Esercizi di Ingegneria del software

2023/2024

UNIVERSITÀ
DI PARMA

Nome: **Di Agostino Manuel**, **Colli Simone**
Insegnamento: Ingegneria del Software
Anno: 2023/2024

Indice

1 Design patterns

1.1 Composite pattern

Esercizio 1.1.1

Si vuole modellare la mappa di un videogioco *open-world* utilizzando una struttura dati ad albero; a seconda del livello di zoom dell'interfaccia è infatti possibile osservare una versione più dettagliata della mappa, corrispondente ad un particolare nodo dell'albero. Esistono quattro categorie di nodi:

- **foglia**: non ulteriormente espandibile, corrisponde ad un preciso punto fisso sulla mappa, importante per la logica del gioco;
- **distretto**: identifica zone più ampie, ad esempio un quartiere cittadino formato dalla composizione di nodi foglie;
- **regione**: aggrega più distretti contigui;
- **stato**: il livello più alto nella gerarchia considerata.

Ad ogni nodo sono inoltre sempre associati un *nome* e una *tipologia* conforme alla seguente interfaccia:

```
public enum NodeType {
    LEAF,
    DISTRICT,
    REGION,
    COUNTRY
}
```

Realizzare un class diagram UML che descriva la struttura della soluzione e fornire un'implementazione in Java che ne soddisfi i requisiti.

1.2 Iterator pattern

Esercizio 1.2.1

[**Richiede 1.1.1**]
È necessario implementare una serie di attraversamenti della struttura dati descritta nell'esercizio 1.1.1; in particolare si vogliono fornire degli *Iteratori* che, dato un *Nodo*, permettano di consultare gli elementi del sottoalbero da esso identificato secondo un determinato criterio. Tutti gli iteratori in questione devono essere conformi alla seguente interfaccia:

```
public interface Iterator {
    public Object first();
    public Object next();
    public Object currentItem();
    public boolean isDone();
}
```

Si richiede l'implementazione dei seguenti iteratori:

- visita in **profondità**
- visita in **ampiezza**
- visita dei soli nodi del **tipo specificato** in fase di costruzione dell'iteratore (`NodeType`)

- visita in **ampiezza** per livelli, a partire da un determinato nodo e **con una profondità massima**, utile al motore di rendering della mappa

1.3 Decorator pattern

Esercizio 1.3.1 (*Ispirato ad un esempio del libro GoF*)

I flussi sono un'astrazione fondamentale nella maggior parte delle strutture di I/O. Un flusso può fornire un'interfaccia per convertire oggetti in una sequenza di byte o caratteri. Questo ci consente di trascrivere un oggetto su un file o su una stringa in memoria per il recupero successivo. Un modo diretto per farlo è definire una classe astratta `Stream` con le sottoclassi `MemoryStream` e `FileStream`. Il corpo della classe `Stream` è il seguente:

```
public abstract class Stream {
    // private data section

    public void PutInt() {
        // impl
    }
    public void PutString() {
        // impl
    }
    public abstract void HandleBufferFull();
}
```

Le classi `MemoryStream` e `FileStream` ridefiniscono il metodo `HandleBufferFull()` per scrivere direttamente sulla memoria RAM e rispettivamente su file.

Supponiamo che la classe astratta `Stream` manipoli il buffer di caratteri del flusso tramite codifica `UTF-8`; si aggiunga la funzionalità di conversione di codifica del testo in `ASCII` standard a `7 bit` senza intaccare le classi/interfacce sopra citate, utilizzando dunque il *Decorator pattern*.

Fornire anche un class diagram UML della soluzione.

1.4 Command pattern

Esercizio 1.4.1

Si vuole realizzare la logica applicativa di un editor di testo che permetta di manipolare file testuali in codifica `ASCII` standard a `7 bit`. È previsto che l'interfaccia `EditableFile` esponga una serie di funzionalità:

- **creazione** del file, a partire da un nome scelto dall'utente
- **eliminazione** del file
- **lettura** completa del file
- **lettura** parziale del file, specificando punto di inizio e fine (numeri riga)
- **concatenamento** di testo alla fine del file
- **modifica** parziale del file, specificando punto di inizio e fine (numeri riga)
- **ridenominazione** del file

- **salvataggio** del file

Bisogna inoltre prevedere la possibilità di annullare fino a *256 modifiche* effettuate dall'utente.

Scrivere un'implementazione in Java del sistema descritto fornendone una descrizione tramite class diagram UML.

Esercizio 1.4.2

Si consideri la seguente interfaccia Java che descrive una struttura dati *queue*:

```
public interface Queue<T> {
    public void enqueue(T elem);
    public T dequeue();
    public void clear();
    public boolean isEmpty();
}
```

L'interfaccia in questione deve essere migliorata attraverso l'interfaccia **UndoQueue**, aggiungendo la funzionalità di undo per l'ultima modifica effettuata. Fornire un implementazione della suddetta usando la classe **SimpleQueue**, che offre un costruttore senza paramentri per la creazione di una coda vuota. Descrivere la struttura mediante un class diagram UML e fornirne un implementazione in Java.

1.5 Visitor pattern

Esercizio 1.5.1

[Richiede 1.1.1]
A partire dalla struttura dati implementata in 1.1.1 è richiesta l'aggiunta delle seguenti azioni:

- possibilità di creare elenchi dei nodi suddivisi per categoria (`LEAF`, `DISTRICT`, `REGION`, `COUNTRY`)
- stampa del nodo specializzata per tipo, nel formato `[NodeType] NodeName`

Non è possibile modificare il codice originario, se non aggiungendo il metodo `accept(Visitor v)` alle classi di tipo `Node`.

1.6 Abstract factory pattern

Esercizio 1.6.1

[Richiede l'esercizio 1.1.1]
Sulla struttura dati implementata in 1.1.1 è richiesto di delegare la creazione dei vari nodi (`LEAF`, `DISTRICT`, `REGION`, `COUNTRY`) ad un'abstract factory. Ipotizzando che i nodi implementino un interfaccia Component (o estendano una classe base con lo stesso nome), implementare un **ConcreteComponetFactory** che deve implementare la seguente interfaccia:

```
public interface ComponentAbstractFactory {
    public Component createLeaf();
    public Component createDistrict();
    public Component createRegion();
    public Component createCountry();
}
```

1.7 Proxy pattern

Esercizio 1.7.1

[Richiede l'esercizio 1.4.1]
A partire dalla logica applicativa dell'editor di testo in 1.4.1 è richiesto di ottimizzare l'utilizzo limitando gli accessi in lettura e scrittura sul file utilizzando un proxy.

1.8 Interpreter pattern

Esercizio 1.8.1

Si prenda in considerazione la seguente grammatica BNF:

$$\begin{aligned} Prop &\longrightarrow Atom \mid Formula \\ Formula &\longrightarrow (Formula) \mid \neg Fomula \mid \\ &\quad Formula \wedge Formula \mid Formula \vee Formula \mid \\ &\quad Formula \Rightarrow Formula \mid Formula \Leftrightarrow Formula \\ Atom &\longrightarrow \{\text{a string}\} \mid \bot \mid \top \end{aligned}$$

modellante il linguaggio $\mathcal{L}_{lp}$ della *logica proposizionale*.

Nell'ipotesi di disporre di un modulo $\mathcal{P}_{lp}$ in grado di costruire una struttura ad oggetti partendo dal testo di uno script (analizzatore lessicale, analizzatore sintattico e analizzatore semantico), rispondere alle seguenti domande eventualmente utilizzando i package `java.lang`, `java.util` e `java.io`:

1. Scrivere le interfacce Java del package `it.unipr.informatica.exercise.l8.model` necessarie a $\mathcal{P}_{lp}$ per descrivere uno script scritto in L8 sapendo già che verrà utilizzato il design pattern visitor con queste interfacce.

2. Scrivere una classe Java `it.unipr.informatica.exercise.lp.LPInterpreter` che utilizzi il design pattern visitor per interpretare uno script del linguaggio $\mathcal{L}_{lp}$ ricevuto mediante oggetti che implementano le interfacce realizzate al punto 1.

2 Tableaux

2.1 Tableaux proposizionali

Esercizio 2.1.1

Svolgere i seguenti tableaux:

5. $(p \Rightarrow q) \Rightarrow (\neg p \Rightarrow \neg q)$
6. $(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$
7. $p \Rightarrow \neg p$
8. $p \equiv \neg p$
9. $(p \equiv q) \equiv (\neg p \equiv \neg q)$
10. $\neg(p \wedge q) \equiv (\neg p \wedge \neg q)$
11. $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$
12. $(\neg p \vee \neg q) \equiv \neg(p \vee q)$
13. $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$
14. $(p \equiv (p \wedge q)) \equiv (q \equiv (p \vee q))$