State pattern

comportamentale

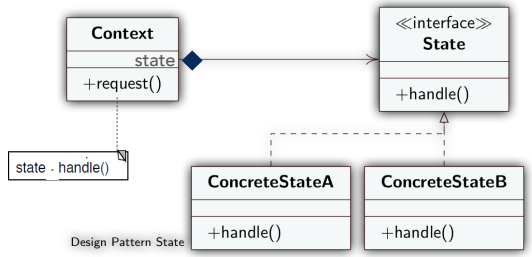
Prof. A. Calvagna

Design pattern State

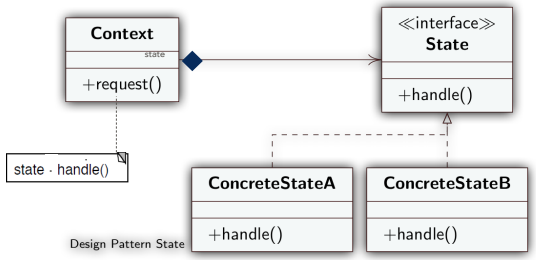
- **Intento**: Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Sembrerà che l'oggetto abbia cambiato classe.

Applicabilità: request() è un algoritmo complesso con molte varianti condizionali, legate allo stato del contesto…

Soluzione: Inserire ogni ramo condizionale in una classe separata



Design pattern State

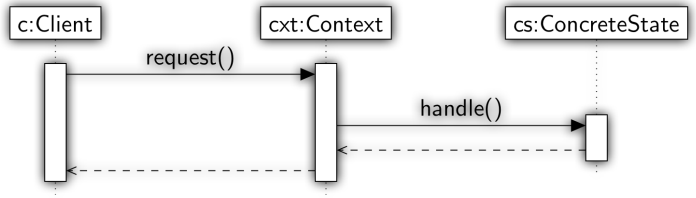


Context definisce l'interfaccia che interessa ai client, e mantiene un'istanza ***state*** di una classe **ConcreteState** che definisce lo stato corrente

L'interfaccia **State** definisce il nome del servizio (o servizi) astratto la cui implementazione deve dipendere dallo stato del Context

I **ConcreteState** sono sottoclassi di State e implementano ciascuna il diverso comportamento da associare ad un diverso stato del Context

Design Pattern State



Collaborazioni:

Il Context è l'interfaccia per le classi client

rimbalza le invocazioni all'oggetto ConcreteState corrente

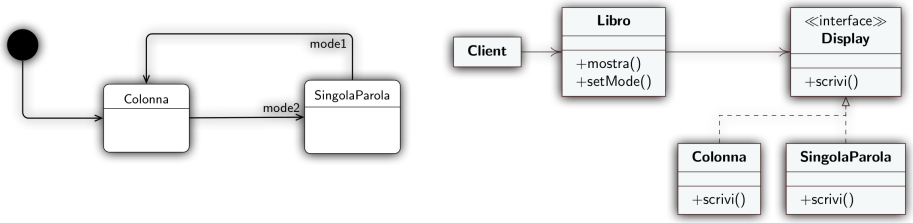
può passare se stesso come argomento

all'oggetto ConcreteState può servire **accedere al contesto…**

Il Context o i ConcreteState decidono **quale stato è il successivo** ed in quali circostanze

Esempio facile

- Si vuole mostrare il testo di un libro su un display, in vari modi strani: in modalità formattata in colonna o una sola parola per volta



Implementazione con il pattern State

```
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 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();
        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 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;
        }
    }
}
```

Senza il pattern state…

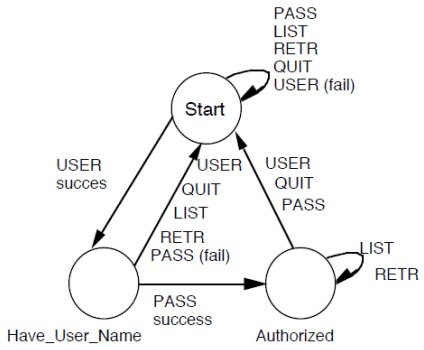
```
public class LibroPrimaDiState {
     private String testo = " ... ";
     private List<String> lista = Arrays.asList(testo.split("[\\s+]+"));
     private int mode = 2;

     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;
     }
}
```

Eseguiamo appLibro (con State) su VSCODE

Esempio più complesso

- Applicazione "Simple Post Office Protocol"
- Un semplice mailbox (solo ricezione) che supporta i seguenti comandi:
  - USER <*username*>
  - PASS <*password*>
  - LIST
  - RETR <*message number*>
  - QUIT
- Comandi USER e PASS :
  - Se l'*username* e la *password* sono validi l'utente può accedere agli altri comandi
- Comando LIST :
  - Parametro (opzionale) *message-number*
  - Se presente, ritorna la dimensione del messaggio indicato
  - Altrimenti, ritorna la dimensione di tutti i messaggi nel mailbox

Logica di funzionamento



L'applicazione ha una serie di funzionalità non indipendenti: legate dallo stato

Esistono delle regole precise che stabiliscono le transizioni (esecuzioni di comandi) che sono ammesse

Posso formalizzarle con una macchina a stati finiti (automa)

Codifico un protocollo d'uso corretto per l'oggetto

Implementazione con switch

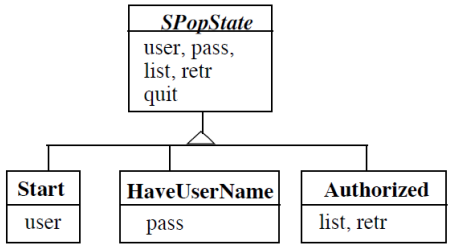
su vscode….SPOPapp

- Codice confuso e poco chiara la logica
- Ripetizioni e duplicazioni
- Difficile da modificare in seguito

- La presenza di switch è un sintomo che suggerisce di usare il pattern State (quantomeno, il polimorfismo)

SPop con Pattern

- Sfrutto il polimorfismo
- No duplicazione
- Comportamenti default



su vscode….SPOPappState

Chi definisce le transizioni? Lo stato

```
class Spop {
   private SPopState state = new Start();

   public void user( String userName ){
      state = state.user( userName );
   }
   public void pass( String password ){
      state = state.pass( password );
   }
   public void list( int messageNumber ){
      state = state.list( messageNumber );
   }
```

È la scelta che dà più flessibilità a run-time

Nuovi stati (e nuove logiche) possono essere incluse a run-time

Il contesto (Spop) resta minimale e riusabile

Chi definisce le transizioni? Il contesto

```
class SPop{
   private SPopState state = new Start();

   public void user( String userName ){
      state.user( userName );
      state = new HaveUserName( userName );
   }
   public void pass( String password ){
      if ( state.pass( password ) )
         state = new Authorized( );
      else
         state = new Start();
}
```

- La **logica** che gestisce i cambiamenti di stato è **separata dalle azioni** corrispondenti e **raccolta** in una sola classe (Context)
  - Più facile comprenderla o modificarla: anche se sparpagliata nei metodi del context, non è distribuita su più oggetti
  - permette di **riusare gli stati** (ora tra loro indipendenti) in contesti (automi) diversi
  - Va bene se le transizioni sono già note a compile-time
  - Gli stati si semplificano ulteriormente

- Altra versione di SPOP con logica cablata nel contesto:

SPOPAppStateBis su VSCODE

Condivisione degli stati

- Nell'esempio visto, tutto il comportamento e di dati del contesto sono in SpopState: **è una situazione estrema**
- In generale, il contesto può avere anche altri dati o metodi che non influenzano il resto del comportamento
- L'altro estremo solo gli **stati senza attributi di istanza**:
- sono stateless o lavorano su dati conservati nel contesto: li posso condividere/riusare in molteplici contesti
  - Posso usare il pattern **singleton** per crearli una volta sola e non distruggerli mai
  - Oppure posso crearli quando servono ed eliminarli quando inutilizzati
  - **Refactoring** di uno stato affinché non abbia più variabili di istanza
    - le sposto nel contesto. Poi…
      - le passo allo stato come parametri (vedi prec. slide), oppure…
      - Passo il contesto e gli stati vi accedono direttamente…

Condivisione degli stati

```
class SPop {
    private SPopState state = new Start();
    String userName;
    String password;
    public String userName() { return userName; }
    public String password() { return password; }
    public void user( String newName ) {
            this.userName = newName ;
            state.user( this );
    }
….etc.
```

```
class HaveUserName implements SPopState {
    public SPopState pass( SPop mailServer ) {
        validate(mailServer.password());
        …etc.
    }
}
```

Passo il contesto e gli stati vi accedono direttamente…

Devo prevedere metodi getter-setter per consentire e disciplinare l'accesso

State vs Strategy

•Anche state ha **object scope**, ma cambiano **frequenza** e **visibilità** dei cambi

Come distinguere strategy da state?

Dalla **frequenza** dei cambi:

Nello strategy, applico al contesto una sola strategia (tra varie possibili) per ogni suo ciclo di vita.

Nello state, il contesto cambia più volte lo stato concreto nell'arco dello stesso ciclo di vita.

Dalla **visibilità** dei cambi:

Nello strategy, tutte le strategie fanno la stessa cosa, ma in modo diverso: i client non vedono differenza nel comportamento invocato sul contesto

Nello state, gli stati concreti producono azioni differenti, per cui i client vedono il contesto reagire in modo cangiante

Altro esempio: Refactoring per applicare lo State pattern

- Progetto e implemento un sistema senza conoscere il pattern, poi lo riprogetto per avvalermene
- parto dai seguenti requisiti
  - un sistema software dovrà fornire la possibilità di prenotare e acquistare un biglietto (per un viaggio).
  - Si potrà annullare la prenotazione, ma non l'acquisto.
  - Ogni biglietto ha un codice, un prezzo, una data di acquisto, il nome dell'intestatario (e i dettagli del viaggio).
  - Per la prenotazione si dovrà dare il nome dell'intestatario.

Progettazione

- **Analisi lessicale (Abbott):** Il sistema software dovrà fornire la possibilità di prenotare e acquistare un biglietto (per un viaggio). Si potrà annullare la prenotazione, ma non l'acquisto. Ogni biglietto ha un codice, un prezzo, una data di acquisto, il nome dell'intestatario (e i dettagli del viaggio). Per la prenotazione si dovrà dare il nome dell'intestatario.
- Identifico nomi (entità/attributi), verbi (operazioni), eventuali aggettivi (attributi)
- Individuo e raggruppo attributi/operazioni nelle entità individuate
- Individuo eventuali stati diversi delle entità (le dinamiche)

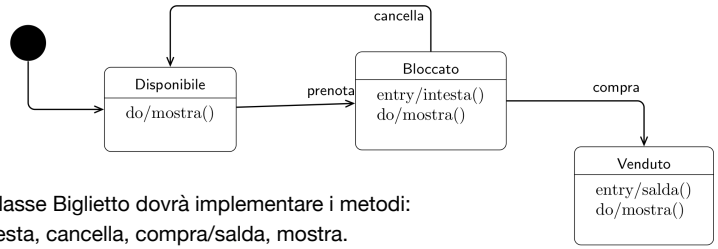
  Classi: Biglietto, Viaggio, Intestatario (le ultime due sono record: solo dati)

  Attributi: codice, prezzo, data, nome

  Operazioni: prenota, acquista, annulla

Diagramma degli stati

Stati di Biglietto: disponibile, bloccato (ovvero prenotato), venduto



Quindi, la classe Biglietto dovrà implementare i metodi: prenota/intesta, cancella, compra/salda, mostra.

Ciascuna operazione controllerà lo stato in cui si trova il biglietto prima di eseguire le azioni necessarie


```
// Codice che implementa i suddetti requisiti (prima versione)
public class Biglietto {
     private String codice = “XYZ”, nome;
     private int prezzo = 100;
     private enum StatoBiglietto { DISP, BLOC, VEND }
     private StatoBiglietto stato = StatoBiglietto.DISP;

   // ogni operazione deve controllare in che stato si trova il biglietto
     public void prenota(String s) {
          switch (stato) {
          case DISP:
               System.out.println("Cambia stato da Disponibile a Bloccato");
               nome = s;
               System.out.println("Inserito nuovo intestatario");
               stato = StatoBiglietto.BLOC;
               break;
          case BLOC:
               nome = s;
               System.out.println("Inserito nuovo intestatario");
               break;
          case VEND:
               System.out.println("Non puo' cambiare il nome nello stato Venduto");
               break;
          }
     }
```



```
public void cancella() {
     switch (stato) {
     case DISP:
          System.out.println("Lo stato era gia' Disponibile");
          break;
     case BLOC:
          System.out.println("Cambia stato da Bloccato a Disponibile");
          nome = "";
          stato = StatoBiglietto.DISP;
          break;
     case VEND:
          System.out.println("Non puo' cambiare stato da Venduto a Disponibile");
          break;
     }
}

public void mostra() {
     System.out.println("Prezzo: " + prezzo + "  codice: " + codice);
     if (stato == StatoBiglietto.BLOC || stato == StatoBiglietto.VEND)
          System.out.println("Nome: " + nome);
}
```



```
     public void compra() {
          switch (stato) {
          case DISP:
               System.out.println("Non si puo’ pagare, bisogna prima intestarlo");
               break;
          case BLOC:
               System.out.println("Cambia stato da Bloccato a Venduto");
               stato = StatoBiglietto.VEND;
               System.out.println("Pagamento effettuato");
               break;
          case VEND:
               System.out.println("Il biglietto era gia' stato venduto");
               break;
          }
     }
}
```

```java
public class Client {
    private Biglietto b = new Biglietto();
    public static void main(String[] args) {
      usaBiglietto();
      nonUsaOk();
   }

    private static void usaBiglietto() {
        b.prenota("Ciccio Falsaperla");
        b.mostra();
        b.compra()
        b.mostra();
   }

    private static void nonUsaOk() {
        b.compra();
        b.cancella();
        b.prenota("Mario Biondi");
    }
}
```

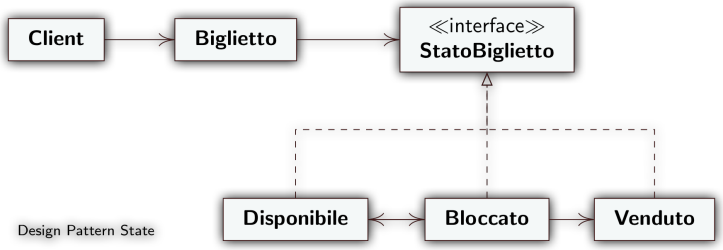
Output dell'esecuzione di MainBiglietto

```
Prezzo: 100  codice: XYZ
Cambia stato da Disponibile a Bloccato
Inserito nuovo intestatario
Prezzo: 100  codice: XYZ
Nome: Mario Tokoro
Cambia stato da Bloccato a Venduto
Pagamento effettuato
Prezzo: 100  codice: XYZ
Nome: Mario Tokoro
Il biglietto era gia' stato venduto
Non puo' cambiare stato da Venduto a Disponibile
Non puo' cambiare il nome nello stato Venduto
```


Analisi del codice

- La classe ha circa 70LOC, metodo più lungo 15LOC, solo 32 linee con ";"
- Ogni metodo ha vari rami condizionali, uno per ogni stato. La logica condizionale rende il codice difficile da modificare
- Il comportamento in ciascuno stato non è ben separato, poiché lo stesso metodo implementa più comportamenti
- Si può arrivare a un design e un codice più semplice, e che separa i comportamenti? Sì, tramite indirettezze
- Una tecnica di refactoring in genere applicabile è **Replace Conditional with Polymorphism**,
  - Le condizioni possono essere trasformate in messaggi, questo riduce i duplicati, aggiunge chiarezza e aumenta la flessibilità del codice
- Ma qui I cambiamenti sono legati allo stato: uso il pattern State, ovvero la tecnica di refactoring **Replace Type Code with State**

Nuovo design, con il pattern State

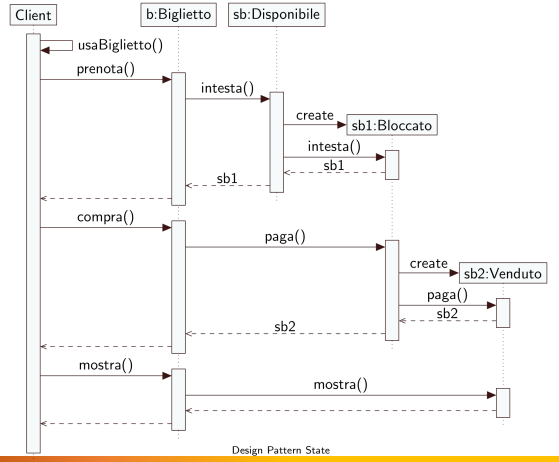


Ci sono **dipendenze** tra gli stati perché le **transizioni** di stato sono state cablate al loro interno, non nel contesto (biglietto)

Ciascuno stato avvia, quando occorre, una transizione, questo ha permesso di **eliminare** l'avvio delle transizioni **da Context**, e quindi **gli switch** su esso

L'interfaccia usata dai ConcreteState **ritorna il riferimento** a un nuovo state (**è detta fluent)**

Vista in fase di esecuzione



L'interfaccia del context può non coincidere con l'interfaccia State: questa può avere segnature diverse, su cui però si devono poter mappare correttamente le funzionalità

Se le transazioni sono cablate nel context, posso riusare i concrete states per creare una diversa applicazione o algoritmo

in questo caso il context è *affine* ad un adapter *multiplo* adatta il client al riuso di multiple funzionalità

Su VSCODE: APPBiglietti

Analisi del codice

- Le LOC sono 2 o 3 per metodo, ci sono 4 classi, e 1 interfaccia
- Totale LOC 140 circa (compresi commenti e linee vuote), solo 53 linee con ";"
- Sono state **eliminate le istruzioni condizionali**: i metodi non devono controllare in quale stato si trovano, poiché la classe si riferisce ad un singolo stato. **Non si ha codice duplicato** per i test condizionali su ciascun metodo
- **I metodi sono più semplici** da comprendere e modificare
- Il codice per ciascuno stato può implementare altre attività senza complicarsi

- Sono **conseguenze** dell'applicazione del design pattern State