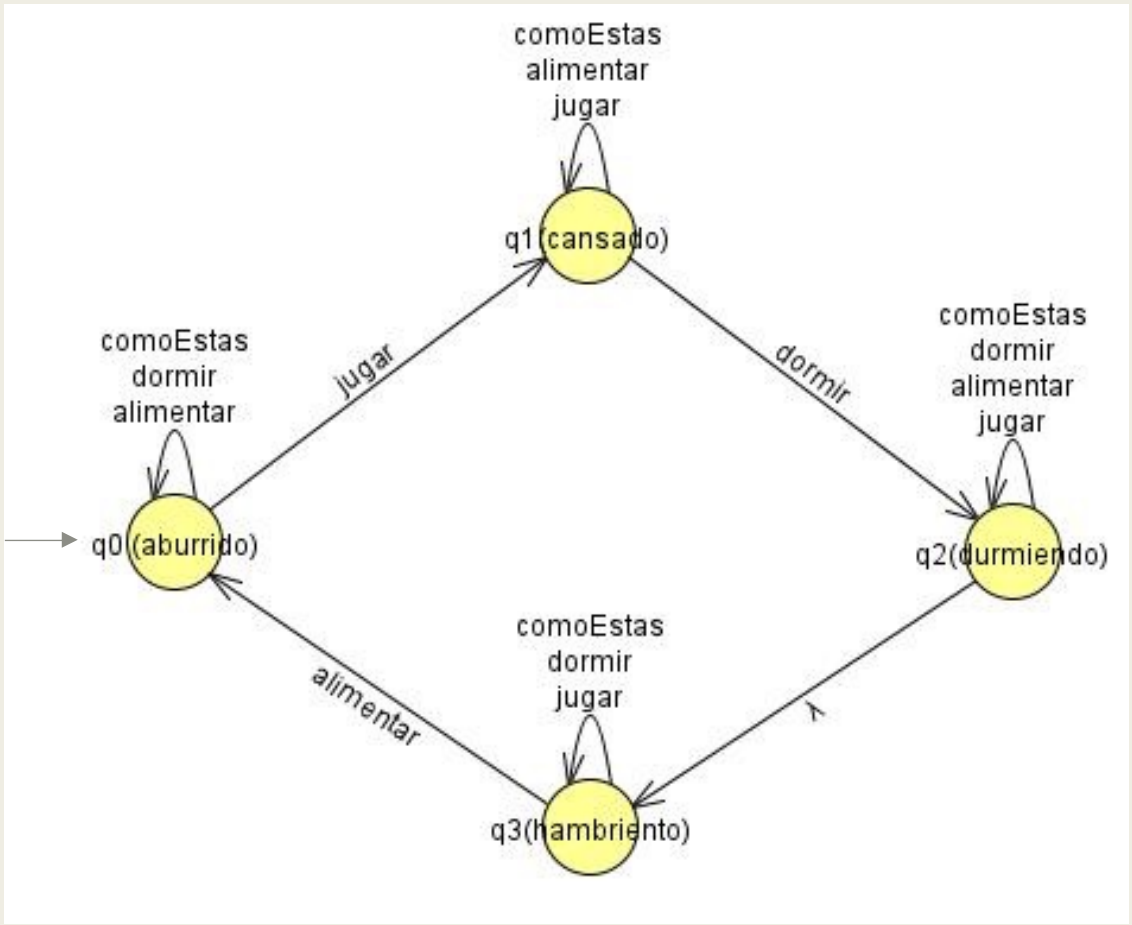
CÓDIGO DE EJEMPLO

- El objeto cuyo estado es susceptible de cambiar (Contexto) contendrá una referencia a otro objeto que define los distintos tipos de estado en que se puede encontrar.





CÓDIGO DE EJEMPLO



CÓDIGO DE EJEMPLO

CÓDIGO → State.java y Tamagochi.java

Tamagotchi.java

```java
package Aplicacion;


public class Tamagotchi {
    private State estado;

    public Tamagotchi(){
        setState(new Aburrido());
    }
    public void setState(State estado){
        this.estado = estado;
        this.estado.setTamagotchi(this);
    }

    public void alimentar() {
        this.estado.alimentar();
    }
    public void dormir(){
        this.estado.dormir();
    }
    public void jugar(){
        this.estado.jugar();
    }
    public void comoEstas(){
        this.estado.comoEstas();
    }
}
```

State.java

```java
package Aplicacion;

public interface State {
    void jugar();
    void alimentar();
    void dormir();
    void comoEstas();

    void setTamagotchi(Tamagotchi tamagotchi);
}

```

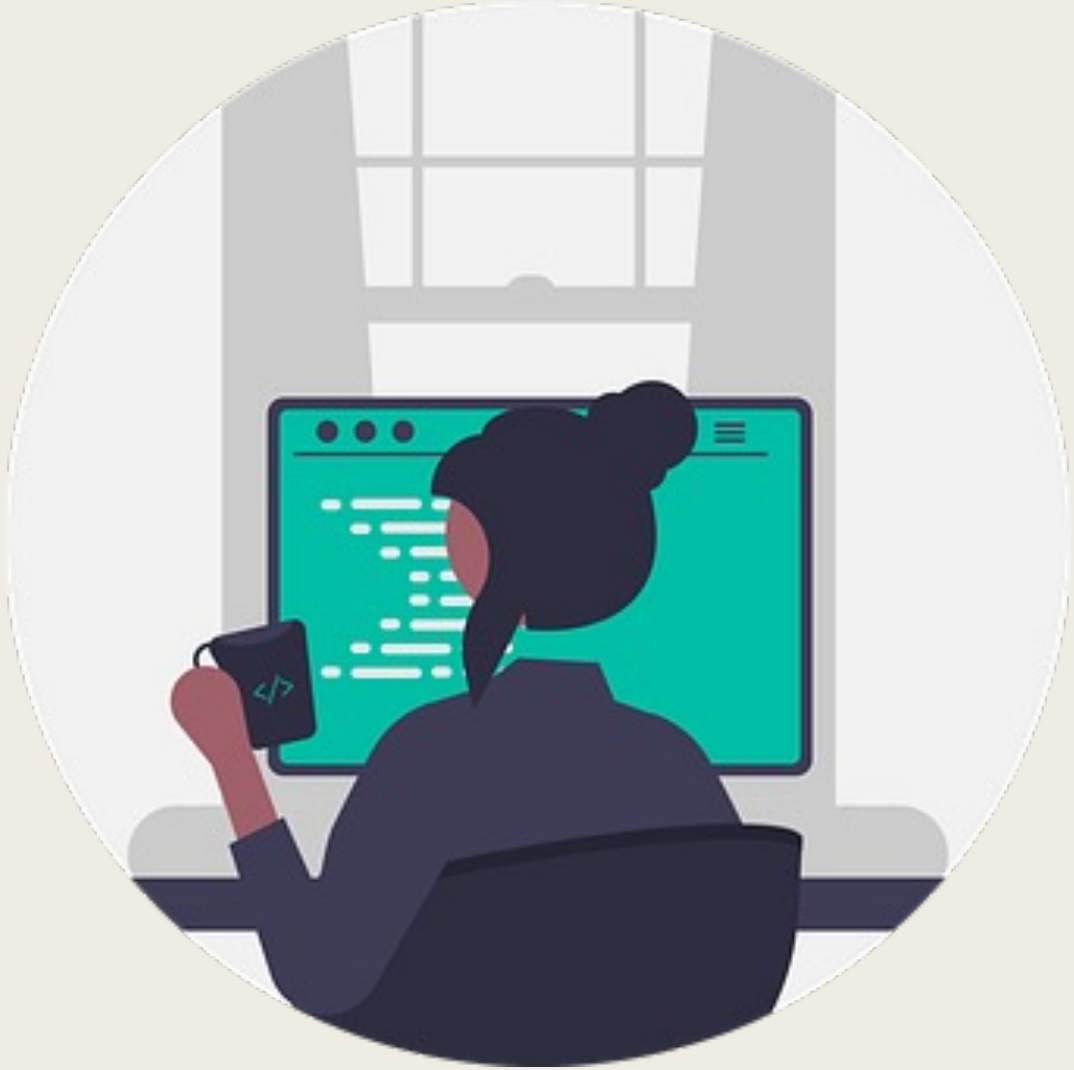
CÓDIGO → concreteStateA y concreteStateB

Aburrido.java

```java
package Aplicacion;

public class Aburrido implements State{
private Tamagotchi tamagotchi;
    @Override
    public void jugar() {
        System.out.println("Juguemos!!!!");
        tamagotchi.setState(new Cansado());
    }

    @Override
    public void alimentar() {
        System.out.println("No quiero comer!");
    }

    @Override
    public void dormir() {
        System.out.println("No quiero dormir!");
    }

    @Override
    public void comoEstas() {
        System.out.println("Estoy aburrido, quiero jugar");
    }

    @Override
    public void setTamagotchi(Tamagotchi tamagotchi) {
        this.tamagotchi = tamagotchi;
    }
}
```

Cansado.java

```java
package Aplicacion;

public class Cansado implements State {

    private Tamagotchi tamagotchi;

    @Override
    public void jugar() {
        System.out.println("Estoy muy cansado para jugar");
    }

    @Override
    public void alimentar() {
        System.out.println("no quiero comer");
    }

    @Override
    public void dormir() {
        System.out.println("Buenas noches uwu");
        tamagotchi.setState(new Durmiendo());
    }

    @Override
    public void comoEstas() {
        System.out.println("Tengo sueño");
    }

    @Override
    public void setTamagotchi(Tamagotchi tamagotchi) {
        this.tamagotchi = tamagotchi;
    }
}

```



VENTAJAS

- Principio de responsabilidad única.
- Principio de abierto/cerrado.
- Simplifica el código del contexto → eliminando voluminosos condicionales de máquina de estados.

INCONVENIENTES

- Excesivo → pocos estados
- Complejo → si los estados son parecidos
- Difícil mantenimiento
- Complejo para el porgramador

BIBLIOGRAFÍA

- Refactoring:
  - *https://refactoring.guru/es/design-patterns/state*
  - *https://refactoring.guru/es/design-patterns/state/python/example*

- Informaticapc.com:
  - *https://informaticapc.com/patrones-de-diseno/state.php*