Università di Roma "La Sapienza"
Facoltà di Ingegneria

Corso di
"PROGETTAZIONE DEL SOFTWARE I"
(Corso di Laurea in Ingegneria Informatica)
Proff. Giuseppe De Giacomo e Marco Cadoli
Canali A-L & M-Z
A.A. 2004-05

Compito d'esame del 15 aprile 2005

SOLUZIONE

Requisiti

L'applicazione da progettare riguarda la gestione degli articoli sottomessi ad una conferenza scientifica. Le persone, di cui interessa nome, cognome ed indirizzo di posta elettronica, possono essere autori o revisori, ma non entrambi. Gli articoli, di cui interessa il titolo e la dimensione in kiloByte del file, sono scritti da almeno un autore. Ad ogni articolo viene assegnato un revisore *senior* e almeno due revisori *junior*. Un revisore, di cui interessa la nazionalità, non può essere contemporaneamente *senior* e *junior*. Su ciascun articolo assegnato loro, i primi esprimono un giudizio positivo o negativo, mentre i secondi assegnano un voto compreso fra 0 e 9.

Un articolo, una volta sottomesso, si trova sotto esame e può essere candidato all'accettazione o al rifiuto. Nel primo caso può essere definitivamente accettato, oppure tornare sotto esame. Il secondo caso è analogo. Solo ad articoli sotto esame possono essere assegnati revisori.

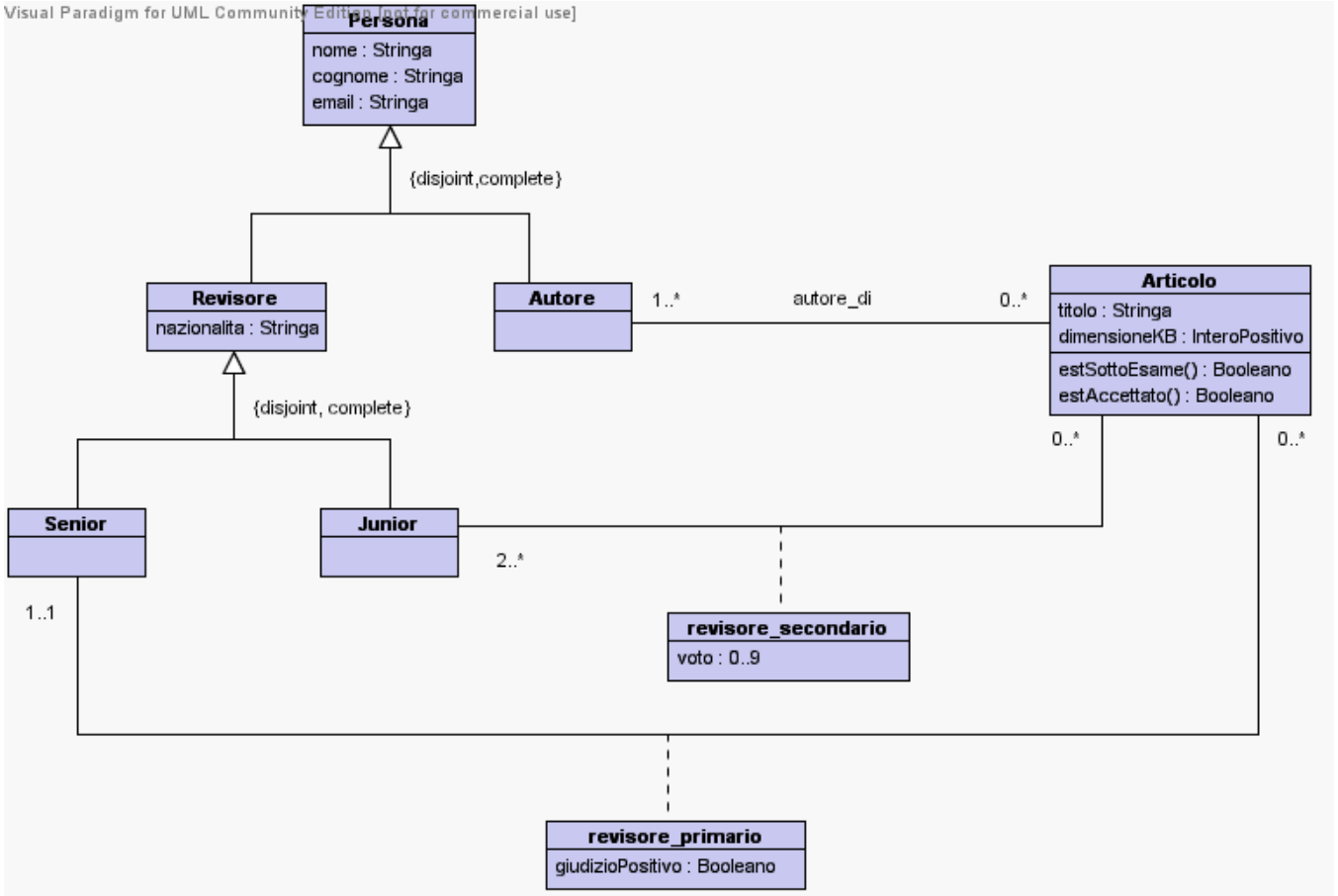
Requisiti (cont.)

Il comitato di indirizzo della conferenza è interessato, come cliente della nostra applicazione, ad effettuare i seguenti controlli:

- data una persona, sapere se è autore di almeno un articolo con giudizio negativo o con media dei voti inferiore a 4, che è stato accettato;
- dato un articolo, sapere se è stato assegnato ad almeno due revisori della stessa nazionalità.

Fase di analisi

Diagramma delle classi


Visual Paradigm for UML Community Edition [not for commercial use]
Persona
nome : Stringa
cognome : Stringa
email : Stringa
{disjoint,complete}
Revisore
nazionalita : Stringa
Autore
1..*
autore_di
0..*
Articolo
titolo : Stringa
dimensioneKB : InteroPositivo
estSottoEsame() : Booleano
estAccettato() : Booleano
{disjoint, complete}
0..*
0..*
Senior
Junior
2..*
1..1
revisore_secondario
voto : 0..9
revisore_primario
giudizioPositivo : Booleano

Commento sul diagramma delle classi

La necessità di disporre nella classe *Articolo* di metodi per conoscere se un articolo è sotto esame e se è stato accettato si evince, rispettivamente, dal vincolo che solo articoli sotto esame possono essere assegnati a revisori e dal fatto che la prima funzionalità dello use case richiede verificare se un articolo è stato accettato o meno.

Diagramma degli stati e delle transizioni classe Articolo

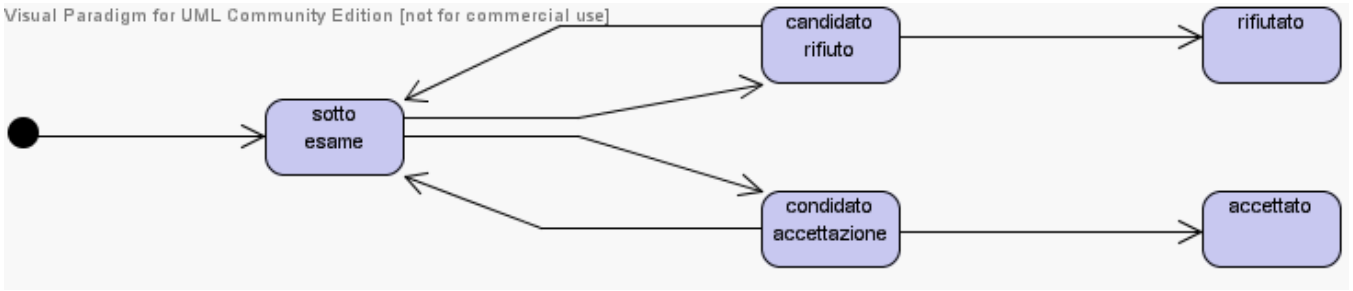

Visual Paradigm for UML Community Edition [not for commercial use]
sotto esame
candidato rifiuto
rifiutato
condidato accettazione
accettato

Diagramma degli use case


Visual Paradigm for UML Community Edition [not for commercial use]
controlli
effettua
comitato
conferenza

Specifica della classe Articolo

InizioSpecificaClasse Articolo

estSottoEsame *(): Booleano*

pre: nessuna

post: *result* è *true* se lo stato dell'articolo è "sotto esame", *false* altrimenti.

estAccettato *(): Booleano*

pre: *this.estSottoEsame() = false*

post: *result* è *true* se o stato dell'articolo è "accettato", *false* altrimenti.

FineSpecifica

Specifica dello use case

InizioSpecificaUseCase **Controlli**

autoreConRevisioneProblematica *(p: Persona): Booleano*

pre: nessuna

post: *result* è pari a *true* se *p* è un autore di un articolo con giudizio negativo o con media dei voti inferiore a 4, che è stato accettato; *false* altrimenti.

articoloRevisionatoStessaNazionalita *(a: Articolo): Booleano*

pre: nessuna

post: *result* è *true* se è stato assegnata ad almeno due revisori della stessa nazionalità; *false* altrimenti.

FineSpecifica

Fase di progetto

Algoritmi per le operazioni delle classi

Adottiamo i seguenti algoritmi:

- Per l'operazione **estSottoEsame** della classe *Articolo*:

```
se (stato corrente = ‘‘sotto esame’’) return true;
altrimenti return false;
```

- Per l'operazione **estAccettato** della classe *Articolo*:

```
se (stato corrente = ‘‘accettato’’) return true;
altrimenti return false;
```

Algoritmi per le operazioni dello use-case

Adottiamo i seguenti algoritmi:

- Per l'operazione **autoreConRevisioneProblematica**:

```
se (p non è istanza di Autore) return false;
altrimenti {
  per ogni link l di tipo autore_di in cui p è coinvolto {
     a = l.Articolo;
     se (a.estAccettato())
         &&
         ll.giudizioPositivo, dove ll è il link di tipo
                revisore_primario in cui a è coinvolto &&
         &&
         votomedio(a) < 4) return true;
  }
  return false
}

dove votomedio(a) è calcolato come segue:

Reale sum = 0;
```

```
Intero cont = 0;
per ogni link l di tipo revisore_secondaro in cui a è coinvolto {
  sum = sum + l.voto;
  cont++;
}
return sum/cont;
```

Algoritmi per le operazioni degli use case (cont.)

- Per l'operazione **articoloRevisionatoStessaNazionalita**:

```
sia l il link di tipo revisore_primario a cui partecipa a;
Insieme naz  = {l.Senior.nazionalita};
per ogni ll di tipo revisore_secondario a cui partecipa a {
   Stringa n = ll.Junior.nazionalita;
   se (n appartiene naz) return true;
   altrimenti naz = naz + {n};
}
return false;
```

Responsabilità sulle associazioni

La seguente tabella delle responsabilità si evince da:

1. i requisiti,

2. la specifica degli algoritmi per le operazioni di classe e use-case,

3. i vincoli di molteplicità nel diagramma delle classi.

Associazione	Classe	ha resp.
autore_di	*Autore*	SÌ[2]
	Articolo	SÌ[3]
revisore_primario	*Articolo*	SÌ[2,3]
	Senior	NO
revisore_secondario	*Articolo*	SÌ[2,3]
	Junior	NO

Strutture di dati

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

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

Per fare ciò, utilizzeremo la classe Java `InsiemeListaOmogeneo`.

API per le strutture di dati

```
// File insiemelista/InsiemeListaOmogeneo.java

package insiemelista;

public class InsiemeListaOmogeneo extends InsiemeLista {
  public InsiemeListaOmogeneo(Class cl)
  public InsiemeListaOmogeneo()
  public int size()
  public boolean isEmpty()
  public boolean contains(Object e)
  public boolean add(Object e)
  public boolean remove(Object e)
  public Iterator iterator()
  public boolean containsAll(Collection c)
  public Object[] toArray()
  public Object[] toArray(Object[] a)
  public boolean equals(Object o)
  public Object clone()
  public String toString()
}
```

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`
InteroPositivo	`int`
0..9	`int`
Insieme	`InsiemeListaOmogeneo`

Per tenere conto del fatto che, nei casi "InteroPositivo" e "0..9" il tipo Java è semanticamente più esteso del corrispondente tipo UML, prevediamo una verifica delle condizioni di ammissibilità sul lato server, perché è una soluzione di migliore qualità.

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**
Persona	*nome*
	cognome
Revisore	*nazionalita*
Articolo	*titolo*

	Proprietà	
Classe UML	**nota alla nascita**	**non nota alla nascita**

Altre considerazioni

Sequenza di nascita degli oggetti: Non abbiamo vincoli particolare se non quelli dettati dalle molteplicità: in particolare possiamo assumere che i revisori, (senior e junior) e gli autori siano già stati creati quando nascono gli oggetti articolo.

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

Per la classe UML *Articolo*, ci dobbiamo occupare della rappresentazione in Java del diagramma degli stati e delle transizioni.

Scegliamo di rappresentare gli stati mediante una variabile `int`, secondo la seguente tabella.

Stato	Rappresentazione in Java	
	tipo var.	`int`
	nome var.	`stato`
sotto esecuzione	valore	1
candidato accettazione	valore	2
candidaro rifiuto	valore	3
candidaro rifiuto	valore	4
accettato	valore	5
rifiutato	valore	6

API delle classi Java progettate

A titolo di esempio, viene fornita la API della classe `Persona`:

```
public abstract class Persona {
// COSTRUTTORE
    public Persona(String nome, String cognome, String email)
// GESTIONE ATTRIBUTI
    public String getNome()
    public String getCognome()
    public String getEmail()
    public void setEmail(String e)
// STAMPA
    public String toString()
}
```

Fase di realizzazione

Considerazioni

La realizzazione in questo caso è assolutamente standard e non richiede particolari accorgimenti.