Java Modeling Language

Tesina Metodi Formali nell'Ingegneria del Software

Alberto Cerullo

Overview

- Introduzione a JML
- Costrutti di JML
- JMLC e ESC/JAVA2
- Esempi d'applicazione
- Altri tool che usano JML

Introduzione a JML

- Java Modeling Language: linguaggio di specifica per moduli Java
- Utile per:
  - Specifica del comportamento delle classi Java
  - Definizione ed implementazione delle decisioni di progetto
- Nasce nell'ambito della "Progettazione a contratto" (pre- e post- condizioni definiscono un contratto tra una classe ed i suoi clienti)

Costrutti di JML

- JML permette di definire asserzioni all'interno del codice Java di una classe
- Il codice JML è inserito all'interno di commenti opportunamente marcati
- Due possibilità:
  - //@ statement JML
  - /*@
    statement JML
    @*/
- Commenti del tipo // @ oppure /* @ @ */ non sono validi

Predicati ed Espressioni

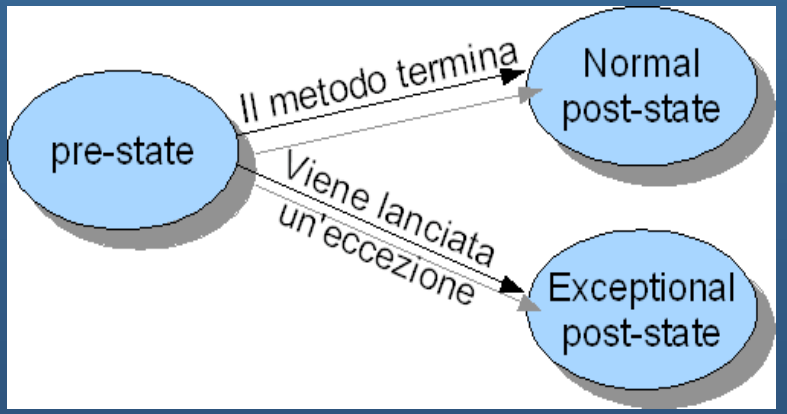
- I predicati JML sono espressioni di specifica (“spec-expressions”) con valore booleano
- Sono espressioni Java estese con espressioni primarie JML e costrutti ad hoc
- Un’espressione JML è considerata valida se:
  - Restituisce il valore true
  - Non causa eccezione
- “Strong validity”

Asserzioni JML

- Nelle asserzioni JML non si possono usare gli operatori “++” o “- -” e gli operatori di assegnamento, perchè causerebbero side effect.
- Asserzioni JML diverse dalle annotazioni Java
- Tutte le keyword JML sono precedute da “\”.
- Esempi:
  - \result espr : valore di ritorno di un metodo
  - \old (espr) e \pre (espr) : valore che aveva espr nel pre-stato di un metodo (non ammettono variabili libere)
    - (\forall int i; 0 <= i && i < 7; \old(i < y)); //illegale
    - (\forall int i; 0 <= i && i < 7; i < \old(y)); // ok

Ciclo di vita di un metodo

- Ogni metodo annotato con specifiche JML deve essere invocato in uno stato in cui le precodizioni siano verificate. Tale stato si chiama “pre-state”
- Se le precodizioni sono verificate :
  - Il metodo termina senza lanciare eccezioni (normal post state)
  - Viene lanciata un’eccezione (exeptional post-state)



Quantificatori

- Sintassi:

  (\quantificatore variabili ;[range ;] body);

- Quantificatori: \forall, \exists, \sum, \max, \min, \product, \num_of
- Esempio:
  - (\forall int i,j; 0 <= i && i < j && j < a.length; a[i] < a[j]);
  - (\forall int i,j; ;0 <= i && i < j && j < a.length ==> a[i] < a[j]);
- La condizione specificata nel range deve individuare un dominio finito

Precondizioni e PostCondizioni (1/2)

- Una precondizione è una (o più) condizione che deve essere vera immediatamente prima dell'esecuzione di un'operazione, mentre una postcondizione è una condizione che deve risultare vera subito dopo l'esecuzione di una operazione.

- Sintassi:

  requires pred;
  ensures pred;

- In una singola specifica possono essere presenti un numero qualsiasi di pre- e/o post-condizioni

  requires P; equivale a requires P && Q;
  requires Q;

Precondizioni e PostCondizioni (2/2)

- E' possibile definire diverse precondizioni e conseguenti postcondizioni non correlate tra loro tramite "also"

```
requires P1 ;
ensures Q1;
also
requires P2 ;
ensures Q2;
```

- oppure:

```
requires P1 || P2
ensures \old(P1) ==>Q1;
ensures \old(P2) ==>Q2;
```

Invariante

- Proprietà che deve essere sempre rispettata
- Deve essere garantita quando termina l'esecuzione del costruttore ed all'inizio e alla fine di un metodo
- Sintassi:

  invariant espr_booleana;

- Esempio:

```
public int priorityLevel;
public int timeStamp;
/*@ invariant
@ priorityLevel >= 0 && timeStamp >= 0;
@*/
```

- E' implicitamente incluso all'interno di pre- e post-condizioni
- In casi di necessità si può violare l'invariante dichiarando un metodo come "helper"

Assert

- Serve per definire asserzioni all'interno dei metodi
- Diverso dall' "assert" Java
- Sintassi

  assert pred [: expression];
- La parte opzionale "expression" deve essere di tipo String ed è il messaggio da visualizzare in caso di violazione

Assignable

- Serve per indicare gli elementi che possono essere modificati da un metodo
- Sintassi:

  assignable obj [, obj]
- Vanno indicati gli oggetti e non i riferimenti.

```
//assignable arr;                                   è errato!
public static void metodo(int [] arr){

//assignable arr[*];                                è corretto!
public static void metodo(int [] arr){
```

- Per affermare che il metodo non modifica nulla si usa assignable \nothing
- In generale:
  - Per I metodi statici vanno indicati gli oggetti nei parametri formali che vanno modificati
  - Nel caso degli array: elementi arr[5] o range arr[1..8] o arr[*]
  - Si usa la forma ".*" per gli oggetti di cui si vuole modificare un campo

signals

- Sintassi:

  signals (Exc [e]) Pred;

- La clausola signal afferma che se l'invocazione del metodo termina con il lancio dell'eccezione di tipo Exc, deve essere garantito il predicato Pred
- Alla stessa famiglia appartiene anche la clausola signals_only.
- Sintassi: **signals_only E1,E2,…,En;**
- Questa clausola indica quali eccezioni possono verificarsi

Side Effect in JML

- Non si può fare side effect nelle asserzioni JML
- All'interno delle asserzioni possono esserci solo metodi puri, cioè metodi che non producono side effect
- Per definire un metodo puro:

```
public /*@ pure @*/ tipo_ritorno metodo_get()
```

- Tale scelta evita modifiche al codice

Specifiche Informali

- Sintassi:

  (* testo che descrive la specifica *)

- Utile nella fase iniziale della progettazione
- o quando non è possibile definire in maniera formale una specifica
- I tool che fanno uso di JML non possono manipolare questo tipo di specifiche

Information Hiding

- Vengono mantenute le regole Java sulla visibilità
- Metodi definiti come “public” hanno anche le specifiche con visibilità pubblica
- La keyword “public” definisce un’asserzione JML pubblica
- Non possono essere usate variabili private all’interno di specifiche pubbliche a meno che non sia definite come “spec_public”
- Sintassi:

  private **/*@ spec_public @*/** tipo variabile;

Variabile modello

- Variabile definita ed usata solo all'interno delle specifiche
- Serve per gestire eventuali modifiche
- Esempio: vogliamo sostituire

  private /*@ spec_public non_null @*/ String nome;

  con

  private /*@ non_null @*/ String nomeCompleto;
- Applichiamo "model":

  //@ public **model** non_null String nome;

  private /*@ non_null @*/ String nomeCompleto;

  //@ private **represents** nome <- nomeCompleto;
- "represents" definisce una funzione d'astrazione, che mappa una rappresentazione concreta di un valore in una astratta

Tool che lavorano con JML

- Runtime assertion checker: **JMLC**
- Static Verification: **ESC/Java2**

JMLC

- Il compiler JML (jmlc) permette di compilare codice java annotato con asserzioni JML
- Permette la verifica a tempo di esecuzione delle asserzioni
- Verifica trasparente all'utente a meno di una violazione di qualche asserzione
- Per compilare il codice:

  jmlc NomeClasse.java
- Per eseguire il codice:

  jmlrac NomeClasse

JMLC

- Le release di questo tool, che sono allo stato attuale nella fase di beta-test, funzionano su piattaforme che presentano come versioni Java la J2SDK 1.4.1 o 1.4.2
- Durante il lavoro per questa tesina ho potuto constatare che ci sono dei problemi di compatibilità tra le ultime release del tool (che è arrivato alla 5.6) e le varie versioni del JDK.
- Una versione stabile, che è quella usata per questo lavoro, è la release 5.5 che funziona correttamente con il JDK 1.4.2_17
- Tra gli sviluppi futuri di JMLC c'è quello di garantire una completa compatibilità con le versioni superiori di Java a partire dalla 1.5.

ESC/Java2

- Prodotto nei laboratori della Compaq Research
- Perfettamente compatibile con le versioni 1.4 di Java
- Tool che cerca errori in codice Java attraverso un'analisi statica del programma (**static checker**)
- Tenta di provare la correttezza delle specifiche a tempo di compilazione
- 3 passi fondamentali:
  - Partendo dal codice sorgente Java e dalle specifiche JML, vengono generati alcuni predicati da verificare.
  - I predicati, insieme ad un modello logico del funzionamento di Java, vengono elaborati da un theorem prover, che verifica che le condizioni siano soddisfatte.
  - Se le condizioni non sono soddisfatte viene segnalato un potenziale bug.

ESC/Java2

- A differenza di JMLC, che testa la correttezza di una specifica, ESC/Java ne prova la correttezza.
- In maniera dettagliata i passi che esegue ESC/Java2 sono i seguenti:
  - fase di parsing (in cui viene controllata la sintassi)
  - fase di typechecking (controllo sui tipi)
  - fase di controllo statico (vengono cercati potenziali bugs). In questa fase viene usato, in maniera trasparente all'utente, il theorem prover chiamato Simplify.

- Le prime due fasi producono "cautions" ed "errori".
- La terza ed ultima fase produce "warnings".

- Il codice annotato con JML viene trasformato in una forma semplificata di un linguaggio definito da Dijkstra chiamato Guarded Command
- Da questa forma vengono generate l'insieme di condizioni da passare al prover Simplify, che accetta formule del primo ordine, di cui cerca di dimostrare la validità

Simplify

- E' il **theorem prover** usato da ESC/Java2
- Quando si deve verificare la validità di una formula F, Simplify come molti theorem prover procede testando la soddisfacibilità della formula negata ¬F
- Per verificare se una formula è soddisfacibile, Simplify effettua una ricerca di tipo backtracking, guidata dalla struttura proposizionale della formula, cercando di trovare un'assegnazione di soddisfacibilità, ossia un'assegnazione di valori di verità che rendono la formula vera e che è consistente con la semantica della teoria sottostante.

Esempio con Simplify

- Vogliamo verificare F: $x < y \Rightarrow (x - 1 < y \wedge x < y + 2)$
- Possiamo scrivere ¬F : $x < y \wedge (x - 1 \geq y \vee x \geq y + 2)$
- Che è divisibile in:

$$x < y$$
$$x - 1 \geq y$$
$$x \geq y + 2$$

- Ogni assegnazione che deve rifiutare ¬ F (cfr. soddisfare F) deve asserire x<y vero, la ricerca quindi inizia asserendo x<y

Esempio con Simplify - Algoritmo

si assume $x < y$

consideriamo la clausola $x - 1 \geq y \vee x \geq y + 2$

primo caso,assumiamo $x - 1 \geq y$

si verifica l'inconsistenza di $x < y \wedge x - 1 \geq y$

facciamo backtracking dal primo caso (si elimina l'assunzione $x - 1 \geq y$

secondo caso, assumiamo $x \geq y + 2$

si verifica l'inconsistenza di $x < y \wedge x \geq y + 2$

facciamo backtracking dal secondo caso

(poiché sono esauriti tutti i casi e non è stato trovata un'assegnazione di soddisfacibilità terminiamo)

viene restituito ¬F insoddisfacibile, dunque F è soddisfacibile.

La strategia di Ricerca

- Usa una struttura dati globale **context**: congiunzione delle formule e assunzioni definite nel caso di ricerca corrente
- Valore booleano **refuted**, settato qaundo il contesto è incosistente
- Insieme **lits**: formule atomiche chiamate letterali
- Insieme **cls**: insieme di clausole, costituite dalla disgiunzione di letterali
- Algo usati: **AsserLit(P), Push (),Pop()**

```
proc Sat() ≡
  enable refinement;
  Refine();
  if refuted then return
  elsif cls is empty then
    output the satisfying assignment lits;
    return
  else
    let c be some clause in cls, and l be some literal of c;
    Push();
    AssertLit(l);

     delete c from cls;
     Sat()
     Pop();
     delete l from c;
     Sat()
   end
end
```

La strategia di Ricerca 2


```
proc Refine() ≡
  while refinement enabled do
    disable refinement;
    for each clause C in cls do
      RefineClause(C);
      if refuted then
        return
      end
    end
  end
end
```

```
proc RefineClause(C : Clause) ≡
  if C contains a literal l such that [lits ⇒ l] then
    delete C from cls;
    return
  end;
  while C contains a literal l such that [lits ⇒ ¬l] do
    delete l from C
  end;
  if C is empty then

    refuted := true
  elseif C is a unit clause {l} then
    AssertLit(l);
    enable refinement
  end
end
```

Esempi

Esempio Massimo (1/2)

- Vogliamo trovare il massimo in un vettore di interi positivi
- Precondizione: ogni elemento dell'array è maggiore di 0
- Postcondizione: il risultato è un valore presente nell'array ed è il più grande di quelli presenti.
- In logica del primo ordine:
- Precondizione:
- Postcondizione:

$$\forall X \quad X \geq 0 \wedge X < vett.length \rightarrow vett[X] > 0.$$

$$(\exists X \quad X \geq 0 \wedge X < vett.length \wedge vett[X] = result) \wedge$$
$$(\forall X \quad X \geq 0 \wedge X < vett.length \rightarrow result \geq vett[X])$$

- In JML
- Precondizione: **requires vett!=null && (\forall int i; 0 <= i && i < vett.length; vett[i] > 0);**
- Postcondizione: **ensures (\exists int i; 0 <= i && i < vett.length; vett[i] == \result) && (\forall int j; 0 <= j && j < vett.length; \result >= vett[j]) ;**

Esempio Massimo (2/2)

Compilazione

```
C:\WINDOWS\system32\cmd.exe
parsing ..\specs\java\lang\Byte.jml
parsing ..\specs\java\lang\Error.jml
parsing ..\specs\java\io\DataInput.refines-spec
parsing ..\specs\java\lang\Runnable.spec
parsing ..\specs\java\security\PublicKey.spec
parsing ..\specs\java\security\Key.spec
parsing ..\specs\java\util\Date.refines-spec
parsing ..\specs\java\util\Calendar.refines-spec
parsing ..\specs\java\util\GregorianCalendar.refines-spec
parsing ..\specs\java\util\ResourceBundle.jml
typechecking ..\specs\java\lang\Object.jml
parsing ..\specs\java\lang\Float.jml
parsing ..\specs\java\lang\Double.jml
parsing ..\specs\java\lang\Integer.jml
parsing ..\specs\java\util\List.spec
parsing ..\specs\java\util\ListIterator.spec
parsing ..\specs\java\util\Observer.spec
parsing ..\specs\java\util\Observable.refines-spec
parsing ..\specs\java\util\SortedMap.spec
parsing ..\specs\java\util\SortedSet.spec
parsing ..\..\Documents and Settings\aspire1690\Impostazioni locali\Temp\jmlrac4
0685.java
typechecking Massimo.java
typechecking Massimo.java
typechecking Massimo.java
typechecking Massimo.java

C:\JML5\bin>_
```

Main corretto con JMLC

```
C:\WINDOWS\system32\cmd.exe
0685.java
typechecking Massimo.java
typechecking Massimo.java
typechecking Massimo.java
typechecking Massimo.java

C:\JML5\bin>jmlrac Massimo
0:1, 1:5, 2:3,
massimo: 5
**********

C:\JML5\bin>_
```

Main con violazione con JMLC

```
C:\WINDOWS\system32\cmd.exe
typechecking Massimo.java
typechecking Massimo.java
typechecking Massimo.java
typechecking Massimo.java

C:\JML5\bin>jmlrac Massimo
0:1, 1:5, 2:-3,
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionEr
ror: by method Massimo.massimo
        at Massimo.main(Massimo.java:725)

C:\JML5\bin>
```

```
C:\WINDOWS\system32\cmd.exe
l-5.2.jar" escjava.Main -classpath   C:\ESCJava2\jmlspecs.jar -classpath . -nowa
rn Deadlock -specs C:\ESCJava2\specs Massimo.java -noCautions
ESC/Java version ESCJava-2.0b3
    [0.078 s 8537872 bytes]

Massimo ...
  Prover started:0.032 s 14949576 bytes
    [2.125 s 14599680 bytes]

Massimo: stampaVettore(int[]) ...
    [6.656 s 18523464 bytes]  passed

Massimo: massimo(int[]) ...
    [0.047 s 17359288 bytes]  passed

Massimo: main(java.lang.String[]) ...
    [1.531 s 19296496 bytes]  passed

Massimo: Massimo() ...
    [0.016 s 19582264 bytes]  passed
  [10.375 s 19583144 bytes total]

C:\ESCJava2>
```

Static verification con ESC/Java2

Esempio con ESC/Java2


```
class Test {
      int[] a;       //@ invariant a !=null;
      int n;         //@ invariant 0<= n && n <= a.length;
      int extractMin() {
      int m = Integer.MAX_VALUE;
      int mindex = 0;
      for (int i = 0; i < n; i++) {
          if (a[i] < m) { mindex =i; m = a[i]; } }
          n- -;
      a[mindex] = a[n];
      return m;
}     Warning: Possible null dereference
      Warning: Array index possibly too large
```

Esempio con ESC/Java2


```
class Test {
      int[] a;          //@ invariant a !=null;
      int n;            //@ invariant 0<= n && n <= a.length;//@ requires n>=0;
      int extractMin() {
      int m = Integer.MAX_VALUE;
      int mindex = 0;
      for (int i = 0; i <= n; i++) {
          if (a[i] < m) { mindex =i; m = a[i]; } }
          n- -;
      a[mindex] = a[n];
      return m;
}
```

Warning: **Possible null dereference (Null)** **Index negative or too large (IndexNegative)** **Già risolto!**

Esempio con ESC/Java2

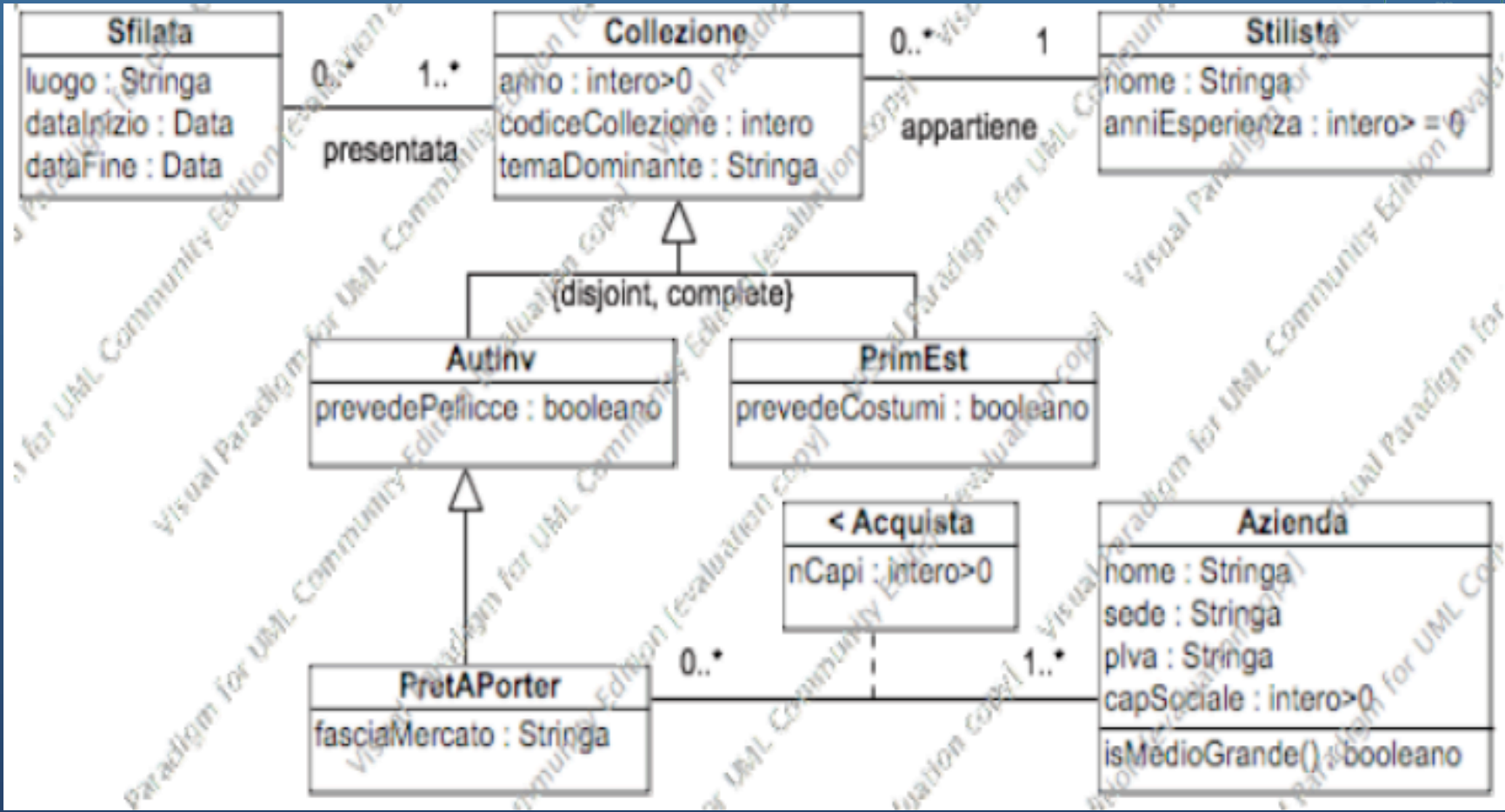
- Con ESC/Java2 non sono considerati possibili warnings sul numero di chiamate di extractMin() che potrebbero non assicurare la precondizione!

Un progetto documentato con JML: Fashion District

- JML può essere usato come valido supporto nella realizzazione di un progetto.
- Dopo la raccolta dei requisiti, può accompagnare le fasi di:
  - Analisi
  - Progetto
  - Realizzazione
- Fashion District permette ad un'agenzia di moda di gestire le collezioni presentate nelle sfilate ed effettuare indagini di mercato

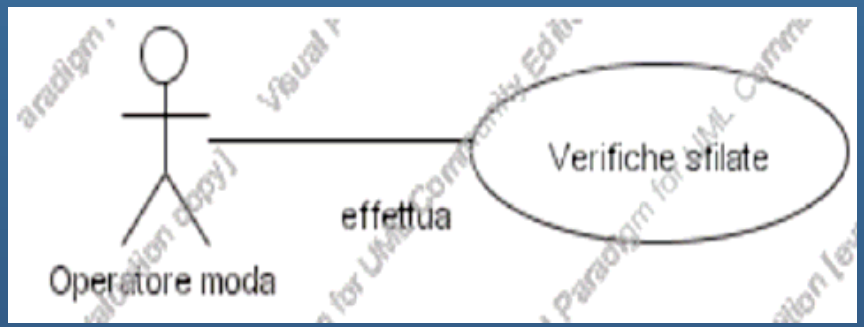
Fase di Analisi

Diagramma delle classi



Fase di Analisi

Diagramma degli use case



SpecificaUseCase VerificheSfilate
verificaEsordienti(s : Sfilata) :
booleano
pre: nessuna
post: Sia C l'insieme degli oggetti di classe Collezione coinvolti in link di tipo
'presentata' con s, ovvero:
C = { c in Collezione | <s, c> in presentata }.
Sia inoltre
T = { t in Stilista | <c,t> in appartiene e c in C }
l'insieme degli oggetti di classe Stilista coinvolti in link
di tipo appartiene con oggetti in C.
r vale true se e solo se
($\sum_{(t \in T)}$ t.anniEsperienza)/|T| < 5.

calcolaVendite(c : Collezione): intero >= 0
pre: nessuna
post: Se c non e' di classe PretAPorter, allora result = 0.
Altrimenti (c e' di classe PretAPorter), detto
L = { <c,a> in c.acquista | a.isMe $\sum_{(l \in L)}$ ande() = true }
result e a
InLICapi

Fase di Analisi

Specifica per il diagramma delle classi

SpecificaClasse Azienda
isMedioGrande() : booleano
pre: nessuna
post: result = true se e solo se this.capSociale > 10 000 000.
FineSpecifica

- Traducibile in JML:

```
/*@ requires capSociale > 10000000;
    ensures \result == true;
    also
    requires capSociale <= 10000000;
    ensures \result == false;
@*/
```

Fase di Progetto

- Scelte di progetto
- Il tipo UML intero>0 tradotto con un semplice “int” in Java richiede la verifica lato server che il valore intero sia effettivamente >0.
- Questa verifica può essere garantita in JML con l’introduzione di un invariante preposto a tale scopo.
- Nel nostro progetto:
  - //@ invariant capSociale>0;
  - //@ invariant anniEsp>=0;

Tipo UML	Tipo Java	Note
Stringa	String	-
Data	Data	Versione senza side effect
intero>0	int	Verifica lato server
intero	int	-
booleano	boolean	-
Insieme	HashSet	Implementa l’interfaccia Set

Fase di Progetto

Specifica realizzativa degli use case

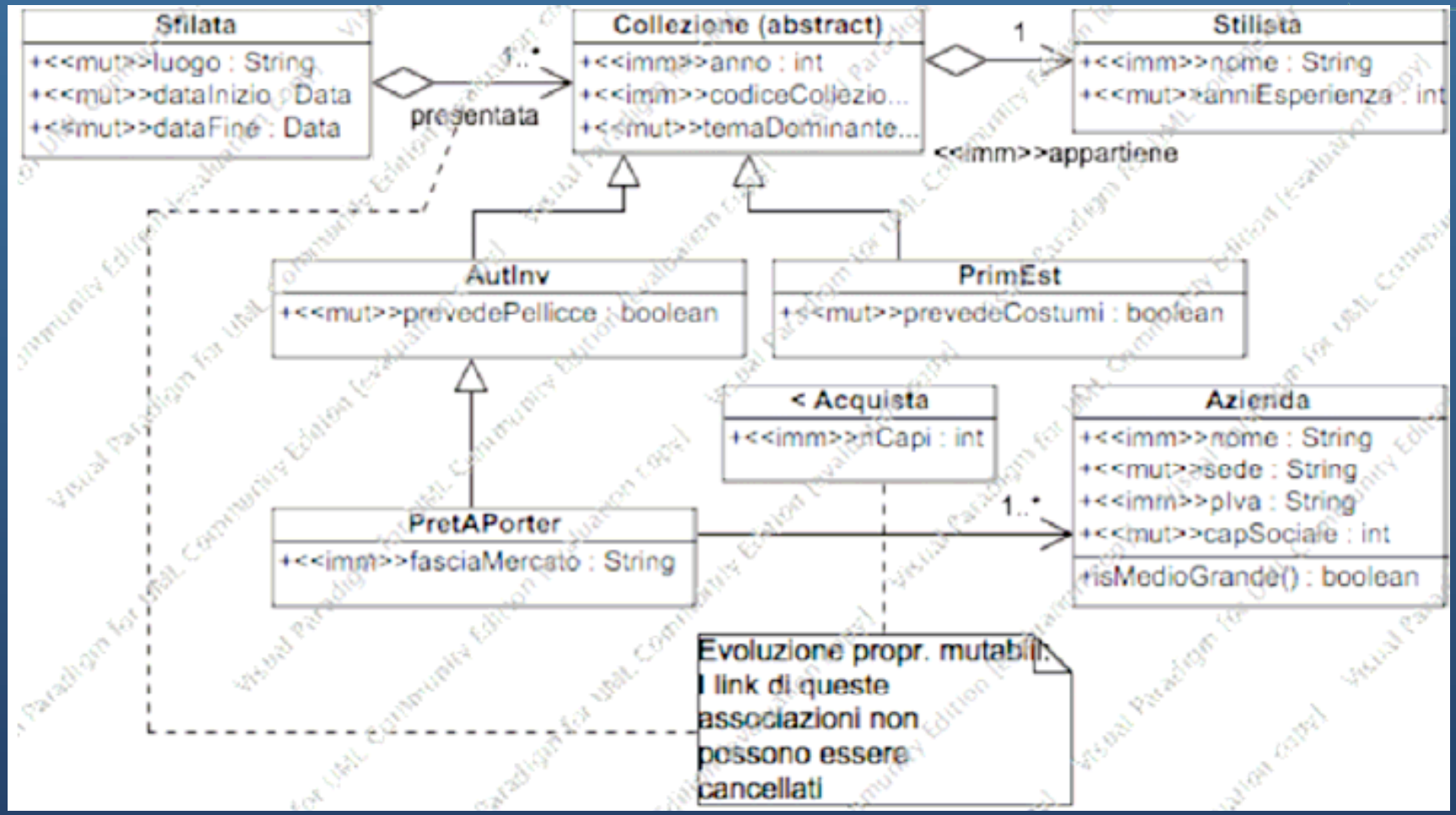
```
+verificaEsordienti(s : Sfilata) : boolean
pre: nessuna
post: algoritmo:
        somma = 0;
        quanti = 0;
        per ogni 'l' in s.presentata {
        somma = somma +
            l.Collezione.appartiene.Stilista.anniEsperienza;
        }
        ritorna ( (somma/|s.presentata|) < 5 );
+calcolaVendite(c : Collezione): int
pre: nessuna
post: Se c non e' di classe PretAPorter, allora ritorna 0.
        Altrimenti (c e' di classe PretAPorter) {
                result = 0;
                per ogni 'l' in c.acquista {
                se l.Azienda.isMedioGrande() = true {
                result += l.nCapi;
                } } }
                ritorna result;
```

```
/*@ requires s!= null;
      ensures (* \result == verifica
      se la media degli anni
      d'esperienza degli stilisti< 5
      *);
@*/
```

```
/*@ requires c!= null &&
PretAPorter.class.isInstance(c);
          ensures (* \result ==
          numero di capi
          acquistati da aziende
          con capitale maggiore di
          10 milioni *);
also
requires requires c!= null && !
PretAPorter.class.isInstance(c);
ensures \result ==0;
@*/
```

Fase di Progetto

Diagramma delle classi realizzativo



Fase di Realizzazione

- Alla luce di quanto visto nelle fasi di Analisi e di Progetto, nella fase di Realizzazione:
  - Gli attributi di tipo Stringa o classi Java, definiti come immutabili, verranno dichiarati non_null
  - Gli attributi con visibilità bassa, ma utili nelle specifiche vanno dichiarati /*@ spec_public @*/
  - Tutti i metodi che non effettuano side effect vanno dichiarati /*@ pure @*/
  - Nel caso di eccezioni si usa la clasusola signals_only o signals.
  - Le variabili modificate da un metodo vanno segnalate come "assignable"
  - Per particolari proprietà facciamo uso di "invariant" (es.intero compreso in un particolare intervallo)

Fase di Realizzazione

Mostriamo ad esempio la classe Stilista.java:

```
public class Stilista {
private /*@ spec_public non_null @*/ final String nome;
private /*@ spec_public @*/ int anniEsp;
//@ invariant anniEsp>=0;
/*@ requires n!=null;
assignable nome,anniEsp;
ensures (nome == n) && (anniEsp == a);
signals_only EccezionePrecondizioni;
@*/
public Stilista(String n, int a) throws EccezionePrecondizioni {
if (n==null) throw new EccezionePrecondizioni("Il nome non puo' essere null");
nome=n;
setAnniEsperienza(a);
}
public /*@ pure @*/ String getNome() { return nome; }
public /*@ pure @*/ int getAnniEsperienza() { return anniEsp; }
public /*@ pure @*/ String toString() {
return nome + " (" + anniEsp + " anni di esperienza)";
}
```

Fase di Realizzazione


```
/*@
requires a>=0;
assignable anniEsp;
ensures anniEsp == a;
signals_only EccezionePrecondizioni;
@*/
public void setAnniEsperienza(int a) throws EccezionePrecondizioni {
if (a<0) throw new EccezionePrecondizioni("a deve essere positivo o nullo");
anniEsp=a;
```

Altri Tool che usano JML

Tool per lo static checking

Tool che effettuano il controllo statico dei programmi:

- LOOP
- JACK

LOOP

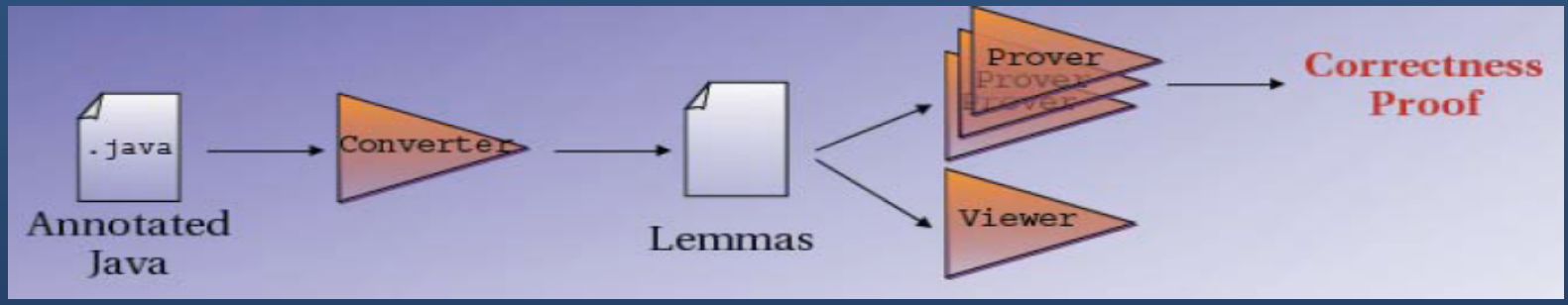
- Logic of Object-Oriented Programming
- Verifica proprietà JML opportunamente trasformate e date in pasto al prover PVS, che presenta un output di difficile interpretazione
- Per ogni metodo da verificare viene creata una “proof obligation”
- PVS è caratterizzato da un linguaggio di specifica basato su logica higher-order.
- L’idea di base consiste nella definizione di “proof” per la cui dimostrazione fa uso del calcolo dei sequenti

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ \mathrm{R.}$$

- Si parte dal nodo radice $\Gamma \vdash \Delta$ applicando la regola R ed ottenendo nuovi sequenti

JACK

- Java Applet Correctness Kit
- Disponibile come plugin di Eclipse
- E' una via di mezzo tra ESC/Java2 e LOOP
- Parte da annotazioni JML ed implementa un calcolo di precondizioni più deboli "weakest precondition" verificate da prover automatici
- I passi compiuti da JACK sono:
  - Conversione di file Java annotati con specifiche JML in lemmi
  - Uso di prover automatici (tra cui Simplify) per verificare la correttezza dei lemmi
  - Uso di viewer dei lemmi che ne nascondono la complessità sottostante



Tool per la generazione di specifiche

Tool che tentano di inferire automaticamente le specifiche del codice:

- Daikon
- Houdini

Daikon

- Realizzato presso i laboratori del MIT
- Genera in maniera automatica specifiche per il codice in input
- Trova in maniera dinamica invarianti in punti specifici del codice
- Crea dei profili d'esecuzione del programma, riporta proprietà che risultano vere, osserva i valori calcolati, li generalizza e restituisce proprietà
- Può presentare imprecisioni
- I valori calcolati vengono osservati in test e se ne verifica l'accuratezza
- L'accuratezza dipende dalla qualità e completezza dei test

Houdini

- Genera annotazioni JML
- Si basa sul seguente algoritmo:
  - Vengono generate annotazioni JML candidate
  - Viene eseguito più volte ESC/Java per eliminare annotazioni non consistenti con il codice
  - Quando tutte le annotazioni sono consistenti, viene eseguito ESC/Java un'ultima volta. Viene restituito l'output con gli eventuali warning
- Houdini lavora su un insieme limitato di annotazioni
- Il numero dei possibili warnings è piuttosto limitato

Runtime Assertion Checker: Jass

- Acronimo di Java with ASSertions
- Precompilatore che supporta asserzioni Java in file .jass o .java
- Viene prodotta una classe valida Java, compilabile con il tradizionale javac
- Diversi controlli: Pre, Post, Inv, Check, Forall, Trace, Opt

- Dalla versione 3 Jass fa uso delle asserzioni JML
- Permette tracciamento delle asserzioni permettendo di monitorare il comportamento dinamico di un oggetto, le chiamate e l'ordine di invocazione dei metodi

JML + Proprietà temporali: JAG

- Generatore di annotazioni JML per verificare proprietà temporali in classi Java
- Formato da più traduttori che permettono di trasformare proprietà dinamiche in annotazioni JML
- Nelle prime versioni era usato come linguaggio di Input JTPL (Java Temporal Pattern Language), un adattamento a Java di LTL
- Permetteva di definire proprietà di safety e di liveness
- Tool ancora in fase di sviluppo

Conclusioni

- JML risulta semplice da imparare (sintassi e semantica molto simile a Java)
- Non necessita di specificare un modello formale, poiché il codice rappresenta il modello stesso .
- Può essere usato:
  - Come supporto nelle varie fasi di definizione e di realizzazione di interi progetti
  - Per commentare codice già esistente
- E' crescente l'interesse intorno a JML
- Ci sono molti tool di supporto che permettono tra l'altro:
  - Runtime checking: aiutando ad identificare possibili errori legati a possibili esecuzioni
  - Static Checking: più precisi e che assicurano la correttezza per qualsiasi esecuzione