Sapienza - Università di Roma
Facoltà di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

Esercitazioni di Progettazione del Software - A.A. 2009/2010
Prova al calcolatore del 16 luglio 2010

Requisiti

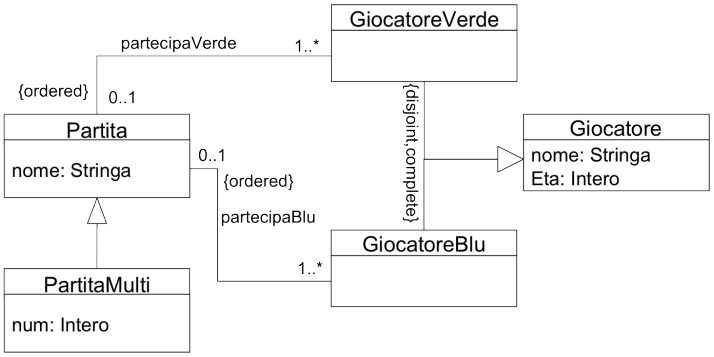
Si vuole realizzare un'applicazione per il *gioco del cerino*. In questo gioco, i partecipanti devono passarsi un cerino durante l'esecuzione di un brano musicale, per evitare di esserne in possesso quando la musica viene interrotta.

Ciascun giocatore è caratterizzato dal proprio nome (una stringa) e dall'età (un intero). I giocatori sono divisi in due squadre: verdi e blu. Una partita è giocata tra due insiemi, ordinati e non vuoti, di giocatori verdi e blu. Ogni giocatore può giocare in al più una partita. Ciascuna partita ha un nome (una stringa). In una variante del gioco, il *gioco dei cerini*, si gioca con $num \geq 2$ cerini. In Figura 1(a) è mostrato il diagramma delle classi corrispondente al dominio.

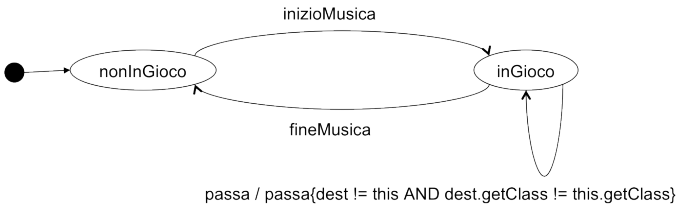
Un giocatore è inizialmente "non in gioco". Quando riceve l'evento "inizio musica" passa allo stato "in gioco". Ciascun giocatore in gioco, quando riceve il cerino (evento "passa"), lo passa ad un altro giocatore (nuovo evento "passa"). I giocatori devono passare il cerino ricevuto ad un qualsiasi giocatore della squadra avversaria (ad es., se è un giocatore verde, può aver ricevuto il cerino solo da un giocatore blu e può passarlo solamente ad un giocatore blu). Quando un giocatore riceve l'evento "fine musica" torna nello stato "non in gioco". I giocatori possono essere modificati solo nello stato "non in gioco". In Figura 1(b) è mostrato il diagramma degli stati e delle transizioni relativo alla classe *Giocatore*.

L'applicazione si svolge come segue:

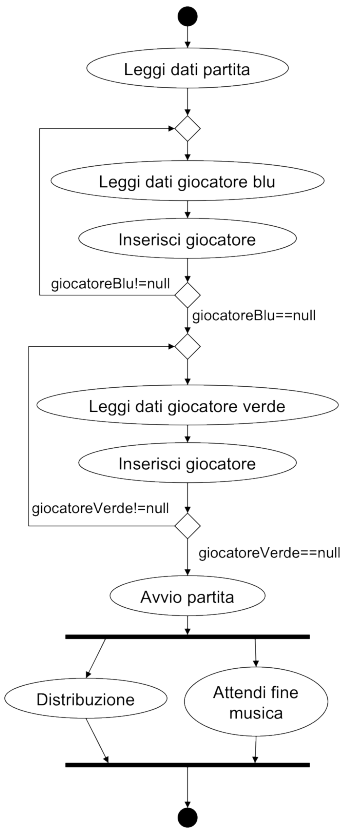
- viene creata un partita, in base ai dati –nome e numero di cerini– forniti in input dall'utente. Se il numero dei cerini è > 1 allora la partita è del *gioco dei cerini*, altrimenti è del *gioco del cerino*;
- iterativamente, i giocatori blu vengono creati ed iscritti alla partita, permettendo all'utente di inserirne i dati, di volta in volta, mediante opportuna attività;
- i giocatori verdi vengono creati ed iscritti alla partita con le stesse modalità dei giocatori blu;
- l'utente decreta l'inizio della partita, avviando la musica (evento "inizio musica" inviato in broadcast a tutti i giocatori);
- vengono avviate due attività contemporanee:
  - un'attività distribuisce tutti i cerini (presenti nelle partita) ad alcuni giocatori, scelti arbitrariamente, inviando loro l'evento "passa"; quindi, i giocatori si scambiano i cerini seguendo le regole mostrate sopra;
  - un'attività rimane in attesa che l'utente interrompa la musica, mediante opportuna interfaccia di I/O, dopodiché invia l'evento "fine musica" in broadcast a tutti i partecipanti.



(a) Diagramma UML delle classi



(b) Diagramma UML degli stati e delle transizioni della classe *Giocatore*



(c) Diagramma UML delle attività

In Figura 1(c) 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, si chiede di intervenire sulle seguenti classi:

- `FinestraPrincipale` (package `applicazione`): implementa una finestra contenente un pulsante per l'avvio dell'attività principale
- `Ascoltatore` (package `applicazione`): alla pressione del pulsante in FinestraPrincipale, deve avviare AttivitaPrincipale
- `AttivitaPrincipale` (package `attivita_composte`)
- `Partita` (package `partita`)
- `GiocatoreFired` (package `giocatore`)
- `ManagerPartecipaVerde` (package `partecipaVerde`)
- `InserisciGiocatore` (package `attivita_atomiche`)

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.

Analisi

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

```
InizioSpecificaStatiClasse Giocatore
     Stato: {inGioco, nonInGioco}
     Variabili di stato ausiliarie:  --
     Stato iniziale:
        stato = nonInGioco
FineSpecifica

InizioSpecificaTransizioniClasse Giocatore
     Transizione: nonInGioco -> inGioco
                     inizioMusica
         Evento: inizioMusica
         Condizione: --
         Azione:
             pre: evento.dest = this
             post: --

     Transizione: inGioco -> inGioco
                     passa / passa{dest != this AND dest.getClass != this.getClass}
         Evento: passa
         Condizione:--
         Azione:
            pre: evento.dest = this
            post: nuovoevento = passa{mitt=this, dest=d}, con d ≠ this tale che:
                 se this ∈ GiocatoreBlu allora d ∈ GiocatoreVerde  and
                 se this ∈ GiocatoreVerde allora d ∈ GiocatoreBlu
```

Transizione: inGioco -> nonInGioco
fineMusica
Evento: fineMusica
Condizione: --
Azione:
pre: evento.dest = this
post: --
FineSpecifica

Attività di I/O

InizioSpecificaAttivitàAtomica LeggiDatiPartita
LeggiDatiPartita ():(Partita)
pre: --
post: Legge il nome *nome* della partita ed il relativo numero *num* di cerini;
se $num = 1$ allora result è un oggetto Partita tale che `result.nome` = *nome*;
altrimenti, result è un oggetto $PartitaMulti$ tale che `result.nome` = $nome \wedge$ `result.num` = num.
FineSpecifica

InizioSpecificaAttivitàAtomica LeggiDatiGiocatoreBlu
LeggiDatiGiocatoreBlu ():(GiocatoreBlu)
pre: --
post: Legge il nome *nome* e l'età *eta* del giocatore blu corrente;
`result` e' un oggetto GiocatoreBlu tale che `result.nome` = $nome \wedge$ `result.eta` = eta.
FineSpecifica

InizioSpecificaAttivitàAtomica LeggiDatiGiocatoreVerde
LeggiDatiGiocatoreVerde ():(GiocatoreVerde)
pre: --
post: Legge il nome *nome* e l'età *eta* del giocatore verde corrente.
`result` è un oggetto $GiocatoreVerde$ tale che `result.nome` = $nome \wedge$ `result.eta` = eta.
FineSpecifica

InizioSpecificaAttivitàAtomica FermaMusica
FermaMusica ():()
pre: --
post: Attende che l'utente interrompa la musica, premendo un bottone.
FineSpecifica

Attività Atomiche

InizioSpecificaAttivitàAtomica InserisciGiocatore
InserisciGiocatore (Giocatore g, Partita p):()
pre: --
post: Inserisce un link `l` tra `g` e `p`;
`l` è di tipo $PartecipaVerde$ se `g` è un oggetto $GiocatoreVerde$;
altrimenti, `l` è di tipo $PartecipaBlu$.
FineSpecifica

InizioSpecificaAttivitàAtomica AvviaPartita
AvviaPartita (Partita p):()
pre: --
post: avvia la partita `p`, inviando l'evento `inizioMusica` in broadcast
FineSpecifica

InizioSpecificaAttivitàAtomica Distribuzione
Distribuzione (Partita p):()

pre: --
post: se p è *Partita*, invia un evento passa ad un giocatore, blu o verde, scelto casualmente tra quelli che partecipano alla partita;
se p è *PartitaMulti*, invia p.num eventi passa ad altrettanti giocatori, verdi o blu, scelti casualmente tra quelli che partecipano alla partita;
FineSpecifica

InizioSpecificaAttivitàAtomica SegnalaFineMusica
SegnalaFineMusica ():()
pre: --
post: Invia l'evento fineMusica a tutti i partecipanti;
FineSpecifica

Attività Composte

InizioSpecificaAttività AttivitaAttendiFineMusica

AttivitaAttendiFineMusica():()

Variabili Processo: –
Inizio Processo
FermaMusica():();
SegnalaFineMusica():();

FineSpecifica

InizioSpecificaAttività AttivitaPrincipale

AttivitaPrincipale():()

Variabili Processo:
partita: Partita -- partita corrente
giocatoreBlu: GiocatoreBlu -- giocatore blu corrente
giocatoreVerde: GiocatoreVerde -- giocatore verde corrente
altroGiocatore: Bool -- altro giocatore da aggiungere?

Inizio Processo

```
LeggiDatiPartita():(partita);
do{
  LeggiDatiGiocatoreBlu():(giocatoreBlu);
  altroGiocatore = (giocatoreBlu != null);
  if (altroGiocatore){;
    InserisciGiocatore(giocatoreBlu, partita);
  }
}while{altroGiocatore};
do{
  LeggiDatiGiocatoreVerde():(giocatoreVerde);
  altroGiocatore = (giocatoreVerde != null);
  if (altroGiocatore){;
    InserisciGiocatore(giocatoreVerde, partita);
  }
}while{altroGiocatore};
AvviaPartita(partita):();
fork{
  thread t1:{ Distribuzione(partita):();}
  thread t2:{ AttivitaAttendiFineMusica():();}
};
```

```
        join t1, t2;

FineSpecifica
```

Progetto

Responsabilità sulle Associazioni

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

Associazione	Classe	Ha Responsabilità
partecipaBlu	Partita	SÌ (O,M)
	GiocatoreBlu	SÌ (O,M)
partecipaVerde	Partita	SÌ (O,M)
	GiocatoreVerde	SÌ (O,M)

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 e la tabella delle assunzioni sulla nascita.

Classe UML	Proprietà Immutabile
Giocatore	nome
	eta
Partita	nome
PartitaMulti	num

	Proprietà	
Classe UML	**Nota alla nascita**	**Non nota alla nascita**
-	-	-

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.