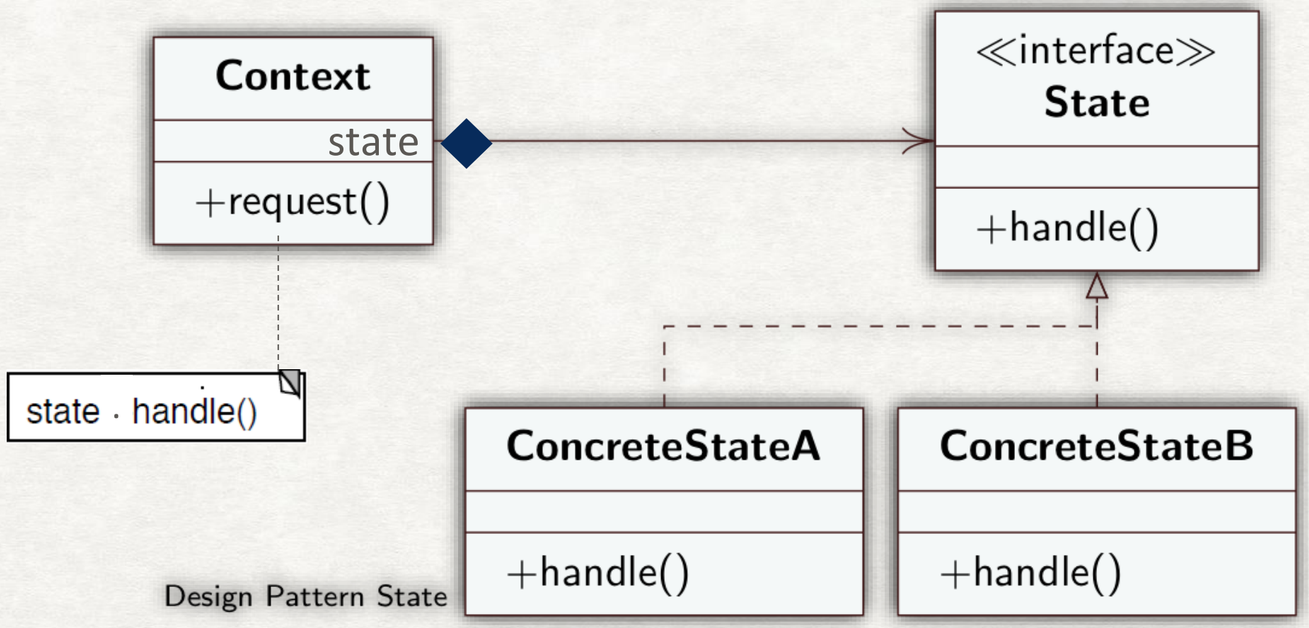
Design pattern State

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

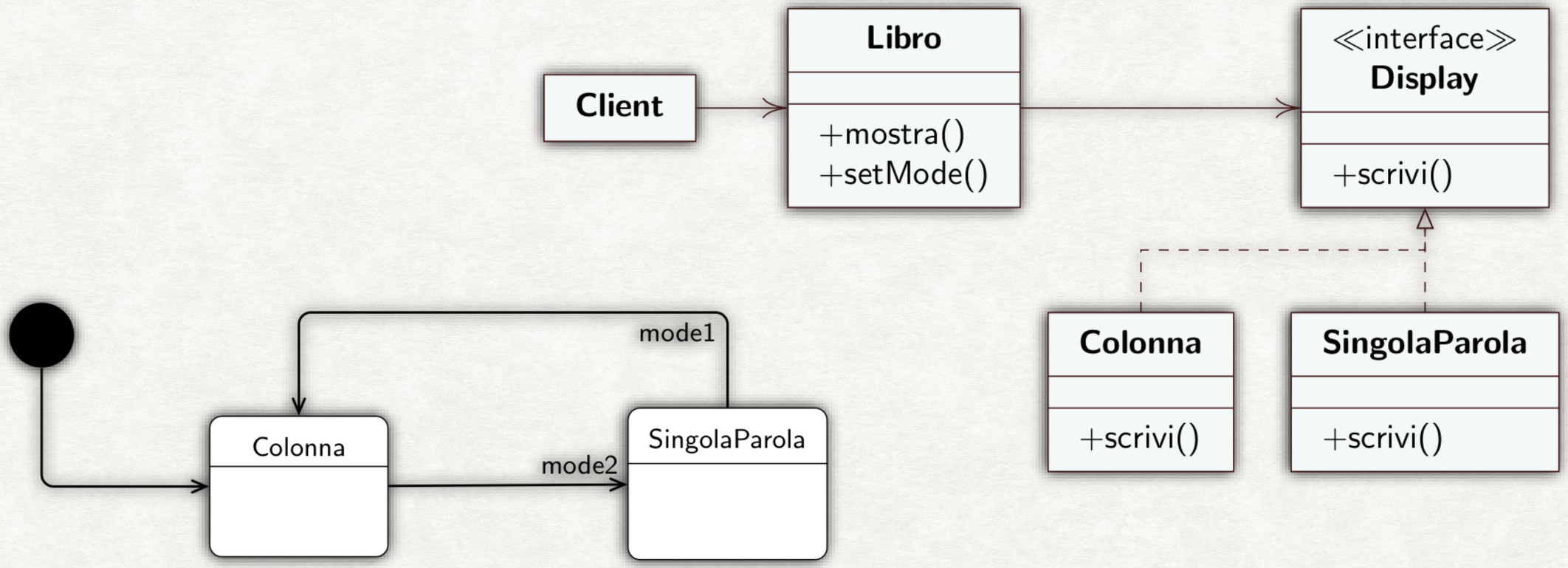


Design pattern State

- 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

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



```
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 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 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();
	}
}
```

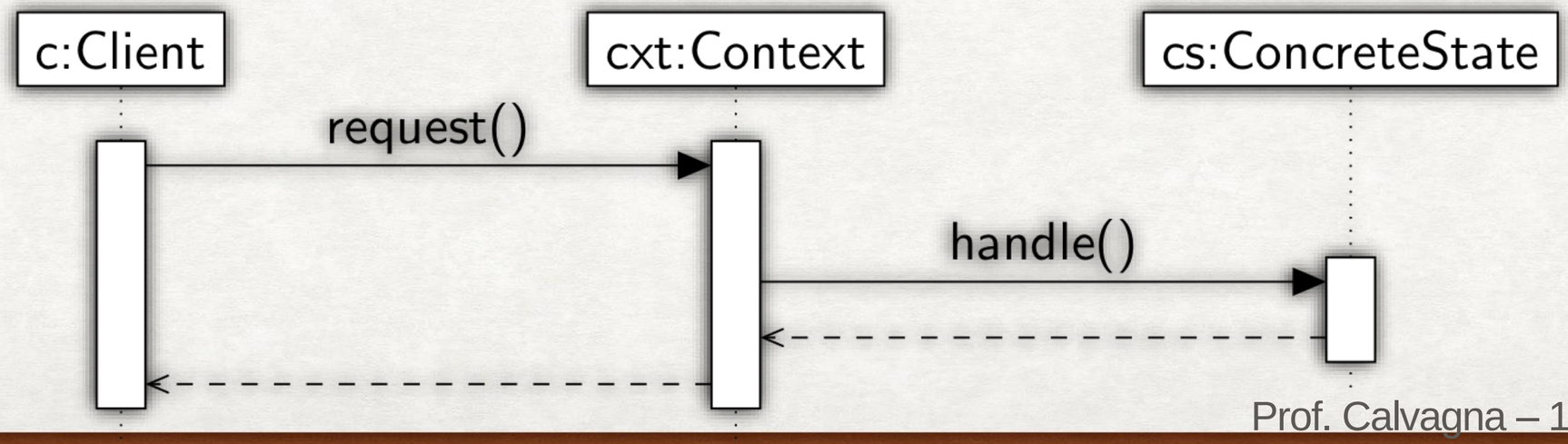
```java
public class SingolaParola implements Display { // ConcreteState
	private int maxLung;
	public void scrivi(List<String> testo) {
		System.out.println();
		trovaMaxLung(testo);
		for (String p : testo) {
			int numSpazi = (maxLung - p.length()) / 2;
			mettiSpazi(numSpazi);
			System.out.print(p);
			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) { }
	}
}
```

- Vediamo il codice in esecuzione su vscode…

```
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;
	}
}
```

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



Pattern STATE

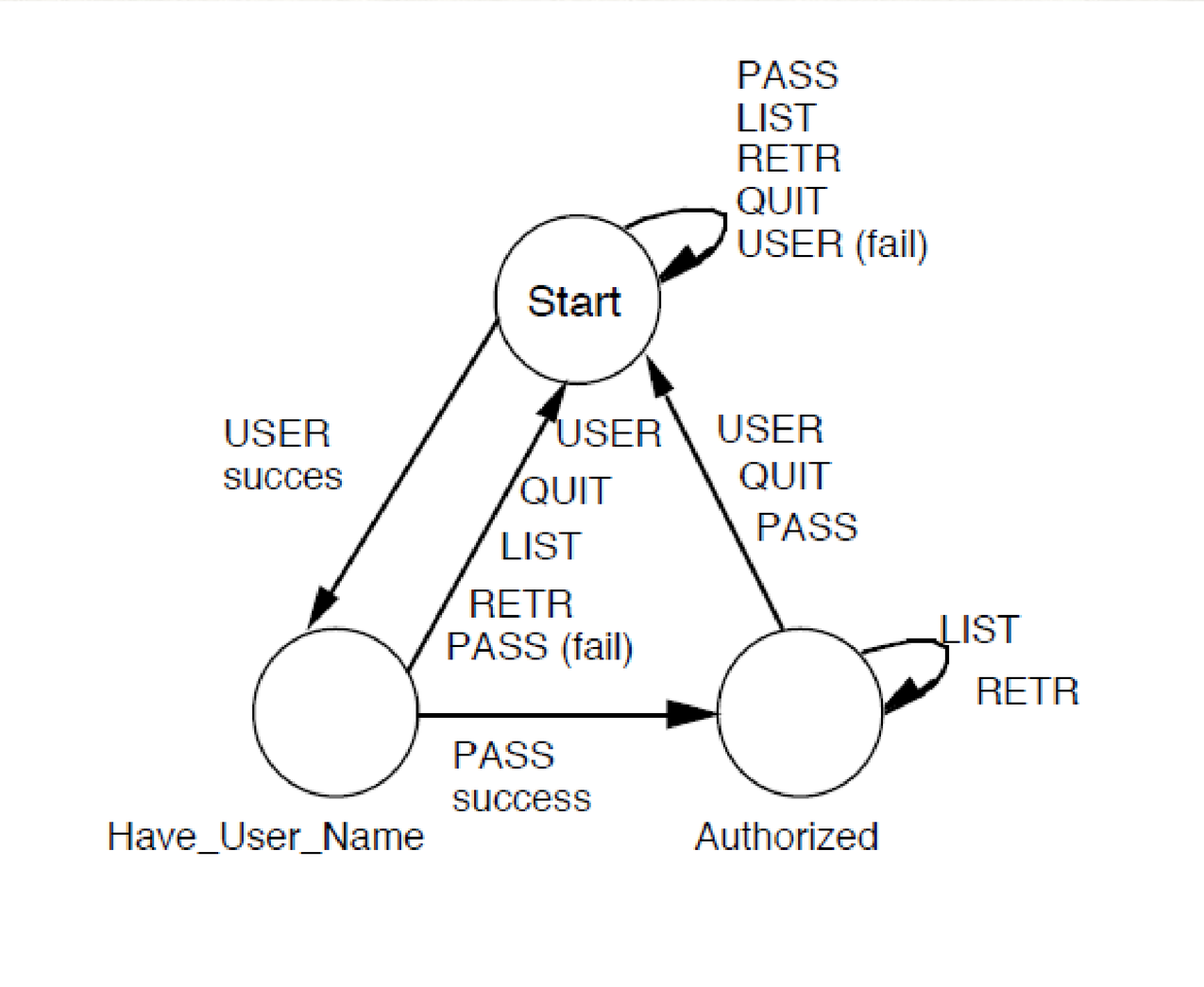
- analizziamolo a fondo con un esempio più complesso:
  - **Applicazione SPOP (Simple Post Office Protocol)**
  - Un semplice mailbox (solo ricezione) che supporti i seguenti comandi:
    - USER <*username*>
    - PASS <*password*>
    - LIST
    - RETR <*message number*>
    - QUIT

- **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

- **Comando RETR**
  - Parametri: *nessuno*
  - Ritorna il testo del messaggio in cima al mailbox e lo rimuove dal mailbox
  - Legale solo per utenti autorizzati
- **Comando QUIT**
  - Nessun parametro
  - termina la sessione correttamente, tornando allo stato iniziale

Logica di funzionamento

- L'applicazione ha una serie di funzionalità non indipendenti
- Esistono delle regole precise che legano tra loro i comandi
- Le posso definire formalmente con una macchina a stati finiti (automa)
- Impongo un protocollo d'uso corretto per un oggetto
- Non tutte le sequenze di interazione (esecuzione) devono essere possibili



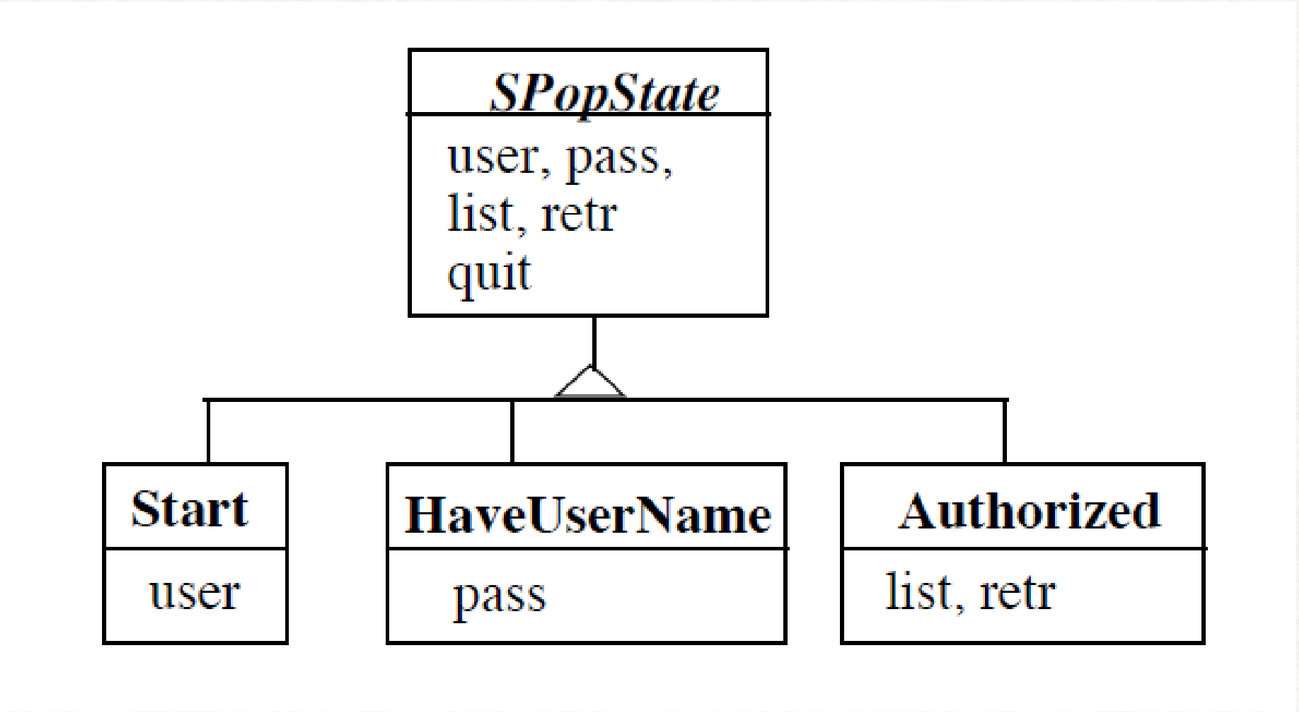
Implementazione con SWITCH

- Vedi codice app su vscode….SPOPapp

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

Implementazione con Pattern

- Sfrutto il polimorfismo
- No duplicazione
- Comportamenti default



- Divido i comportamenti diversi i classi corrispondenti ai diversi stati
- Logica di funzionamento chiara, in questo caso, cablata negli stati

Implementazione con State

- Vedi codice su vscode… SPOPAppState

- Altra versione con logica cablata nel contesto: SPOPAppStateBis

Pausa 15 min

Varianti

- Quanto stato nella classe stato?
  - Nell’esempio:
    - Tutto lo stato ed il comportamento sono in SpopState e nelle sue sottoclassi concrete: è una situazione estrema
    - In generale il contesto può avere dati e metodi oltre quelli dello stato astratto
      - Sono quelli che non cambiano lo stato
      - solo alcuni aspetti del contesto alterano il suo comportamento

Chi definisce le transazioni?

- Le ho definite nelle sottoclassi concrete di State (SpopState)
- È la scelta che dà più flessibilità
  - Nuovi stati (e nuove logiche) possono essere incluse a run-time
  - Il contesto (Spop) resta minimale e riusabile

```
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 );
    }
```

Dove definire le transazioni?

- Le posso cablare nel contesto
  - permette di riusare gli stati in più automi con transizioni diverse (contesti diversi)
  - Va bene se le transizioni sono fissate a compile-time, per ogni contesto
  - Gli stati si semplificano
  - Leggo tutta la logica in una classe
  - Analogie con AOP: metodo/aspetto

```
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();
}
```

Transizioni nel contesto

- Vediamo implementazione su vscode: applicazione SpopStateBis…

Condivisione degli stati

- Stati senza variabili di istanza possono essere condivisi 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 conservo da un’altra parte: nel contesto. Poi…
      - le passo allo stato come parametri (vedi scorso es.)
      - Passo il contesto e gli stati vi accedono direttamente…

Condivisione degli stati

- Passo il contesto e gli stati vi accedono direttamente…
- Devo prevedere metodi getter-setter per consentire e disciplinare l'accesso

```
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.
    }
}
```

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 esplicitata 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, ma le classi sono più semplici

State vs Strategy

- Come distinguerli?
  - **Frequenza delle varianti di comportamento**
    - Nello strategy il contesto seleziona tipicamente una tra varie possibili strategie una volta per tutte
    - Nello state il contesto cambia più volte il suo stato concreto nell'arco del suo ciclo vitale
  - **Visibilità delle varianti di comportamento**
    - Nello strategy tutte le strategie fanno la stessa cosa, ma in modo diverso: i client non vedono differenza nel comportamento del contesto
    - Nello state gli stati concreti producono azioni differenti, per cui i client vedono il contesto reagire in modo cangiante

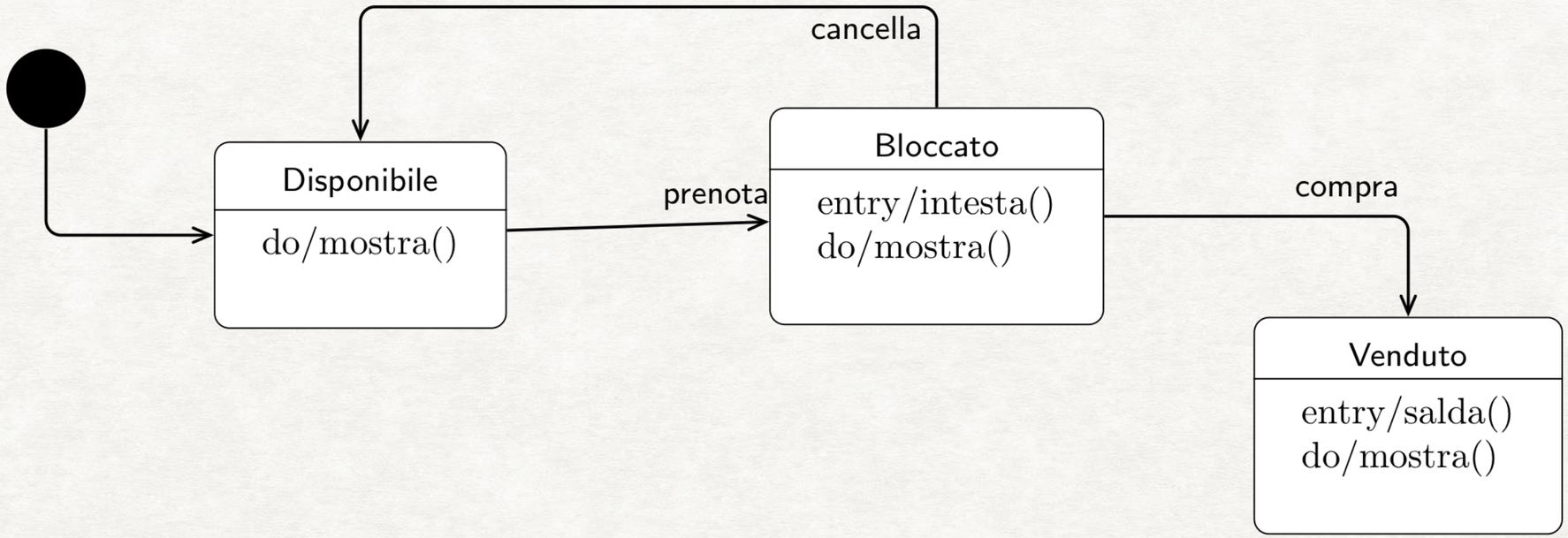
Esercizio guidato

- **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

- 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.
- Classi: Biglietto
- Attributi: codice, prezzo, data, nome
- Operazioni: prenota, acquista, annulla
- La classe Biglietto si può trovare in uno degli stati: disponibile, bloccato (ovvero prenotato), venduto

Diagramma degli stati



- Quindi, la classe Biglietto dovrà implementare i metodi: prenota, cancella, compra, 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;
        }
    }
}
```

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

      private static void usaBiglietto() {
            b.prenota("Mario Tokoro");
            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
- Le condizioni possono essere trasformate in messaggi, questo riduce i duplicati, aggiunge chiarezza e aumenta la flessibilità del codice
- La tecnica di refactoring Replace Conditional with Polymorphism (ovvero Sostituisci i rami condizionali con il polimorfismo), indica come fare
- Ovvero, si tratta del design pattern … State, ovvero Replace Type Code with State

```java
// StatoBiglietto e' uno State
public interface StatoBiglietto {
    public void mostra();
    public StatoBiglietto intesta(String s);
    public StatoBiglietto paga();
    public StatoBiglietto cancella();
}
```

```java
// Disponibile e' un ConcreteState
public class Disponibile implements StatoBiglietto {
    @Override public void mostra() {  }

    @Override public StatoBiglietto intesta(String s) {
        System.out.println("DISP Cambia stato da Disponibile a Bloccato”);
        StatoBiglietto sb = new Bloccato();
        return sb.intesta(s);
//      return new Bloccato().intesta(s);
    }

    @Override public StatoBiglietto paga() {
        System.out.println("DISP Non si puo’ pagare, bisogna prima intestarlo");
        return this;
    }

    @Override public StatoBiglietto cancella() {
        System.out.println("DISP Lo stato era gia' Disponibile");
        return this;
    }
}
```

```java
// Bloccato e' un ConcreteState
public class Bloccato implements StatoBiglietto {
	private String nome;

	@Override public void mostra() {
		System.out.println("BLOC Nome: "+nome);
	}

	@Override public StatoBiglietto intesta(String s) {
		System.out.println("BLOC Inserito nuovo intestatario");
		nome = s;
		return this;
	}

	@Override public StatoBiglietto paga() {
		System.out.println("BLOC Cambia stato da Bloccato a Venduto");
		return new Venduto(nome).paga();
	}

	@Override public StatoBiglietto cancella() {
		System.out.println("BLOC Cambia stato da Bloccato a Disponibile");
		return new Disponibile();
	}
}
```

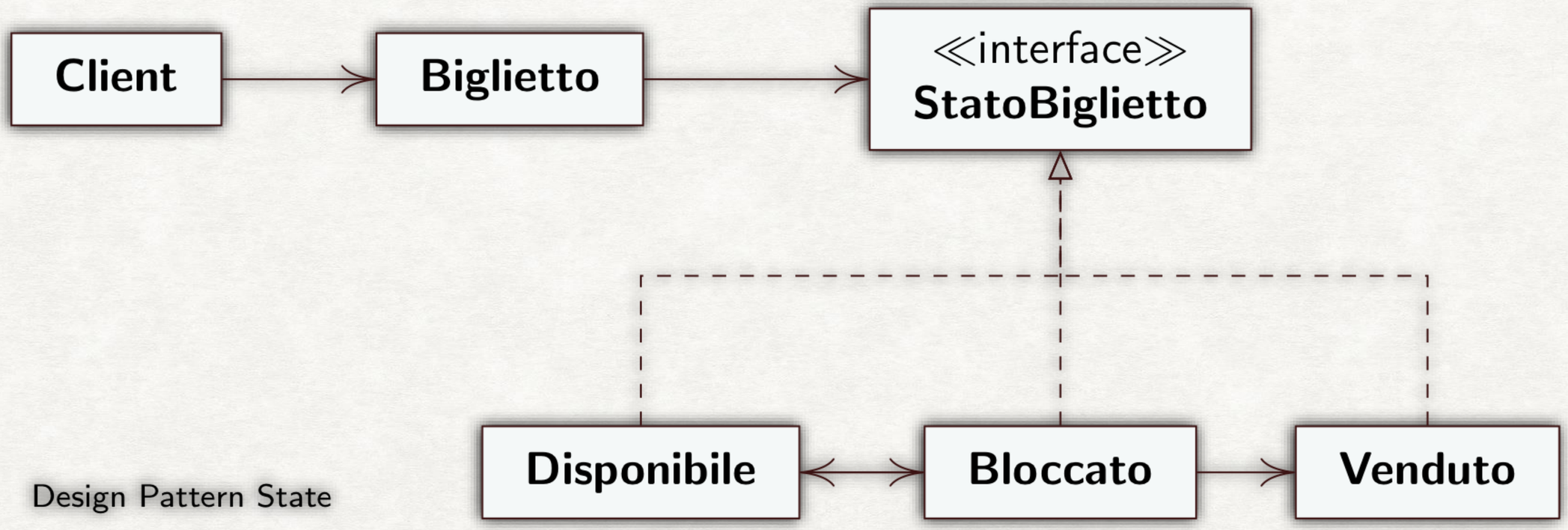
```
import java.time.LocalDateTime;
public class Venduto implements StatoBiglietto { // Venduto e' un ConcreteState
    private final String nome;
    private LocalDateTime dataPagam;

    public Venduto(String n) { nome = n; }

    @Override public void mostra() { System.out.println("VEND Nome: " + nome); }
    @Override public StatoBiglietto intesta(String s) {
        System.out.println("VEND Non puo' cambiare il nome nello stato Venduto");
        return this;
    }
    @Override public StatoBiglietto paga() {
        if (dataPagam == null) {
            dataPagam = LocalDateTime.now();
            System.out.println("VEND Pagamento effettuato");
        } else
            System.out.println("VEND Il biglietto era gia' stato pagato");
        return this;
    }
    @Override public StatoBiglietto cancella() {
        System.out.println("VEND Non puo' cambiare stato da Venduto a Disponibile");
        return this;
    }
}
```

```
// Biglietto e' un Context

public class Biglietto {
	private String codice = "XYZ";
	private int prezzo = 100;

	private StatoBiglietto sb = new Disponibile();

	public void mostra() {
		System.out.println("Prezzo: " + prezzo + "  codice: " + codice);
		sb.mostra();
	}

	public void prenota(String s) {
		sb = sb.intesta(s);
	}

	public void cancella() {
		sb = sb.cancella();
	}

	public void compra() {
		sb = sb.paga();
	}
}
```

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

    private static void usaBiglietto() {
          b.prenota("Mario Tokoro");
          b.mostra();
          b.compra();
          b.mostra();
    }

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

Output dell'esecuzione di MainBiglietto

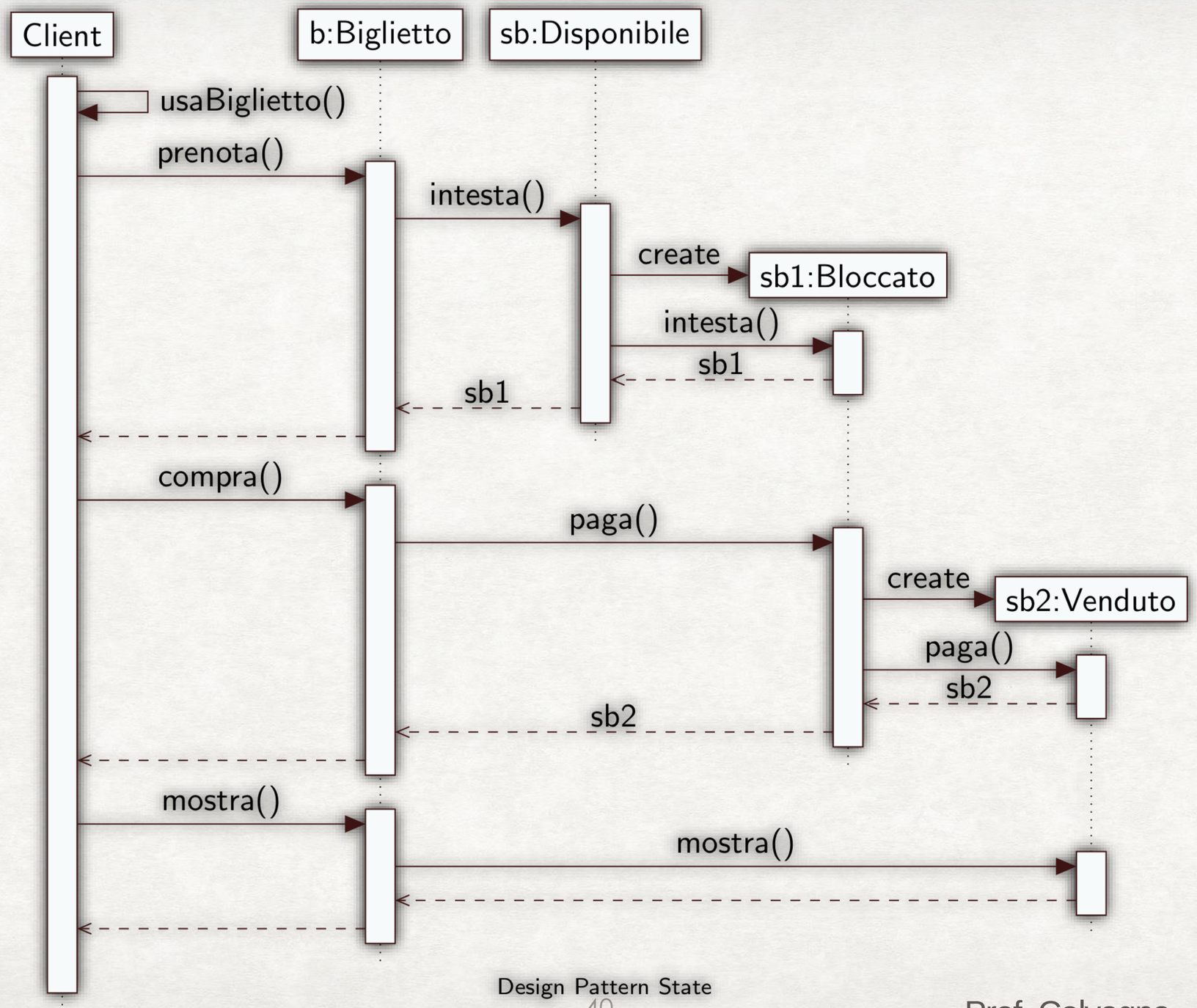
```
Prezzo: 100  codice: XYZ
DISP Cambia stato da Disponibile a Bloccato
BLOC Inserito nuovo intestatario
Prezzo: 100  codice: XYZ
BLOC Nome: Mario Tokoro
BLOC Cambia stato da Bloccato a Venduto
VEND Pagamento effettuato
Prezzo: 100  codice: XYZ
VEND Nome: Mario Tokoro

VEND Il biglietto era gia' stato venduto
VEND Non puo' cambiare stato da Venduto a Disponibile
VEND Non puo' cambiare il nome nello stato Venduto
```



Design Pattern State

Conseguenze

- 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
- Ogni metodo è più semplice da comprendere e modificare
- 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 permette di ritornare il riferimento a un nuovo state (è detta fluent)
- Il codice per ciascuno stato può implementare altre attività senza complicarsi
- La presenza di switch è un sintomo che suggerisce di usare il polimorfismo
- 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 ";"


Client
b:Biglietto
sb:Disponibile
usaBiglietto()
prenota()
intesta()
create
sb1:Bloccato
intesta()
sb1
sb1
compra()
paga()
create
sb2:Venduto
paga()
sb2
sb2
mostra()
mostra()