

HOMework 4

Classi Astratte e Nidificate

Riflessione

Eccezioni e I/O

Tipi Enumerativi

Esercizio 0 (Prerequisiti)

- Chi non avesse concluso la scrittura dei test, lo faccia in questo homework, prima di fare le modifiche al codice in risposta agli esercizi che lo compongono

Esercizio 1 (AbstractComando)

- Scrivere una classe astratta **AbstractComando** per eliminare le implementazioni “vuote” dei metodi **setParametro()** dalle classi concrete che implementano l'interfaccia **Comando** (in particolare dalle classi che modellano comandi privi di parametri come ad es. **ComandoGuarda**, **ComandoAiuto**)
- Scrivere nuovi test per la nuova classe astratta e rifattorizzare i test della gerarchia **Comando** già sviluppati durante lo svolgimento dei precedenti homework

Esercizio 2 (**AbstractPersonaggio**)

- Scrivere la classe astratta **AbstractPersonaggio** e le classi concrete che la estendono
 - **Strega**
 - **Mago**
 - **Cane**
- Scrivere le classi **ComandoSaluta** e **ComandoInteragisci** che modellano i comandi attraverso i quali il giocatore può rispettivamente salutare e interagire con un personaggio
- Queste modifiche sono descritte anche nelle trasparenze reperibili nella pagina del materiale didattico del corso:

P00-classi-astratte-enum

Esercizio 3 (ComandoRegala)

- Modificare la classe astratta **AbstractPersonaggio** introducendo il metodo astratto:

```
public String riceviRegalo(Attrezzo attrezzo, Partita partita)
```

- Scrivere la classe **ComandoRegala**, attraverso la quale il giocatore può regalare un attrezzo al personaggio presente nella stanza
 - ✓ in una stanza può trovarsi un solo personaggio: affinché un attrezzo possa essere regalato il parametro del comando *regala* deve essere il nome di uno degli attrezzi presenti nella borsa

Esercizio 3 (cont.)

- Nelle classi **Cane**, **Strega**, **Mago** implementare il metodo astratto:

```
public String riceviRegalo(Attrezzo attrezzo)
```

- un cane riceve un regalo: se questo è il suo cibo preferito lo accetta, e butta a terra un attrezzo; altrimenti morde e toglie un CFU
 - una strega riceve un regalo, che trattiene scoppiando a ridere
 - un mago riceve un regalo, gli dimezza il peso e lo lascia cadere nella stanza
- La stringa restituita rappresenta il messaggio che deve essere prodotto dal comando quando eseguito (analogamente al comando *interagisci*)

Esercizio 4

(FabbricaComandiRiflessiva)

- Introdurre ed utilizzare la fabbrica di comandi basato sull'uso delle API per l'introspezione per la creazione degli oggetti **Comando**
- Ricontrollare tutto per assicurarsi che finalmente l'elenco dei comandi disponibili nel gioco sia effettivamente specificato una sola volta
- In tutto il codice:
 - ✓ A riprova, basta controllare quali modifiche sono necessarie per aggiungere un nuovo comando e quindi renderlo *perfettamente* funzionante ed integrato con il resto del gioco (ad es. deve figurare tra i comandi disponibili nell'elenco prodotto dal comando **aiuto**)

Esercizio 5 (**CaricatoreLabirinto**)

- Modificare la classe **Labirinto** affinché la specifica del labirinto venga letta da file testuale utilizzando la classe **CaricatoreLabirinto**
 - ✓ Una *bozza* è fornita nella pagina del materiale didattico
- Va modificata e rifattorizzata per due ottimi motivi:
 - dovrà, a sua volta, avvalersi di **LabirintoBuilder**
 - Contiene degli errori
- Scrivere dei test di unità per correggere gli errori di **CaricatoreLabirinto**
 - ✓ per favorire leggibilità, ed autocontenimento dei test, e per non vincolarli all'effettiva presenza di file, usare fixture specificate tramite stringhe direttamente nei test:
 - *suggerimento*: ricorrere a **StringReader**
 - ✓ Scrivere diversi test-case su fixture di complessità crescenti: labirinto «monolocale», «bilocale», ecc.

Esercizio 6

- Modificare la classe `CaricatoreLabirinto` affinché sia possibile caricare anche personaggi, stanze chiuse, stanze buie ecc. ecc.

Esercizio 7 (`diadia.properties`)

- Modificare l'applicazione affinché la specifica delle costanti non sia cablata nel codice ma sia *esternalizzata* in un opportuno file di properties `diadia.properties` da distribuire assieme al codice
- Ad es.
 - il numero di CFU iniziali
 - il peso max della borsa
- Esportare l'applicativo in formato `.jar` e verificarne il funzionamento in un ambiente diverso da quello di sviluppo nonostante la dipendenza verso risorse aggiuntive rispetto al codice (ad es. il file `diadia.properties`)
 - ✓ *Suggerimento:* non cablare nel codice il percorso fisico del file di properties, ma *solo* il suo nome logico

Esercizio 8 (Tipizzazione Lasca)

- Modificare l'applicazione affinché siano utilizzati i tipi enumerativi laddove ancora resistono dei concetti di primo ordine per il gioco che risultino tuttavia ancora troppo lascamente tipati
 - ✗ Ad esempio come stringhe...
 - ✓ Sono sempre le stesse 4; le chiamiamo per nome!

Esercizio 9 (Classi Nidificate)

- Trasformare `LabirintoBuilder` in una classe statica nidificata di `Labirinto`
 - Rendere il costruttore di `Labirinto` privato
 - Aggiungere quindi un factory method statico e pubblico:

```
public static LabirintoBuilder Labirinto.newBuilder()
```

- E' preferibile dichiarare la classe nidificata pubblica o privata? perché?
- Scrivere dei test di unità a supporto della verifica di correttezza del codice prima e dopo questi cambiamenti

Esercizio 10

(IOSimulator con JCF)

- Rimuovere ogni riferimento agli array anche nella classe **IOSimulator**
 - Valutare in alternativa l'utilizzo di **List** o **Map**
 - Se volessi ricordare per ogni riga letta quali messaggi ha prodotto?
 - Scrivere/ampliare i test per simulare *interi* partite e non solo *singoli* metodi
 - Iniziare con partite semplici con pochi comandi
 - Aumentare la complessità creando test che comprendono più comandi in fila
 - Se necessario creare ogni volta labirinti ad hoc tramite l'utilizzo di **Labirinto.newBuilder()**
- ✓ *Attenzione*: questi non sono affatto test di unità

Esercizio 11 (facoltativo)

- `IOConsole` deve essere istanziata una sola volta
- Per verificare questo requisito prima della consegna:
 - 1) Scaricare il file `ControlliPrimaDellaConsegna.jar` dal sito del corso posizionandolo nella stessa cartella contenente il file `.jar` del progetto come prodotto in precedenza: `diadia.jar`
 - ✓ (vedi es. 7 <<)
 - 2) Eseguire direttamente dall'interno della cartella il comando

```
java -cp ControlliPrimaDellaConsegna.jar:diadia.jar  
  
it.uniroma3.diadia.DiaDia
```
 - 3) Giocare!
 - ✓ Provando a “eseguire tutto il codice almeno una volta”
 - 4) La stampa `[Classe it.uniroma3.diadia.IOConsole - Numero istanziammenti:1]` indicherà quante volte viene istanziata la classe `IOConsole`

Esercizio 11 (continua)

- Il file `ControlliPrimaDellaConsegna.jar` contiene
 - L'interfaccia `IO`
 - Una versione modificata della classe `IOConsole` per contare e stampare il numero di suoi esemplari prodotti
- ✓ N.B. L'ordine degli argomenti dopo `-cp` è fondamentale:

```
java -cp ControlliPrimaDellaConsegna.jar:diadia.jar  
                                             it.uniroma3.diadia.DiaDia
```

- La classe `IOConsole` presente in `ControlliPrimaDellaConsegna.jar` viene caricata dalla JVM prima di quella presente nel file `diadia.jar` e per questo motivo la “sovrascrive”
 - La JVM si «rifiuta» di rileggere il `.class` di una classe già caricata, per ovvi motivi di sicurezza

Esercizio 11 (continua)

```
package it.uniroma3.diadia;
import java.util.Scanner;

public class IOConsole implements IO {

    private static int numeroIstanziamenti = 0;

    public IOConsole() {
        numeroIstanziamenti++;
        this.mostraMessaggio("[Classe " + getClass().getName() +
            " - Numero istanziamenti:" + numeroIstanziamenti + "]");
    }

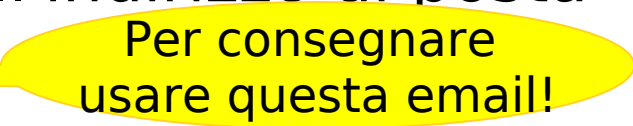
    public void mostraMessaggio(String msg) {
        System.out.println(msg);
    }

    public String leggiRiga() {
        Scanner scannerDiLinee = new Scanner(System.in);
        String riga = scannerDiLinee.nextLine();
        // scannerDiLinee.close();
        return riga;
    }
}
```


Esercizio 12 (facoltativo)

- Reinserire la riga cancellata nel precedente listato e contenente l'invocazione di `Scanner.close()`
- Utilizzare la forma sintattica *try-with-resource*
- Cambiare tutto il codice affinché non si presenti l'errore dovuto alla prematura chiusura di `System.in`
 - *Suggerimento*: creare lo scanner direttamente nel metodo che deve gestirne l'intero ciclo di utilizzo, dalla creazione, al rilascio, per tutta la durata di ogni partita:
 - ✓ Il `main()`
- Quindi ripetere l'esercizio 11 con la nuova versione del codice

TERMINI E MODALITA' DI CONSEGNA

- La soluzione deve essere inviata al docente entro le 21:00 del 14 giugno 2020 come segue:
 - Svolgere in gruppi di max 2 persone
 - Esportare (con la funzione File->Export di Eclipse) il progetto realizzato nel file **homework4.zip**
 - Inviare il file **homework4.zip** all'indirizzo di posta elettronica poo.roma3@gmail.com  Per consegnare usare questa email!
 - Nel corpo del messaggio riportare eventuali malfunzionamenti noti, ma non risolti
 - L'oggetto (subject) *DEVE* iniziare con la stringa **[2020-HOMEWORK4]** seguita dalle matricole
 - Ad es.: **[2020-HOMEWORK4] 512345 554321**