

HOMEWORK 2

Polimorfismo ed
Estensione di Classi

Esercizio 0

- ✓ Chi non avesse concluso la scrittura dei test per il precedente homework, lo faccia in questo homework, prima di fare le modifiche al codice, raggiungendo una situazione iniziale in cui numerosi test di unità hanno successo e confermano il corretto funzionamento a tempo di esecuzione del codice sviluppato sinora

Esercizio 1

- Implementare tutte le ristrutturazioni discusse nella dispensa sul polimorfismo
 - Scrivere quindi i test della classe `ComandoVai`
 - Implementare le classi corrispondenti a tutti i comandi previsti sino ad ora nel gioco
 - «aiuto», «fine», «prendi», «posa»; «guarda»
 - aggiungere la classe `ComandoGuarda`: «guarda» stampa le informazioni sulla stanza corrente e sullo stato della partita
 - aggiungere la classe `ComandoNonValido`
 - scrivere i test per le classi `ComandoPosa`, `ComandoPrendi`
- N.B. bisogna rifattorizzare *anche* i vecchi test mantenendo sempre i test ed il codice principale allineati

Esercizio 2

- Introdurre l'interfaccia *FabbricaDiComandi* e la classe `FabbricaDiComandiFisarmonica`
- Scrivere i test di unità su questa classe concreta ma limitarsi alla sola verifica del corretto riconoscimento dei comandi
 - *Suggerimento:* per scrivere questi test, aggiungere i metodi `getNome()` e `getParametro()` all'interfaccia *Comando*
 - ✓ N.B.: evitare invece di soffermarsi sull'istanziamento della corretta classe concreta associata a ciascun comando perché ancora non abbiamo gli strumenti opportuni

Esercizio 2 (continua)

- Il progetto sta crescendo: riorganizziamo meglio le classi introducendo anche il package
 - `it.uniroma3.diadia.comandi`
ove collocare i comandi e la fabbrica

Esercizio 3

- In HW1 (esercizio 5) abbiamo rifattorizzato il codice affinché l'uso diretto di `System.out` e `System.in` fosse prima evitato e poi “centralizzato” in `IOConsole`
- ✓ Assicurarsi di aver svolto l'esercizio 5 del precedente homework

Esercizio 3 (continua)

- Completiamo il processo di disaccoppiamento dall'I/O tramite l'introduzione di un apposita interfaccia denominata `IO` (presentata nella prossima slide>>) che astragga `IOConsole`:
 - Creare l'interfaccia e posizionarla nel package `it.uniroma3.diadia`
 - Cambiare `IOConsole` affinché la implementi
 - Cambiare tutto il codice affinché ogni riferimento ad `IOConsole` sia rimpiazzato da un (meno vincolante) riferimento tipato `IO`

Esercizio 3: Interfaccia IO

```
package it.uniroma3.diadia;  
  
public interface IO {  
    public void mostraMessaggio(String  
messaggio);  
    public String leggiRiga();  
}
```

- ✓ L'unica istanza (della sua implementazione) deve essere creata dal metodo `DiaDia.main()`

```
public class DiaDia {  
    ...  
    public static void main(String[] argc) {  
        IO io = new IOConsole();  
        DiaDia gioco = new DiaDia(io);  
        gioco.gioca();  
    }  
    ...}
```


Esercizio 3: Vantaggi

- ✓ In futuro sarà più facile cambiare l'implementazione dell'interfaccia `IO` creata nel metodo `main()` ed ipotizzare forme di interazione diverse da quelle sinora usate. Ad esempio:
 - una GUI (>>): dotando il gioco di una vera e propria parte grafica
 - un sistema automatico che simuli delle partite (>> vedi esercizio 9)

Esercizio 4

- Le recenti modifiche cambiano l'implementazione del gioco senza modificarne affatto il comportamento
- Subito dopo averle effettuate, verificare mediante i test sviluppati in questo e negli homework precedenti la correttezza del codice per confermare che non si siano introdotti nuovi errori
 - ✓ ovvero che non ci sia *regressione*
- In presenza di test che cominciano a fallire solo ora, mentre in precedenza avevano successo, utilizzare i fallimenti e la diagnostica per correggere gli errori
 - ✓ **Cominciando sempre dai test più semplici**
 - ✓ Se non si riesce subito a trovare i bug, aggiungere altri test-case sino a renderne palesi le cause

Esercizio 5

- Implementare ed introdurre nel gioco la «stanza magica», come descritto nelle dispense sull'estensione di classi
- Realizzare due distinte versioni di **Stanza**
 - **Stanza**
 - versione con campi privati: rispetta il principio dell'*information hiding* facendo utilizzare da parte delle classi estese solo la parte pubblica della classe base
 - **StanzaProtected**: campi protetti
- Corrispondentemente, realizzare anche due versioni della classe derivata:
 - **StanzaMagica**: estende **Stanza**
 - **StanzaMagicaProtected**: estende **StanzaProtected**

TDD (Facoltativo)

- ✓ N.B. È perfettamente lecito e consigliabile fare l'esercizio 8 anche prima degli esercizi 6&7

Esercizi 6-7

- Vogliamo introdurre nel gioco due ulteriori stanze particolari
 - La «stanza buia»: se nella stanza non è presente un attrezzo con un nome particolare (ad esempio "lanterna") il metodo `getDescrizione()` di una stanza buia ritorna la stringa *"qui c'è un buio pesto"*
 - La «stanza bloccata»: una delle direzioni della stanza non può essere seguita a meno che nella stanza non sia presente un oggetto con un nome particolare (ad esempio "passepartout")
- Creare le classi **StanzaBuia** e **StanzaBloccata** come estensioni della classe **Stanza**

Esercizio 6: Stanza Buia

- La classe `StanzaBuia` deve avere una variabile di istanza di tipo `String`: memorizza il nome dell'attrezzo che consente di avere la descrizione completa della stanza
- Il metodo `getDescrizione()` va sovrascritto affinché produca la descrizione usuale o la stringa "qui c'è buio pesto" a seconda che nella stanza ci sia o meno l'attrezzo richiesto per "vedere"
- Il nome dell'attrezzo necessario viene impostato attraverso il costruttore

Esercizio 7: Stanza Bloccata

- La classe **StanzaBloccata** deve avere due variabili di istanza di tipo **String** per memorizzare:
 - il nome della direzione bloccata
 - il nome dell'attrezzo che consente di sbloccare la direzione bloccata
- Il metodo **getStanzaAdiacente(String dir)** va riscritto (override)
 - se nella stanza non è presente l'attrezzo sbloccante, il metodo ritorna un riferimento alla stanza corrente
 - altrimenti ha l'usuale comportamento (ritorna la stanza corrispondente all'uscita specificata)

Esercizio 7: Stanza Bloccata (continua)

- Dentro la classe `StanzaBloccata` riscrivere anche il metodo `getDescrizione()` affinché produca una descrizione opportuna
- Anche in questo caso il nome dell'attrezzo sbloccante (ad es. 'piedediporco') e il nome della direzione bloccata vanno impostati attraverso il costruttore

Esercizio 8

- Scrivere i test per le classi **StanzaBuia** e **StanzaBloccata** implementate secondo le indicazioni espresse nelle trasparenze precedenti
- ✓ N.B. È perfettamente lecito e consigliabile fare questo esercizio anche prima degli esercizi 6&7
- Suggerimenti: cercare di mantenere i test «unitari»
 - ✓ non scomodare intere «Partite» solo per testare la logica di particolari stanze
 - ✓ i labirinti minimali per questi test sono «monocali» e/o «bilocali»?

Esercizio 9 (facoltativo)

- Scrivere una nuova classe `IOSimulator` (nel package `it.uniroma3.diadia`) che implementi l'interfaccia `IO`
- I metodi `mostraMessaggio()` e `leggiRiga()` dovranno rispettivamente scrivere / leggere i messaggi che sono scritti a video / letti da tastiera
 - Possono essere letti/conservati in array definiti come variabili di istanza
 - Il metodo `leggiRiga()` consentirà di “iniettare” le righe che desideriamo far figurare come istruzioni (di solito immesse dall'utente)
 - Il metodo `mostraMessaggio()` consentirà di conoscere i messaggi stampati durante la partita (a supporto di eventuali asserzioni)

Esercizio 9 (continua)

- Sviluppare dei test automatici che sappiano simulare intere partite senza bisogno di input da console
 - scrivono comandi e leggono risposte rimpiazzando interazione “manuale” con i servizi offerti da **IOSimulator**
 - Ideale per ampliare gli scenari del testing automatico sino a coprire intere partite
 - ✓ N.B. Non si tratta di unit-test
 - Ma per praticità li scriviamo con JUnit
 - Lo unit-testing va esattamente nella direzione opposta: si verifica il corretto funzionamento di piccoli frammenti di codice, NON di *inter*e partite!

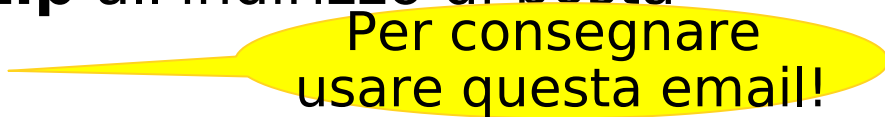
Esercizio 9 (test di accett.)

- ✓ Test basati sull'I/O
 - Chiamati *test di accettazione*
 - “Accoppiati” all'I/O (ovvero alle righe immesse ed alle stampe effettuate) e *non* ai dettagli del codice che legge e scrive
- Uno degli strumenti più efficaci per comunicare precisamente la logica dell'intero applicativo:
 - La sequenza di comandi impartiti e di stampe risultanti è comprensibile anche ad un “non-programmatore”
 - Ad es. al committente finale

Controlli Prima della Consegna

- Assicurarsi che l'esercizio 3 di questo homework e l'esercizio 5 del precedente siano stati svolti correttamente. Nello specifico verificare che:
 - ✓ Non ci siano chiamate di metodo tramite `System.in` o `System.out` al di fuori del corpo della classe `IOConsole`
 - ✓ `IO` e `IOConsole` si trovino nel package `it.uniroma3.diadia`
 - ✓ `IOConsole` implementi `IO`
 - ✓ `IOConsole` sia correttamente istanziato una sola volta ed iniettato dal metodo `main()` al costruttore della classe `DiaDia`
 - ✓ In tutto il codice principale i riferimenti siano tipati `IO` e non `IOConsole`
 - ✓ Se si è svolto anche l'esercizio 9 è normale avere riferimenti tipati `IOSimulator` nei corrispondenti test

TERMINI E MODALITA' DI CONSEGNA

- Cominciare a lavorare a questo homework solo dopo aver consegnato tutti gli homework precedenti
- La soluzione deve essere inviata al docente entro le 21:00 del 3 maggio 2020 come segue:
 - Svolgere in gruppi di max 2 persone
 - Esportare (con la funzione File->Export di Eclipse) il progetto realizzato nel file **homework2.zip**
 - Inviare il file **homework2.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, ed il numero dell'ultimo esercizio svolto con successo
 - L'oggetto (subject) *DEVE* iniziare con la stringa **[2020-HOMEWORK2]** seguita dalle matricole
 - Ad es.: **[2020-HOMEWORK2] 412345 454321**