Quando ho oggetti che devo modificare il loro comportamento il base al loro stato allora questi algoritmi li possiamo mettere in modo modulare. Il cambiamento di stato sia tale che quando si cambia lo stato si cambia l'intera classe che esegue quel comportamento. Se voglio un oggetto che modifichi il suo compirtamento in base al suo stato sia abbastanza consueto nella OOP?(lo stato in OOP sarebbe il cambiamento dei suoi attributi), NON STO GIà OTTENENDO UN OBBIETTIVO DEL DP? SI. Ma non riesco a fare il secondo obbiettivo del DP, ovvero cambiare lo stato in modo da cambiare classe, inoltre, devo trattare tutto in maniera modulare. Quando siamo davanti a situazione di codici in cui ci sono vari rami condizionali e in ogni ramo condizionale ci sono varie operazioni che facciamo, allora stiamo evidenziando il problema che il desing patern può trattare. (Per ogni operazione che dobbiamo fare lo facciamo dipendere dallo stato).

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

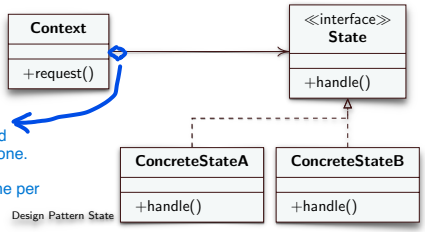
Supponiamo avere la classe Studente
ed abbiamo l'operazione aggiornaStudente.
Allora queste due operazioni mettiamo un ramo condizionale,condizione verificata facciamo delle cose, se no lo facciamo l'altro, in una possiamo mettere la gestione dell'errore nell'altra no.. etc..
Ogni ramo condizionale ha quindi una serie di operazioni che svolge.

Il concetto di stato e transizione lo trattiamo con i rami condizionati. IL DP dice che
se ci troviamo in queste condizioni, anzichè ADOTTARE la soluzione dei rami condizionali UTILIZZIAMO UNA NUOVA SOLUZIONE, separa OGNI RAMO CONDIZIONALE in una classe assesstante , il comportamento di uno stato è svolto da un CONCRETESTATE, ovvero si occupa di eleencare le istruzioni valido per lo stato in cui tu ti trovi

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

Poi ci serve un interfaccia state che definisce le operazioni che il concreteState dovrà utilizzare.
IL context è la classe che conosce quale concreteState deve utilizzare. Grazie a questa soluzione di questo dp lo stato è realizzato dal singolo concreteState, e lo stato lo conosce il Context.



Oltre a definire questi ruoli, il nome del metodo suggerito da Context è request(), mentre i nomi dei metodi suggeriti per i contextState SONO HANDLE(). Quindi il context fonisce un interfaccia con piu metodi che io ho pensato che possono essere utili per le classi CLIENT.

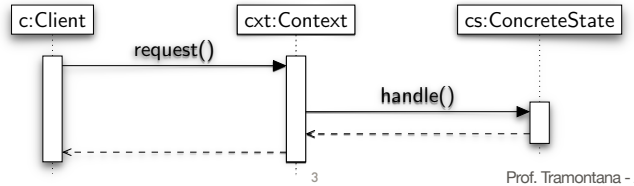
rombo in uml sta ad indicare aggregazione. Aggregazione che sottolinea il fatto che per poter definire bene l'oggetto che sta aggregando, il punto in cui
parte la freccia, abbiamo bisogno dei vari servizi che sono i concreteState

IMPORTANTE :i vari concreteState non devono essere conosciuti dal client, ma solo il CONTEXT LO CONOSCE, anzi è vietato che i client la conoscono

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

handle potrebbe avere parametro attributo che tiene dentro la classe context,oppure tutta la classe Context.
Il concreteState esegue il metodo e quindi quel comportamento e finire la sua esecuzione, lo ritorna al context e di conseguenza poi lo torna al Client



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

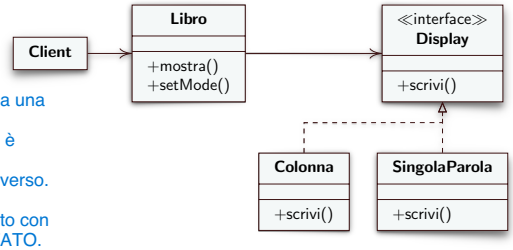
grazie al dp, possiamo aggiungere in un qualunque momento un nuovo
stato, quindi dobbiamo solo implementare una nuova classe, tutto il resto delle classi spesso non dobbiamo toccare. Aggiungere la transizione dovrebbe essere facile-->posso farlo agevolmente nella classe Context

La semplicità che ho raggiunto all'interno dei singoli contreteState e context, se ho piu operazioni da implementare tutte definite all'nterno dell'interfaccia State,ognuna che si comporta
in maniera differente in uno stato. Se avessi metto tutte queste operazioni
in una classe è facile confondere le varie implemtentazioni e nel codice può venire piu complicato perchè è tutto mischiato

State : abbiamo individuato che abbiamo due comportamenti diversi e ci possiamo trovare solo in uno dei due
abbiamo due classi che svolgono il ruolo di concreteState

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

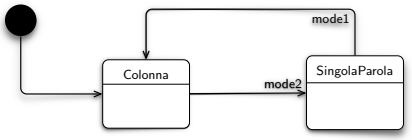


si chiama diagramma UML
degli stati. Ogni diagramma UML ha una
sua
definizione. Quello degli stati infatti è
differente da quello
delle classi, il diagrammi infatti è diverso.

Nel diagramma degli stati il quadrato con
gli angoli arrotondati indica uno STATO.

La freccia che collega gli stati
rappresenta una transizione tra uno stato
ALL'ALTRO.

Lo spazio BIANCO che lascio dentro il
rettangolo : quando entro, poi c'è quando
sono nello stato che è la parola chiave
che significa puoi fare quelle azioni in
quello stato e poi exit che indica l'azione
che devi fare quando esci dallo stato, a
volte non si mette niente
ma si lascia in bianco perchè non sempre
ci sono delle azioni prestabilite




```
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();
   }
}
```



```
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) { }
   }
}
```



```
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;
   }
}
```