

# Mocking

<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>  
<https://dzone.com/refcardz/mockito>

Per ogni test di unità ci dovrebbe essere:

- un unico componente reale
  - quindi una unica `new` che crea il **SUT**
- i vari DOC implementati tramite mocking
  - non si mockano in genere classi delle librerie standard

In certi casi (classi più complesse) può essere utile/necessario mockare parte dell'oggetto sotto esame per renderlo (più facilmente) testabili:

- si può creare oggetto reale e poi farne uno spy per sovrascrivere alcuni metodi
- si può iniettare un oggetto mockato in un attributo della classe in esame
  - se c'è costruttore che lo permette... facile
  - se non c'è si può provare a usare annotazione `@Injectock` che realizza un semplice meccanismo di `DependencyInjection`

# Observer pattern mocking

- test del model con mock degli Observers

```
@Test
void modelTest {
    //SETUP
    Model model = new Model();
    Observer obs = mock(Observer.class);
    Observer obs1 = mock(Observer.class);

    //EXERCISE
    model.addObserver(obs);
    model.addObserver(obs1);
    model.setTemp(42.0);

    //VERIFY
    verify(obs).update(eq(model), eq("42.0"));
    verify(obs1).update(eq(model), eq("42.0"));
}
```

# Observer pattern mocking (cont)

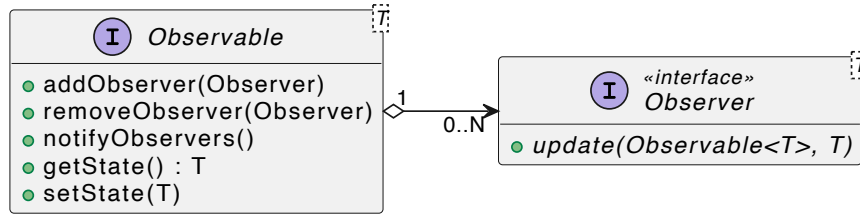
- test di un observer con un modello non generico ma di cui ho solo interfaccia di cui fornisco una versione dummy

```
@Test
void observerTest {
    //SETUP
    abstract class MockObservableIModel extends Observable implements Model {};
    MockObservableIModel model = mock(MockObservableIModel.class);
    when(model.getTemp()).thenReturn(42.42);

    //EXERCISE
    observer.update(model, null);

    //VERIFY
    verify(model).getTemp();
    assertThat(observer.getVal()).isCloseTo(42.42, Offset.offset(.01));
}
```

# Versione Generica



```
interface Observer<T> {
    void update(Observable<T> model, T state);
}

interface Observable<T> {
    void addObserver(Observer<T> observer);
    void removeObserver(Observer<T> observer);
    void notifyObservers();
    T getState();
}
```



```
public class State {  
    private double temp;  
    public State(double temp) { this.temp = temp; }  
    public double getTemp() { return temp; }  
    public void setTemp(double temp) { this.temp = temp; }  
}
```

```
public class Model extends State implements Observable<Double> {  
    private final List<Observer<Double>> observers = new ArrayList<>();  
    @Override public void addObserver(Observer<Double> observer) {  
        observers.add(observer);  
    }  
    @Override public void removeObserver(Observer<Double> observer) {  
        observers.remove(observer);  
    }  
    @Override public void notifyObservers() {  
        for (Observer<State> observer : observers)  
            observer.update(this, getState());  
    }  
    @Override public Double getState() {  
        return getTemp();  
    }  
    @Override public void setTemp(Double state) {  
        super.setTemp(state);  
        notifyObservers();  
    }  
}
```

# Mockito dependency Injection

```
@ExtendWith(MockitoExtension.class)
public class TestPartita {
    @Mock
    Tavolo mockedTable;

    @InjectMocks
    Partita SUT;

    @Test
    void primoTest () {
        assertThat(SUT.controllaSeCartaPresenteSuTavolo(Card.get(Rank.ACE, Suit.CLUBS))).isFalse();
    }

    @Test
    void secondoTest () {
        when(mockedTable.inMostra(Card.get(Rank.ACE, Suit.CLUBS))).thenReturn(true);

        assertThat(SUT.controllaSeCartaPresenteSuTavolo(Card.get(Rank.ACE, Suit.CLUBS))).isTrue();
    }
}
```

# funzione per fare mocking di Iterable

Ve l'ho data durante il laboratorio

```
public class MockUtils {  
    @SafeVarargs  
    public static <T> void whenIterated(Iterable<T> p, T... d) {  
        when(p.iterator()).thenReturn((Answer<Iterator<T>>) invocation -> List.of(d).iterator());  
    }  
}
```

come la usiamo?

# possibile uso

```
@Test
void scegliSuccess() {
    Giocatore player1 = mock(Giocatore.class);
    Giocatore player2 = mock(Giocatore.class);
    Giocatore me = mock(Giocatore.class);

    when(player1.getMazzettoTop()).thenReturn(Rank.FIVE);
    when(player2.getMazzettoTop()).thenReturn(Rank.TEN);

    Partita partita = mock(Partita.class);
    MockUtils.whenIterated(partita, me, player2, player1);

    List<Card> mano = List.of(Card.get(Rank.ACE, Suit.CLUBS), Card.get(Rank.KING, Suit.DIAMONDS), Card.get(Rank.TEN, Suit.HEARTS));
    SelettoreCarta FAILED = mock(SelettoreCarta.class);

    SelettoreCarta strategy = new SelettoreRubaMazzetto(FAILED, me);

    assertThat(strategy.scegli(mano, partita)).isEqualTo(Card.get(Rank.TEN, Suit.HEARTS));
}
```



# Come si identificano le classi e le relazioni?

## TDD e Pattern

- stiamo sperimentando a laboratorio

## Noun Extraction

- È basato sulle specifiche
  - ad esempio i commenti esplicativi delle UserStory/Use Case
- Vengono estratti tutti i **sostantivi**, o **frasi sostantivizzate**
- Si considerano tutti i candidati e poi si comincia a sfoltire
- Poi si cercano le relazioni (probabilmente date dai **verbi**)

# Esempio

- The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks.
- Member of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only member of staff may borrow journals.
- The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.


# Criteri di sfoltimento

- **Ridondanza**
  - sinonimi, nomi diversi per lo stesso concetto
    - library member, member of the library
    - loan e short term loan
- **vaghezza**, nomi generici
  - items
- **nomi di eventi e operazioni**
  - loan
- **metalinguaggio**
  - system, rules
- **esterno al sistema**
  - library, week
- **attributi**
  - se ci fosse stato *name of the member*

# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff

 Book

 CopyOfBook

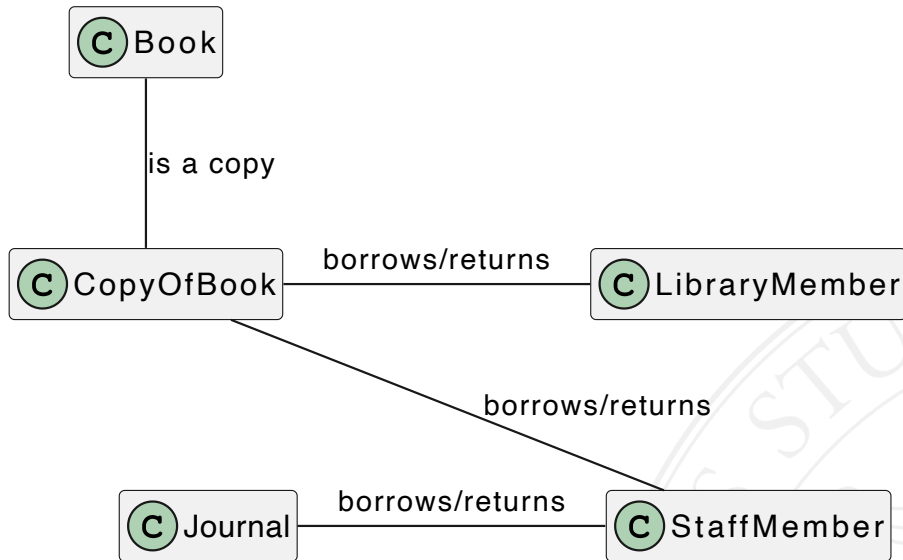
 LibraryMember

 Journal

 StaffMember

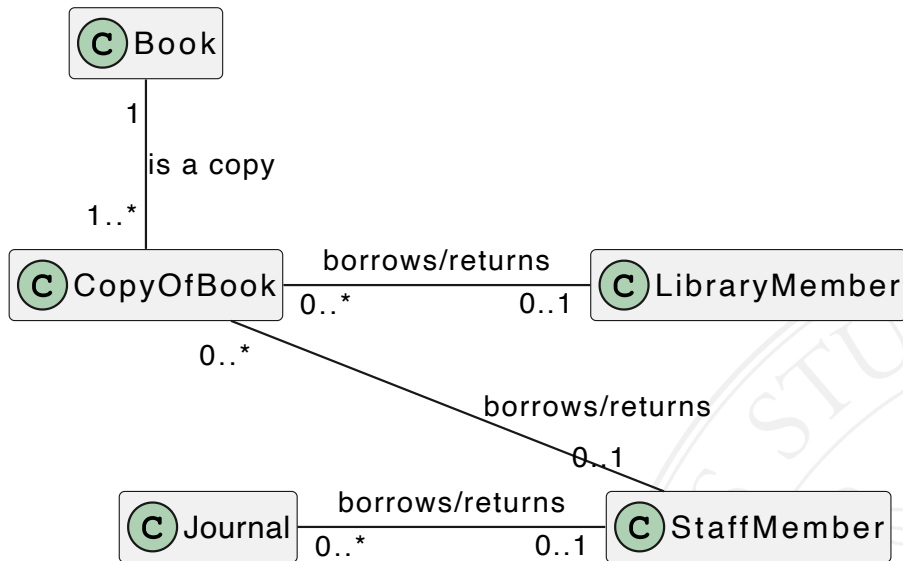
# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff



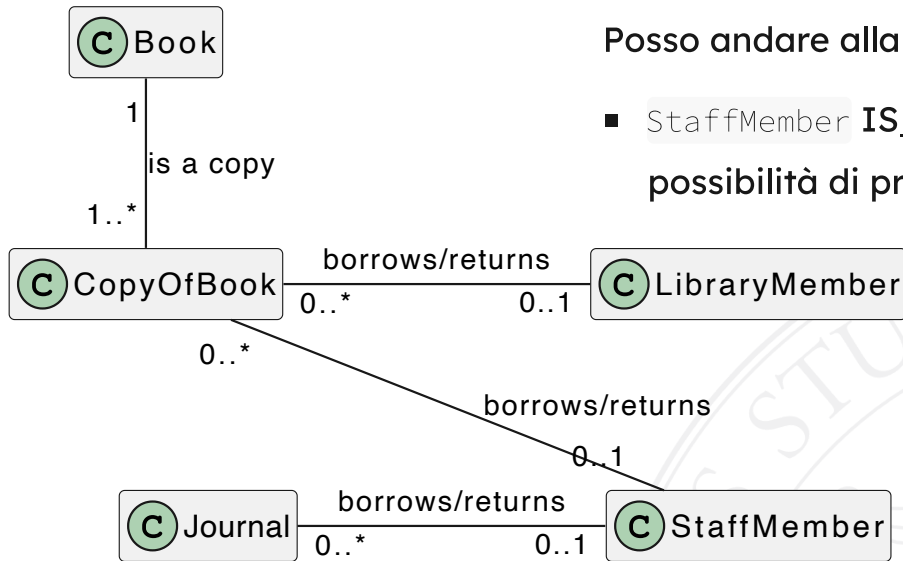
# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff



# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff

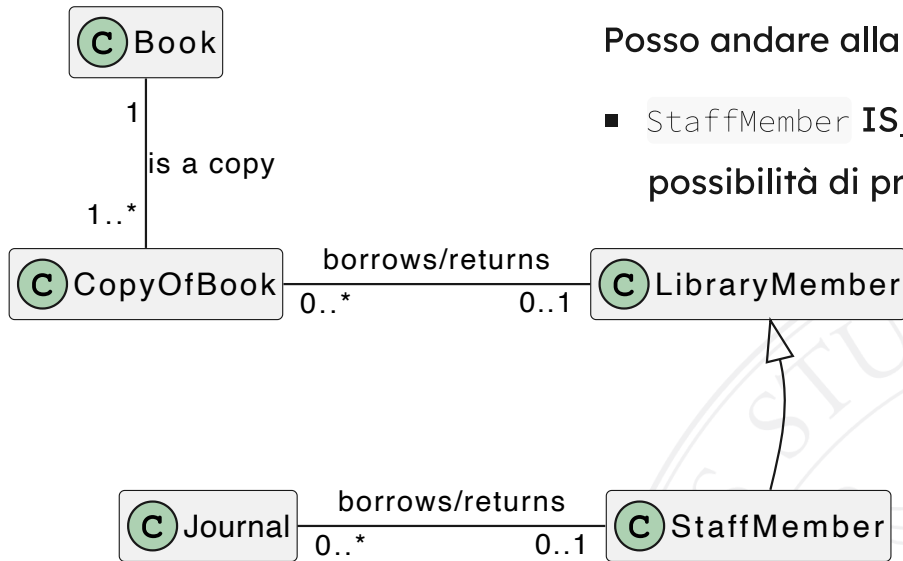


Posso andare alla ricerca di **generalizzazioni**:

- **StaffMember** **IS\_A** **LibraryMember** con in più la possibilità di prendere **Journal**

# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff



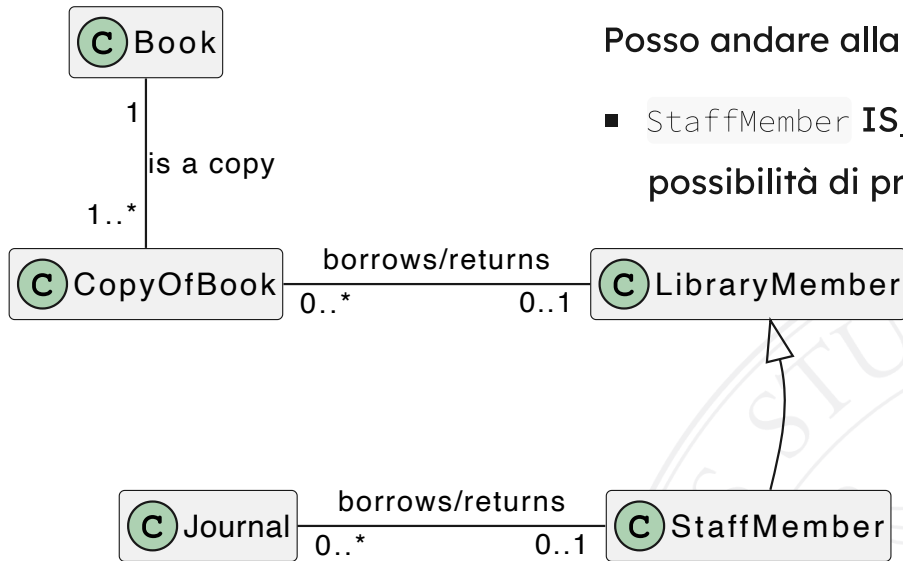
Posso andare alla ricerca di **generalizzazioni**:

- StaffMember **IS\_A** LibraryMember con in più la possibilità di prendere Journal



# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff



Posso andare alla ricerca di **generalizzazioni**:

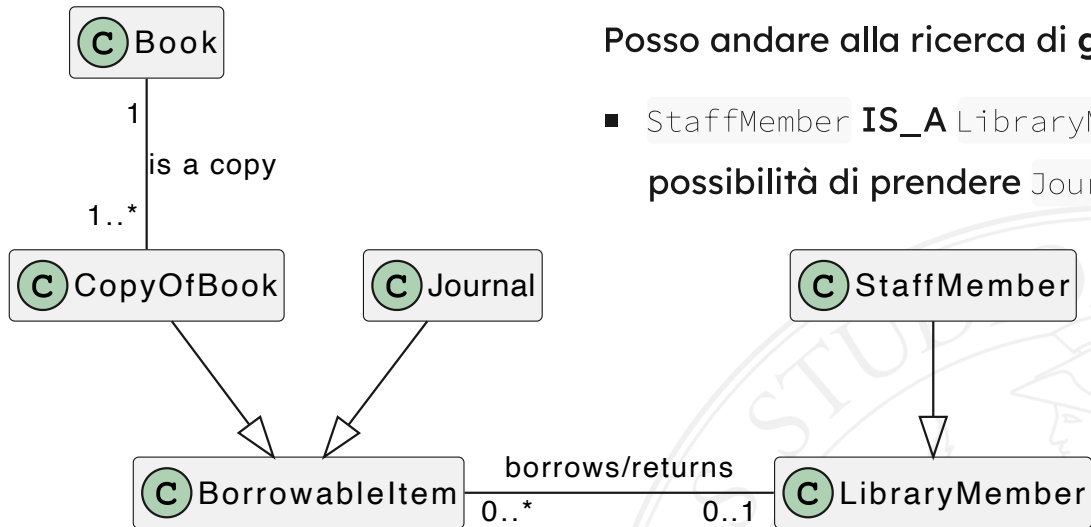
- **StaffMember** **IS\_A** **LibraryMember** con in più la possibilità di prendere **Journal**

Posso riconsiderare alcuni termini "scartati":

- **Items** era termine appunto generico per **CopyOfBook** e **Journal**

# Esempio: cosa rimane?

journal, book, copy (of book), library member, member of staff



Posso andare alla ricerca di **generalizzazioni**:

- **StaffMember** **IS\_A** **LibraryMember** con in più la possibilità di prendere **Journal**

Posso riconsiderare alcuni termini "scartati":

- **Items** era termine appunto generico per **CopyOfBook** e **Journal**

# Stato: concreto vs astratto

Lo **stato concreto** di un oggetto dipende dalla sua implementazione.

- È definito dai valori dei suoi attributi
- La cardinalità dell'insieme degli stati possibili (**state space**) *esplode* molto facilmente:
  - un `int` →  $2^{32}$  stati ( $\approx 4 * 10^9$ )
    - Se ho due interi  $2^{32} * 2^{32}$
  - un `Deck` → tutte le permutazioni di qualunque numero delle 52 carte ( $\approx 2 * 10^{68}$ )
  - una `String` → tutte le permutazioni di serie di (fino  $2^{32}$ ) caratteri... !!!

Lo **stato astratto** di un oggetto è un sottoinsieme *arbitrario* degli stati concreti

- meglio se **significativo**

# Casi particolari

## Oggetti senza stato

Esistono?

- **SI:** *Stateless Objects*

Ad esempio gli *oggetti funzione* spesso sono senza stato: sono appunto astrazioni funzionali

## Oggetti che hanno un solo stato

Esistono?

- **SI:** Sono gli oggetti immutabili

E allora perché abbiamo detto che `String` (che sappiamo essere *immutabile*) ha un numero di stati quasi infinito?

- una **classe** immutabile non ha un solo stato

Carlo Bellettini e Mattia Monga - INGEGNERIA DEL SOFTWARE - 2023-24

# State Diagram

Derivamo dagli automi a stati finiti

Un automa è una n-upla  $\langle S, I, U; \delta, t, s_0 \rangle$

- $S$  l'insieme finito e non vuoto degli stati
- $I$  l'insieme finito dei possibili ingressi
- $U$  l'insieme finito delle possibili uscite
- $\delta$  la funzione di transizione
- $t$  la funzione di uscita
- $s_0$  lo stato iniziale

# Funzione di transizione $\delta$

Stabilisce i possibili passaggi da uno stato ad un altro

$$\delta: S \times I \rightarrow S$$

- dato uno stato ed un segnale di input stabilisce qual è il prossimo stato
- può essere una funzione parziale, cioè non essere definita per tutte le possibili coppie

## Stato iniziale $s_0$

Esiste un solo stato attivo in ogni momento

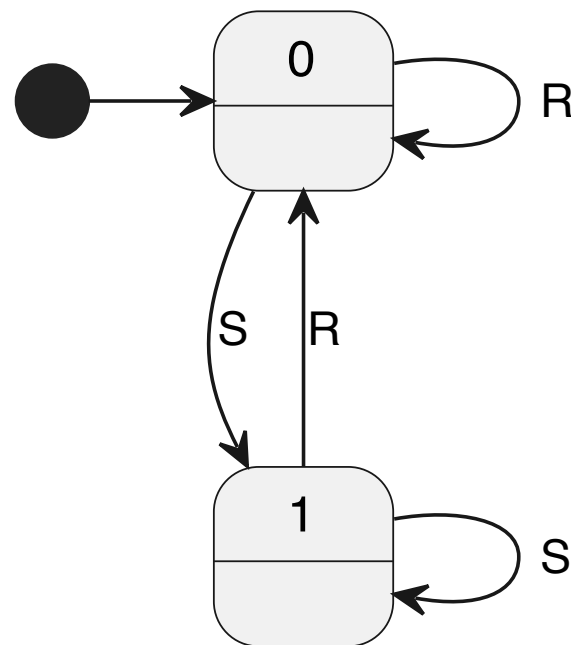
Qual è quello iniziale?

- Deve essere stabilito a priori :  $s_0 \in S$

# Un esempio di automa che conoscete

## Flip-flop Set-Reset

- insieme degli stati:
  - 0 e 1
- insieme dei simboli di ingresso:
  - R e S
- possibili transizioni:
  - archi etichettati
- funzione di uscita:
  - $t: S \rightarrow U$
- lo stato iniziale



# Automi di Mealy vs. automi di Moore

- Al contrario degli *automi di Moore* visti precedentemente, le uscite dipendono non solo dallo stato raggiunto ma anche da come viene raggiunto
  - $t: S \times I \rightarrow U$
- Sono quindi da attaccarsi agli archi



# In UML

- Il diagramma degli stati è derivato da *StateCharts*
  - Estensione a macchine a stati finiti viste
- Viene usato per definire il comportamento di un oggetto (di una classe di oggetti)

# La classe copia

Nell'esempio della biblioteca, la copia di un libro.

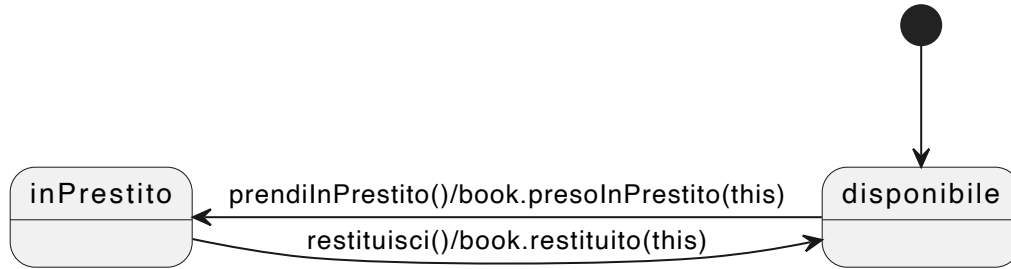
- Sono identificabili due stati significativi:
  - È disponibile
  - È già inPrestito
- Sono disponibili due metodi che si possono invocare sulla classe. Eventi a cui deve rispondere...
  - `prendiInPrestito`
  - `restituisce`

# Osservazione

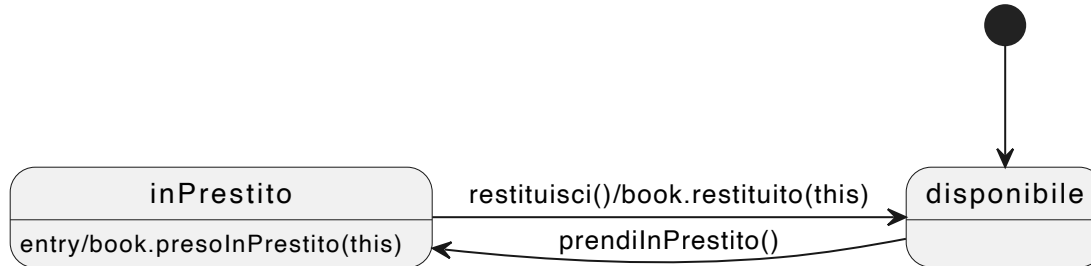
Quali aspetti abbiamo modellato?

- Cosa succede a voler modellare un registro di 32 bit?
- Se il numero di stati significativi di un sistema e' alto, gli automi possono non essere un buon metodo
- Se ad esempio volessimo (o dovessimo) tenere conto in maniera precisa del valore di una variabile di un oggetto...

# Le azioni



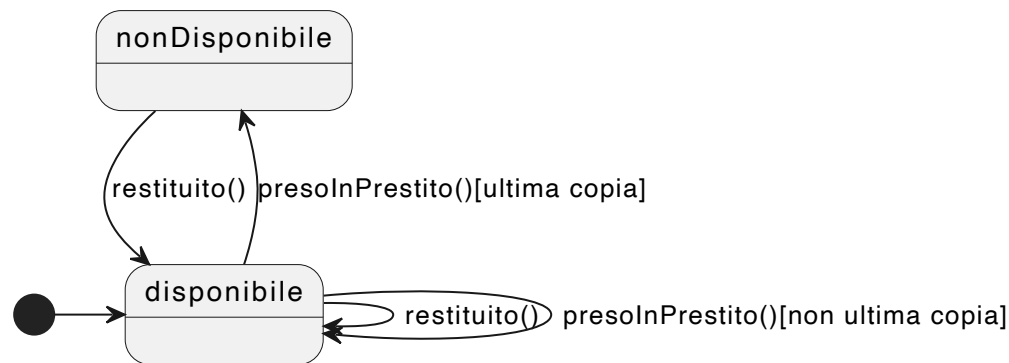
## Azioni interne allo stato



# Guardie

Servono per disambiguare transizioni causate da uno stesso evento e uscenti da stesso stato

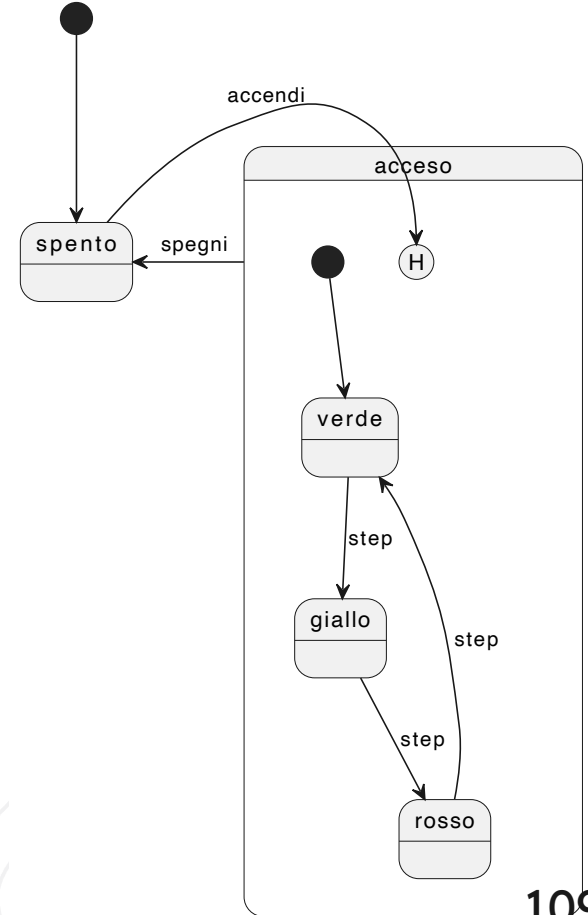
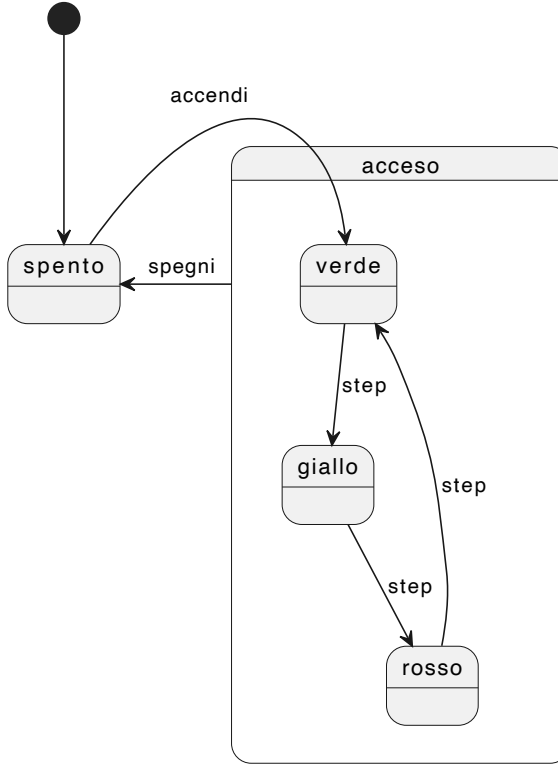
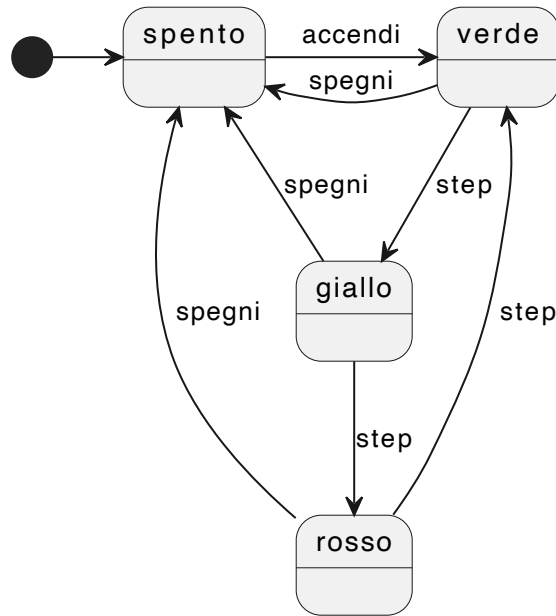
- Esempio libreria: il libro



# Altri tipi di eventi

- Time event:
  - After (duration): Indica una durata massima di permanenza nello stato
- Change event:
  - When (condizione): La condizione è espressa in termini di valori degli attributi

# Superstate



# Concorrenza

