Sapienza Università di Roma
Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica ed Automatica, Ingegneria dei Sistemi Informatici

Esercitazioni di Progettazione del Software
A.A. 2012/2013

Prova al calcolatore del 6 settembre 2013

Requisiti

Si vuole realizzare un'applicazione per simulare il funzionamento di semafori stradali. Una simulazione è caratterizzata da un codice e dalla data in cui viene effettuata. Ogni simulazione comprende un insieme ordinato di semafori (almeno due) e ogni semaforo è coinvolto in una simulazione alla volta. Ogni semaforo ha un nome ed è caratterizzato dalla durata in secondi della luce gialla e della luce verde.

In Figura 1 è mostrato il diagramma delle classi corrispondente al dominio.

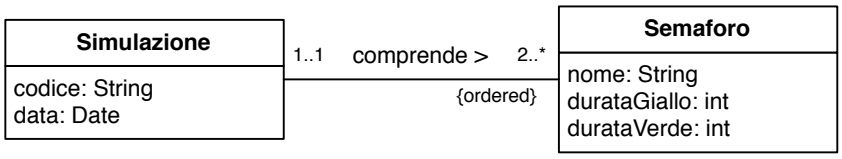

Figura 1: Diagramma UML delle classi

Durante una simulazione un solo semaforo alla volta diventa verde, in base all'ordine in cui i semafori sono coinvolti nella simulazione. Quando l'ultimo semaforo diventa rosso, la simulazione continua riprendendo dal primo semaforo della lista che diventa verde.

Un semaforo si trova inizialmente nello stato *Spento*, in attesa di essere acceso; quando riceve l'evento *accendi*, il semaforo si accende (macro-stato *Acceso*) sul colore rosso, passando nello stato *Rosso*. Il semaforo si comporta poi come segue.

- Quando è nello stato *Rosso* e riceve l'evento *verde*, la luce verde si accende, il semaforo passa nello stato $Verde$ e invia a se stesso l'evento *verde* avente come payload il valore della durata del verde del semaforo stesso.
- Quando è nello stato $Verde$ e riceve l'evento *verde*, se il payload dell'evento è positivo il semaforo resta nello stato $Verde$ (il tempo previsto per la durata del verde non è ancora passato) e dopo

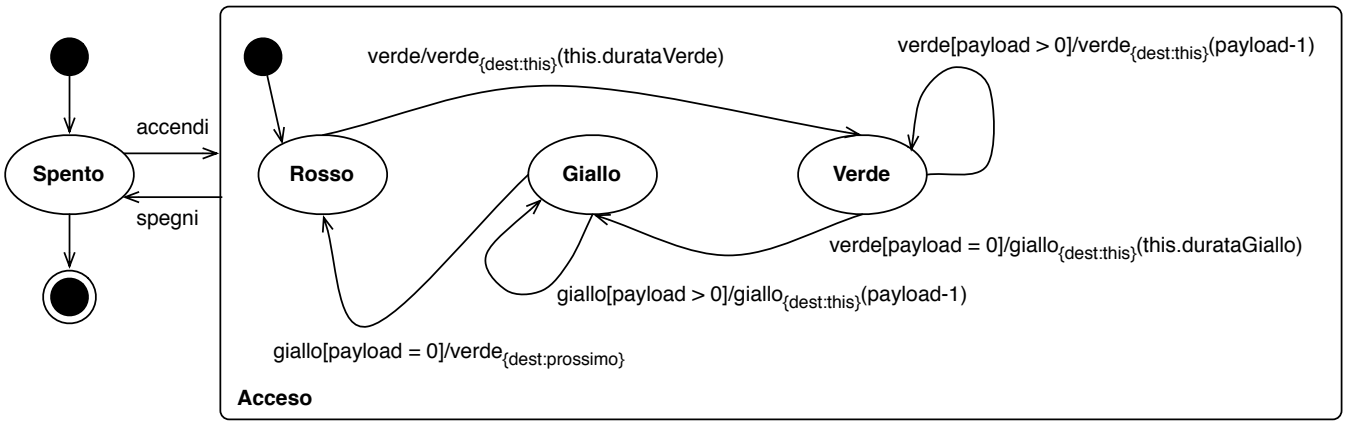

Figura 2: Diagramma degli stati e delle transizioni relativo alla classe $Semaforo$

un'attesa di un secondo invia a se stesso l'evento *verde* con payload pari al payload dell'evento ricevuto decrementato di uno; se invece il payload dell'evento è pari a zero, la luce gialla si accende, il semaforo passa nello stato *Giallo* e invia a se stesso l'evento *giallo* avente come payload il valore della durata del giallo del semaforo stesso.

- Quando è nello stato *Giallo* e riceve l'evento *giallo*, se il payload dell'evento è positivo il semaforo resta nello stato *Giallo* (il tempo previsto per la durata del giallo non è ancora passato) e dopo un'attesa di un secondo invia a se stesso l'evento *giallo* con payload pari al payload dell'evento ricevuto decrementato di uno; se invece il payload dell'evento è pari a zero, la luce rossa si accende, il semaforo torna nello stato *Rosso* e invia l'evento *verde* al prossimo semaforo coinvolto nella simulazione.
- Un semaforo *Acceso* può essere spento in qualsiasi momento tramite l'invio dell'evento *spegni* che lo riporta nello stato *Spento*.

In Figura 2 è mostrato il diagramma degli stati e delle transizioni relativo alla classe $Semaforo$.

Una sessione di interazione con l'applicazione si svolge come segue:

- l'utente inserisce i dati della simulazione, specificando il codice che la caratterizza (la data viene generata dal sistema);
- iterativamente vengono definiti i semafori coinvolti nella simulazione; in particolare, per ogni semaforo da definire per la simulazione:
  - l'utente inserisce i dati del semaforo, specificando il nome e la durata del giallo e del verde;
  - il semaforo viene aggiunto alla simulazione;
  - si procede poi con l'eventuale semaforo successivo, sulla base della scelta dell'utente;
- si verifica poi l'eseguibilità della simulazione:
  - se la simulazione comprende almeno due semafori la simulazione può essere eseguita e si procede come specificato nel seguito;

  - altrimenti, viene mostrato un messaggio di errore e l'applicazione termina;

- viene inizializzato l'ambiente di simulazione e viene poi visualizzata la simulazione tramite opportuna interfaccia grafica che mostra il funzionamento dei semafori e consente all'utente di generare gli eventi per controllarne l'evoluzione; in particolare, l'utente può:
  - accendere tutti i semafori (con invio in broadcast dell'evento *accendi*)
  - avviare la simulazione (con invio dell'evento *verde* al primo semaforo)
  - spegnere tutti i semafori (con invio in broadcast dell'evento *spegni*)
- al termine della simulazione, si procede con l'eventuale simulazione successiva, sulla base della scelta dell'utente.

In Figura 3 è riportato il diagramma delle attività corrispondente.

La prova consiste nel completare o modificare il codice fornito insieme al testo, in modo da soddisfare i requisiti sopra riportati. Seguendo le indicazioni riportate nei commenti al codice[1], si chiede di intervenire sulle seguenti classi:

- `FinestraPrincipale` (package `app.gui`) (si vedano le considerazioni in fondo al documento di specifica)
- `ManagerComprende` (package `app.dominio`)
- `SemaforoFired` (package `app.dominio`)
- `AggiungiSemaforo` (package `app.attivita.atomiche`)
- `VerificaSimulazione` (package `app.attivita.atomiche`)
- `AttivitaPrincipale` (package `app.attivita.complesse`)

Tempo a disposizione: **3 ore.**

Gli elaborati non accettati dal compilatore saranno considerati insufficienti.

Per facilitare la comprensione del codice e lo svolgimento della prova, nel seguito sono riportati i documenti di specifica risultanti dalle fasi di analisi e di progetto.

[1] le porzioni di codice su cui intervenire sono identificate dal commento `/* DA COMPLETARE A CURA DELLO STUDENTE */`

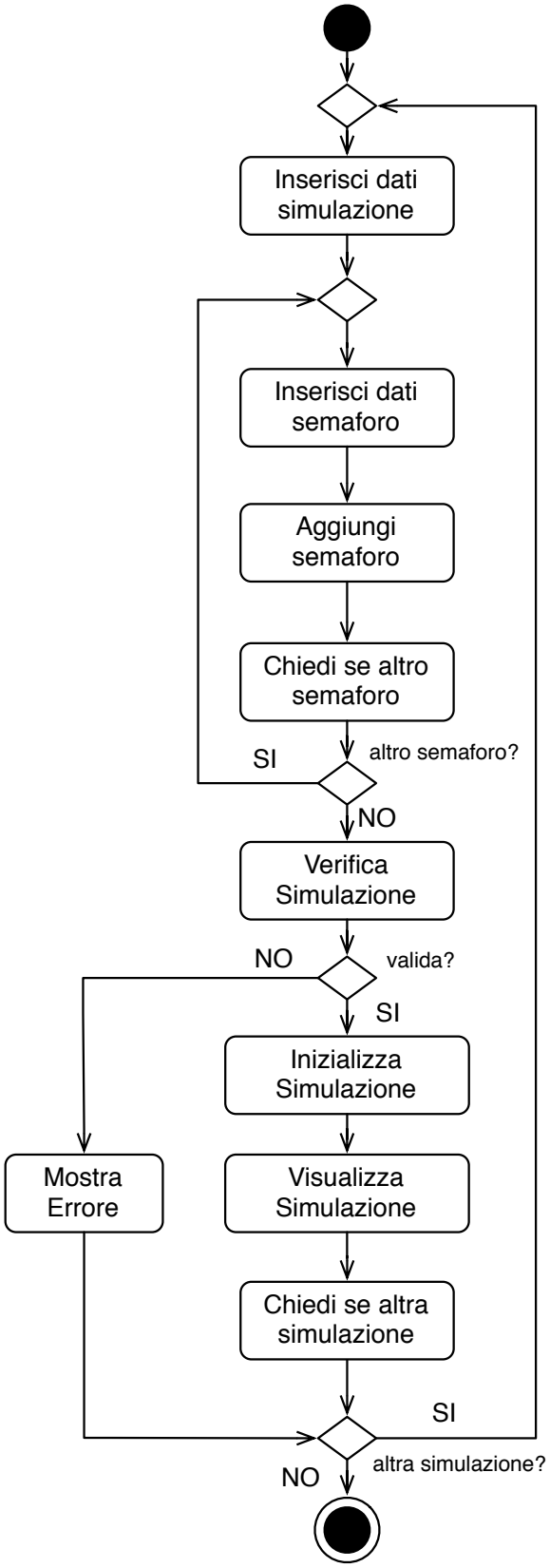

Figura 3: Diagramma delle attività

Analisi

Specifica del diagramma degli stati e delle transizioni della classe *Semaforo*

```
InizioSpecificaStatiClasse Semaforo

    Stato: {spento, acceso{rosso, giallo, verde}}
    Variabili di stato ausiliarie: –
    Stato iniziale:
       stato = spento

FineSpecifica


InizioSpecificaTransizioniClasse Semaforo

    Transizione: spento --> rosso
                    accendi
        Evento: accendi
        Condizione: --
        Azione: --


    Transizione: rosso --> verde
                    verde/verde{dest:this}(this.durataVerde)
        Evento: verde
        Condizione: --
        Azione:
            pre: --
            post: nuovoevento = verde{dest=this}(this.durataVerde)

    Transizione: verde --> verde
                    verde[payload > 0]/verde{dest:this}(payload-1)
        Evento: verde(payload:int)
        Condizione: [payload > 0]
        Azione:
            pre: --
            post: Attendi un secondo;
                  nuovoevento = verde{dest=this}(payload-1)

    Transizione: verde --> giallo
                    verde[payload = 0]/giallo{dest:this}(this.durataGiallo)
        Evento: verde(payload:int)
        Condizione: [payload = 0]
        Azione:
            pre: --
            post: nuovoevento = giallo{dest=this}(this.durataGiallo)
```

```
Transizione: giallo --> giallo
                giallo[payload > 0]/giallo{dest:this}(payload-1)
    Evento: giallo(payload:int)
    Condizione: [payload > 0]
    Azione:
        pre: --
        post: Attendi un secondo;
              nuovoevento = giallo{dest=this}(payload-1)

Transizione: giallo --> rosso
                giallo[payload = 0]/verde{dest:prossimo}
    Evento: giallo(payload:int)
    Condizione: [payload = 0]
    Azione:
        pre: --
        post: nuovoevento = verde{dest=prossimo}
```

NOTA: si veda il metodo ausiliario `prossimoSemaforo()` nella classe *SemaforoFired*

```
Transizione: acceso{rosso,giallo,verde} --> spento
                                      spegni
    Evento: spegni
    Condizione: --
    Azione: --

FineSpecifica
```

Attività di I/O

```
InizioSpecificaAttivitàAtomica InserisciDatiSimulazione
    InserisciDatiSimulazione():(Simulazione)
    pre: --
    post: Legge il codice di una simulazione fornito in input dall'utente.
          result è la simulazione creata a partire dal codice inserito, con data generata dal sistema.
FineSpecifica

InizioSpecificaAttivitàAtomica InserisciDatiSemaforo
    InserisciDatiSemaforo():(Semaforo)
    pre: --
    post: Legge nome, durata del giallo e del verde di un semaforo, forniti in input dall'utente.
          result è il semaforo creato a partire dai dati inseriti.
FineSpecifica

InizioSpecificaAttivitàAtomica ChiediSeAltroSemaforo
    ChiediSeAltroSemaforo():(Bool)
    pre: --
    post: Chiede all'utente se vuole aggiungere un altro semaforo.
          result è true in caso affermativo, false altrimenti.
FineSpecifica
```

`InizioSpecificaAttivitàAtomica` `VisualizzaSimulazione`
`VisualizzaSimulazione(s:Simulazione):()`
`pre: --`
`post:` Mostra la finestra di visualizzazione della simulazione del funzionamento dei semafori.
`FineSpecifica`

`InizioSpecificaAttivitàAtomica` `ChiediSeAltraSimulazione`
`ChiediSeAltraSimulazione():(Bool)`
`pre: --`
`post:` Chiede all'utente se vuole eseguire un'altra simulazione.
`result` è `true` in caso affermativo, `false` altrimenti.
`FineSpecifica`

`InizioSpecificaAttivitàAtomica` `MostraErrore`
`MostraErrore():()`
`pre: --`
`post:` Visualizza un messaggio di errore che informa l'utente che la simulazione non può essere eseguita.
`FineSpecifica`

Attività Atomiche

`InizioSpecificaAttivitàAtomica` `AggiungiSemaforo`
`AggiungiSemaforo(sim:Simulazione, sem:Semaforo):()`
`pre: --`
`post:` Crea un link *link* di tipo *Comprende* tale che $link.simulazione =$ `sim` e $link.semaforo =$ `sem`.
`FineSpecifica`

`InizioSpecificaAttivitàAtomica` `VerificaSimulazione`
`VerificaSimulazione(sim:Simulazione):(Bool)`
`pre: --`
`post:` Verifica se la simulazione `sim` comprende almeno due semafori;
`result` è `true` se `sim.quantiSemafori() > 1`, `false` altrimenti.
`FineSpecifica`

`InizioSpecificaAttivitàAtomica` `InizializzaSimulazione`
`InizializzaSimulazione(sim:Simulazione):()`
`pre: --`
`post:` Inizializza l'`Environment`, inserendovi (come Listener) tutti i semafori legati alla simulazione `sim` da un link di tipo *Comprende*; successivamente, attiva i Listener.
`FineSpecifica`

Attività Composte

InizioSpecificaAttività AttivitaPrincipale

AttivitaPrincipale():()

Variabili Processo:
simulazione: Simulazione -- simulazione corrente
semaforo: Semaforo -- semaforo corrente
altroSemaforo: Bool -- inserire altro semaforo?
simulazioneValida: Bool -- simulazione valida?
altraSimulazione: Bool -- eseguire altra simulazione?

Inizio Processo:

```
do {

  InserisciDatiSimulazione():(simulazione);

  do {
    InserisciDatiSemaforo():(semaforo);
    AggiungiSemaforo(simulazione, semaforo)();
    ChiediSeAltroSemaforo():(altroSemaforo);
  } while(altroSemaforo);

  VerificaSimulazione(simulazione):(simulazioneValida);

  if (simulazioneValida) {
    InizializzaSimulazione(simulazione):();
    VisualizzaSimulazione(simulazione):();
    ChiediSeAltraSimulazione():(altraSimulazione);
  }
  else {
    mostraErrore():();
  }

} while(altraSimulazione);
```

FineSpecifica

Progetto

Responsabilità sulle Associazioni

R: Requisiti; O: Specifica delle Operazioni/Attività; M: Vincoli di Molteplicità

Associazione	Classe	Ha Responsabilità
comprende	Simulazione Semaforo	SI (M,O,R) SI (M,O)

Strutture di Dati

Rappresentiamo le collezioni omogenee di oggetti mediante le classi `Set` ed `HashSet` del Collection Framework di Java. Inoltre, rappresentiamo le collezioni omogenee ordinate di oggetti mediante le classi `List` e `ArrayList` del Collection Framework di Java.

Tabelle di Gestione delle Proprietà delle Classi UML

Riassumiamo le scelte differenti da quelle di default mediante la tabella delle proprietà immutabili.

Classe UML	Proprietà Immutabile
Simulazione	codice data
Semaforo	nome

Altre Considerazioni

Non dobbiamo assumere una particolare sequenza di nascita degli oggetti.

Non esistono valori di default per qualche proprietà che siano validi per tutti gli oggetti.

La finestra principale dell'applicazione deve essere simile a quella in Figura 4.

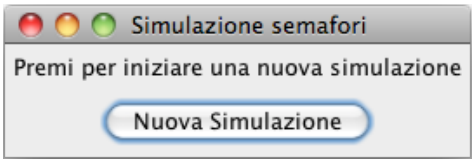

Figura 4: La finestra principale