# 20. State

(GoF pag. 305)

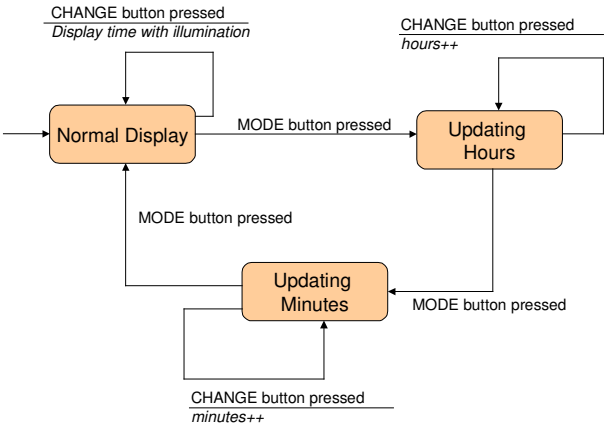
## 20.1. Descrizione

Consente ad un oggetto modificare il suo comportamento quando il suo stato interno cambia.

## 20.2. Esempio

Si pensi ad un orologio che possiede due pulsanti: MODE e CHANGE. Il primo pulsante serve per settare il modo di operazione di tre modi possibili: “visualizzazione normale”, “modifica delle ore” o “modifica dei minuti”. Il secondo pulsante, invece, serve per accendere la luce del display, se è in modalità di visualizzazione normale, oppure per incrementare in una unità le ore o i minuti, se è in modalità di modifica di ore o di minuti.

Il seguente diagramma di stati serve a rappresentare il comportamento dell’orologio:




In questo esempio, un approccio semplicistico conduce all’implementazione del codice di ogni operazione come una serie di decisioni:

```
operation buttonCHANGEpressed{

   if( clockState = NORMAL_DISPLAY )
      displayTimeWithLight();
   else if( clockState = UPDATING_HOURS )
      hours++;
   else if( clockState = UPDATING_MINUTES )
      minutes++;
   ...

}
```

Il problema di questo tipo di codice è che si rende più difficile la manutenzione, perché la creazione di nuovi stati comporta la modifica di tutte le operazioni dove essi sono testati. Da un'altra parte non si tiene una visione dello stato, in modo di capire come agisce l’oggetto

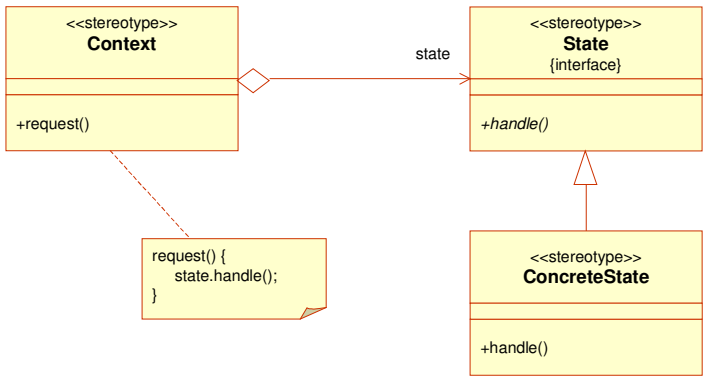
(l'orologio in questo caso), a seconda del proprio stato, perché questo comportamento è spezzato dentro l'insieme di operazioni disponibili.

Si vuole definire un meccanismo efficiente per gestire i diversi comportamenti che devono avere le operazioni di un oggetto, secondo gli stati in cui si trovi.

## 20.3. Descrizione della soluzione offerta dal Pattern

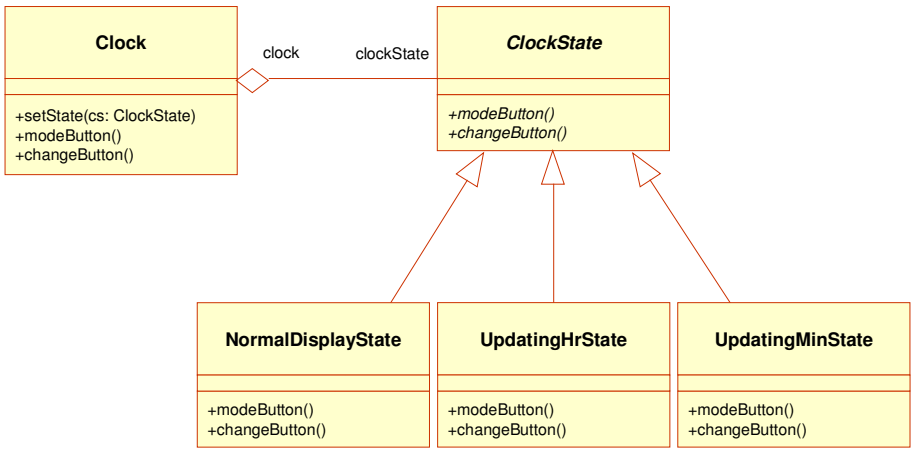
Il pattern "*State*" suggerisce incapsulare, all'interno di una classe, il modo particolare in cui le operazioni di un oggetto (Context) vengono svolte quando lo si trova in quello stato. Ogni classe (ConcreteState) rappresenta un singolo stato possibile del Context e implementa una interfaccia comune (State) contenente le operazioni che il Context delega allo stato. L'oggetto Context deve tenere un riferimento al ConcreteState che rappresenta lo stato corrente.

## 20.4. Struttura del Pattern




## 20.5. Applicazione del Pattern

### Schema del modello



### Partecipanti

- **Context**: classe Clock.
  - Specifica una interfaccia di interesse per i clienti.
  - Mantiene una istanza di **ConcreteState** che rappresenta lo stato corrente
- **State**: classe astratta ClockState.
  - Specifica l'interfaccia delle classi che incapsula il articolare comportamento associato a un particolare stato del **Context**.
- **ConcreteState**: classi NormalDisplayState, UpdatingHrState e UpdatingMinState.
  - Ogni classe implementa il comportamento associato ad uno stato del **Context**.

### Descrizione del codice

La classe astratta `ClockState` (**State**) specifica l'interfaccia che ogni **ConcreteState** deve implementare. Particolarmente questa interfaccia offre due metodi `modeButton` e `changeButton` che sono le operazioni da eseguire se viene premuto il tasto MODE o il tasto CHANGE dell'orologio. Queste operazioni hanno comportamenti diversi secondo lo stato in cui ritrova l'orologio. La classe `ClockState` gestisce anche un riferimento all'oggetto `Clock` a chi appartiene, in modo che i particolari stati possano accedere alle sue proprietà:

```
public abstract class ClockState {

   protected Clock clock ;

   public ClockState(Clock clock) {
      this.clock = clock;
   }

   public abstract void modeButton();
   public abstract void changeButton();

}
```

Il **ConcreteState** `NormalDisplayState` estende `ClockState`. Il suo costruttore richiama il costruttore della superclasse per la gestione del riferimento al rispettivo oggetto `Clock`. Il metodo `modeButton` semplicemente cambia lo stato dell'orologio da "visualizzazione normale" a "aggiornamento delle ore" (creando una istanza di `UpdatingHrState` e associandola allo stato corrente dell'orologio). Il metodo `changeButton` accende la luce del display per visualizzare l'ora corrente (si ipotizzi che la luce si spegne automaticamente):

```
public class NormalDisplayState extends ClockState {

   public NormalDisplayState(Clock clock) {
        super( clock );
        System.out.println( "** Clock is in normal display.");
   }

   public void modeButton() {
      clock.setState( new UpdatingHrState( clock ) );
   }
```

```
   public void changeButton() {
        System.out.print( "LIGHT ON: " );
      clock.showTime();
   }

}
```

La classe `UpdatingHrState` rappresenta lo stato di modifica del numero delle ore dell'orologio. In questo caso, però, il metodo `modeButton` cambia lo stato a "modifica dei minuti" (creando una istanza di `UpdatingMinState` e associandola al Clock). D'altra pare, il metodo `changeButton` incrementa l'ora corrente in una unità.

```
public class UpdatingHrState extends ClockState {

   public UpdatingHrState(Clock clock) {
        super( clock );
        System.out.println(
              "** UPDATING HR: Press CHANGE button to increase hours.");
   }

   public void modeButton() {
      clock.setState( new UpdatingMinState( clock ) );
   }

   public void changeButton() {
      clock.hr++;
      if(clock.hr == 24)
         clock.hr = 0;
        System.out.print( "CHANGE pressed - ");
        clock.showTime();
   }

}
```

La classe `UpdatingMinState` rappresenta lo stato di "modifica dei minuti". In questo stato, il metodo `modeButton` porta l'orologio allo stato di "visualizzazione normale" (tramite la creazione e associazione all'orologio di una istanza di `NormalDisplayState`). Il metodo `changeButton`, invece, incrementa di una unità i minuti dell'orologio.

```
public class UpdatingMinState extends ClockState {

   public UpdatingMinState(Clock clock) {
        super( clock );
        System.out.println(
           "** UPDATING MIN: Press CHANGE button to increase minutes.");
   }

   public void modeButton() {
      clock.setState( new NormalDisplayState( clock ) );
   }

   public void changeButton() {
      clock.min++;
             if(clock.min == 60)
         clock.min = 0;
        System.out.print( "CHANGE pressed - ");
        clock.showTime();
   }

}
```

La classe `Clock` rappresenta il **Context** dello stato. Lo stato corrente di ogni istanza di `Clock` viene gestito con un riferimento verso un oggetto **ConcreteState** (variabile `clockState`), tramite l'interfaccia `ClockState`. Al momento della creazione, ogni clock viene settato alle ore 12:00 e con nello stato di visualizzazione normale.

```
public class Clock {
```

```
   private ClockState clockState;
   public int hr, min;

   public Clock() {
      clockState = new NormalDisplayState( this );
   }

   public void setState( ClockState cs ) {
       clockState = cs;
   }

   public void modeButton() {
      clockState.modeButton();
   }

   public void changeButton() {
      clockState.changeButton();
   }

   public void showTime() {
        System.out.println( "Current time is Hr : " + hr + " Min: "
                            + min );
   }

}
```

Finalmente si presenta il codice dell'applicazione che crea una istanza di Clock ed esegue le seguenti operazioni:

1. Preme per primo il tasto CHANGE: dato che l'orologio è nello stato di visualizzazione normale, dovrebbe mostrare l'ora corrente con la luce del display accesa.

2. Preme il tasto MODE : attiva lo stato di modifica delle ore.

3. Preme due volte il tasto CHANGE: cambia l'ora corrente alle ore 14.

4. Preme il tasto MODE: attiva lo stato di modifica dei minuti

5. Preme quattro volte il tasto CHANGE: cambia il numero dei minuti a 4.

6. Preme il tasto MODE: ritorna allo stato di visualizzazione normale.

```
public class StateExample {

   public static void main ( String arg[] ) {

     Clock theClock = new Clock();
      theClock.changeButton();
      theClock.modeButton();
      theClock.changeButton();
      theClock.changeButton();
      theClock.modeButton();
      theClock.changeButton();
      theClock.changeButton();
      theClock.changeButton();
      theClock.changeButton();
      theClock.modeButton();
   }

}
```

**Osservazioni sull'esempio**

In questo esempio il cambiamento di stato del **Context** è gestito tramite le stesse operazioni degli stati. Vale dire, una particolare operazione eseguita in uno stato, crea un nuovo stato e lo assegna come stato corrente.

Nell'esempio descritto gli stati non vengono riutilizzati, cioè, ogni volta che si cambia di stato, viene creato un nuovo oggetto **ConcreteState**, intanto che il vecchio si perde. Sarebbe più efficiente tenere una singola istanza di ogni **ConcreteState** e assegnare quella corrispondente, tutte le volte che l'orologio cambia stato.

**Esecuzione dell'esempio**

```
C: \Design Patterns\Behavioral\State\Example1>java StateExample

** Clock is in normal display.
LIGHT ON: Current time is Hr : 0 Min: 0

** UPDATING HR: Press CHANGE button to increase hours.
CHANGE pressed - Current time is Hr : 1 Min: 0
CHANGE pressed - Current time is Hr : 2 Min: 0

** UPDATING MIN: Press CHANGE button to increase minutes.
CHANGE pressed - Current time is Hr : 2 Min: 1
CHANGE pressed - Current time is Hr : 2 Min: 2
CHANGE pressed - Current time is Hr : 2 Min: 3
CHANGE pressed - Current time is Hr : 2 Min: 4

** Clock is in normal display.
```

## 20.6. Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in considerazione.