

# Laboratorio di Progettazione di Sistemi Software

## Materiale per il progetto - Esercizi

Valentina Presutti (A-L)

Riccardo Solmi (M-Z)

# Indice degli argomenti

---

- Introduzione e primi esercizi sull'esempio di riferimento del corso
- Rappresentazione di un modello
  - Soluzione specifica vs. generica
- Aggiunta modulare di operazioni
  - Visitors, type switch, Iterators
- Manipolazione di un modello
  - API specifiche vs. generiche
- Aggiunta modulare di costrutti
  - Factory estendibili di esempi e prototipi

# Introduzione

---

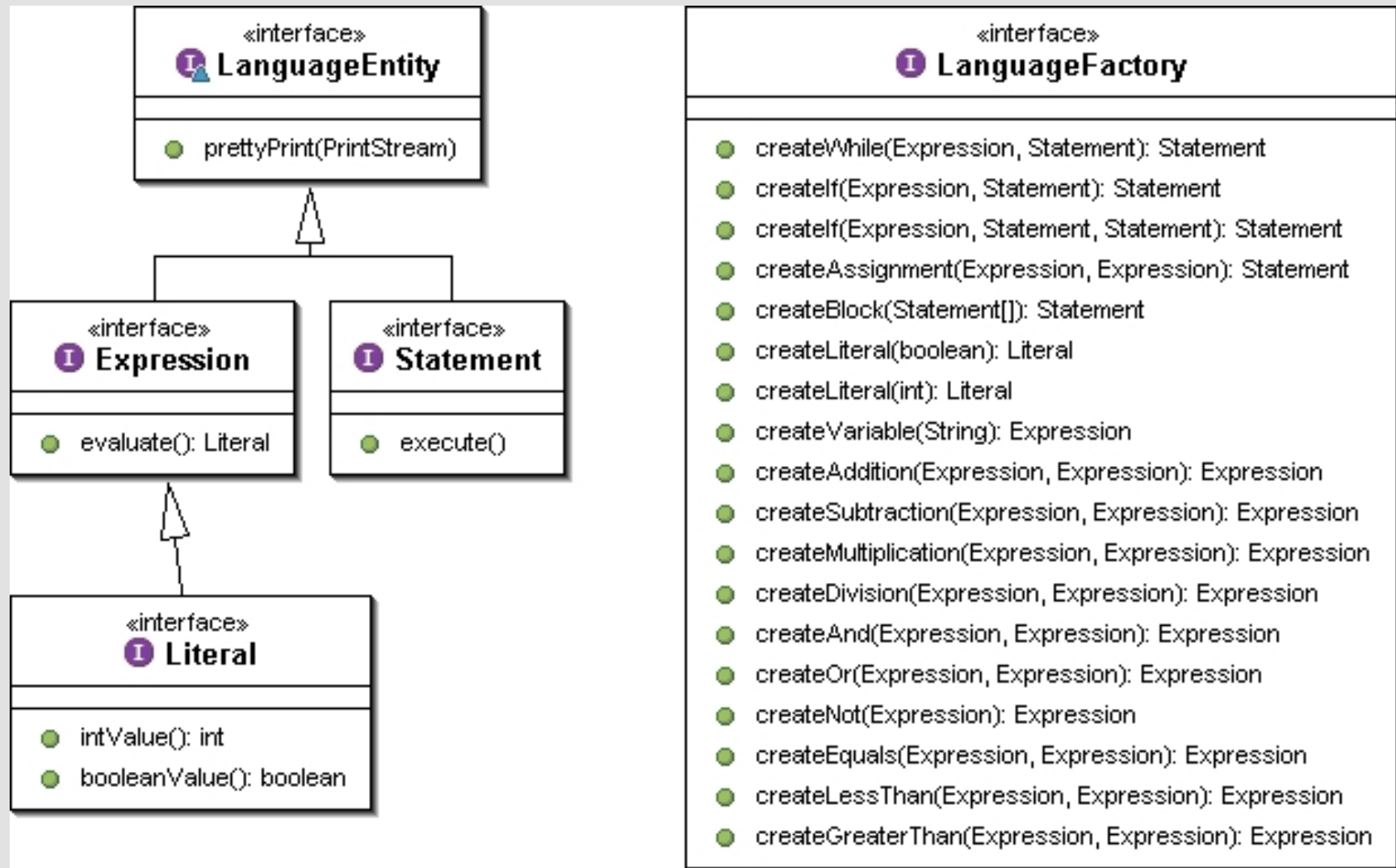
- Questo documento contiene la descrizione del processo di sviluppo del progetto del corso di labss 2006/07
- Le scelte progettuali e le alternative sono presentate usando il catalogo di design pattern visto a lezione
- Le trasformazioni che dal progetto iniziale (primo esercizio) portano a sviluppare 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.

# Progetto labp2001

---

- **Esercizio: implementare il progetto labp2001**
  - Importate il progetto vuoto dato (labp2001implvuota.zip)
  - Le specifiche sono incluse (labp2001implvuota/docs/api/index.html)
- **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.)

# Diagramma delle classi delle specifiche



# Uso dei diagrammi UML per comprendere esercizio

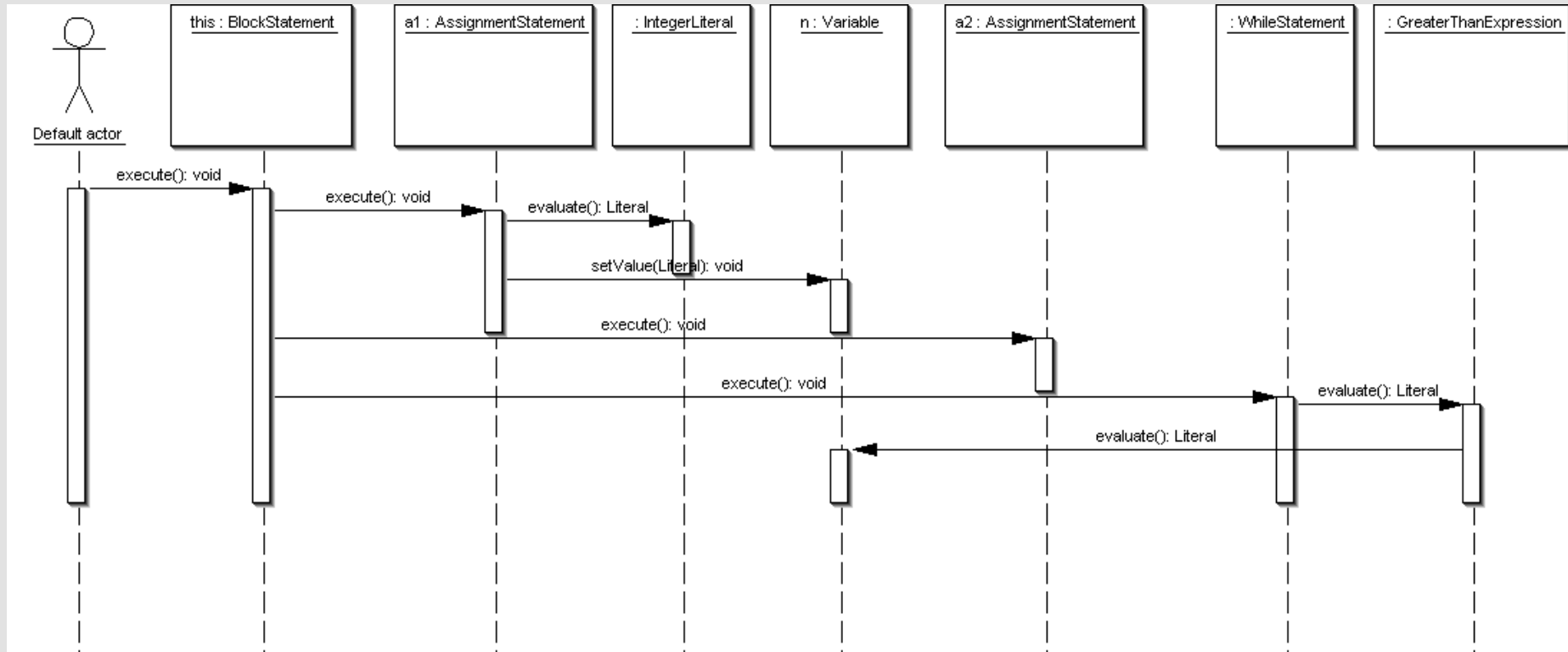
---

- 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

# Diagramma degli oggetti del programma fattoriale

---

# Diagramma delle sequenze di execute su fattoriale



# 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`)?

# 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:
  - labss.lang.il.model
- Refactoring: rinominare il package dell’implementazione in:
  - labss.lang.il.model.impl

# 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?

# Rappresentazione specifica vs. generica del modello

---

- 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, <, >, =).
- Nella scelta di quali/quante classi concrete implementare, qual è lo spazio delle soluzioni?
  - Soluzione specifica: Una per ogni costrutto linguistico (if, while, +, ...)
  - Soluzione generica: 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?
- Posso arrivare a fare una soluzione abbastanza generica da funzionare con tutti i linguaggi?

# Esercizi su soluzione con poche classi generiche

---

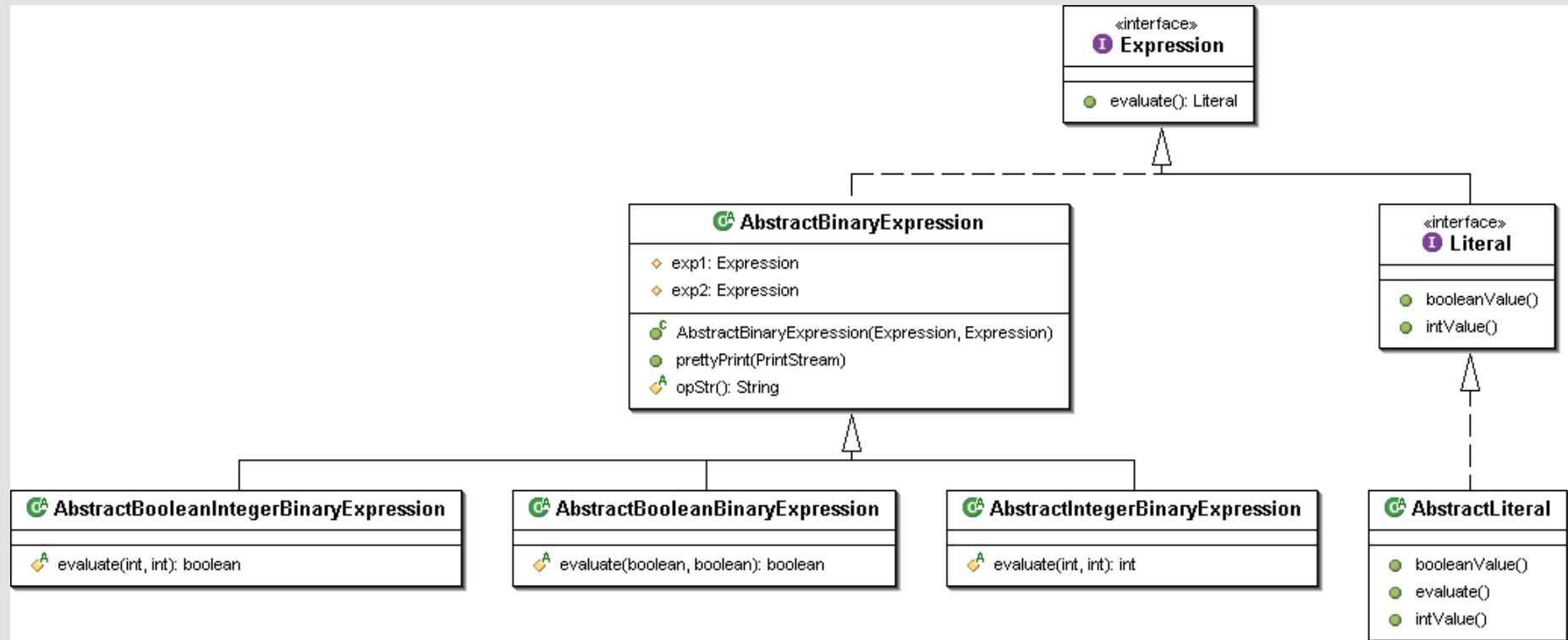
- Come faccio a dare ai metodi `evaluate` e `prettyPrint` un comportamento polimorfo?
- Cosa mi conviene passare in più al costruttore: un intero e/o una stringa?
- E' una soluzione orientata agli oggetti?
  - Osservazione: il metodo della `factory` sa esattamente cosa vuole costruire (ad esempio una 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? Potrei usare una `enum`?
- Scenario d'uso: `deploy` dinamico di un linguaggio

# Soluzione specifica che introduce classi astratte

---

- Mantengo una classe per ogni costrutto linguistico ma raccolgo in una classe astratta le parti comuni alle implementazioni delle operazioni binarie e ai literal
- 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 usando solo pull up e override?
  - Introduzione e applicazione del design pattern `TemplateMethod`
- In un contesto generativo lo sforzo di fattorizzare la parte comune dell'implementazione conserva la stessa importanza?

# Diagramma classi espressioni astratte



# 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?

# 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

# 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? (design pattern Monostate)

# 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
  - Rendo `final` i metodi template e protetto le operazioni ridefinibili che introducono.
  - Rendo privati i campi delle classi esponendo eventualmente dei metodi per accedervi (getters/setters)
- L'uso del modificatore “`final`” mi aiuta ulteriormente ad imporre le mie scelte (esempi Template Method, costanti)

# Possibili linee di sviluppo del progetto

---

- 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
- 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
- Posso rendere modulari entrambe le aggiunte di cui sopra?

# Rappresentazione di operazioni polimorfe

---

- Una operazione polimorfa definita su un modello ha bisogno di accedere a due fonti di informazioni (senza contare input/output):
  - La struttura dati su cui operare (i.e il modello)
  - Lo stato di avanzamento dell'operazione stessa
- A seconda che l'operazione sia implementata all'interno del modello o all'esterno le due fonti di informazione sono raggiungibili in modo diverso:
  - Nella soluzione “spalmata” sulle classi del modello ogni variante polimorfa dell'operazione ha accesso diretto ai dati del modello e deve farsi passare e passare a sua volta lo stato di avanzamento dell'operazione
  - Nelle soluzioni modulari (esterne al modello) ogni variante polimorfa ha accesso diretto allo stato di avanzamento e si deve fare passare lo stato dell'oggetto su cui operare

# Rappresentazione di operazioni polimorfe (cont.)

---

- *L'invocazione* di una operazione polimorfa dipende dalla sua rappresentazione:
  - Soluzione spalmata sul modello – normale object dispatching eseguito sul modello (parametro implicito)
  - Soluzione modulare – di solito richiede l'istanziamento della classe che rappresenta l'operazione e l'invocazione di un metodo che richiede il passaggio del modello come parametro esplicito (Si possono nascondere i dettagli implementativi in un metodo statico)
- *Selezione della variante polimorfa da eseguire su un oggetto:*
  - Soluzione spalmata sul modello – il polimorfismo fornito dall'object dispatching rende automatica la selezione della variante polimorfa dell'operazione corrispondente al tipo runtime dell'oggetto.
  - Soluzione modulare – è necessario introdurre un nostro meccanismo di dispatching per selezionare la variante polimorfa
    - Le soluzioni illustrate nei prossimi lucidi si distinguono proprio nel modo di realizzare questo dispatching polimorfo

# Soluzione basata su if annidati e instanceof

---

- Nell'implementazione vista fin'ora, le varianti di ciascuna operazione polimorfa sono sparse sulle classi che rappresentano i costrutti; vogliamo renderle modulari raggruppandole in un'unica classe.
- Esercizio: riscrivere l'operazione prettyPrint in una classe separata dai costrutti (in un package ....visitors)
  - Utilizzare degli *if* annidati di *instanceof* per realizzare la selezione della variante polimorfa da eseguire su un oggetto
- E' possibile condividere l'implementazione di varianti polimorfe? Confrontare con i pattern visti per la soluzione spalmata sul modello
- Posso rendere costante e trascurabile il costo del dispatching facendomi aiutare dal modello nella selezione della variante polimorfa?

# 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 sul 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 di Manipolazione di un modello

---

- Un modello deve essere navigabile e modificabile anche dopo la sua costruzione
  - Per supportare la costruzione top down di modelli
  - Per supportare le operazioni modulari
- Per *API di manipolazione* si intende un insieme di metodi applicabili alle entità di un modello al fine di navigare, modificare o interrogare le relazioni di associazione delle entità stesse.
- I metodi principali di manipolazione sono i *getter* e i *setter*.
- Si possono definire a livello del tipo delle singole entità (API specifiche) oppure a livello del tipo interfaccia più astratto del modello (API generiche).

# API specifiche per manipolare un modello

---

- Le API di manipolazione specifica di un modello prevedono l'aggiunta di metodi getter e setter specifici per le varie entità
  - In ciascuna classe che rappresenta una entità del modello aggiungo:
    - per ogni proprietà definita in quella classe, una coppia di metodi getter e setter pubblici per manipolarla
- Osservare che le API specifiche non fanno parte delle interfacce. Chi può usarle?
  - `((WhileStatement) If.createWhile(..., ...)).setExp(...);`
  - `((WhileStatement) whileStm.clone()).setExp(...);`
- Nel caso di entità che rappresentano una collezione di oggetti che API posso definire per manipolarle?
  - Ad esempio `BlockStatement` che è una sequenza di istruzioni
- 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 un modello

---

Generalizzando l'idea del Composite si possono introdurre dei metodi di manipolazione che usano il primo parametro per selezionare l'associazione su cui operare.

## Manipolazione *by index*

- Le API per manipolare i composite, in particolare *size*, *get* e *set*, possono essere implementate da tutte le entità non solo dalle collezioni
  - Posso attribuire un ordinale alle associazioni di una entità e usarlo per definire il comportamento delle API *by index*

## Manipolazione *by name*

- Posso introdurre una enumerazione di tutti i nomi delle associazioni definite in un modello e usarla per selezionare l'associazione di una entità su cui operare.
  - Suggerimento: introdurre anche un metodo *indexOf* per ridirigere le API *by name* sull'implementazione delle API *by index*

Una API generica per sua natura usa come tipi dei parametri e tipi di ritorno il tipo interfaccia più astratto del modello di conseguenza vengono meno tutte le garanzie statiche tipiche di una API specifica

- Per questo motivo solitamente l'API generica affianca quella specifica

# Visitors con API generiche

---

- Il costo implementativo dell'aggiunta di API generiche si ripaga nel momento in cui ce ne avvantaggiamo della definizione modulare del comportamento.
- 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, ...)
- Operazione = Strategia di Attraversamento + Comportamento Specifico

# Strategie di Attraversamento Componibili

---

- **Traversal primitivi:**
  - Identity
  - Failure
  - Sequence(v1, v2)
  - IfThen(v1, v2), IfElse(v1, v2), IfThenElse(v1, v2, v3)
  - Not(v)
  - One(v1, ...), Some(v1, ...), All(v1, ...)
  - TraverseOne(v), TraverseSome(v), TraverseAll(v), TraverseParent(v)
- **Esempi di Traversal derivabili per composizione:**
  - Try(v) = IfElse(v, Identity)
  - TopDown(v) = Sequence(v, TraverseAll(TopDown(v) )
  - TopDownWhile(v) = IfThen(v, TraverseAll(TopDownWhile(v) )
  - TopDownUntil(v) = IfElse(v, TraverseAll(TopDownUntil(v) )
  - BottomUp(v) = Sequence(TraverseAll(BottomUp(v)), v)
  - DownUp(v1, v2) = Sequence(v1, Sequence(TraverseAll(DownUp(v1,v2) ), v2)
  - OnceTopDown(v) = IfElse(v, TraverseOne(OnceTopDown(v))
  - Innermost(v) = BottomUp(Try(Sequence(v, Innermost(v)))
  - Outermost(v) = TopDown(Try(Sequence(v, Outermost(v)))

# Soluzione basata su iteratore

---

- 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)
  - 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 *EntityIterators*
- Servirsi di iteratori aggiuntivi (*SingletonIterator*) per evitare la logica condizionale

# Iteratori Componibili

---

- **Iteratori primitivi:**
  - Empty
  - Self
  - Child, ChildReverse, ChildByName, ChildByIndex
  - FollowingSibling, PrecedingSibling
  - Descendant, DescendantOrSelf
  - Parent, Ancestor, AncestorOrSelf, AncestorReverse
  - Filter(i, predicate)
  - Sequence(i1, ...)
  - Compose(i1, ...)

# Operazioni modulari: push style vs pull style

---

- Scrivere un iteratore che mostra le variabili usate e confrontarlo con la soluzione basata su visitor
- Osservare la granularità della soluzione: classe vs. blocco di istruzioni
- Osservare le possibilità di intervenire sul controllo di flusso.
  - Posso interrompere l'operazione?
  - Posso sospendere l'operazione e riprenderla in seguito?
- Come scelgo la variante polimorfa da eseguire?

# Aggiunta modulare di costrutti: Factory estendibili

---

- La LanguageFactory (Abstract Factory) ora costruisce istanze di singoli costrutti
  - Definisce un metodo per ogni costrutto
  - I metodi in genere hanno dei parametri
- **Insufficienza del pattern Abstract Factory**
  - Se voglio permettere al cliente di aggiungere degli esempi di codice istanziabili su richiesta
  - Se voglio fare un editor e supportare le operazioni di copia/incolla (e Drag and Drop) di frammenti di codice
- **Ad uso interno continuo ad usare anche la Abstract Factory**
  - Nella definizione degli esempi e dei prototipi per le factory estendibili

# Factory estendibile per esempi di codice

---

- Aggiungere interfaccia *TemplateFactory* con un metodo *create* per costruire un esempio di codice (template):
  - `LanguageEntity create() // factory method`
- Aggiungere classe *TemplateManager* con i metodi:
  - `LanguageEntity create(String name)`
  - `void put(String name, TemplateFactory templateFactory)`
  - `void remove(String name)`
- L'implementazione può usare una *Map* per mappare i nomi dei template sulle factory da usare per costruirli.
- Osservare che *create* mi restituisce un *LanguageEntity* e non mi permette di passare degli argomenti
  - Come si può rimediare?

# Osservazioni

---

- Il Template Manager è più adatto per costruire esempi di codice piuttosto che singoli costrutti
  - Un esempio in genere non richiede parametri
  - Posso accedere a tutti i vostri esempi senza concordare con voi quanti sono e come si chiamano
- I template vengono costruiti solo se richiesti
- Per configurare un template posso
  - Modificare il metodo factory in modo da accettare parametri
  - Assegnare dei valori iniziali di default ed esporre dei metodi setter (servono API generiche).

# Factory estendibile per frammenti di codice

---

- Scenario: solo a runtime conosco il frammento di codice che voglio aggiungere alla factory
- Ho bisogno di un meccanismo per fare una *copia* di un frammento di codice dato
  - Introduzione design pattern Prototype
- .. e di un Prototype Manager che mi permetta di aggiungere prototipi e di clonarli su richiesta
  - Quanto è simile al Template Manager?

# Meccanismo di duplicazione (clone)

---

- Ho due meccanismi di creazione: istanziazione e clonazione
  - Linguaggi class based vs. linguaggi object based
  - Anche un linguaggio class based può avere la clonazione
- La **new** crea un oggetto (istanza) a partire da una “ricetta”: la classe.
  - Esempio: Expression var = **new** Variable(“i”);
  - Posso passare parametri al costruttore;
  - Il risultato ha un tipo concreto
- La **clone()** crea un oggetto facendo una copia di un oggetto già esistente.
  - Esempio Expression var2 = (Variable) var.clone();
  - L’oggetto è già configurato bene, eventualmente lo modifico in seguito usando dei metodi (setters)
  - Il risultato ha tipo Object.

# La clone di Java

---

- **clone()** è un metodo definito in `Object` (tutti lo ereditano)
  - 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 raggiungibili tramite riferimenti
  - La clone di Java è shallow, la si può ridefinire deep a piacere

# Clonazione di un frammento di codice

---

- Per clonare un frammento di codice ho bisogno di una clone profonda
  - Posso rendere pubblico e usabile il metodo clone in una classe astratta ed estenderlo in tutti i costrutti del linguaggio
  - Ogni tipo concreto deve far proseguire la copia in profondità in tutti i suoi campi strutturali (di tipo riferimento)
- **Esercizio: aggiungere al progetto il supporto alla clonazione**

# Manager di prototipi

---

- Aggiungere classe *PrototypeManager* con i metodi:
  - `LanguageEntity create(String name)`
  - `void put(String name, LanguageEntity prototype)`
  - `void remove(String name)`
- Osservare che *create* mi restituisce un *LanguageEntity* e non mi permette di passare degli argomenti
- Osservazione: mi farebbe comodo definire dei prototipi incompleti. Uso dei *null*? Design pattern Null Object
  - Aggiungere costrutti “null” `NullStatement` e `NullExpression` in modo che la `prettyPrint`, la `clone` e la `execute` vadano mentre le altre operazioni provochino una eccezione
- Esercizio: provare ad integrare le funzionalità del `PrototypeManager` nel `TemplateManager`

# Osservazioni

---

- **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
  - I prototipi vengono costruiti e allocati anche se nessuno li richiede
- **Esercizio: la clone di Variable rende non funzionante execute(). Come posso rimediare?**