

Design Pattern State

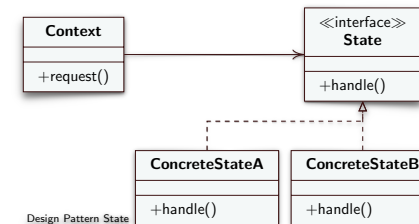
- **Intento:** Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Far sembrare che l'oggetto abbia cambiato la sua classe
- **Problema**
 - Il comportamento di un oggetto dipende dal suo stato e il comportamento deve cambiare a run-time in base al suo stato
 - Le operazioni da svolgere hanno vari grandi rami condizionali che dipendono dallo stato
 - Lo stato è spesso rappresentato dal valore di una o più variabili enumerative costanti
 - Spesso varie operazioni contengono la stessa struttura condizionale

1

Prof. Tramontana - Aprile 2019

Design Pattern State

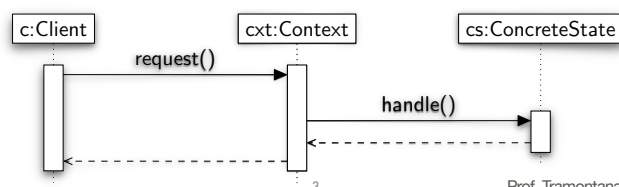
- Soluzione
 - Inserire ogni ramo condizionale in una classe separata
 - **Context** definisce l'interfaccia che interessa ai client, e mantiene un'istanza di una classe ConcreteState che definisce lo stato corrente
 - **State** definisce un'interfaccia che incapsula il comportamento associato ad un particolare stato del Context
 - **ConcreteState** sono le sottoclassi che implementano ciascuna il comportamento associato ad uno stato del Context



Prof. Tramontana - Aprile 2019

Design Pattern State

- Collaborazioni
 - Il **Context** passa le richieste dipendenti da un certo stato all'oggetto **ConcreteState** corrente
 - Un **Context** può passare se stesso come argomento all'oggetto **ConcreteState** per farlo accedere al contesto se necessario
 - Il **Context** è l'interfaccia per le classi client
 - Il **Context** o i **ConcreteState** decidono quale stato è il successivo ed in quali circostanze



Prof. Tramontana - Aprile 2019

Design Pattern State

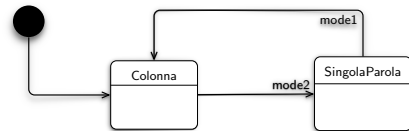
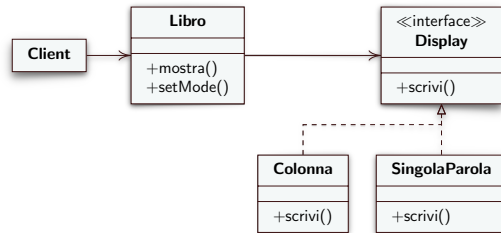
- Conseguenze
 - Il comportamento associato ad uno stato è localizzato in una sola classe (**ConcreteState**) e si partiziona il comportamento di stati differenti. Per tale motivo, si posso aggiungere nuovi stati e transizioni facilmente, creando nuove sottoclassi. Incapsulare le azioni di uno stato in una classe impone una struttura e rende più chiaro lo scopo del codice
 - La logica che gestisce il cambiamento di stato è separata dai vari comportamenti ed è in una sola classe (**Context**), anziché (con istruzioni if o switch) sulla classe che implementa i comportamenti. Tale separazione aiuta ad evitare stati inconsistenti, poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante
 - Il numero di classi totale è maggiore, le classi sono più semplici

4

Prof. Tramontana - Aprile 2019

Esempio

- Si vogliono avere vari modi per scrivere il testo di un libro su un display: in modalità una colonna, due colonne, o una singola parola per volta



5

Prof. Tramontana - Aprile 2019

```

public class Libro { // Context
    private String testo = "Darwin's _Origin of Species_ persuaded the world that the "
        + "difference between different species of animals and plants is not the fixed "
        + "immutable difference that it appears to be.";
    private List<String> lista = Arrays.asList(testo.split("[\\s+]+"));
    private Display mode = new Colonna();
    public void mostra() {
        mode.scrivi(lista);
    }
    public void setMode(int x) {
        switch (x) {
            case 1: mode = new Colonna(); break;
            case 2: mode = new SingolaParola(); break;
        }
    }
}

public interface Display { // State
    public void scrivi(List<String> testo);
}

public class Colonna implements Display { // ConcreteState
    private final int numCar = 38;
    private final int numRighe = 12;
    public void scrivi(List<String> testo) {
        int riga = 0;
        int col = 0;
        for (String p : testo) {
            if (col + p.length() > numCar) {
                System.out.println();
                riga++;
                col = 0;
            }
            if (riga == numRighe) break;
            System.out.print(p + " ");
            col += p.length() + 1;
        }
    }
}

public class Client {
    public static void main(String[] args) {
        Libro l = new Libro();
        l.mostra();
        l.setMode(2);
        l.mostra();
    }
}
    
```

6

Prof. Tramontana - Aprile 2019

```

public class SingolaParola implements Display { // ConcreteState
    private int maxLung;

    public void scrivi(List<String> testo) {
        System.out.println();
        mettiSpazi(30);
        trovaMaxLung(testo);
        for (String p : testo) {
            int numSpazi = (maxLung - p.length()) / 2;
            mettiSpazi(numSpazi);
            System.out.print(p);
            if (p.length() % 2 == 1) numSpazi++;
            mettiSpazi(numSpazi);
            aspetta();
            cancellaRiga();
        }
        System.out.println();
    }
    private void mettiSpazi(int n) {
        for (int i = 0; i < n; i++) System.out.print(" ");
    }
    private void cancellaRiga() {
        for (int i = 0; i < maxLung; i++) System.out.print("\b");
    }
    private void trovaMaxLung(List<String> testo) {
        for (String p : testo) if (maxLung < p.length()) maxLung = p.length();
    }
    private static void aspetta() {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {}
    }
}
    
```

7

Prof. Tramontana - Aprile 2019

```

public class LibroPrimaDiState {
    private String testo = " ... ";
    private List<String> lista = Arrays.asList(testo.split("[\\s+]+"));
    private int mode = 2;

    public void mostra() {
        switch (mode) {
            case 1:
                // vedi metodo scrivi della classe SingolaParola
                break;
            case 2:
                // vedi metodo scrivi della classe Colonna
                break;
        }
    }

    public void setMode(int x) {
        mode = x;
    }
}
    
```

8

Prof. Tramontana - Aprile 2019