LEZIONE 57-58
SINTESI DI COMPORTAMENTI DA UML STATE MACHINES

Ingegneria del Software e Progettazione Web

Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

la famiglia di diagrammi UML

- structure diagrams
  - class diagram
  - object diagram
  - component diagram
  - deployment diagram
  - composite structure diagram
  - package diagram
- behavior diagrams
  - state machine diagram
  - activity diagram
  - use case diagram
  - interaction diagrams
    - sequence diagram
    - communication diagram
    - interaction overview diagram
    - timing diagram

la famiglia di diagrammi UML

- stru
  - c
  - o
  - c
  - d
  - c
  - p

ATTENZIONE!!!!
la presentazione dettagliata delle
UML State Machines è stata affrontata
in Lezione 39 e Lezione 40
... di seguito un breve richiamo
ai concetti principali come
introduzione all'argomento
della lezione

un po di storia ...

- derivano dagli state charts proposte David Harel
  - riadattate al mondo Object Oriented
- includono anche alcuni aspetti delle macchine di Moore e Mealy
- ovvero
  - macchina a stati e transizioni
  - enfatizzano il flusso di controllo da stato a stato

quando e perchè ?

- vengono usate per modellare l'aspetto dinamico del comportamento di un sistema o parti di esso
- in particolare
  - modellano il comportamento di oggetti reattivi
  - modellano il comportamento di oggetti "stateful"
  - sono anche usati per modellare comportamenti di attori, use-case o metodi

rappresentazione grafica in UML

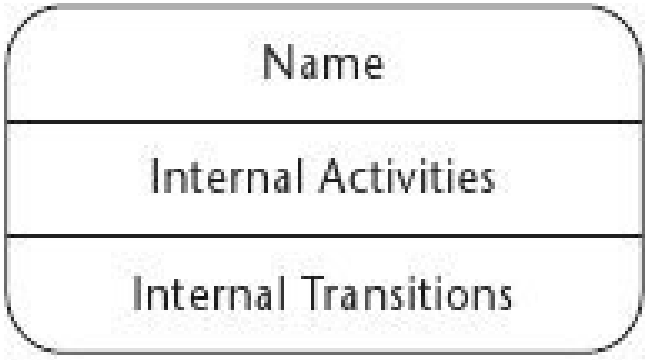
- graficamente sono rappresentate da una specie di grafo
  - nodi e archi
- concettualmente aggiungono alla nozione di grafo un po' di semantica
- in particolare:
  - nodo: stato, attività, azioni
  - arco: transizioni, eventi, invocazione di operazioni

stato

- rappresenta una situazione (nella vita di un oggetto) durante la quale delle condizioni vengono soddisfatte e delle azioni o attività possono essere eseguite
- può essere semplice oppure composto
  - esempio: un aereo può trovarsi nei seguenti 5 stati
    - `On`, `Off`, `TakingOff`, `Landing`, `Flying`
- lo stato di un oggetto varia nel tempo ma è sempre determinato da
  - i valori dei suoi attributi
  - le istanze di relazioni con altri oggetti
  - le attività che sta svolgendo

uno stato in UML

- graficamente: rettangolo con gli angoli smussati
- concettualmente: composto da 3+1 parti



  - nome: stringa di testo
  - attività interne: azioni (interne) eseguite in risposta ad eventi o ad invocazioni di operazione. user-defined/standard
  - transizioni interne: cambiamenti di stato interni che avvengono al soddisfacimento di una condizione
  - (decomposizione: solo per state machine gerarchiche)

attività interne

- sono caratterizzare dai seguenti concetti:

```
event-name args-list ‘/’ action-expression
```

  - entry: attività eseguita nel momento in cui si entra in uno stato. Concettualmente è una attività atomica (i.e. non può essere interrotta)
  - exit: attività eseguita nel momento in cui si esce da uno stato. Concettualmente è una attività atomica
  - do: identifica un’attività in esecuzione mentre l’oggetto è nello stato della transizione. La sua esecuzione può essere interrotta alla ricezione di un evento
  - (include: invocazione a una submachine. l’azione contiene il nome della submachine invocata)

transizioni

- è una relazione tra due stati
  - indica un cambiamento di stato da parte di un oggetto al verificarsi di un certo evento e di una certa condizione
- graficamente è una freccia con un’etichetta che
  - esce dallo stato di partenza
  - entra nello stato di destinazione a seguito della transizione

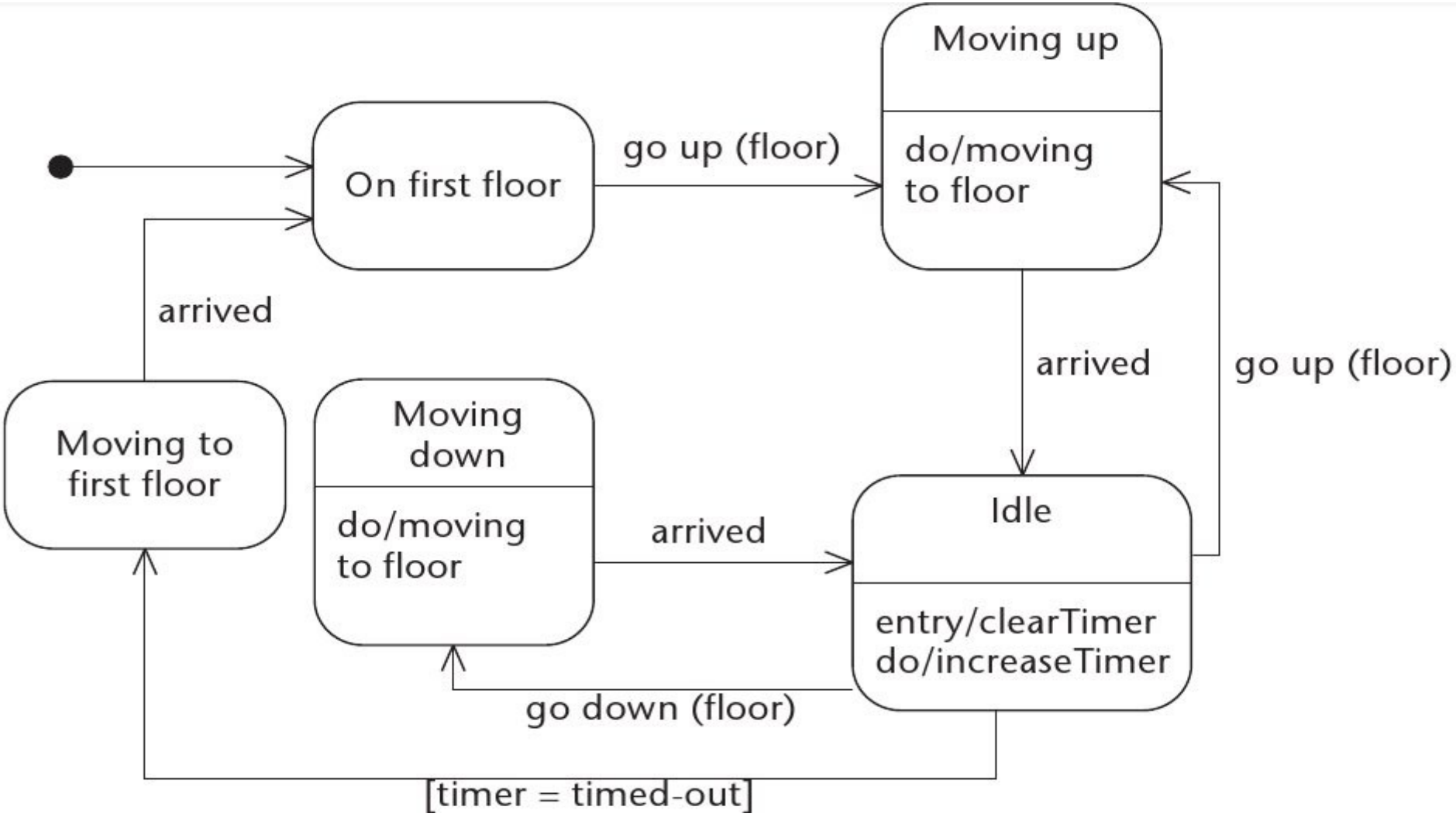
transizioni – rappresentazione

- 5-tupla composta da :
  - stato sorgente
  - evento scatenante
  - condizione di guardia
  - effetto : azione OR generatore di eventi
  - stato destinatario
- sono caratterizzate da :

```
event-signature[guard]/action^send-clause
```

- dove :
  - `send-clause = dest-expr.event-name args-list`
  - `event-signature = event-name args-list`

transizioni – esempio



state machine – specificare i comportamenti

state machine – specificare i comportamenti

- il comportamento delle classi può essere specificato:
  - come un algoritmo
  - esplicitamente con una state machine
  - in entrambi i modi
- difficoltà da state machine a O.O.
  - rappresentazione e gestione dei segnali/eventi
  - la specifica ufficiale di un UML è agnostica rispetto alle soluzioni implementative
  - i modelli UML di per sé (cioè senza un trasformatore associato) sono ambigui

una possibile soluzione

- le transizioni di stato di una classe avvengono solo attraverso le operazioni pubbliche esportate dalla classe
  - *SM* può implementare il comportamento di *C* se ogni transizione di *SM* corrisponde ad una operazione pubblica di *C* e viceversa.
- le classi che devono ricevere/gestire un segnale/ evento devono esportare una operazione corrispondente che riceve il segnale come argomento
  - implementare eventi e segnali come classi (con propri attributi ed operazioni)

una possibile soluzione

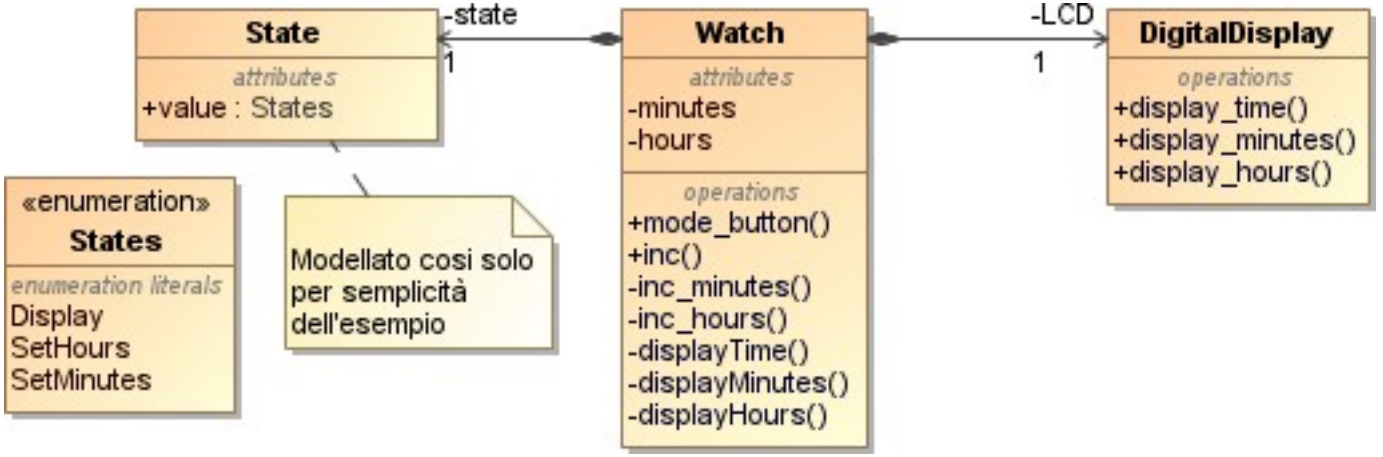
- le t
  sol
  dal
  - S
    t
    p
- le c
  eve
  cor
  arg
  - i
    (

ATTENZIONE!!!!
il metodo proposto
(e dettagliato nel seguito)
fa riferimento a molti aspetti implementativi, quindi è opportuno considerarlo nella fasi di raffinamento della PROGETTAZIONE (e.g. in Logic View) verso una possibile implementazione; ma non è adatto nelle fasi/viste di ANALISI (e.g. in Use Case View)

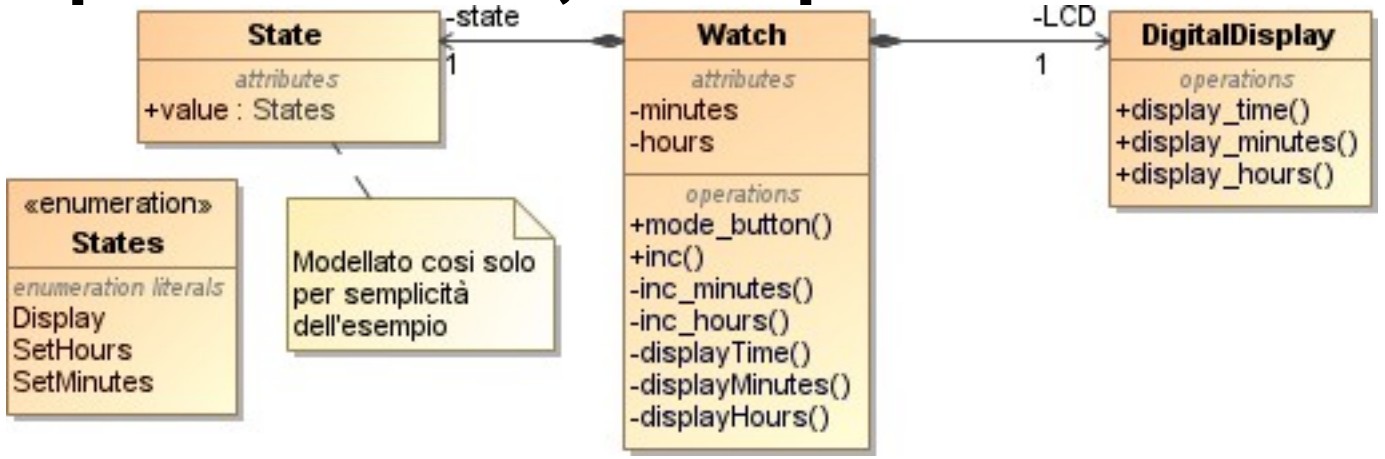
esempio – state machine

- si vuole modellare il comportamento di un orologio digitale.
- l'orologio:
  - si compone di una unità dedicata al controllo del display
  - ha diverse modalità di funzionamento (e.g. regolazione ora, regolazione minuti, display orario)
  - permette di scegliere la modalità di funzionamento
  - permette di ricevere speciali tipi di input (e.g. tick per incremento dell'ora)

esempio – struttura statica CD

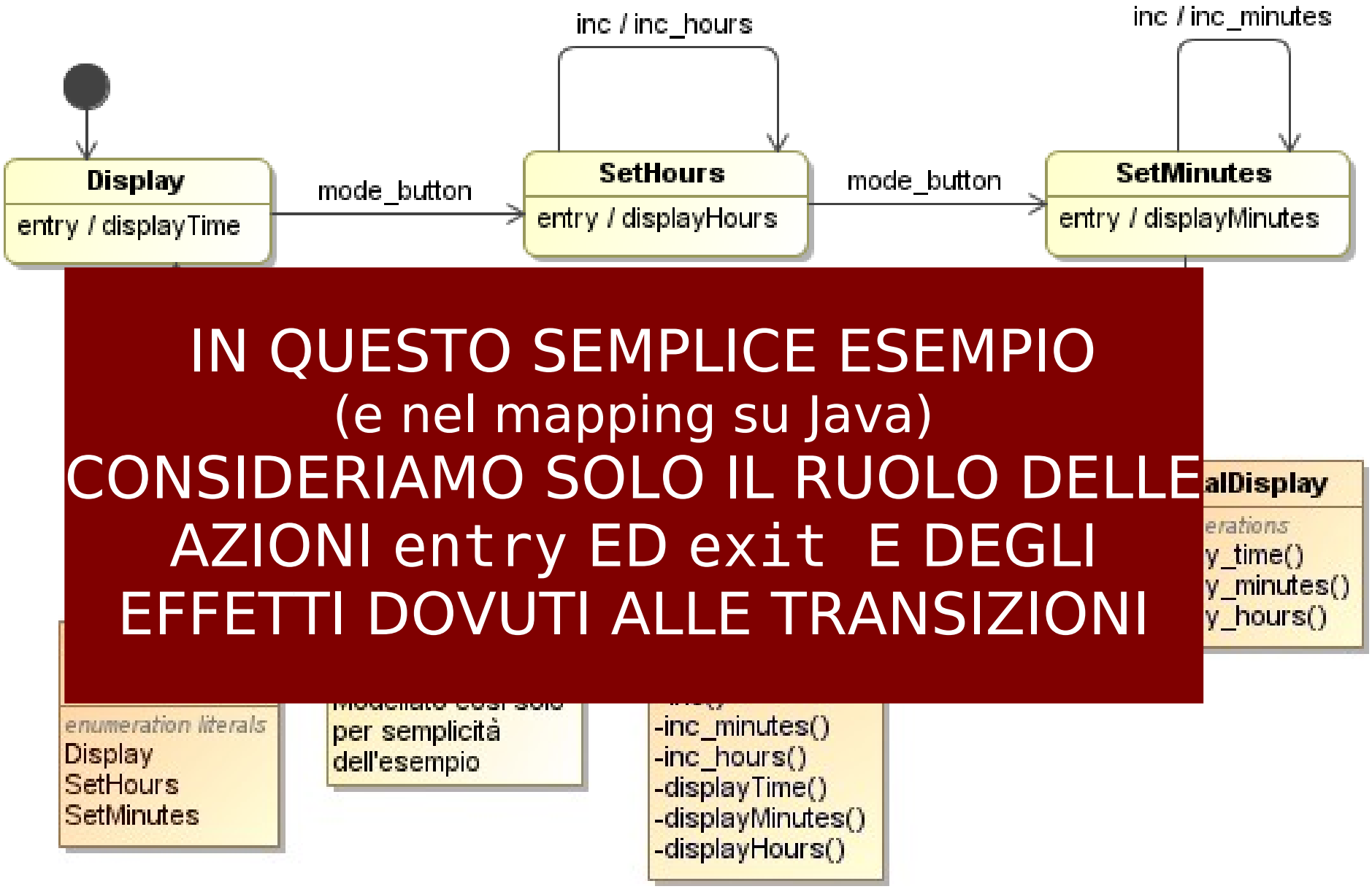


esempio – CD, map in Java

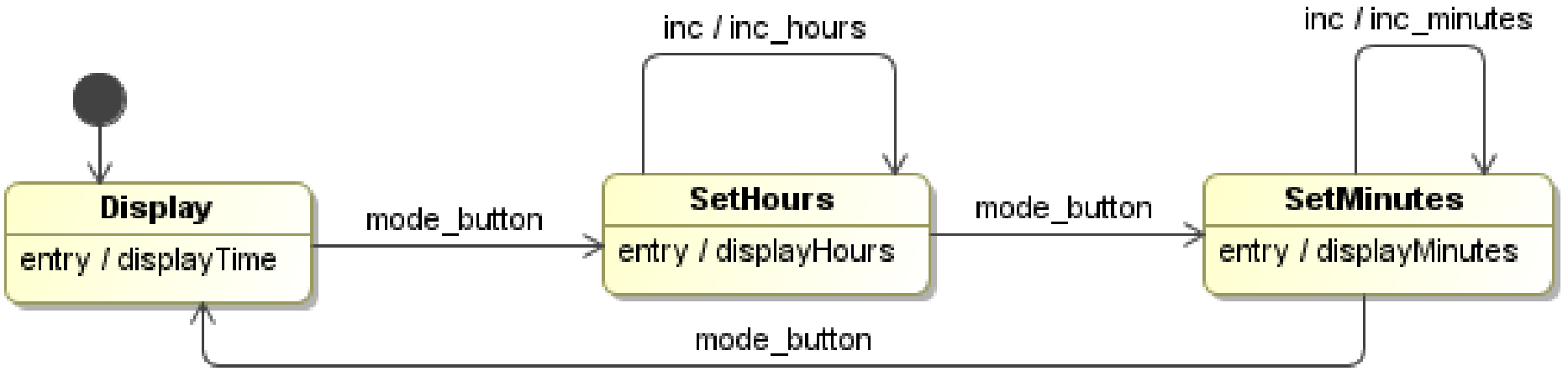


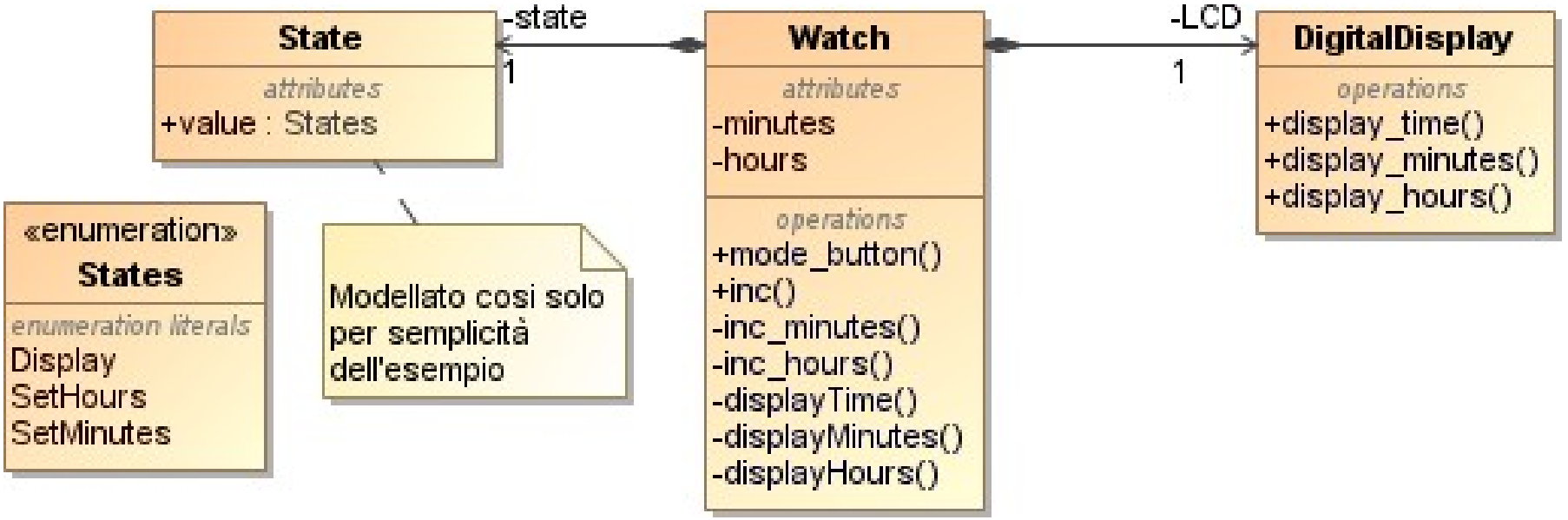

```
public class Watch {
  private State state;
  private DigitalDisplay LCD;
  public Watch (){
    state = new State();
    LCD = new DigitalDisplay();
    state.value = State.Display;
    LCD.display_time();
  }
  public void mode_button(){ ... }
  public void inc(){ ... }
  ... ...
}
```

esempio – state machine

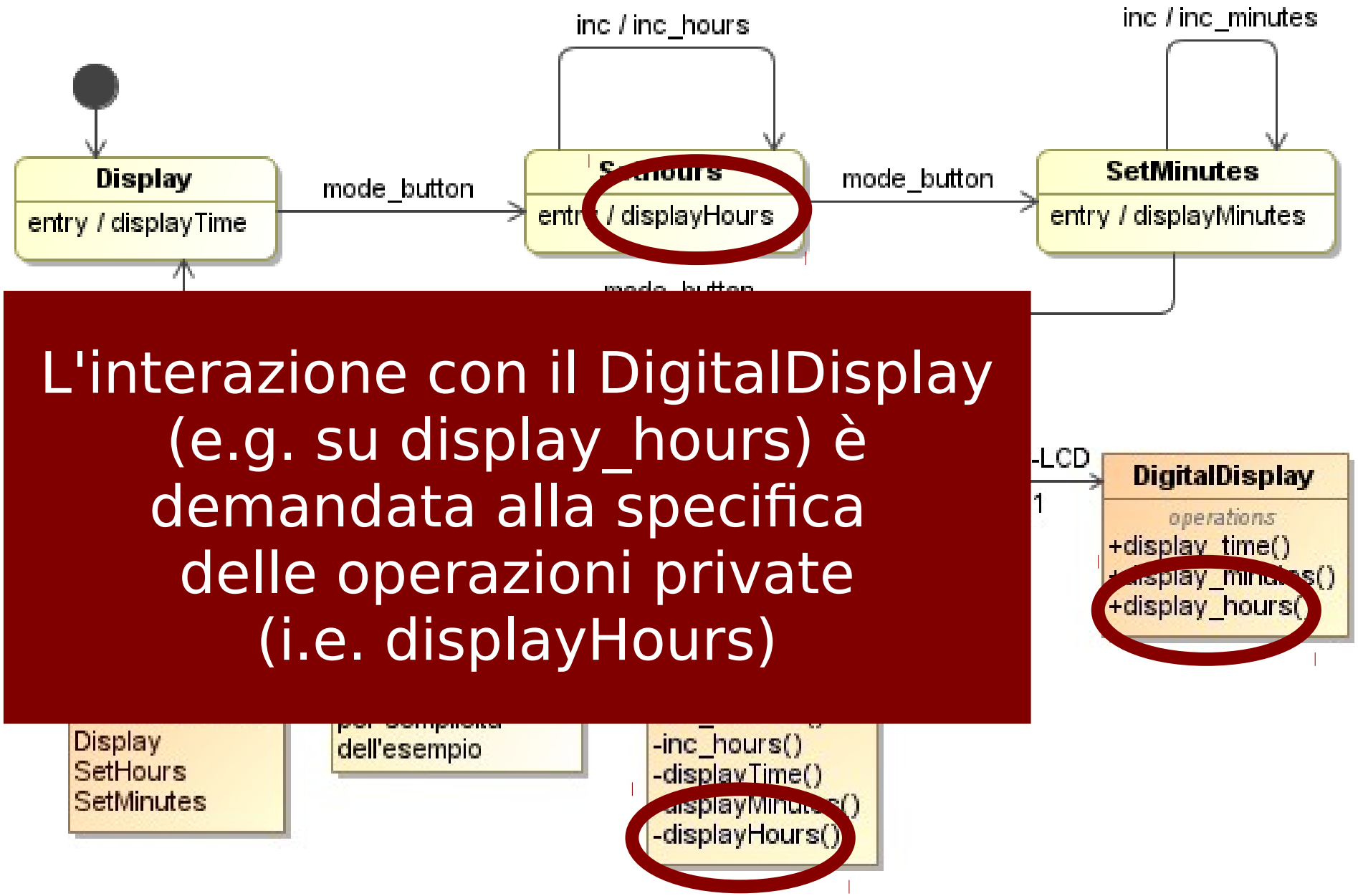


esempio – state machine





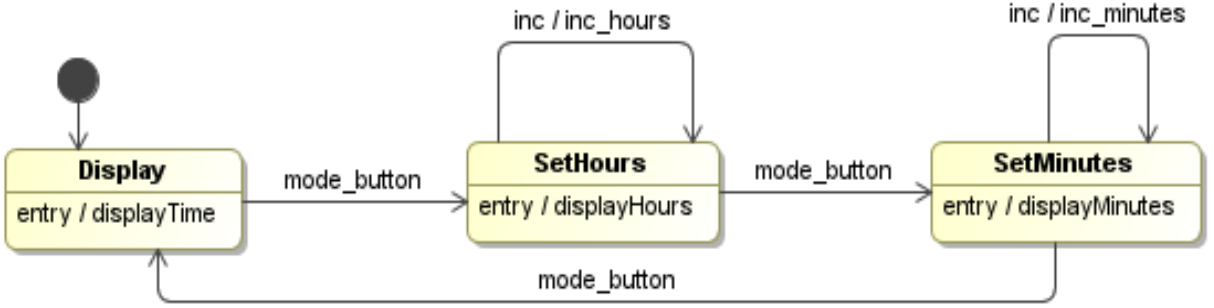
esempio – state machine



esempio – state machine && Java

```
public void mode_button(){
  switch (state.value){
    case … … :
     // <check the guard, if it exists>
     if ( … ){
        // <exit the old state>
        … …
       // <effect of the transition>
       … …
     // <change state>
        … …
       // <entry the new state>
       … …
      }
    break;
    … …
  }
}
```

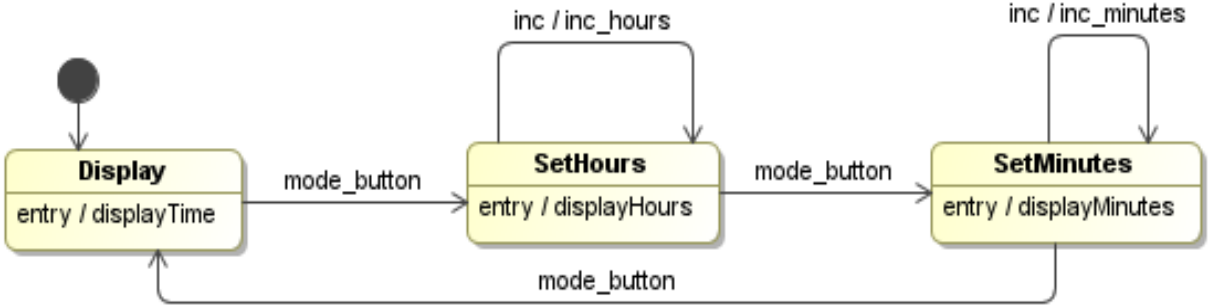
```
public void inc() {
  switch (state.value){
    case … … :
      … …
    break;
    … …
    case … … :
      … …
    break;
    }
  }
}
```



esempio – state machine && Java

```
public void mode_button(){
    switch (state.value){
      case State.Display:
 //no cond., no exit, no effects
        state.value=State.SetHours;
        this.displayHours();
      break;
      case State.SetHours:
 //no cond., no exit, no effects
        state.value=State.SetMinutes;
        this.displayMinutes();
      break;
      case State.SetMinutes:
 //no cond., no exit, no effects
        state.value = State.Display;
        this.displayTime();
      break;
    }
 }
```

```
public void inc() {
 switch (state.value){
   case State.Display :
     //nothing to do
   break;
   case State.SetHours:
    //no cond., no exit,
     this.inc_hours();
     state.value = State.SetHours;
     this.displayHours();
   break;
   case State.SetMinutes:
   //no cond., no exit,
     this.inc_minutes();
     state.value = State.SetMinutes;
     this.displayMinutes();
   break;
 }
```



catalogo GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

catalogo GoF

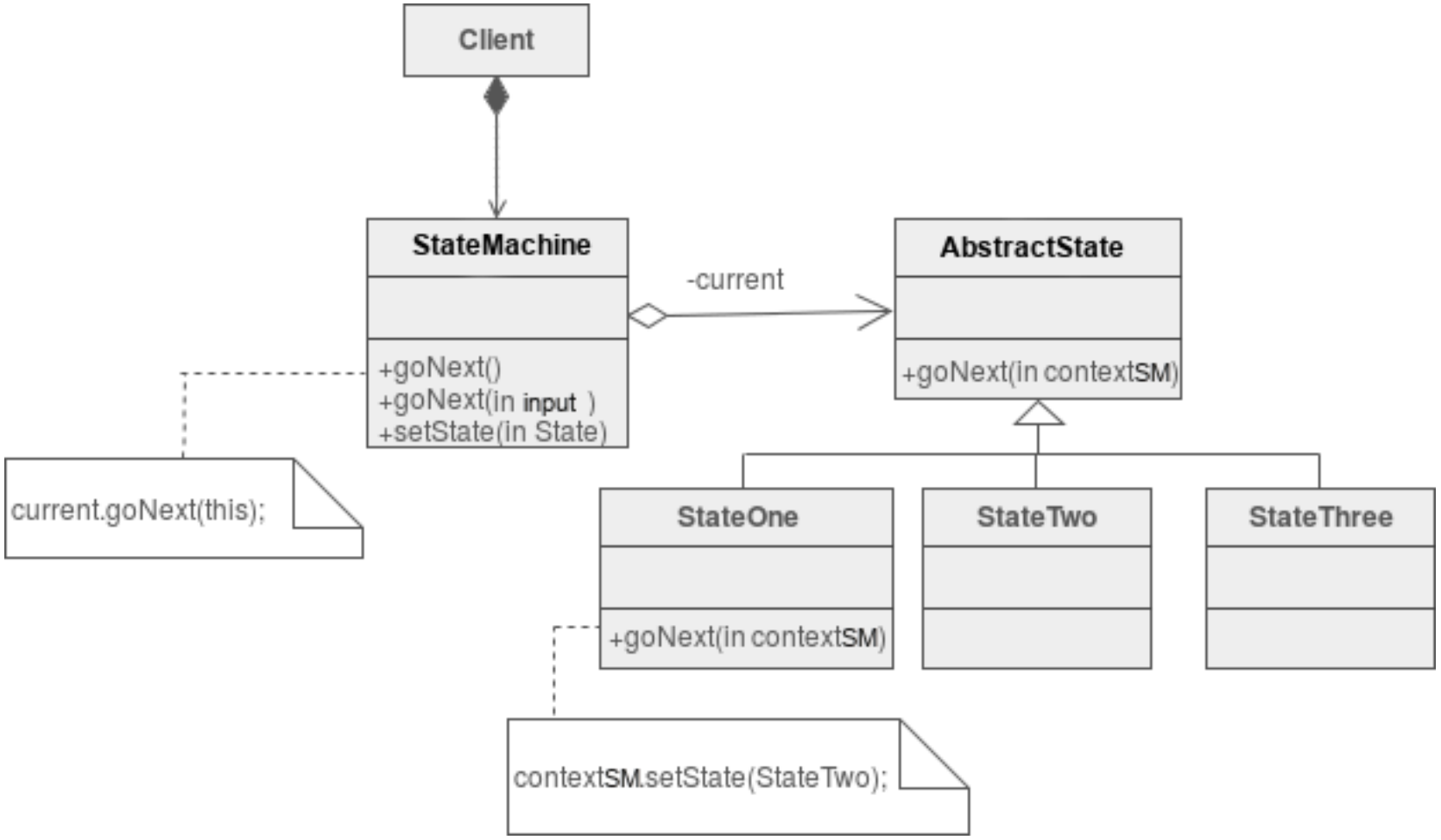
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

state – 1

- Scopo
  - consentire ad un oggetto di cambiare il suo comportamento a seguito di un cambiamento del suo stato interno
- Sinonimi
  - Strategy Pattern (dove la “strategia” rappresenta gli stati)
- Motivazione
  - definire una object-oriented state machine
- Applicabilità
  - definire una classe “di contesto” (i.e. `StateMachine`) che rappresenta il contratto tra un client ed il comportamento che si vuole modellare
  - rappresentare la nozione astratta di stato (i.e. `AbstractState`)
  - comportamenti specifici degli stati sono definiti nella sotto-classi di `AbstractState`

state – 2

- Struttura



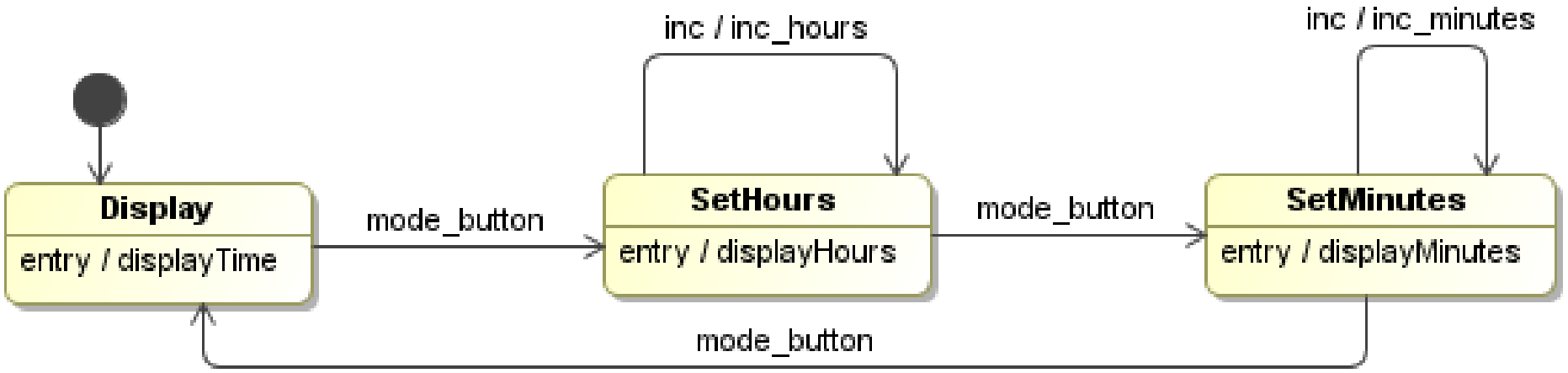
state – 3

- Partecipanti
  - Client
    - rappresenta la classe alla quale si vuole associare un comportamento per mezzo di una state machine
  - StateMachine (in alcuni testi riferito come "**Context**")
    - specifica la macchia a stati riferita dal `Client`
    - definisce l'insieme degli eventi attivabili sulla macchia a stati come visti dal `Client`
    - mantiene un riferimento ad un `ConcreteState` attraverso la sua astrazione `AbstractState`
  - AbstractState (in alcuni testi riferito semplicemente come "**State**")
    - specifica l'interfaccia che incapsula la logica del comportamento associato ad un determinato stato
  - ConcreteState (StateOne, StateTwo, StateThree)
    - implementa il comportamento associato ad un particolare stato
    - una sottoclasse per ogni stato che si vuole modellare

state – 4

- Collaborazioni
  - `Client` notifica `StateMachine` per cambiare il suo stato
    - lo inizializza o transisce allo stato successivo
  - `StateMachine` mantiene il riferimento all'istanza che rappresenta lo stato attuale in cui si trova il `Client`
  - a seguito dell'invocazione dell'operazione `goNext()` su `StateMachine`, verrà invocato l'opportuno metodo `goNext(StateMachine)` relativo all'istanza di `ConcreteState` legata con la specifica `StateMachine`
- Conseguenze
  - non sono specificati vincoli su dove implementare le transazioni:
    - vantaggi: è facile aggiungere un nuovo stato da gestire
    - svantaggi: ogni classe derivata da `AbstractState` (i.e. i sotto-stati) hanno un alto grado di coupling, ogni classe stato deve avere una buona conoscenza delle altre sottoclassi

esempio – state GoF



provare a modellare ed implementare
la state machine
discussa in precedenza attraverso
l'applicazione del GoF STATE

esempio – state GoF possibile soluzione