# DESIGN PATTERNS – Parte 6

## State
## Proxy

## STATE

Il Design Pattern State nasce dall'esigenza di poter gestire gli stati di una classe senza dover usare dei costrutti come gli Enum e quindi delle switch. Prendiamo il caso di una classica macchinetta delle bibite... avremmo un diagramma delle classi tipo questo:

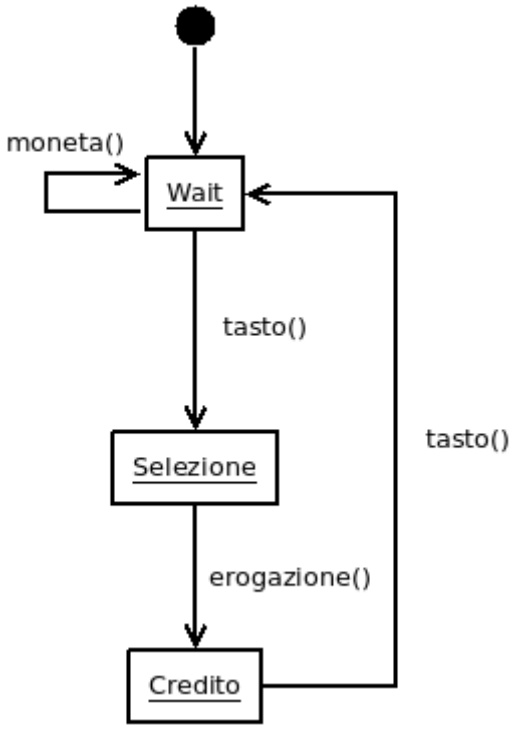

Vediamo in cosa consiste la classe astratta Stato. Questa classe implementa tutti i metodi che la macchinetta potrebbe eseguire indipendentemente dallo stato in cui si trova. Fatto questo, ogni classe che è sotto-classe di Stato non sono altro che degli stati specifici i quali implementano dei metodi che sono degli ovverriding dei metodi di Stato. I metodi su cui non avviene l'overriding rimangono vuoti. Per comprendere al meglio questo funzionamento vediamo prima il codice e poi il diagramma degli stati.
Gli stati da parte loro, vengono implementati come singleton, così da essere “veramente” unici.

```
public abstract class Stato {

	public void tasto(Macchinetta m) { }

	public void moneta(Macchinetta m) { }

	public void erogazione(Macchinetta m) { }

}
```

```
public class Macchinetta {

     private Stato stato;

     private double prezzo, credito = 0;

     public Macchinetta() {
          stato = Wait.self;
     }

     public void tasto() { stato.tasto(this); }

     public void moneta() { stato.moneta(this); }

     public void erogazione() { stato.erogazione(this); }

     public void setStato(Stato s) { stato = s; }

     public void addCredito(double d) { credito = credito + d; }

     public void resto() { credito = credito - prezzo; }

}

public class Wait extends Stato {

     public final Stato self = new Wait();

     private Wait() { }

     public void moneta(Macchinetta m){
          m.addCredito( leggiMoneta() );
     }

     public void tasto(Macchinetta m){ m.setStato(Selezione.self);}

     private double leggiMoneta(){ /* … */ }

}

public class Selezione extends Stato {

     public final Stato self = new Selezione();

     private Selezione() { }

     public void erogazione(Macchinetta m) {
          m.setStato( Credito.self; }
     }

}
```

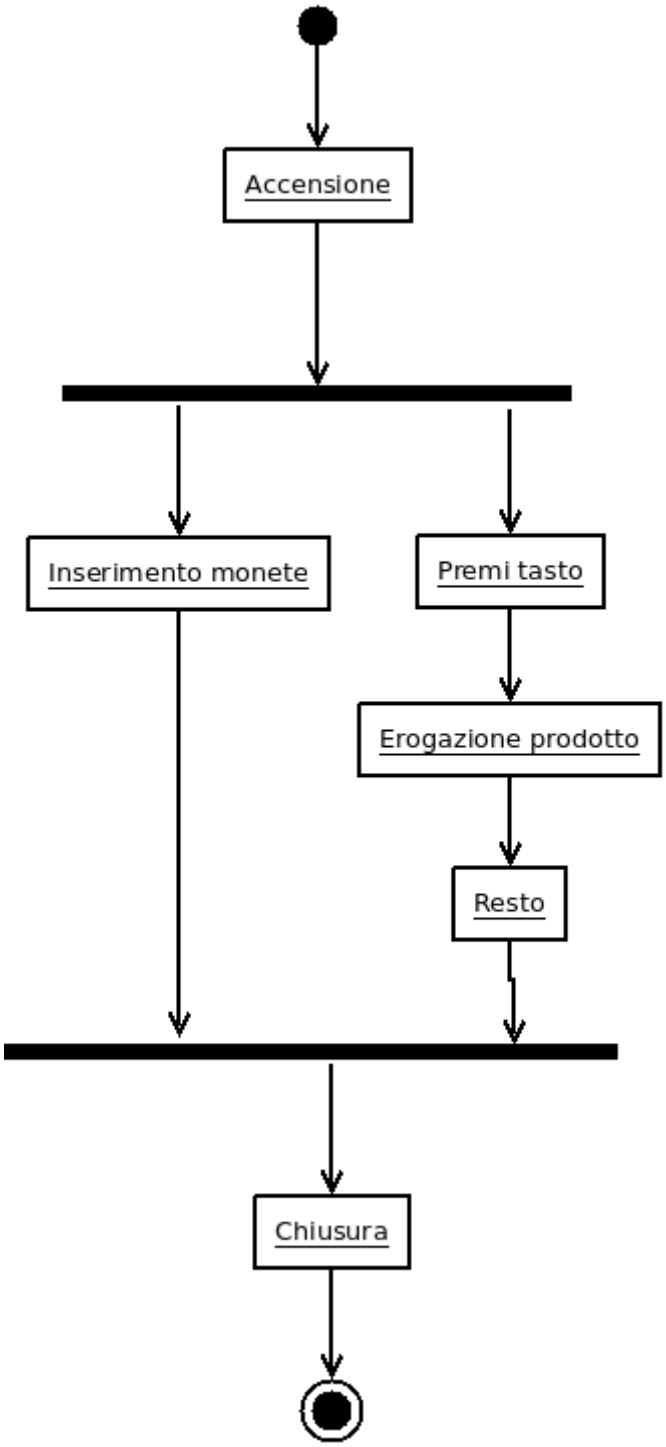
```
public class Credito extends Stato {

     public final Stato self = new Credito();

     private Credito() { }

     private void tasto(Macchinetta m){
          m.resto();
          m.setStato( Wait.self; }
     }
}
```

In pratica abbiamo visto che la macchinetta passa da uno stato all'altro attraverso una serie di azioni. Vediamo dunque lo schema degli stati:




Possiamo notare che questo diagramma non ha fine.. ma soprattutto non tiene conto di quello che l'utente potrebbe fare da un momento all'altro. Per esempio, mettiamo che l'utente dopo l'evento erogazione inserisca una moneta, che succede? Questo caso non ne abbiamo tenuto conto.
Un altro caso è se la macchinetta si chiude di colpo, che succede? Per questi eventi è sempre buona norma includerli utti dentro quello che si chima Super-stato.

Simile al diagramma degli stati è il diagramma degli eventi, che è utile ugualmente per comprendere per quali stati/azioni una classe deve passare.




Il diagramma mostra dettagliatamente come la macchinetta abbia un più delineato passaggio di stati.. in questo modo potremmo riscrivere un diagramma degli stati più dettagliato.

Un modo utile per distinguere i vari stati con l'utilizzo del diagramma delle azioni è quello di compilare una cosidetta tabella degli stati.
Le colonne della tabella saranno:
sorgente, destinazione, evento, guardia e attività.

Sorgente è lo stato in cui si trova attualmente la macchinetta.
Destinazione è lo stato in cui la macchinetta transiterà se accade un evento.
Evento è ciò che fa avvenire la transizione
Guardia è una o più condizioni per cui l'evento può accadere.
Attività è ciò che lo stato farà dopo la transizione.

Dal diagramma
Vediamo dunque sta tabella degli stati:

| **SORGENTE** | **DESTINAZIONE** | **EVENTO** | **GUARDIA** | **ATTIVITÀ** |
|---|---|---|---|---|
| *Accensione* | *Wait* | on() | | Accensione |
| *Wait* | *Wait* | moneta() | | Inserimento di una moneta |
| *Wait* | *Selezione* | eroga() | credito>=prezzo | Eroga il prodotto |
| *Selezione* | *Credito* | credito() | credito>0 | Rilascia il resto |
| *Credito* | *Wait* | resto() | | Eggetta le monete |
| * | *Chiusura* | off() | | Chiusura |

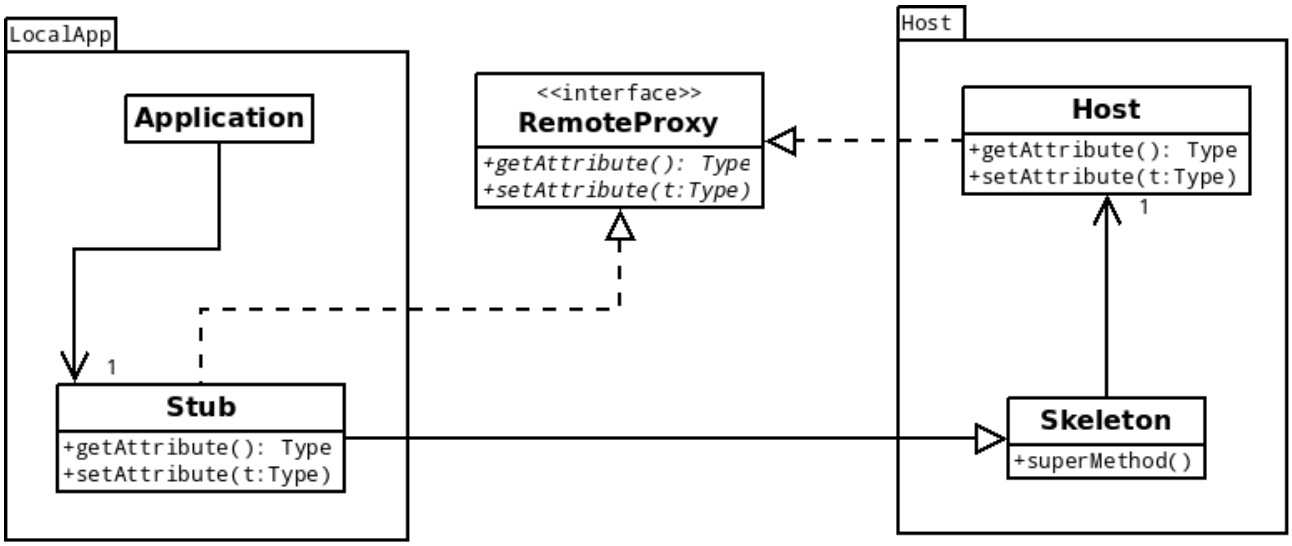
Parlando prima si superStato vediamo che Chiusura è proprio uno di essi.

## PROXY

Quando c'è il problema che non è possibile accedere sempre direttamente ad un certo oggetto, si usa il Pattern Proxy.
Quando avviene questo problema, questo pattern va a sostituire l'oggetto originale facendo credere all'applicazione di accedere all'oggetto voluto, quando invece non lo sta facendo.

## REMOTE PROXY

Remote Proxy è un caso speciale di Proxy, nel quale l'oggetto si trova in un'altra rete. Per rendere possibile l'accesso a questo oggetto si crea un livello intermedio che fa da ponte tra i due mondi.
Esempio:




Un esempio di RemoteProxy sono gli RMI in Java.
Il nostro caso è un'applicazione Application che si trova in un ambiente locale (LocalApp) che vuole accedere all'oggetto Host di un ambiente esterno (Host). Purtroppo non può accedervi direttamente.. un modo è quello di usare un RemoteProxy che sarà un'interfaccia comune ai due ambienti.
L'interfaccia verrà implementata da un oggetto locale (Stub) e da Host, così che per Application sembri che siano la stessa cosa. Fatto questo si fa estendere a Stub Skeleton, una classe di Host che avrà un certo metodo superMethod che quando verrà richiamato chiamerà lo stesso equivalente metodo dall'oggetto Host e il risultato restituito sarà il risultato del metodo Stub, in questo modo Application crederà di usare Host quando in realtà sta usando Stub.

## FAILOVER PROXY

Failover Proxy è usato quando l'accessibilità al dato non è sempre garantito, si usa quindi un oggetto fittizio per sostituirlo momentaneamente finché non è

possibile di nuovo accedere al servizio.
Per esempio avendo uno schema del genere vediamo che se Client non può accedere ad Implementation accederà a FailoverProxy che comunque implementa un'interfaccia comune rendendo interscambiabili le due classi.