Programmazione Orientata agli Oggetti

Polimorfismo:
Studio di caso

Riprendiamo lo Studio di Caso

- Nelle lezioni precedenti abbiamo già individuato (e rimosso) alcuni problemi nel codice dello studio di caso:
  - La responsabilità di gestire il labirinto deve essere assegnata ad una classe opportuna (la classe **`Labirinto`**)
  - Analogamente la responsabilità di gestire le informazioni relative al giocatore (borsa, CFU) devono essere assegnate ad una classe opportuna (la classe **`Giocatore`**): da fare per esercizio!
- La qualità del codice rimane bassa
  - La classe **`DiaDia`** implementa tutti(!) i possibili comandi del gioco
  - Analizziamo le conseguenze di quest'ultimo punto

Introdurre Nuovi Comandi

- Per introdurre un nuovo comando dobbiamo:
  - aggiungere un elemento nell'array **`ElencoComandi`**
  - aggiungere un metodo nella classe **`DiaDia`**
  - modificare il metodo **`processaIstruzione(String)`**
- Ma proviamo a ragionare in termini di responsabilità ed a "pensare in avanti"

Pensiamo in Avanti

- È ragionevole supporre che in futuro nel nostro gioco possano essere introdotti nuovi comandi
- Se per ogni comando introdotto dobbiamo modificare la classe **`DiaDia`** come abbiamo appena descritto, tale classe crescerà a dismisura

Una Soluzione (1)

- Il problema nasce dal fatto che la classe **`DiaDia`** conosce e realizza i dettagli di tutti i comandi
  - dovrebbe limitarsi a chiamare l'esecuzione di un comando, senza conoscerne i dettagli
- Le operazioni corrispondenti all'esecuzione di ogni comando dovrebbero essere codificate direttamente da un oggetto **`Comando`**
  - ma abbiamo tanti diversi comandi, ognuno con le sue peculiarità ...
- Per ovviare al problema un programmatore esperto ci suggerisce di sfruttare le potenzialità del polimorfismo, ristrutturando il codice come indicato di seguito

Una Soluzione (2)

- La classe **`Comando`** va trasformata in una interface, che rappresenti un generico comando
- L'interface **`Comando`** deve offrire il metodo

  `public void esegui(Partita partita)`
- Tutti i comandi del gioco saranno realizzati da oggetti istanze di classi che implementano l'interface ***`Comando`***:
  - l'implementazione del metodo **`esegui(Partita partita)`** codifica la semantica del comando specifico
- (Per ora) la classe **`DiaDia`** sulla base delle istruzioni lette da tastiera istanzia l'implementazione opportuna del comando
- Al comando istanziato chiederà quindi di eseguire il metodo **`esegui(Partita partita)`**, senza preoccuparsi di come avverrà l'esecuzione

La Classe `DiaDia`

```
private boolean processaIstruzione(String istruzione) {
    Comando comandoDaEseguire;
    FabbricaDiComandi factory = new FabbricaDiComandi()

    comandoDaEseguire = factory.costruisciComando(istruzione);
    comandoDaEseguire.esegui(this.partita);
    if (this.partita.vinta())
            System.out.println("Hai vinto!");
    if (!this.partita.giocatoreIsVivo())
            System.out.println("Hai esaurito i CFU...");
    return this.partita.isFinita();
}
```

- L'oggetto **`factory`** (istanza di **`FabbricaDiComandi`**) ha la responsabilità di creare un oggetto **`Comando`**. Non ci interessano le specificità di ogni singolo comando disponibile
  - vedremo in seguito (>>) come è fatta la classe **`FabbricaDiComandi`**
- Invochiamo semplicemente il metodo **`esegui()`** (che è polimorfo): in sostanza lasciamo al comando la responsabilità di eseguire il comando
- Spariscono dalla classe tutti i metodi che implementano i comandi.
  - ✔ Per disporre di un nuovo comando basta introdurre la sua classe, senza dover modificare la classe **`DiaDia`**

L'Interface *Comando*

```
public interface Comando {

   /**
      * esecuzione del comando
      */
      public void esegui(Partita partita);

}
```

La Classe `ComandoVai`

- Proviamo a creare una implementazione (la classe **`ComandoVai`**)
- La classe **`ComandoVai`** implementa il comando che permette di cambiare stanza
- Scriviamone il codice

Implementazione di *Comando*

```
public class ComandoVai implements Comando {
  private String direzione;

  public ComandoVai(String direzione) {
       this.direzione = direzione;
  }

  /**
     * esecuzione del comando
     */
  @Override
  public void esegui(Partita partita) {
     // qui il codice per cambiare stanza …
  }
}
```

La classe ComandoVai (2)

```
@Override
public void esegui(Partita partita) {
    Stanza stanzaCorrente = partita.getStanzaCorrente();
    Stanza prossimaStanza = null;
    if (direzione==null) {
        System.out.println("Dove vuoi andare?
                            Devi specificare una direzione");
        return;
    }
    prossimaStanza = stanzaCorrente.getStanzaAdiacente(this.direzione);
    if (prossimaStanza==null) {
        System.out.println("Direzione inesistente");
        return;
    }
    partita.setStanzaCorrente(prossimaStanza);
    System.out.println(partita.getStanzaCorrente().getNome());
    partita.getGiocatore().setCfu(partita.getGiocatore().getCfu()-1);
}
```

Osservazioni (1)

- Chi ha la responsabilità di creare gli oggetti **`Comando`**?
- Questa responsabilità è ragionevole sia affidata non ad un metodo della classe **`DiaDia`**, ma ad una classe dedicata
- Una classe che *fabbrica* comandi
  - **`FabbricaDiComandi`**
  - fabbrica un oggetto **`Comando`** a partire dall'istruzione digitata

- Rimangono alcuni problemi (per ora ci accontentiamo…)
  - In particolare:
    - accoppiamento forte con la gestione dell'I/O
    - il codice a fisarmonica non viene definitivamente eliminato, è stato solamente confinato dentro **`FabbricaDiComandi`**

Osservazioni (2)

- I comandi possono avere un parametro
- Come facciamo ad impostarne il valore?
  - attraverso il costruttore
    Es. **`ComandoVai(String direzione)`**
  - oppure, introducendo (nella interface) un metodo setter
    `setParametro(String parametro)`
- La seconda soluzione impone che *tutte* le classi che implementano ***`Comando`*** abbiano questo metodo
  - anche quelle che rappresentano comandi senza parametri (come «aiuto» o «fine»)
  - ✔ Non necessariamente un problema (il corpo del metodo sarà vuoto); sicuramente poco *elegante*
- Le due soluzioni sono equivalenti. Preferiamo comunque la seconda
  - (per motivi evidenti solo in seguito >>)

L'Interface *Comando* (rivista)

```
public interface Comando {

   /**
      * esecuzione del comando
      */
      public void esegui(Partita partita);

   /**
      * set parametro del comando
      */
      public void setParametro(String parametro);

}
```

La Classe ComandoVai (rivista)

```
public class ComandoVai implements Comando {
  private String direzione;
  @Override
  public void esegui(Partita partita) {
    Stanza stanzaCorrente = partita.getStanzaCorrente();
    Stanza prossimaStanza = null;
    if (this.direzione==null) {
        System.out.println("Dove vuoi andare?
                                Devi specificare una direzione");
        return;
    }
    prossimaStanza = stanzaCorrente.getStanzaAdiacente(this.direzione);
     if (prossimaStanza==null) {
        System.out.println("Direzione inesistente");
        return;
     }
     partita.setStanzaCorrente(prossimaStanza);
     System.out.println(partita.getStanzaCorrente().getNome());
     partita.getGiocatore().setCfu(partita.getGiocatore().getCfu()-1);
  }
  @Override
  public void setParametro(String parametro) {
     this.direzione = parametro;
  }
}
```

Creazione di Oggetti *Comando*

```
public class FabbricaDiComandi {

    public Comando costruisciComando(String istruzione) {
       Scanner scannerDiParole = new Scanner(istruzione);
       String nomeComando = null;
       String parametro = null;
       Comando comando = null;

       if (scannerDiParole.hasNext())
         nomeComando = scannerDiParole.next(); // prima parola: nome del comando
       if (scannerDiParole.hasNext())
         parametro = scannerDiParole.next();   // seconda parola: eventuale parametro

       if (nomeComando == null)
         comando = new ComandoNonValido();
       else if (nomeComando.equals("vai"))
         comando = new ComandoVai();
       else if (nomeComando.equals("prendi"))
         comando = new ComandoPrendi();
       else if (nomeComando.equals("posa"))
         comando = new ComandoPosa();
       else if (nomeComando.equals("aiuto"))
         comando = new ComandoAiuto();
       else if (nomeComando.equals("fine"))
         comando = new ComandoFine();
       else if (nomeComando.equals("guarda"))
         comando = new ComandoGuarda();
       else comando = new ComandoNonValido();
       comando.setParametro(parametro);
       return comando;
    }
}
```

Osservazioni

- Abbiamo migliorato significativamente la qualità del codice della classe **`DiaDia`**
  - risulta ora più coesa (con quali responsabilità?)
  - non è più accoppiata ai *dettagli* dei singoli comandi
  - abbiamo rimosso il codice a *fisarmonica*
    - ora confinato nella classe **`FabbricaDiComandi`** (anche se in una forma molto più semplice e pulita): non deteriora la qualità del codice di **`DiaDia`**
    - questa anomalia sarà completamente risolta in seguito (>>)
  - Predisponiamo il codice a questa prevedibile evoluzione:
    - creiamo una interface **`FabbricaDiComandi`**,
    - ridenominiamo la classe attuale (che implementa tale interface) **`FabbricaDiComandiFisarmonica`**
  - In pratica astraiamo dai dettagli implementativi della fabbrica, in attesa di una implementazione alternativa (>>)

Creazione di Oggetti *Comando*

```
public interface FabbricaDiComandi {
    public Comando costruisciComando(String istruzione);
}

public class FabbricaDiComandiFisarmonica implements FabbricaDiComandi {
  @Override
  public Comando costruisciComando(String istruzione) {
    Scanner scannerDiParole = new Scanner(istruzione);
    String nomeComando = null;
    String parametro = null;
    Comando comando = null;

    if (scannerDiParole.hasNext())
       nomeComando = scannerDiParole.next();// prima parola:   nome del comando
    if (scannerDiParole.hasNext())
       parametro = scannerDiParole.next();  // seconda parola: eventuale param.

    if (nomeComando == null)
       …
    return comando;
  }

}
```