Laboratorio di Progettazione di Sistemi Software

Materiale per il progetto

con esercizi 2

Valentina Presutti (A-L)
Riccardo Solmi (M-Z)

Indice degli argomenti

- **Aggiunta modulare di operazioni**
  - Visitors, type switch, Iterators
  - API generiche di modellazione

Possibili linee di sviluppo del progetto

- **Cosa devo modificare per aggiungere/modificare un costrutto al linguaggio (ad esempio l'istruzione *for*)?**
  - Posso rendere modulare l'aggiunta?
    - Introduzione al design pattern Template Manager
    - Introduzione ai design pattern Prototype e Prototype Manager
- **Cosa devo modificare per aggiungere/modificare una operazione sul linguaggio (ad esempio *typeCheck*)?**
  - Posso rendere modulare l'aggiunta?
    - Introduzione ai design pattern Visitor e Iterator
- **Posso rendere modulari entrambe le aggiunte di cui sopra?**

Aggiunta modulare di operazioni (behavior)

- **Le varianti di ciascuna operazione polimorfa ora sono sparse su tutti i costrutti; vogliamo renderle modulari.**
- **Esercizio: riscrivere l'operazione prettyPrint in una classe separata dai costrutti (in un package ….visitors)**
  - Soluzione basata su *if* annidati di *instanceof*
- **Osservare uso dei parametri e uso delle variabili di istanza nelle due soluzioni. A cosa servono? Qual è più efficiente?**
- **Osservare accesso alle proprietà dei costrutti nelle due soluzioni. Qual è più efficiente?**
- **Osservare meccanismi di condivisione di varianti polimorfe**
- **Posso rendere costante e trascurabile il costo del dispatching?**

Soluzione basata su double dispatching (Visitor)

- **Introduzione design pattern Visitor**
- **Implementazione:**
  - Aggiungere interfaccia *LanguageVisitor* con un metodo per ogni costrutto linguistico (E) secondo il seguente schema:
    - void visit(E) oppure void visitE(E)
  - Aggiungere in LanguageEntity metodo:
    - void accept(LanguageVisitor)
  - Implementare accept in tutti i costrutti.
    - Non basta in uno astratto ereditato da tutti?
- **Scrivere *PrettyPrintVisitor* ispirandosi a *prettyPrint()*.**
  - Osservare circolarità nelle dipendenze tra interfacce e implementazioni
  - LanguageVisitor trade-off *visit* con tipi concreti o astratti

Soluzione basata su type switch

- **Introduzione design pattern Enumeration**
- **Definire una enumerazione tipata dei costrutti che compongono il linguaggio**
  - Implementazione basata su classi Enum e EnumValue
  - Aggiungere i seguenti metodi in LanguageEntity:
    - EnumValue getType()
    - int getOrdinal()
  - Devo implementarli in tutti i costrutti?
- **Usare enumerazione per sostituire gli *if* annidati di *instanceof* con uno *switch* e dei *case* sui valori della enumerazione.**

Operazioni sparse su modello vs modulari

- **Posso ancora avere diverse implementazioni compatibili usando i Visitors? E con il *type switch*?**
- **Mi conviene sostituire tutte le operazioni del modello con altrettante operazioni modulari?**
- **Possono convivere le diverse soluzioni viste?**
  - Quali metodi mi conviene definire modulari e quali lasciare sparsi sul modello?
- **Lo switch mi permette di definire un comportamento di default posso farlo anche con i visitors?**
- **Esercizio: scrivere PrettyPrintVisitor con indentazione**
- **Esercizio: scrivere InterpreterVisitor con binding corretto**

API specifiche per manipolare il modello

- **Il modello deve essere navigabile/modificabile anche dall'esterno**
  - Per supportare factory estendibili
  - Per supportare operazioni modulari
- **Aggiungere a tutti i costrutti del modello i getter e i setter pubblici per tutte le proprietà**
  - BlockStatement è una sequenza di istruzioni. Che operazioni definisco per manipolarlo?
- **Osservare che le API specifiche non fanno parte delle interfacce. Chi può usarle?**
  - ((WhileStatement) lf.createWhile(…, …)).setExp(…);
  - ((WhileStatement) whileStm.clone()).setExp(…);
- **Posso aggiungere tutti i metodi di manipolazione all'interfaccia LanguageEntity per rendere più usabile il modello?**
  - Il design pattern Composite suggerisce di aggiungere solo i metodi per manipolare le collezioni

API generiche per manipolare il modello

Manipolazione by index

- **Le API per manipolare i composite, in particolare *size*, *get* e *set*, possono essere implementate da tutte le entità (costrutti)**
- **Conseguenze:**
  - Posso chiedere ad un costrutto quante proprietà strutturali ha
  - Posso leggere/scrivere tutte le proprietà di un costrutto senza conoscere il suo tipo concreto

Manipolazione by name

- **Introdurre una enumerazione delle proprietà strutturali: *LanguagePropertiesEnum* e una coppia di metodi *get/set* in *LanguageEntity* che le usano**
  - Suggerimento: introdurre anche una *indexOf* per mapparle sulle API by index

Visitors con API generiche

- **Posso applicare le API generiche per scrivere dei visitor più corti e più riusabili**
  - Scrivere visitor che attraversa tutto il modello top down e che funzioni per qualsiasi modello
  - Scrivere visitor che mostra le variabili usate
- **Posso definire dei visitor generici per navigare il modello con diverse strategie (top down, bottom up, …)**
- **Quando uso una strategia di attraversamento definita con un visitor il controllo ce l'ha il visitor (push style API)**
  - Il visitor fa tutto l'attraversamento, io ridefinisco il suo comportamento nei punti in cui voglio intervenire
  - Posso fare in modo di avere io il controllo dell'attraversamento? (pull style API)

Soluzione basata su iteratore

- **Introduzione design pattern Iterator**
- **Implementazione:**
  - Usare l'interfaccia standard di Java *Iterator* (senza *remove*) per implementare:
    - un *EntityIterator* per singole entità (costrutti)
    - un *ModelIterator* per eseguire una iterazione top down di tutto il modello
      - Implementarlo per composizione di *EntityIterator*s
      - Servirsi di iteratori aggiuntivi (*SingletonIterator*) per evitare la logica condizionale

Operazioni modulari: push style vs pull style

- **Scrivere un iteratore che mostra le variabili usate e confrontare con la soluzione basata su visitor**
- **Osservare la granularità della soluzione: classe, blocco**
- **Osservare le possibilità offerte dal controllo.**
  - Posso interrompere l'operazione?
  - Posso sospendere l'operazione e riprenderla in seguito?
- **Come scelgo la variante polimorfa da eseguire?**