State Pattern

Introducción y nombre

State. De Comportamiento. Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

Intención

Busca que un objeto pueda reaccionar según su estado interno. Si bien muchas veces esto se puede solucionar con un boolean o utilizando constantes, esto suele terminar con una gran cantidad de if-else, código ilegible y dificultad en el mantenimiento. La intención del State es desacoplar el estado de la clase en cuestión.

También conocido como

Objects for State, Patrón de estados.

Motivación

En determinadas ocasiones se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra. Esto resulta complicado de manejar, sobretodo cuando se debe tener en cuenta el cambio de comportamientos y estados de dicho objeto, todos dentro del mismo bloque de código. El patrón State propone una solución a esta complicación, creando un objeto por cada estado posible.

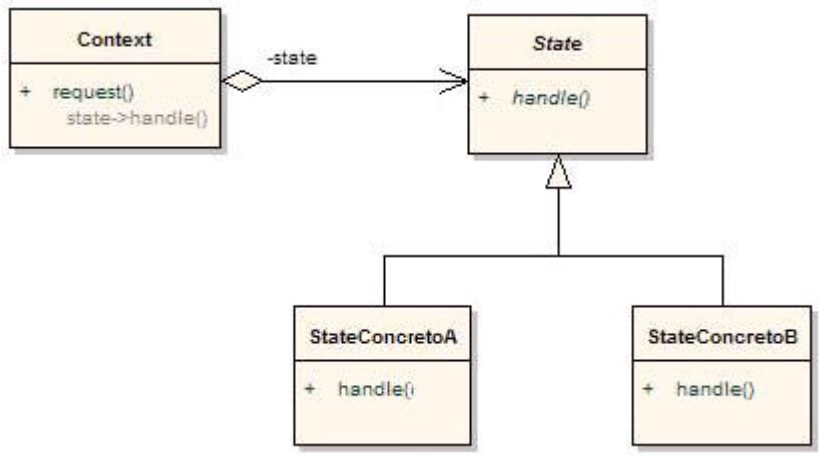
Solución

Este patrón debe ser utilizado cuando:

El comportamiento de un objeto depende de un estado, y debe cambiar en tiempo de ejecución según el comportamiento del estado.

Cuando las operaciones tienen largas sentencias con múltiples ramas que depende del estado del objeto.

Diagrama UML



Participantes

Context: mantiene una instancia con el estado actual

State: define interfaz para el comportamiento asociado a un determinado estado del Contexto.
StateConcreto: cada subclase implementa el comportamiento asociado con un estado del contexto.

Colaboraciones

El Context delega el estado específico al objeto StateConcreto actual. Un objeto Context puede pasarse a sí mismo como parámetro hacia un objeto State. De esta manera la clase State puede acceder al contexto si fuese necesario. Context es la interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos State. Una vez hecho esto estos clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context como los objetos de StateConcreto pueden decidir el cambio de estado.

Consecuencias

Se localizan fácilmente las responsabilidades de los estados concretos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.
Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.
Facilita la ampliación de estados mediante una simple herencia, sin afectar al Context.
Permite a un objeto cambiar de estado en tiempo de ejecución.
Los estados pueden reutilizarse: varios Context pueden utilizar los mismos estados.
Se incrementa el número de subclases.

Implementación

La clase Context envía mensajes a los estados concretos dentro de su código para brindarle a éstos la responsabilidad que debe cumplir el objeto Context. Así el objeto Context va cambiando las responsabilidades según el estado en que se encuentra. Para llevar a cabo esto se puede utilizar dos tácticas: que el State sea una interfaz o una clase abstracta. Para resolver este problema, siempre se debe intentar utilizar una interfaz, exceptuando aquellos casos donde se necesite repetir un comportamiento en los estados concretos: para ello lo ideal es utilizar una clase abstracta (state) con los métodos repetitivos en código, de manera que los estados concretos lo hereden. En este caso, el resto de los métodos no repetitivos deberían ser métodos abstractos.
Un tipo muy importante a tener en cuenta: el patrón no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de solucionar esto: definiendo estas transiciones dentro de la clase contexto, la otra es definiendo estas transiciones en las subclaes de State. Es más conveniente utilizar la primer solución cuando el criterio a aplicar es fijo, es decir, no se modificará. En cambio la segunda resulta conveniente cuando este criterio es dinámico, el inconveniente aquí se presenta en la dependencia de código entre las subclases.

Código de muestra

Veamos un código muy sencillo de entender:

[code]
```
public interface SaludState {
    public String comoTeSentis();
}

public class DolorDeCabeza implements SaludState {
    @Override
```

```
    public String comoTeSentis() {
        return "Todo mal: me duele la cabeza";
    }
}

public class DolorDePanza implements SaludState {
    @Override
    public String comoTeSentis() {
        return "Todo mal: me duele la panza";
    }
}

public class Saludable implements SaludState {
    @Override
    public String comoTeSentis() {
        return "Pipi Cucu!!";
    }
}

public class Persona {
    private String nombre;
    private SaludState salud;

    public Persona() {
        salud = new Saludable();
    }

    public void estoyBien() {
        salud = new Saludable();
    }

    public void dolorDeCabeza() {
        salud = new DolorDeCabeza();
    }

    public void dolorDePanza() {
        salud = new DolorDePanza();
    }

    public String comoTeSentis() {
        return salud.comoTeSentis();
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public SaludState getSalud() {
        return salud;
    }

    public void setSalud(SaludState salud) {
        this.salud = salud;
    }
}

public class Main {
    public static void main(String[] args) {
        Persona p = new Persona();
        p.setNombre("Juan");
        System.out.println(p.comoTeSentis());

        p.dolorDeCabeza();
```

```
        System.out.println(p.comoTeSentis());
    }
}
```

La salida por consola es:
Pipi Cucu!!
Todo mal: me duele la cabeza
[/code]

Cuándo utilizarlo

Este patrón se utiliza cuando un determinado objeto tiene diferentes estados y también distintas responsabilidades según el estado en que se encuentre en determinado instante. También puede utilizarse para simplificar casos en los que se tiene un complicado y extenso código de decisión que depende del estado del objeto.

Imaginemos que vamos a un banco y cuando llegamos nos colocamos en la fila de mostrador: si la misma esta abierta, seguiremos en la fila. En cambio, si esta cerrada nos colocaremos en otra fila o tomaremos alguna decisión acorde. Por otro lado, si vemos un cartel que dice "enseguida vuelvo" quizás tenemos que contemplar el tiempo disponible que tenemos. Es decir, para nosotros, el comportamiento de un banco cambia radicalmente según el estado en el que se encuentre. Para estas ocasiones, es ideal el uso de un patrón de estados.

Patrones relacionados

El patrón State puede utilizar el patrón Singleton cuando requiera controlar que una sola instancia de cada estado. Lo puede utilizar cuando se comparten los objetos como Flyweight existiendo una sola instancia de cada estado y esta instancia es compartida con más de un objeto.