

Laboratorio di Progettazione di Sistemi Software

Materiale per il progetto con esercizi

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

Indice degli argomenti

- **Progetto labp2001 e refactoring in labss_il_model**
- **Astrazione operazioni binarie e literal**
- **Aggiunta costrutti e operazioni**
- **Meccanismo di notifica e meccanismo di duplicazione**
- **Supporto composite**
- **Interfacce + classi astratte e adattatori**
- **Supporto undo/redo**
- **Predicati e configurazione azioni da menu contestuale e DnD**
- **View e Controller**
- **EditorKit**

© 2002-2004 Riccardo Solmi

2

Introduzione

- **Questo documento contiene la descrizione del processo di sviluppo del progetto del corso di labss 2003/04**
- **Le scelte progettuali e le alternative sono presentate usando il catalogo di design pattern visto a lezione**
- **Le trasformazioni che dal progetto iniziale (primo esercizio) portano a sviluppare tutto il materiale per il progetto finale (progetto da svolgere) sono descritte usando il catalogo di refactoring presentato a lezione.**
- **Le domande e gli esercizi proposti vengono iniziati a lezione e sono da completare a casa.**

© 2002-2004 Riccardo Solmi

3

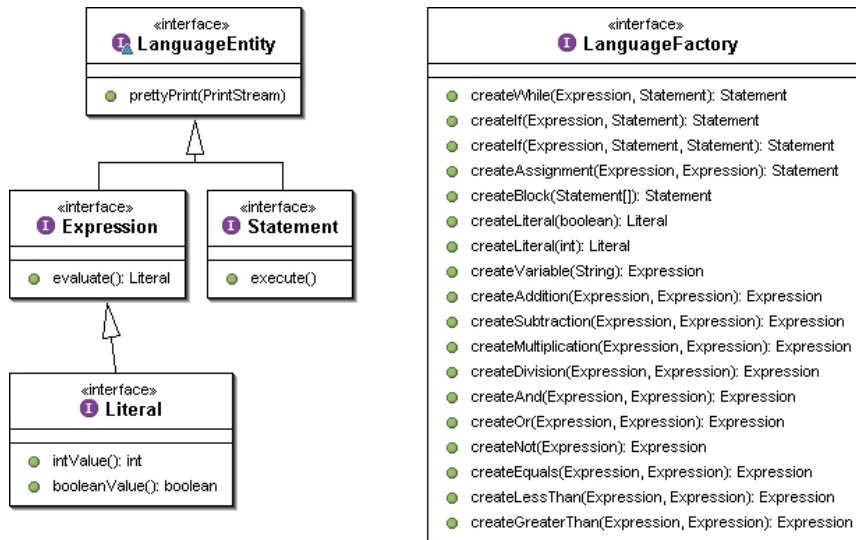
Progetto labp2001

- **Esercizio: implementare il progetto labp2001**
- **Importate il progetto vuoto dato (comprendente libreria di specifiche)**
- **Usare il supporto di Eclipse per definire:**
 - le classi concrete, i metodi ereditati, i costruttori
- **Ad esempio si può partire dal main di Test:**
 - scommentare l'istanziamento della factory;
 - procedere guidati dalle lampadine sugli errori per creare le classi concrete;
 - usare il menu contestuale *source* per introdurre implementazioni di metodi ereditati e i costruttori usando i campi;
 - per implementare i metodi servirsi del menu contestuale che si apre quando si digita "oggetto punto" (esempio: exp1.)

© 2002-2004 Riccardo Solmi

4

Diagramma delle classi delle specifiche



© 2002-2004 Riccardo Solmi

5

Uso dei diagrammi UML per comprendere esercizio

- **In cosa consiste un programma scritto con il progetto dato?**
 - Studiare l'esempio di calcolo del fattoriale in Test
 - Usare un diagramma degli oggetti per rappresentarlo
- **Come faccio a seguire il funzionamento dei metodi polimorfi `execute` e `prettyPrint`?**
 - Usare diagramma delle sequenze applicato al metodo `execute` sul blocco di istruzioni che calcola il fattoriale

© 2002-2004 Riccardo Solmi

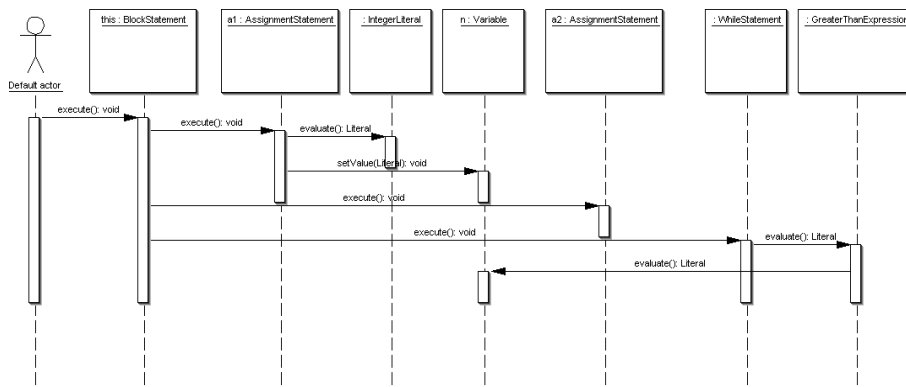
6

Diagramma degli oggetti del programma fattoriale

© 2002-2004 Riccardo Solmi

7

Diagramma delle sequenze di execute su fattoriale



© 2002-2004 Riccardo Solmi

8

Osservazioni e domande

- **Notare che seguendo le lampadine di Eclipse si riescono a creare tutte le classi che servono per compilare il progetto.**
- **Poi, usando i suggerimenti quando scrivete "oggetto punto" si viene indirizzati bene anche nella scrittura dei metodi `execute/evaluate` e `prettyPrint`.**
- **Osservare che posso implementare come prima classe, ad esempio, `WhileStatement` compreso il metodo `execute` che usa `evaluate` su una espressione.**
 - Come fa Java a permettermi di compilare la classe `WhileStatement` se non ho ancora implementato nessuna espressione (e quindi nessuna `evaluate`)?

© 2002-2004 Riccardo Solmi

9

Refactoring del progetto in `labss_il_model`

- **Refactoring: rinominare progetto e spostarlo in una cartella con il nuovo nome: `"labss_il_model"`**
- **Refactoring: importare la libreria di interfacce (copia/incolla package dal progetto specifiche al nuovo)**
- **Refactoring: rinominare il package delle specifiche in:**
 - `unibo.cs.labss.editor.il.model`
- **Refactoring: rinominare il package dell'implementazione in:**
 - `unibo.cs.labss.editor.il.model.impl`

© 2002-2004 Riccardo Solmi

10

Osservazioni e domande

- **Notare che tutti i costruttori nella soluzione di riferimento hanno i parametri.**
- **Avrei potuto usare dei costruttori senza parametri e aggiungere dei metodi setter?**
 - Cosa cambia nell'implementazione
 - Cosa cambia a livello di design?

Astrazione operazioni binarie e literal

- **Osservazione: il diagramma delle classi è molto piatto**
- **Osservare la forte somiglianza tra le implementazioni dei due tipi literal e tra le implementazioni di tutte le operazioni binarie (+, -, *, /, and, or, not, <, >, =).**
- **Quante classi concrete devo/posso fare?**
 - Una per ogni costrutto linguistico (if, while, +, ...)
 - Oppure una per tutte le operazioni binarie e una per i due literal (ad esempio: MieExpression e MieLiteral)
- **PS. L'istruzione "if" è disponibile in due varianti: con e senza else mi conviene fare una classe o due?**

Esercizi su soluzione con classe unica

- **Come faccio a dare ai metodi evaluate e prettyPrint un comportamento polimorfo?**
- **Cosa mi conviene passare in più al costruttore: un intero e/o una stringa?**
- **E' una soluzione orientata agli oggetti?**
 - Osservazione: il metodo della factory sa esattamente cosa vuole costruire (ad esempio una addizione)
 - Ogni volta che eseguo evaluate (o prettyPrint) il metodo si chiede come si deve comportare (+, -, *, ...)
 - Cosa succede se aggiungo una operazione binaria?
 - Il parametro che uso per specificare l'operazione è tipato?

© 2002-2004 Riccardo Solmi

13

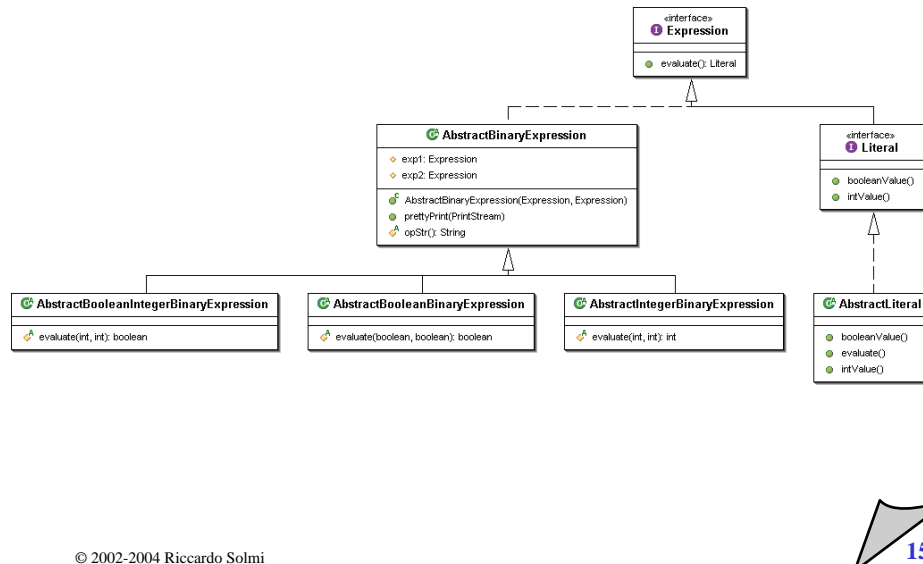
Soluzione che introduce classi astratte

- **Mantengo una classe per ogni costrutto linguistico ma raccolgo in una classe astratta le parti comuni alle implementazioni**
- **Refactoring: introduzione classi astratte AbstractBinaryExpression e AbstractLiteral**
- **Refactoring: pull up campi (exp1 e exp2) e costruttore**
- **Osservazione: i metodi evaluate e prettyPrint pur non essendo uguali sono molto simili**
- **Posso astrarre il comportamento dei metodi?**
 - Introduzione e applicazione del design pattern TemplateMethod

© 2002-2004 Riccardo Solmi

14

Diagramma classi espressioni astratte



15

Tipi astratti vs tipi concreti

- **Notare che in tutte le classi concrete i tipi dei parametri e del risultato dei metodi, e i tipi dei campi sono tutti astratti (interfacce).**
- **Posso usare tipi concreti nei campi?**
 - Che conseguenze ho a livello di design?
- **Posso usare tipi concreti nei parametri e nei risultati?**
 - Che conseguenze ho a livello di design?
 - Posso avere diverse implementazioni del progetto compatibili tra loro?
- **Principio: program to an interface, not an implementation**
 - Posso astrarmi ovunque dalle classi concrete?

© 2002-2004 Riccardo Solmi

16

Istanziamento delle classi concrete

- **Posso istanziare solo classi concrete.**
 - Cosa vuole dire `new Expression() {...}`?
 - istanzia una interfaccia?
 - Posso nascondere al cliente le operazioni di istanziazione?
- **La factory nel progetto serve per controllare l'istanziazione delle classi concrete.**
 - Posso istanziare direttamente le classi concrete del Test?
 - Viceversa, posso obbligare i clienti ad usare la factory?
- **Refactoring: estrai metodo `fact()` da test**
- **Introduzione ai design pattern creazionali**
 - Abstract Factory, FactoryMethod

© 2002-2004 Riccardo Solmi

17

Controllo sul numero di istanze

- **Di quante istanze di Factory posso avere bisogno?**
 - Posso imporre ai clienti l'uso di una sola istanza?
- **Come si riconosce una classe che posso istanziare una volta per tutte?**
 - Posso istanziare solo due oggetti per le costanti booleane?
- **Introduzione ed uso design pattern Singleton**
- **Un Singleton si istanzia in modo diverso da una classe normale (con un metodo o campo statico)**
 - Posso rendere singleton una classe senza che i miei clienti se ne accorgano?

© 2002-2004 Riccardo Solmi

18

Osservazioni sul controllo della visibilità

- **I modificatori di visibilità (`public`, `protected`, `package`, `private`) sono applicabili ai campi, ai metodi, ai costruttori e alle classi (concrete, astratte, interfacce).**
- **Usando i modificatori di visibilità posso imporre le mie scelte di design. Esempi:**
 - Nascondo i costruttori delle classi gestite da factory
 - Uso `private` per singleton e `package` per factory
 - Proteggo le operazioni che fanno parte di un template
 - Rendo privati i campi delle classi.
- **L'uso del modificatore “`final`” mi aiuta ulteriormente ad imporre le mie scelte (esempi Template Method, costanti)**

Struttura a composite

- **Osservare la struttura gerarchica usata per rappresentare il modello di un sorgente scritto con IL.**
 - Come sono scritti i metodi?
 - Design pattern Composite
- **Cosa devo modificare per aggiungere/modificare un costruito al linguaggio (ad esempio l'istruzione `for`)?**
- **Cosa devo modificare per aggiungere/modificare una operazione sul linguaggio (ad esempio `typeCheck`)?**
- **Posso rendere modulari le aggiunte di cui sopra?**
 - Introduzione ed uso dei design patterns Prototype e Visitor

Supporto Factory estendibili

- **Aggiungere interfaccia *Factory* con un metodo *create* che istanzia un oggetto:**
 - Object create()
- **Aggiungere interfaccia *LanguageFactoryRegistry* con i metodi:**
 - Object create(String type)
 - void put(String type, Factory factoryMethod)
 - void remove(String type)
- **L'implementazione può usare una *Map* per mappare i nomi dei tipi sui metodi factory da usare per istanziarli.**
- **Osservare che *create* mi restituisce un *Object* e non mi permette di passare parametri al costruttore**

© 2002-2004 Riccardo Solmi

21

Supporto ai Visitor

- **Aggiungere interfaccia *LanguageVisitor* con un metodo per ogni costrutto linguistico (xxx) secondo il seguente schema:**
 - void visit(XXX) oppure void visitXXX(XXX)
- **Aggiungere in *LanguageEntity* metodo:**
 - void accept(LanguageVisitor)
- **Scrivere *PrettyPrinterVisitor* ispirandosi a *prettyPrinter()*.**
- ***LanguageVisitor* trade-off visit con tipi concreti o cast**
- **Osservare circolarità nelle dipendenze tra interfacce e implementazioni**

© 2002-2004 Riccardo Solmi

22

Metodi sparsi su modello vs Visitors

- **Mi conviene sostituire tutte le operazioni del modello con altrettanti Visitors?**
- **Posso ancora avere diverse implementazioni compatibili usando i Visitors?**
- **Possono convivere le due soluzioni?**
 - Quali metodi mi conviene definire come visitors e quali lasciare sparsi sul Composite?

Indentazione e parentesi

- **Esercizio: migliorare la *PrettyPrintVisitor* in modo da produrre codice indentato in stile Java**
- **Osservazione: se stampo un programma formato dal prodotto di una somma ed un intero ottengo: $1 + 2 * 3$ anziché $(1+2) * 3$.**
- **Esercizio: aggiungere il supporto alla precedenza degli operatori**
 - Aggiungere nell'interfaccia *Expression*:
`void hasPrecedence(Expression op)`
 - Modificare *PrettyPrintVisitor* in modo da racchiudere tra parentesi le espressioni con precedenza minore del nodo padre
 - Suggerimento: aggiungere ad ogni tipo di espressione qualcosa di confrontabile che rappresenti la sua precedenza

Meccanismo di notifica

- **Serve per aggiungere un osservatore al modello in modo che venga informato ogni volta che il modello viene modificato**
 - Design pattern Observer
- **Osservazione: devo controllare tutti i punti del codice in cui cambio le associazioni che definiscono la struttura di una istanza del modello**
- **Refactoring: incapsulamento campi**
 - Rendo privati i campi
 - aggiungo dei metodi getter/setter per accedervi
- **+ Observable nel model per generare eventi**
- **- 1 per modello vs 1 per entità**

© 2002-2004 Riccardo Solmi

25

Insufficienza Abstract Factory del linguaggio

La Abstract Factory costruisce istanze di singoli costrutti

- Definisce un metodo per ogni costrutto
- I metodi in genere hanno dei parametri
- **Per questi ed altri motivi non mi basta una abstract factory se voglio fare un editor**
 - Esempio Drag and Drop con copia di un pezzo di codice
- **Introduzione design pattern Prototype**
- **Ad uso interno continua ad essere utile anche la factory**
 - Nella costruzione degli esempi e dei prototipi, ...

© 2002-2004 Riccardo Solmi

26

Meccanismo di duplicazione (clone)

- **Due meccanismi di istanziazione: creazione e clonazione**
 - Linguaggi class based vs linguaggi object based
 - Anche un linguaggio class based può avere la clone
- **La *new* crea una istanza a partire da una “ricetta”: la classe.**
 - Esempio: Expression var = **new** Variable(“i”);
 - Posso passare parametri al costruttore
- **La clone crea una istanza facendo una copia di un oggetto già esistente.**
 - Esempio var.clone();
 - L’oggetto è già configurato bene, eventualmente lo modifico con delle *set* successive

© 2002-2004 Riccardo Solmi

27

La clone di Java

- **clone() è un metodo definito in Object**
 - Copia la memoria dove risiede l’oggetto.
 - Per poterlo usare bisogna implementare l’interfaccia *Cloneable* (che non definisce nessun metodo)
 - E’ *protected*, per chiamarlo da fuori bisogna ridefinirlo come pubblico
 - Può provocare l’eccezione *CloneNotSupportedException*
- **Shallow vs deep clone**
 - Un oggetto può contenere riferimenti ad altri oggetti
 - Si dice profonda (deep) una clone che copia anche gli oggetti
 - La clone di Java è shallow, la si può ridefinire deep a piacere

© 2002-2004 Riccardo Solmi

28

Clonazione di un modello

- **Per clonare un modello ho bisogno di una clone profonda**
 - Posso rendere pubblico e usabile il metodo clone in una classe astratta ed estenderla da tutte le entità del modello
 - Ogni tipo concreto deve far proseguire la copia in profondità in tutti i suoi campi (di tipo riferimento)
 - In alternativa posso definire un visitor che si occupa della copia in profondità

Scelta design pattern creazionali

- **Un sistema software in genere fa uso di molti pattern creazionali, alcuni solo ad uso interno altri vengono esposti nelle API e servono ai clienti del sistema**
- **Voi dovete scrivere un editor, ma a partire da un framework che vi do.**
 - La LanguageFactory è ad uso interno e potete definirla come vi pare (o anche non definirla affatto)
 - Gli esempi devono essere scritti in modo che ogni gruppo possa definire gli esempi che vuole e che vengano istanziati solo quando richiesti
 - Frammenti del modello (singoli costrutti, blocchi di codice) devono essere copiabili e completi (senza puntatori null)

Factory estendibile di esempi

```
public interface IExamplesFactory {  
    public IEntity createEmptyModel();  
    public IEntity create(String type);  
    public void put(String type, ExampleFactory factoryMethod);  
    public void remove(String type);  
  
    public Set keySet();  
    public static interface ExampleFactory {  
        public IEntity create();  
    }  
}
```

© 2002-2004 Riccardo Solmi

31

Conseguenze

- **Posso accedere a tutti i vostri esempi senza concordare con voi quanti sono e come si chiamano**
- **Gli esempi vengono costruiti solo se richiesti**

© 2002-2004 Riccardo Solmi

32

Manager di prototipi

```
public interface IPrototypeManager {  
    public IEntity clone(String type);  
    public void put(String type, IEntity prototype);  
    public void remove(String type);  
    public Set keySet();  
  
    public boolean containsKey(String type);  
    public boolean isInstance(String type, IEntity entity);  
}
```

Conseguenze

- **Costruisco costrutti ma posso definire anche prototipi più complessi**
 - Esempio, vedi assistenza alla generazione di sorgenti di Eclipse (creazione costruttori, metodi di accesso, ...)
- **Posso aggiungere nuovi prototipi (anche a runtime)**
 - Posso introdurre nuovi costrutti aggiungendoli direttamente tra i prototipi (senza classi del modello)
 - L'utente dell'editor può chiedere che un blocco di codice venga aggiunto ai prototipi, in modo da usarlo velocemente tutte le volte che gli serve

Esercizio

- **Per costruire gli oggetti che compongono il controller, ho bisogno di un metodo che dato un elemento del modello mi costruisca un controller del tipo giusto per “controllarlo”.**
- **Assumiamo di avere una corrispondenza uno a uno tra i tipi del modello e quelli del controller**
- **L’interfaccia richiesta è riportata sotto; come faccio ad implementarla?**
 - Posso fare di meglio rispetto una serie di **if instanceof ... ?**

```
public interface IPartFactory {  
    public AbstractPart createPart(IEntity modelEntity);  
}
```

© 2002-2004 Riccardo Solmi

35

Supporto editing

- **Accesso ad informazioni comuni a tutto il modello**
 - Meccanismo di notifica, radice, identificatore del modello
- **Supporto proprietà con nome e proprietà enumerabili**
 - Le prime richiedono una coppia di metodi *get/setProprietà*
 - Le seconde (dette composite) richiedono una interfaccia simile a una collezione
 - Conviene definire un’unica interfaccia o due?
 - Introduzione al design pattern Composite
- **Metodo *replaceChild* per sostituire una proprietà qualsiasi**
 - Funziona indipendentemente dal tipo di proprietà
- **Elementi foglia (place holders) e radice (contenitore)**

© 2002-2004 Riccardo Solmi

36

Interfaccia IEntity

```
public interface IEntity {  
    public String getModelId();  
    public IModel getModel();  
    public void setModel(IModel model);  
  
    public boolean isComposite();  
    public int size();  
    public int indexOf(IEntity child);  
    public List getChildren();  
    public void addChild(int index, IEntity child);  
    public void removeChild(IEntity child, IEntity emptyReplacement);  
    public void replaceChild(IEntity oldChild, IEntity newChild);  
}
```

© 2002-2004 Riccardo Solmi

37

Interfaccia IModel

```
public interface IModel {  
    public String getModelId();  
    public void setModelId(String modelId);  
  
    public IEntity getRoot();  
    public void setRoot(IEntity root);  
  
    public void addModelListener(PropertyChangeListener l);  
    public void removeModelListener(PropertyChangeListener l);  
  
    public void notifyChanged(IEntity source, String propertyName, boolean oldValue, boolean newValue);  
    public void notifyChanged(IEntity source, String propertyName, int oldValue, int newValue);  
    public void notifyChanged(IEntity source, String propertyName, String oldValue, String newValue);  
    public void notifyChanged(IEntity source, String propertyName, IEntity oldValue, IEntity newValue);  
    public void notifyChanged(IEntity source, String propertyName, Object oldValue, Object newValue);  
}
```

© 2002-2004 Riccardo Solmi

38

Interfacce + classi astratte e adattatori

- **L'interfaccia è il vincolo minore sia per chi la pubblica che per chi la implementa.**
 - Si concordano un insieme di signature di metodi
 - Se le interfacce cambiano i clienti devono modificare le implementazioni
- **Se è possibile fornire implementazioni (concrete o astratte) ragionevoli per le interfacce conviene pubblicarle**
 - Il cliente implementa velocemente le interfacce
 - Il fornitore può cambiare le interfacce e modificare le implementazioni (in genere) senza compromettere la compatibilità con i clienti
- **In ogni caso, se il cliente ha già una propria implementazione/interfacce o comunque vuole essere completamente libero nelle scelte, può farlo.**
 - Introduzione del design pattern Adapter

© 2002-2004 Riccardo Solmi

39

Classi astratte e concrete del modello fornite

- **AbstractEntity**
 - Per le entità che hanno solo proprietà con nome
 - Introduce *setChildrenModel* da ridefinire in modo da propagare verso i figli le chiamate a *setModel*.
 - La *setModel* data mantiene l'invariante sul modello richiamando la *setChildrenModel* (template method)
- **AbstractCompositeEntity (estende AbstractEntity)**
 - Per le entità composite (esempio BlockStatement)
- **Model**
 - Implementazione concreta di *IModel*
 - La radice del modello è tenuta a fare una *setModel* e a settare il model Id.

© 2002-2004 Riccardo Solmi

40

Supporto undo/redo

- **Le operazioni per modificare il modello che abbiamo definito sono distruttive: una volta applicate non è possibile tornare allo stato precedente del modello.**
- **Come si fa a supportare le azioni di undo/redo?**
 - Mi conviene modificare le azioni sul modello?
 - O aggiungere un nuovo strato di metodi e/o classi?
 - Introduzione al design pattern Command
- **Di serie vi vengono forniti 3 comandi:**
 - CompositeAddCommand, CompositeRemoveCommand
 - ReplaceChildCommand
 - Altri possono essere aggiunti per editare le proprietà con nome (esempi in IL: costanti e variabili)

© 2002-2004 Riccardo Solmi

41

Editing interattivo

- **L'editing interattivo è possibile per mezzo di menu contestuali e operazioni di Drag and Drop (+ editing diretto via tastiera per le costanti)**
- **Il framework che vi viene fornito gestisce tutta la parte interattiva legata al funzionamento, ma deve essere configurato con:**
 - l'insieme delle azioni possibili
 - le regole per stabilire quando abilitarle
- **Come si può progettare il sistema in modo che queste parti interattive dell'editor modifichino il proprio comportamento in funzione del linguaggio che si sta utilizzando?**
 - Introduzione al design pattern Strategy

© 2002-2004 Riccardo Solmi

42

Predicati

- **Quando l'utente compie le operazioni di input per richiedere l'apertura del menu contestuale, il sistema seleziona la strategy corrispondente al model Id della selezione**
- **L'insieme delle operazioni definite sul modello viene poi filtrato usando delle regole (predicati).**
 - Solo le azioni che soddisfano il predicato vengono mostrate
- **Un predicato è un oggetto che implementa la seguente interfaccia:**

```
public interface IEnablerPredicate {  
    public boolean evaluate(AbstractPart selectedPart, Object userdata);  
}
```

© 2002-2004 Riccardo Solmi

43

Predicati standard forniti

- **Per comodità viene fornita una classe (EnablerPredicateFactory) che definisce diversi predicati di uso comune più alcuni per costruire predicati complessi a partire da quelli semplici.**
 - and, or, not
 - instanceOf, parent_instanceOf, sameType, differentType
 - insertMode, replaceMode
 - isComposite, hasCompositeParent
 - dndSinglePart, dndOver, dndPartsInstanceOf
- **La classe può essere estesa con altri predicati specifici per il proprio modello (vedi esempio IL)**

© 2002-2004 Riccardo Solmi

44

Azioni da menu contestuale

- **Per definire le azioni disponibili da menu contestuale bisogna implementare l'interfaccia *IActionFactory* o più semplicemente estendere *AbstractActionFactory***
- ***AbstractActionFactory* implementa *create* come template method e richiede di definire due metodi che restituiscono due array: uno per le azioni di replace e l'altro per quelle di inserimento.**

```
public interface IActionFactory {  
    public IAction[] create(IWorkbenchPart workbenchPart,  
        IPrototypeManager prototypeManager);  
}
```

© 2002-2004 Riccardo Solmi

45

Definizione di una azione (da menu contestuale)

- **La definizione di una azione richiede nell'ordine:**
 - Predicato
 - Nome prototipo da inserire/sostituire
 - Stringa che deve apparire nel menu contestuale
 - Trasformatore (solo per azioni replace)
- **Il trasformatore è un oggetto che definisce un metodo per passare delle informazioni dall'oggetto che si sta sostituendo al sostituto.**

```
public interface IEntityTransformer {  
    public void transform(IEntity oldEntity, IEntity newEntity);  
}
```

© 2002-2004 Riccardo Solmi

46

Comandi Drag and Drop

- **I comandi di Drag and Drop sono configurabili in modo analogo alle azioni del menu contestuale**
- **Ogni editor deve implementare l'interfaccia che segue.**
- **Anche in questo caso il modo più semplice è partire dall'implementazione concreta che viene fornita**
- **La factory dei comandi definisce anche il comportamento del tasto delete**

```
public interface IDnDCommandFactory {  
    public Command createCommand(DnDRequest request);  
}
```

© 2002-2004 Riccardo Solmi

47

Richieste di Drag and Drop

- **Il Drag and Drop è complicato dal fatto che la richiesta coinvolge un insieme di parti che vengono sottoposte a drag e una destinazione dove viene eseguito il drop**
- **L'operazione che viene eseguita, inoltre può essere uno spostamento (move) oppure una copia (clone o share).**
 - Se la move sposta il figlio da un composite ad un altro vengono eseguite due richieste
MOVE_ORPHAN_CHILD + MOVE_ADD_CHILD
 - Se la move è interna ad un composite viene richiesta
MOVE_CHILD
 - La copia si attiva con un tasto modificatore (CTRL o ALT di solito) e produce una richiesta
CLONE_CHILD oppure SHARE_CHILD

© 2002-2004 Riccardo Solmi

48

View

- Per ogni tipo di entità del modello che si vuole mostrare nell'editor, è necessario definire una (classe) figura.
- In accordo all'architettura MVC, la figura non ha accesso ne al modello ne al controller.
- Per definire una figura bisogna estendere la classe *SelectableFigure*.
- In linea di principio si può ridefinire il metodo *paintFigure* di una figura e disegnare liberamente quello che si vuole
- In pratica però si preferisce partire dalle figure predefinite (rettangoli, ovali, label) e comporle con un *layout manager* che ne determina l'impaginazione (composite + strategy).

Definizione di figure

- Per comodità viene fornita una classe *GrammarRuleFigure* capace di mostrare figure che rappresentano costrutti grammaticali di un linguaggio.
 - Gli elementi vengono ordinati da sinistra a destra su più righe
- Il costruttore della figura definisce il contenuto e il layout chiamando i seguenti metodi:
 - *addKeyword*, *addOperator*, *addToken*,
 - *addDelimiter*, *addParenthesis*, *addIndent*
 - *addChildPlace*, *nextRow*
- Si possono definire dei metodi aggiuntivi per nascondere una parte della figura (ad esempio le parentesi).

Controller - Parts

- **Le responsabilità del controller dell'architettura MVC sono ripartite tra una gerarchia di "parti" e un insieme di strategie per gestire le interazioni (policies and actions).**
 - La configurazione delle azioni l'abbiamo già descritta
 - Le policies sono gestite internamente dalla libreria fornita
 - Qui ci occupiamo delle Parti
- **Ogni entità del modello deve essere controllata esattamente da una istanza del controller (parte).**
 - Il metodo *createPart* di *IPartFactory* mappa entità del modello su parti
- **Una parte ha accesso sia all'entità del modello che controlla (usando *getModel*) sia alla figura che la rappresenta (usando *getFigure*)**
 - Il metodo *createFigure* di una parte mappa una parte con una figura
- **Inoltre una parte è collegata sia alle parti "padre" che ai "figli" formando così una struttura gerarchica che riproduce la struttura della parte del modello che si vuole mostrare (*getParent, getChildren*)**

© 2002-2004 Riccardo Solmi

51

Gerarchia di parti

- **Tutte le Parti devono estendere la classe *AbstractPart* (o una delle sue sottoclassi) e implementare i seguenti metodi:**
 - *IFigure createFigure()*
 - *List getModelChildren()*
 - *void propertyChangeUI(event)* e *void refreshVisuals()*
- **Vengono fornite alcune sottoclassi di *AbstractPart* specializzate:**
 - *CompositeColumnPart* e *CompositeRowPart* per controllare le entità che sono dei composite (es. *BlockStatement*)
 - *AbstractRootPart* per controllare la radice del modello
 - *EmptyPart* per controllare parti del modello non definite
 - Schema soluzione per costanti editabili (es. *Literal, Variable*)
- **NB Diversamente da quanto visto per il modello, non viene fornita una interfaccia per i controller perché si assume che la maggior parte del lavoro venga fatta dal framework**

© 2002-2004 Riccardo Solmi

52

Definizione di parti: composite, radice e foglie

- **Le classi per i composite e *EmptyPart* sono usabili direttamente, senza ulteriore configurazione**
- **La classe *AbstractRootPart* richiede la definizione del metodo *getModelChildren()***
 - La *AbstractRootPart* controlla l'entità del modello che viene rappresentata come sfondo dell'editor
 - Di solito *getModelChildren* restituisce una lista con un solo elemento: la “vera” radice del modello dal punto di vista dell'utente dell'editor.

Definizione di parti: foglie editabili

- **E' il caso più complicato e deve essere implementato prendendo come esempio le parti costanti e le variabili di IL.**
- **La complicazione è dovuta alla necessità di gestire l'editing del valore tramite tastiera (Direct Edit).**
 - Bisogna definire una *DirectEditPolicy* appropriata
 - E un *Command* che esegua l'operazione di aggiornamento del valore della proprietà editata sul modello
 - Vedi esempio in IL: Variable

Definizione di parti: nodi interni

- **Controllano una entità del modello che ha solo proprietà con nome.**
- **Bisogna estendere `AbstractPart` e ridefinire:**
 - `getModelChildren` deve restituire una lista contenente tutti e soli i valori delle proprietà che si vogliono presentare
 - `createFigure` deve istanziare una figura capace di rappresentare le proprietà ritornate da `getModelChildren` più eventuali abbellimenti sintattici
 - Opzionalmente, si possono definire dei metodi di refresh per mostrare/nascondere alcuni abbellimenti della figura.
 - Esempi: graffe solo in presenza di blocchi, parentesi solo per disambiguare espressioni
 - Devono essere richiamati in `getModelChildren`

© 2002-2004 Riccardo Solmi

55

Editor Kit

- **Tutte le factory che compongono un editor più gli eventuali metodi eseguibili sul modello sono accessibili da un'unica classe che implementa l'interfaccia: `IEditorKit`**
- **Viene fornita una classe astratta `AbstractEditorKit`**

© 2002-2004 Riccardo Solmi

56

Violazioni del mapping 1 a 1 modello-controller

- **Nel modello di IL le variabili sono rappresentate da istanze della classe `Variable`; quando una stessa variabile deve apparire in più punti del programma, usiamo un'unica istanza che pertanto appare in più punti del modello.**
 - Un oggetto variabile per tutte le occorrenze della variabile
- **Un “vero” interprete considera due occorrenze di una variabile riferimenti ad una stessa variabile facendo un binding che associa nomi di variabili a valori in accordo alle regole di scoping del linguaggio.**
 - Tanti oggetti quanti sono i riferimenti + binding separato
- **InterpreterVisitor contiene una classe `BindingManager` che segue questo secondo approccio.**

© 2002-2004 Riccardo Solmi

57

Esempio di aggiunta di un costrutto a IL

- **Voglio aggiungere un costrutto per stampare il valore di una espressione su standard output (tipo: `System.out.print`)**
- **Per aggiungere `PrintStatement` devo :**
 - Creare `PrintStatement`
 - Aggiungere `visitPrint` in `IILVisitor`
 - Implementare `visitPrint` in `InterpreterVisitor`, `PrettyPrinterVisitor` e `ILPartFactoryVisitor`
 - Creare `PrintStatementPart`
 - Creare `PrintStatementFigure`
 - Aggiungere `createPrint` in `IILFactory` e `ILLanguageFactory`
 - Aggiungere prototipo in `ILPrototypeManager`
 - Modificare esempio di fattoriale per stampare risultato

© 2002-2004 Riccardo Solmi

58

Serializzazione del modello

- **Il modello del linguaggio IL viene salvato/caricato usando la serializzazione standard Java**
- **Perché funzioni è necessario che tutte le classi del modello implementino l'interfaccia *Serializable* (senza metodi) e possiedano un costruttore pubblico senza parametri.**
 - *AbstractEntity* implementa già *Serializable* per tutte le entità
 - Notare in IL che la classe *OperatorGroup*, non essendo una entità, implementa esplicitamente l'interfaccia *Serializable*.
- **NB La serializzazione standard di Java non è pensata per fornire persistenza di lungo periodo ad un modello pertanto è abbastanza fragile**
 - Quando si fanno cambiamenti alle classi del modello è facile che i file salvati in precedenza non siano più leggibili
 - I due metodi di salvataggio/caricamento del modello sono implementati nell'editor kit dato ed eventualmente possono essere sostituiti.