Programmazione Orientata agli Oggetti

Classi Astratte

Contenuti

- Classi astratte
- Metodi astratti
- Classi astratte o interface?

Introduzione

- Abbiamo visto come l'estensione può essere uno strumento utile per il riuso del codice
- Le classi estese ereditano anche l'implementazione della classe base
  - La classe estesa può avere variabili di istanza e metodi aggiuntivi oltre a quelli ereditati dalla classe base
  - La classe estesa può ridefinire metodi della classe base
- La classe estesa è un sottotipo della classe base, quindi, in virtù del principio di sostituzione possiamo usare una classe estesa al posto della classe base

Introduzione

- In alcuni casi può essere utile definire classi base che sono pensate **solo** per essere estese
- Queste classi contengono una implementazione parziale
  - variabili di istanza
  - l'implementazione di alcuni metodi; di altri si definisce solo la segnatura
- I metodi incompleti di implementazione vengono definiti completamente nelle classi estese

Classi astratte

- Una classe astratta serve a questo scopo
  - Una classe astratta contiene una definizione parziale della implementazione
  - Una classe astratta non può essere istanziata, ma possono essere istanziate le classi che la estendono
  - Ovviamente vale la relazione sottotipo-supertipo, e quindi il principio di sostituzione: le istanze delle classi che estendono una classe astratta possono essere usate ogniqualvolta abbiamo un riferimento alla classe base

Esempio (sul caso di studio)

- Supponiamo di voler introdurre nel nostro gioco dei personaggi (es. mostri, maghi, ecc.)
- I personaggi sono nelle stanze del gioco (per semplicità accontentiamoci di un personaggio per stanza)

```
import …
public class Stanza {
     private String nome;
     private Map<String, Stanza> uscite;
     private Map<String, Attrezzo> nome2attrezzo;
     private Personaggio personaggio;

…
public void setPersonaggio(Personaggio personaggio) {
     this.personaggio = personaggio;
}

public Personaggio getPersonaggio() {
     return this.personaggio;
}
…
}
```

Esempio (sul caso di studio)

- I personaggi hanno un nome (una stringa) ed una descrizione (una stringa)
- I personaggi possono rispondere al saluto del giocatore (introduciamo il comando "**`saluta`**" per salutare il personaggio presente nella stanza)
- I personaggi possono agire (introduciamo il comando "**`interagisci`**" che invoca l'interazione con il personaggio presente nella stanza)

Esempio

- Possiamo avere numerose tipologie di personaggi
- Ad esempio potremmo avere:
  - *Maghi*: un mago possiede un attrezzo che ci può donare se interagiamo con lui
  - *Streghe*: se interagiamo con una strega questa ci trasferisce in una stanza tra quelle adiacenti. Poiché è permalosa, se non la abbiamo salutata, ci "trasferisce" nella stanza che contiene meno attrezzi; se la abbiamo salutata in quella che contiene più attrezzi
  - *Cani:* il cane morde! Ogni morso ci abbassa di una unità i nostri CFU

Esempio

- Tutte le tipologie di personaggi condividono una parte della implementazione (le variabili per memorizzare nome e descrizione, metodi accessori, costruttori, il codice che gestisce la risposta al saluto)
- Tutte le tipologie di personaggi hanno un metodo che rappresenta l'azione svolta dal personaggio
- Tuttavia non ha senso definire il comportamento di un personaggio generico
- Ogni tipologia di personaggio ha un comportamento specifico (il mago ci dona un attrezzo, la strega ci sposta in una stanza, il cane morde, ecc.) ovvero il codice che implementa l'azione è specifico: ogni tipologia di personaggio ha la propria implementazione.
- In altri termini, il personaggio è definibile solo in astratto
  - alcune sue proprietà possono essere concrete (descritte da una implementazione)
  - altre solo in astratto (non possiamo esplicitarne l'implementazione)

Esempio: la classe astratta `Personaggio`

```
package it.uniroma3.personaggi;
import …

public abstract class Personaggio {
   private String nome;
   private String presentazione;
   private boolean haSalutato;

   public Personaggio(String nome, String presentazione){
       this.nome = nome;
       this.presentazione = presentazione;
       this.haSalutato = false;
   }

   public String getNome() {
       return this.nome;
   }

   public boolean haSalutato() {
       return this.haSalutato;
   }
```

Esempio: la classe astratta `Personaggio` (cont.)

```
    public String saluta() {
        String risposta = "Ciao, io sono "+this.getNome()+".";
        if (!haSalutato)
                risposta += this.presentazione;
        else
                risposta += "Ci siamo gia' presentati.";
        this.haSalutato = true;
        return risposta;
    }

    abstract public String agisci(Partita partita);

    public String toString() {
        return this.getNome();
    }
}
```

La classe ComandoInteragisci

```
package it.diadia.comandi;

import ...
public class ComandoInteragisci implements Comando {
   private String messaggio;

   public void esegui(Partita partita) {
        Personaggio personaggio;
        personaggio = partita.getStanzaCorrente().getPersonaggio();
        this.messaggio = personaggio.agisci(partita);
        return;
   }

   public String getErrore() {
        return null;
   }

   public String getMessaggio() {
        return this.messaggio;
   }

   public void setParametro(String parametro) {
   }
}
```

Esempio: la classe (concreta) Mago

```
public class Mago extends Personaggio {
    private static final String MESSAGGIO_DONO = "Sei un vero simpaticone, " +
          "con una mia magica azione, troverai un nuovo oggetto " +
          "per il tuo bel borsone!";
    private static final String MESSAGGIO_SCUSE = "Mi spiace, ma non ho piu' nulla da
    darti ...";

    private Attrezzo attrezzo;

    public Mago(String nome, String presentazione, Attrezzo attrezzo) {
          super(nome, presentazione);
          this.attrezzo = attrezzo;
    }

    public String agisci(Partita partita) {
          String msg;
          if (attrezzo!=null) {
                    partita.getStanzaCorrente().addAttrezzo(attrezzo);
                    this.attrezzo = null;
                    msg = MESSAGGIO_DONO;
          }
          else {
                    msg = MESSAGGIO_SCUSE;
          }
          return msg;
    }
}
```

Classi e metodi astratti

- Le classi astratte hanno il modificatore **`abstract`** nella definizione
- Le classi astratte non possono essere istanziate

```
Personaggio personaggio = new Personaggio("","",); //NON COMPILA!
```

- Il codice di una classe astratta viene ereditato dalle classi la estendono
- I metodi astratti hanno **`abstract`** nella segnatura
- I metodi astratti non hanno corpo
- La presenza di almeno un metodo astratto rende una classe astratta
  - ma una classe può essere astratta anche se non ha nessun metodo astratto
- Le sottoclassi "concrete" devono completare l'implementazione (a meno che non siano a loro volta astratte), ovvero devono implementare gli eventuali metodi astratti

Classi astratte

- Servono a definire implementazioni parziali che verranno completate nelle classi concrete che le estendono
- Rispetto alle interface ?
  - Come le interface non possono essere istanziate
  - Diversamente dalle interface riportano una implementazione parziale e vengono estese
- Molto usate nei framework

Classi astratte vs. interface

- Classe astratta
  - Pro: Permette di riutilizzare l'implementazione
  - Contro: Limita la possibilità di estensione
- Interface
  - Pro: Nessun limite di estensione
  - Contro: non c'è codice riutilizzato
- Come scegliere?
  - Se c'è codice ("complesso") da riutilizzare: classe astratta, altrimenti interface

Testare le classi astratte

- Qual è l'obiettivo? Testare i metodi implementati
- Si definisce un "mock": una classe, definita solo al fine del testing, che estende la classe astratta, ed implementa i metodi astratti senza fare nulla (ritornando una costante se devo ritornare qualcosa diverso da void)
- Si testano solo i metodi non astratti
- NB: Questa soluzione può risultare insufficiente se un metodo concreto invoca un metodo astratto…

Testare le classi astratte

```
public class MockPersonaggio extends Personaggio {

   public MockPersonaggio(String nome, String presentazione) {
        super(nome, presentazione);
   }

   public String agisci(Partita partita) {
        return “done”;
   }
}
```

- Scriviamo la classe di test **`TestMockPersonaggio`** che testi i metodi (non astratti) ereditati da **`Personaggio`**

Esercizio

- Definire la classe **`Cane`**, che estende la classe **`Personaggio`**: quando interagiamo con un cane, questi morde, togliendoci CFU!
- Definire il comando *interagisci*
- Definire la classe **`Strega`** come riportato nella descrizione precedente.
  NB: è necessario modificare la classe **`Stanza`**, aggiungendo il metodo **`List<String> getDirezioni()`**, che restituisce l'elenco delle direzioni presenti nella stanza