SAPIENZA Università di Roma

A.A. 2008-2009

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Esercitazioni di Progettazione del Software
(Canale A-L)

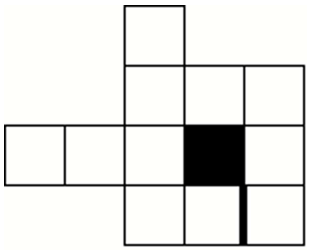
Realizzazione di Applicazioni Basate su Eventi

Fabio Patrizi

Requisiti

Nel videogioco *Gold Seekers*, un giocatore (umano) sfida il computer in una corsa all'oro.

I giocatori si muovono su diverse caselle quadrate. Di ciascuna casella, interessa conoscere tutte quelle ad essa adiacenti, lungo le direzioni nord, sud, est, ovest. Ciascuna casella può essere adiacente ad al più un'altra per direzione, e deve sempre averne almeno una adiacente (ovvero, tutte le caselle devono essere *connesse*). Una configurazione di esempio è mostrata in figura. La casella nera indica assenza di caselle, mentre il bordo spesso indica che le caselle interessate non sono adiacenti.



La presenza di un pozzo ed un sacchetto d'oro, o di due o più sacchetti d'oro nella stessa casella non è ammessa.

Gli elementi di gioco possono essere di tre tipi: sacchetti d'oro, pozzi d'acqua o cercatori d'oro. Ciascun sacchetto è caratterizzato dal proprio peso, ovvero una misura della quantità d'oro che contiene. In ogni istante, ciascun cercatore può aver raccolto nessuno o più sacchetti.

Per raccogliere un sacchetto, il cercatore deve posizionarsi sulla casella che lo ospita. Una volta raccolto, il sacchetto non è più contenuto nella cella.

Requisiti (cont.)

Ciascun cercatore è caratterizzato dal proprio nome (una stringa).

Un cercatore può essere negli stati: *disidratato*, *dissetato* o *non in gioco*. Inzialmente, è nello stato *non in gioco*. Quando il gioco inizia, ha una riserva d'acqua di 5 litri e passa allo stato *dissetato* . Quando la sua riserva d'acqua raggiunge il valore 0, il cercatore entra nello stato *disidratato*. Quando un cercatore è disidratato, va in cerca d'acqua, altrimenti va in cerca d'oro.

In ogni momento, il cercatore può ripristinare la propria riserva d'acqua occupando una casella con pozzo. In tal caso, la riserva raggiunge il valore 10 ed il cercatore torna nello stato dissetato. Ad ogni turno, il cercatore consuma 0,5 litri d'acqua. Quando è disidratato, il cercatore non può raccogliere i sacchetti d'oro, sebbene possa muoversi liberamente sulla mappa, occupando anche le celle contenenti i sacchetti d'oro (che non sono, però, raccolti).

Esistono due tipi di cercatori: autonomi e controllati.

I cercatori controllati vengono comandati dall'utente. Prima di effettuare ogni mossa, il cercatore rimane in attesa che il giocatore indichi la direzione (mediante opportuna interfaccia) verso cui muoversi.

I giocatori muovono uno alla volta, seguendo una politica basata su turni. Il raggiungimento di una casella contenente un pozzo o un sacchetto d'oro e l'azione di raccolta o ripristino dell'acqua sono da considerarsi effettuate in un unico turno.

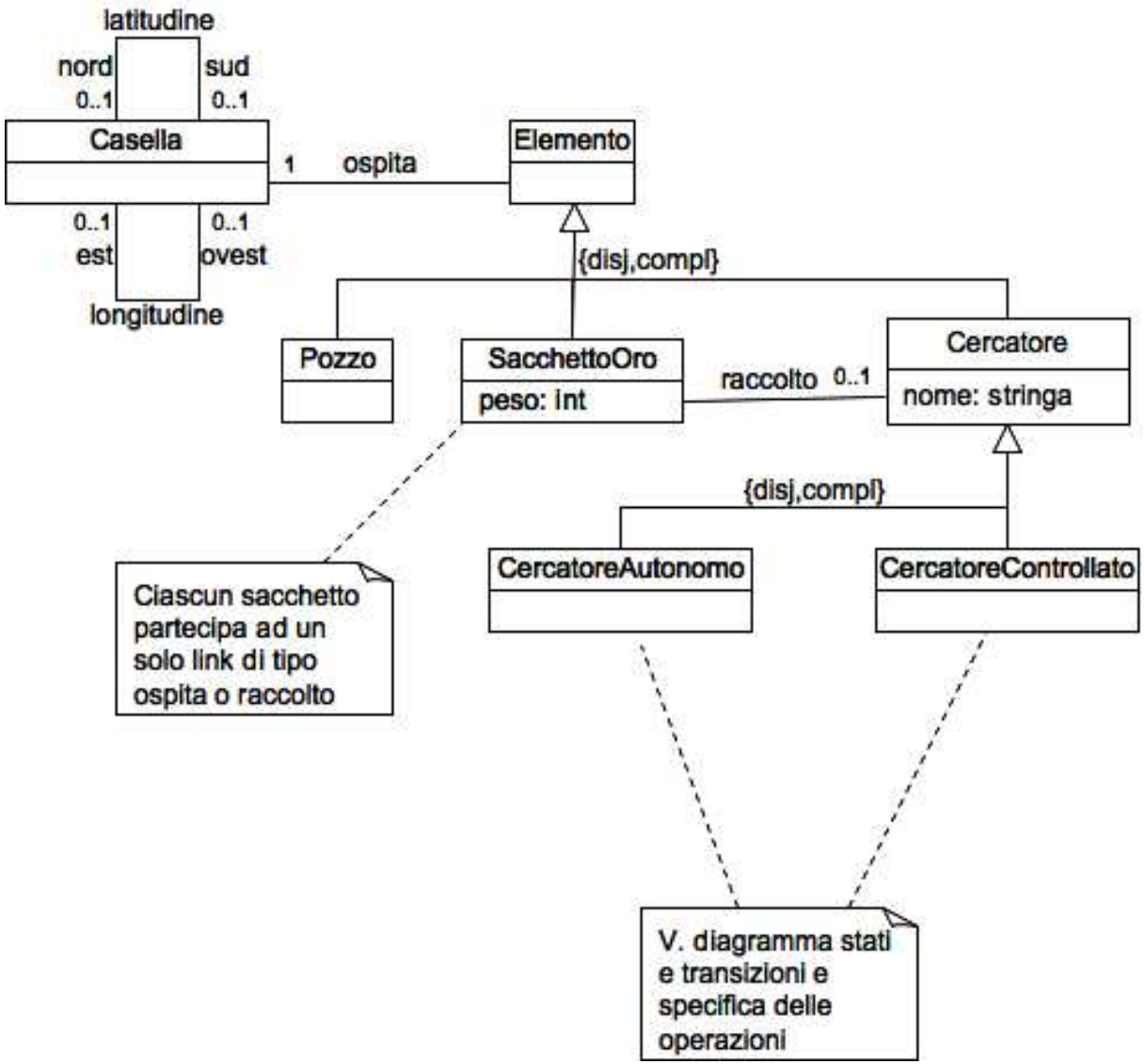
Requisiti (cont.)

Quando sono nello stato *dissetato*, i cercatori autonomi si muovono di un passo verso una delle caselle più vicine contenenti oro. Raggiunta una di esse, raccolgono l'oro. Quando sono nello stato *disidratato*, si muovono di un passo verso la casella più vicina contenente un pozzo e, una volta raggiunta, ripristinano la riserva d'acqua.

Quando non ci sono più sacchetti d'oro presenti, i cercatori tornano nello stato *non in gioco*. Si deve prevedere un'operazione che, dato un insieme di giocatori, restituisca quelli che hanno raccolto la maggior quantità d'oro.

Fase di Analisi

Diagramma UML delle Classi



Vincoli Aggiuntivi (Invarianti)

Ogni casella deve sempre averne almeno una adiacente

```
context Casella
        inv ( self.latitudine.nord->size() + self.latitudine.est->size() +
               self.latitudine.sud->size() + self.latitudine.ovest->size()= 1)
```

Nessuna casella è adiacente a se stessa:

```
context Casella
        inv ( self.latitudine.nord <> self and  self.latitudine.sud <> self and
                self.latitudine.est <> self and self.latitudine.ovest <> self )
```

(Inoltre, potremmo garantire la consistenza del campo di gioco imponendo ulteriori vincoli come, ad esempio, che le associazioni *longitudine* e *latitudine* siano delle *relazioni d'ordine stretto*).

Non possono coesistere un pozzo ed un sacchetto d'oro o due sacchetti d'oro nella stessa casella

```
context Casella
        inv not (self.ospita->exists(os | os.Elemento.oclIsKindOf(SacchettoOro)) and
                       (self.ospita->exists(os | os.Elemento.oclIsKindOf(Pozzo)) or
                        self.ospita->exists(os1 | os1<>os
                                                  os1.Elemento.oclIsKindOf(SacchettoOro))
                       )
```

Ciascun sacchetto è coinvolto in un solo link *ospita* o *raccolto*

```
context SacchettoOro
        inv (self.ospita->size()+self.raccolto->size()=1)
```

Specifica Operazioni Utente

Da completare

Diagramma UML degli Stati e delle Transizioni

Da completare

Specifica Stati Classe Cercatore

Da completare

Specifica Transizioni Classe CercatoreAutonomo

Da Completare

Specifica Transizioni Classe CercatoreControllato

Da completare

Fase di progetto

Responsabilità sulle associazioni

Da Completare

Strutture di dati

Abbiamo la necessità di rappresentare collezioni omogenee di oggetti, a causa:

- dei vincoli di molteplicità 0..* delle associazioni,
- delle variabili necessarie per vari algoritmi.

Per fare ciò, utilizzeremo le classi del collection framework di Java: `Set`, `HashSet`.

Corrispondenza fra tipi UML e Java

Riassumiamo le nostre scelte nella seguente tabella di corrispondenza dei tipi UML.

Tipo UML	Rappresentazione in Java
stringa	`String`
booleano	`boolean`
int	`int`
reale	`double`
Insieme	`HashSet`

Tabelle di gestione delle proprietà di classi UML

Riassumiamo le nostre scelte differenti da quelle di default mediante la *tabella delle proprietà immutabili* e la *tabella delle assunzioni sulla nascita*.

Classe UML	**Proprietà immutabile**
SacchettoOro	*peso*
Cercatore	*nome*

	Proprietà	
Classe UML	**nota alla nascita**	**non nota alla nascita**
Elemento	-	ospita

Altre considerazioni

Sequenza di nascita degli oggetti: Non dobbiamo assumere una particolare sequenza di nascita degli oggetti.

Valori alla nascita: Non sembra ragionevole assumere che per qualche proprietà esistano valori di default validi per tutti gli oggetti.

Rappresentazione degli stati in Java

Da completare

Rappresentazione degli eventi in Java

Rappresentiamo gli eventi tramite la classe `Evento`, (pre)definita durante il corso

Abbiamo bisogno di estendere `Evento` per modellare l'evento `InizioGioco`

Per gestire gli eventi, la classe `Cercatore` dovrà implementare l'interfaccia `Listener`, (pre)definita durante il corso

Lo scambio di eventi è gestito dalla classe `Environment` (pre)definita durante il corso

API delle classi Java progettate

Da completare

Fase di Realizzazione

Completare e correggere, secondo le istruzioni contenute nei commenti al codice, le seguenti classi:

- `Cercatore`,
- `CercatoreAutonomo`,
- `CercatoreControllato`,
- `TipoLinkRaccolto`,
- `ManagerRaccolto`,

- `ControlloreGioco` (v. seguito per approfondimenti).

Si richiede, inoltre, di realizzare l'operazione utente come metodo della classe `ControlloreGioco`.

Osservazioni

Per ragioni implementative, ogni volta che lo stato di gioco viene modificato (ad es., un cercatore viene spostato, un sacchetto d'oro raccolto, etc.), è necessario aggiornare la finestra di gioco, così da visualizzare gli ultimi cambiamenti.

Per fare ciò decidiamo di delegare alla classe `ControlloreGioco` il compito di muovere effettivamente i cercatori.

Inoltre, `ControlloreGioco` gestirà anche l'abilitazione dei comandi per gli spostamenti, nel caso del `CercatoreControllato`, tramite il metodo `abilitaMovimento()` che, quando invocato, abilita i pulsanti Nord, Sud, Est, Ovest presenti nell'interfaccia grafica.

Pertanto:

- Sia `CercatoreAutonomo` che `CercatoreControllato` dovranno avere un riferimento a `ControlloreGioco`, in modo da invocare i metodi opportuni al momento di spostarsi

- `CercatoreControllato` dovrà effettivamente implementare i metodi per spostare i cercatori.

In particolare, i metodi per lo spostamento sono: `muoviNord(Cercatore c)`, `muoviEst(Cercatore c)`, `muoviSud(Cercatore c)`, `muoviOvest(Cercatore c)`, i quali prendono in input il cercatore da spostare e ne cambiano la posizione di una casella nella direzione desiderata (se non esistono caselle nella direzione richiesta, non viene esguita nessuna operazione). Dopo aver fatto ciò, ciascun metodo invoca il metodo `aggiorna` della finestra principale del gioco (cf. codice sorgente).