**Università degli Studi di Bergamo**
**Facoltà di Ingegneria dell'Informazione**

**Informatica III**

# Elaborato 1: JML

| Revisione | Data | Redatto |
|---|---|---|
| 1.2 | 2006/02/06 | Silvio Moioli |

# Indice dei contenuti

## Storia delle revisioni

| Rev. n | Data | Redatto | Descrizione |
|---|---|---|---|
| 1.0 | 2006/02/06 | Silvio Moioli | Versione iniziale. |
| 1.1 | 2006/02/07 | Silvio Moioli | Aggiunti tutti i costrutti JML. |
| 1.2 | 2006/02/08 | Silvio Moioli | Aggiunti aspetti realizzativi e diagramma UML. |

# Introduzione

## Scopo del documento

Presentazione del primo elaborato del corso, applicazione della metodologia "Design by contract" tramite il linguaggio JML e gli strumenti associati.

## Definizioni, abbreviazioni e sigle

- JML: Java Modeling Language;
- UML: Unified Modeling Language;
- IDE: Integrated Development Environment;

## Riferimenti bibliografici e Web

- http://www.cs.ucf.edu/~leavens/JML/, homepage del progetto JML;
- http://www.eclipse.org, homepage dell'IDE Eclipse;
- http://ant.apache.org, homepage dello strumento Apache Ant;

## Approccio seguito in questo elaborato

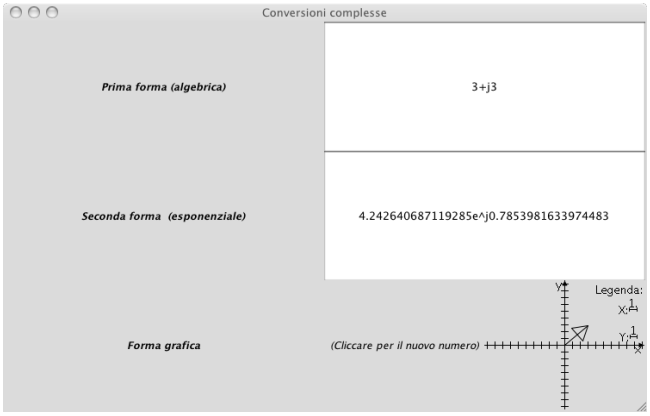
L'elaborato consiste nell'aggiunta di contratti JML ad un programma Java sviluppato precedentemente. Si è preferito questo approccio rispetto alla stesura di un programma completamente nuovo in quanto si ritiene che in questo modo si possano identificare meglio i problemi pratici all'utilizzo dello strumento.

I contratti sono stati posti nelle classi giudicate più critiche per il corretto funzionamento del programma allo scopo di aumentare il grado di confidenza sulla loro correttezza, in particolare evidenziando anomalie altrimenti più difficili da scoprire. La scelta di prendere in considerazione soltanto le classi principali è stata presa sia per limitare le dimensioni dell'elaborato sia perché, probabilmente, questo è lo scenario di utilizzo più comune per JML.

## Presentazione del programma

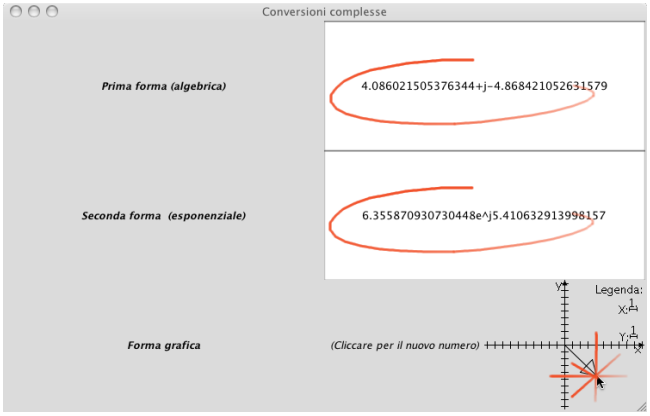
## Funzionalità

Il programma oggetto dell'elaborato è un'interfaccia didattica per lo studio dei numeri complessi. In particolare, dalla schermata principale è possibile convertire un numero complesso nelle tre forme generalmente utilizzate in Analisi e nell'Elettrotecnica: rappresentazione algebrica, esponenziale (modulo e fase) e grafica (attraverso un vettore).




Modificando il contenuto di una delle caselle di testo, si aggiorna automaticamente il contenuto dell'altra e il grafico in basso. In caso di errore di digitazione, la casella di testo diventa rossa.


Infine, cliccando in un punto del grafico, il programma calcola e aggiorna le rappresentazioni esponenziale e algebrica.

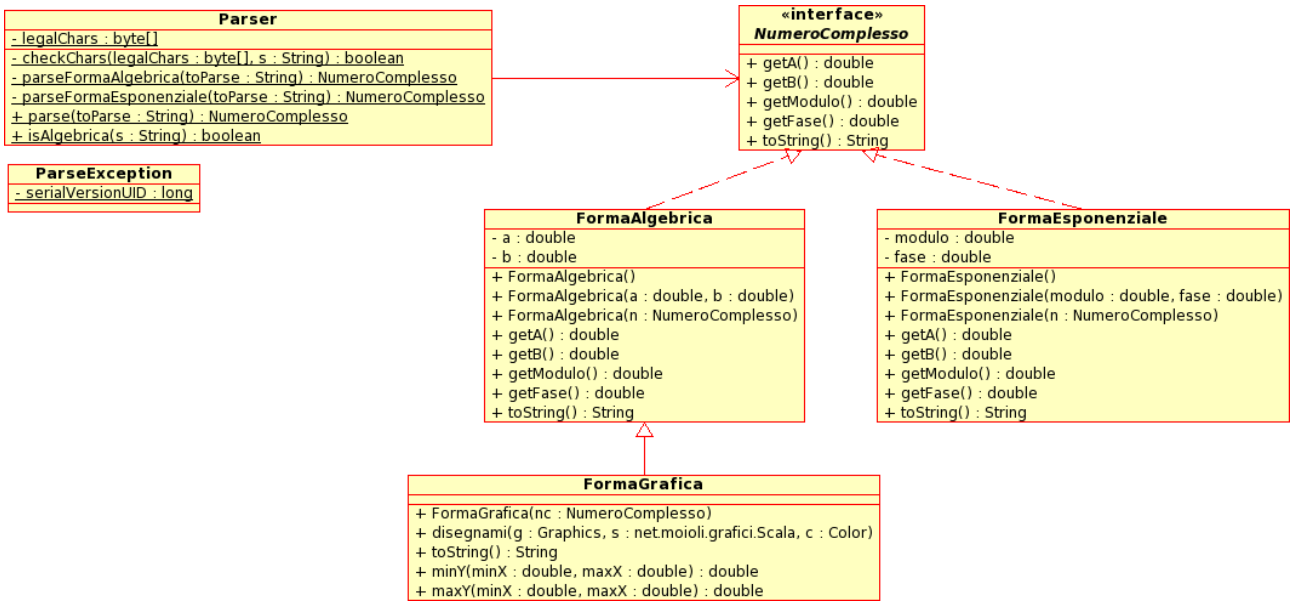



### Classi principali

Le classi che compongono il programma possono essere suddivise in tre categorie:

- un insieme di classi, tra cui `Main`, che si occupano dell'interfaccia Swing e dell'interazione con l'utente;
- un secondo insieme di classi, raggruppate nel package `net.moioli.grafici`, che si occupano del ridisegno del grafico, della legenda e del vettore;
- un terzo insieme che costituisce il modello dei dati (classi `NumeroComplesso`, `FormaAlgebrica`, ecc.)

Ai fini di questo elaborato le classi più interessanti sono quelle del terzo insieme, descritte sinteticamente nel seguente diagramma UML.




## Esempi dei contratti JML

Viene riportato un esempio di contratto per ognuno dei costrutti JML principali.

### Metodi puri

In JML un metodo è puro (indicato dalla parola chiave `pure`) quando non modifica lo stato dell'oggetto sul quale è invocato. Annotare un metodo come puro è utile per utilizzare il suo risultato in altri contratti (JML vieta l'uso di metodi non-puri nei contratti).

Un esempio è il metodo `getA()` della classe `FormaAlgebrica`, che ritorna la parte reale del numero complesso rappresentato dall'oggetto relativo:

```
/**
 * Rappresenta un numero complesso nella forma "a+ib".
 */
public class FormaAlgebrica implements NumeroComplesso {


...


/**
 * Ritorna la parte reale del numero complesso, ovvero il coefficiente
"a"
 * nella forma "a+ib". Implementa NumeroComplesso.
 *
 * @return la parte reale
 */
public /*@ pure @*/ double getA() {
      return a;
}
```

### Precondizioni

JML permette di definire le precondizioni di un metodo tramite la parola chiave `requires`.

Un esempio è questa definizione del costruttore di `FormaAlgebrica`:

```
/**
 * Crea un nuovo numero complesso le cui coordinate vengono passate dal
 * chiamante.
 *
 * @param a la coordinata x del numero (parte reale)
 * @param b la coordinata y del numero (parte immaginaria)
 */
/*@
  @ requires a != Double.NaN;
  @ requires b != Double.NaN;
  @*/
public FormaAlgebrica(double a, double b) {...
```

### Postcondizioni

Analogamente al punto precedente, JML permette di specificare anche postcondizioni utilizzando la parola chiave `ensures`. Il costruttore precedente può quindi essere arricchito come segue:

```
/**
 * Crea un nuovo numero complesso le cui coordinate vengono passate dal
 * chiamante.
 *
 * @param a la coordinata x del numero (parte reale)
```

```
 * @param b la coordinata y del numero (parte immaginaria)
 */
/*@
  @ requires a != Double.NaN;
  @ requires b != Double.NaN;
  @ ensures this.a == a;
  @ ensures this.b == b;
  @*/
public FormaAlgebrica(double a, double b) {
```

### Valori ritornati nelle postcondizioni

Se è necessario controllare il valore ritornato da un metodo nelle postcondizioni di un contratto, è possibile riferirvisi con la parola chiave `\return`, come nell'esempio qui riportato, che assicura che il risultato del metodo `getFase()` sia un numero nell'intervallo $[0, 2\pi]$.

```
/**
 * Rappresenta un numero complesso nella forma "a+ib".
 */
public class FormaAlgebrica implements NumeroComplesso {
    /**
    * Ritorna la fase del numero complesso, ovvero il valore dell'espressione
     * "arctg(b/a)" nella forma "a+ib". Implementa NumeroComplesso.
     *
     * @return la fase
     */
    /*@
      @ also ensures 0 <= \result && \result < 2 * Math.PI;
      @*/
    public /*@ pure @*/ double getFase() {...
```

### Precondizioni e postcondizioni su campi privati

Per effettuare i controlli runtime, le classi di JML devono avere accesso ai membri della classe a cui si applicano i contratti. Se tali membri sono privati, è necessario specificarlo con l'apposita parola chiave `spec_public` come nell'esempio seguente, che si riferisce ancora alla classe `FormaAlgebrica`.

```
/**
 * Rappresenta un numero complesso nella forma "a+ib".
 */
public class FormaAlgebrica implements NumeroComplesso {
      /** Parte reale del numero complesso. */
      private /*@ spec_public @*/ double a;

      /** Parte immaginaria del numero complesso. */
      private /*@ spec_public @*/ double b;
```

### Invarianti

Si dicono invarianti le proposizioni che, per la correttezza del programma, devono valere prima e dopo ogni chiamata a metodo (eccettuato, al più, il costruttore). In JML gli invarianti si definiscono tramite la parola chiave `invariant`.

Nell'esempio qui riportato, preso dalla classe `Grafico`, l'invariante assicura che il riferimento `c` non sia mai `null`:

```
/**
 * Decorator per CompatibleCollection, permette di disegnare una collezione di
 * oggetti disegnabili.
 */
public class Grafico implements CompatibleCollection, IGrafico {


/** Collezione interna degli oggetti da disegnare. */
/*@ invariant c != null; @*/
public /*@ non_null @*/ CompatibleCollection c = new CollectionWrapper(new
LinkedList());
```

Gli oggetti di tipo `Grafico` sono sostanzialmente collezioni di oggetti che implementano `Disegnabile`, tra cui è sempre presente una `Legenda` e due `AssiCartesiani`. Dal punto di vista dei design pattern `Grafico` è un Decorator della classe `CompatibleCollection`, che a sua volta è un Wrapper di `Collection`. La necessità di definire quest'ultima classe deriva da un'incompatibilità di un metodo in `Collection` con `JPanel`, la classe base degli oggetti Swing.

### Invarianti per campi privati

Si possono definire anche invarianti per campi privati, ma con una sintassi leggermente diversa (la motivazione è spiegata al paragrafo precedente).

E' qui riportato un altro esempio tratto da `Grafico`, che controlla che il massimo e il minimo valore delle ascisse rappresentati siano sempre coerenti.

```
/**
 * Decorator per CompatibleCollection, permette di disegnare una collezione di
 * oggetti disegnabili.
 */
public class Grafico implements CompatibleCollection, IGrafico {
      /** Indica il minimo valore della x rappresentato */
      private double minX = -10;


      /** Indica il massimo valore della x rappresentato */
      /*@ private invariant maxX>minX; @*/
      private double maxX = 10;
```

### Contratti di sottoclassi

Se una classe eredita da un'altra e ridefinisce metodi che hanno già un un contratto JML è necessario utilizzare la parola chiave `also` per indicare che il nuovo contratti si aggiunge ai vecchi. Questo vale anche per le classi che implementano interfacce già coperte da contratti.

Ad esempio `FormaAlgebrica` implementa la classe `NumeroComplesso` definendo il corpo del metodo `getModulo()`, quindi i contratti possono essere scritti come segue:

```
/**
 * Interfaccia comune a tutte le rappresentazioni dei numeri complessi.
 */
public interface NumeroComplesso {
    ...
    /**
       * Ritorna il modulo del numero complesso, ovvero il valore
dell'espressione
       * "sqrt(a^2+b^2)" nella forma "a+ib".
       *
       * @return il modulo
       */
      public /*@ pure @*/ double getModulo();
      ...


/**
 * Rappresenta un numero complesso nella forma "a+ib".
 */
public class FormaAlgebrica implements NumeroComplesso {
    ...
    /**
       * Ritorna il modulo del numero complesso, ovvero il valore
dell'espressione
       * "sqrt(a^2+b^2)" nella forma "a+ib".
       *
       * @return il modulo
       */
      /*@
        @ also ensures Math.abs(\result * \result - (a * a + b * b))<0.001;
        @*/
      public double getModulo() {
            return Math.sqrt(a * a + b * b);
      }
```

La parola chiave `also` è richiesta e il compilatore di JML si rifiuterà di compilare la classe se non è presente.

### Metodi di utilità dalle classi della libreria di JML

Nell'esempio precedente si è visto un modo per controllare che un risultato numerico sia approssimativamente uguale a un altro:

```
/*@ also ensures Math.abs(\result * \result - (a * a + b * b))<0.001; @*/
```

Esiste anche un modo più compatto per controllare la stessa condizione, e richiede l'uso del metodo `approximatelyEqualTo` della classe `JMLDouble`. Quest'ultima è una delle classi dispobnibili nella libreria di JML, e va importata regolarmente come mostrato dall'esempio seguente.

```
import org.jmlspecs.models.JMLDouble;


/**
 * Rappresenta un numero complesso nella forma "a+ib".
 */
public class FormaAlgebrica implements NumeroComplesso {
    ...
    /**
       * Ritorna il modulo del numero complesso, ovvero il valore
dell'espressione
       * "sqrt(a^2+b^2)" nella forma "a+ib".
       *
       * @return il modulo
       */
      /*@
      @ also ensures JMLDouble.approximatelyEqualTo(\result * \result, a * a + b
* b, 0.001);
        @*/
      public /*@ pure @*/ double getModulo() {...
```

### Doppia implicazione

Nei contratti JML oltre agli operatori logici standard di Java (`&&` e `||`) si può usare anche la doppia implicazione (`<==>`) ove opportuno. Ad esempio, il seguente contratto sul metodo getFase() specifica che il valore ritornato deve essere compreso tra 0 e π/2 se il numero complesso ha parte reale e immaginaria positive (il vettore si trova nel primo quadrante).

```
/**
 * Ritorna la fase del numero complesso, ovvero il valore dell'espressione
 * "arctg(b/a)" nella forma "a+ib".
 *
 * @return la fase
 */
/*@
  @ also ensures 0 <= \result && \result < 2 * Math.PI;
  @ also ensures (0 <= \result && \result <= Math.PI / 2) <==> (a>=0 && b>=0);
  @*/
public /*@ pure @*/ double getFase() {...
```

### Quantificatori (esistenziale e universale)

JML supporta anche i quantificatori ( ∃ e ∀ ) tramite le parole chiave `\exists` e `\forall`. Nell'esempio seguente il quantificatore universale viene utilizzato per assicurare la buona riuscita del metodo `addAll()`, che aggiunge tutti gli elementi di una collezione.

```
/*@
  @ also requires (\forall Object x; arg0.contains(x); !this.contains(x));
  @ also ensures (\forall  Object x; arg0.contains(x); this.contains(x));
  @*/
public boolean addAll(Collection arg0) {
      return this.c.addAll(arg0);
}
```

### Non-nullità di variabili e parametri

In JML è anche possibile specificare che una variabile o un parametro non assumano mai valore `null`. Per farlo è sufficiente aggiungere la parola chiave `non_null`, come mostrato nell'esempio seguente tratto dalla classe `Grafico`.

```
/*@
  @ also requires !this.contains(arg0);
  @ also ensures this.contains(arg0);
  @ also ensures c.getCollectionSize() == \old(c.getCollectionSize()) + 1;
  @*/
public boolean add(/*@ non_null @*/ Object arg0) {
      return c.add(arg0);
}
```

### Confrontare variabili con il valore assunto prima dell'esecuzione di un metodo

In alcuni casi, le postcondizioni di un metodo richiedono la conoscenza del valore che le variabili d'istanza o i parametri hanno assunto prima dell'esecuzione del metodo stesso. In questi casi è possibile "recuperare" il vecchio valore con la parola chiave `\old` all'interno del contratto.

```
/*@
  @ also ensures c.getCollectionSize() == \old(c.getCollectionSize()) + 1;
  @*/
public boolean add(/*@ non_null @*/ Object arg0) {
      return c.add(arg0);
}
```

### Asserzioni

Le asserzioni permettono di verificare arbitrarie condizioni in punti qualsiasi del codice. JML permette anche le asserzioni, che si possono esprimere come normali condizioni all'interno di commenti `/*@@*/` o `//@`.

```
/**
 * Disegna il grafico sull'oggetto Graphics passato
 *
 * @param g l'oggetto su cui disegnare
 */
public void disegnami(/*@ non_null @*/ Graphics g, /*@ non_null @*/ Rectangle
clip) {
      // Imposta i settaggi per il rendering
```

```
      //@ assert g instanceof Graphics2D;
      Graphics2D g2d = (Graphics2D) g;
```

# Aspetti implementativi

## Compilazione automatica dei contratti con Apache Ant, Eclipse e jmlrac

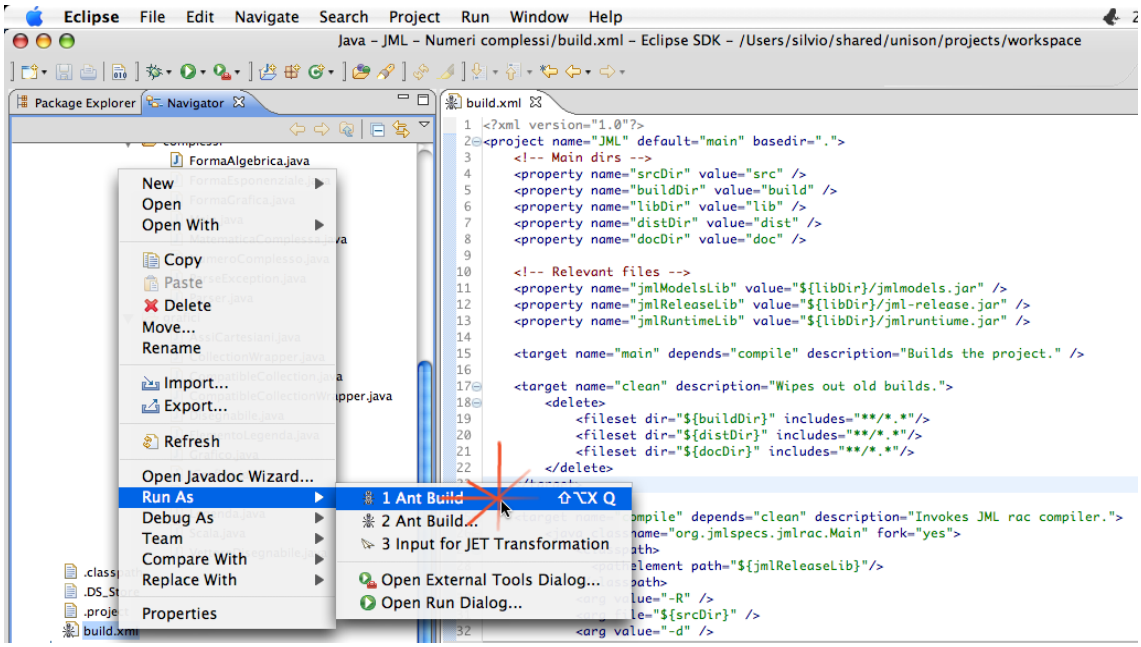
Per compilare i sorgenti con contratti JML è necessario l'utilizzo di un apposito strumento, `jmlrac`, che è disponibile nel pacchetto della distribuzione sia con interfaccia grafica che a linea di comando. Per velocizzare il processo di compilazione e renderlo più agevole, si è deciso di integrarlo nell'ambiente Eclipse tramite uno script di Apache Ant.

Lo script, in pratica, richiama la versione a linea di comando di `jmlrac` con i parametri corretti, ed è facilmente adattabile ad altri programmi.

```
<?xml version="1.0"?>
<project name="JML" default="main" basedir=".">
      <!-- Main dirs -->
      <property name="srcDir" value="src" />
      <property name="buildDir" value="build" />
      <property name="libDir" value="lib" />
      <property name="distDir" value="dist" />
      <property name="docDir" value="doc" />

      <!-- Relevant files -->
      <property name="jmlModelsLib" value="${libDir}/jmlmodels.jar" />
      <property name="jmlReleaseLib" value="${libDir}/jml-release.jar" />
    <property name="jmlRuntimeLib" value="${libDir}/jmlruntiume.jar" />

      <target name="main" depends="compile" description="Builds the project." />

      <target name="clean" description="Wipes out old builds.">
            <delete>
                <fileset dir="${buildDir}" includes="**/*.*"/>
                <fileset dir="${distDir}" includes="**/*.*"/>
                <fileset dir="${docDir}" includes="**/*.*"/>
            </delete>
      </target>

      <target name="compile" depends="clean" description="Invokes JML rac
compiler.">
            <java classname="org.jmlspecs.jmlrac.Main" fork="yes">
                  <classpath>
                    <pathelement path="${jmlReleaseLib}"/>
                  </classpath>
                  <arg value="-R" />
```

```
                <arg file="${srcDir}" />
                <arg value="-d" />
                <arg file="${buildDir}" />
            </java>
        </target>
</project>
```

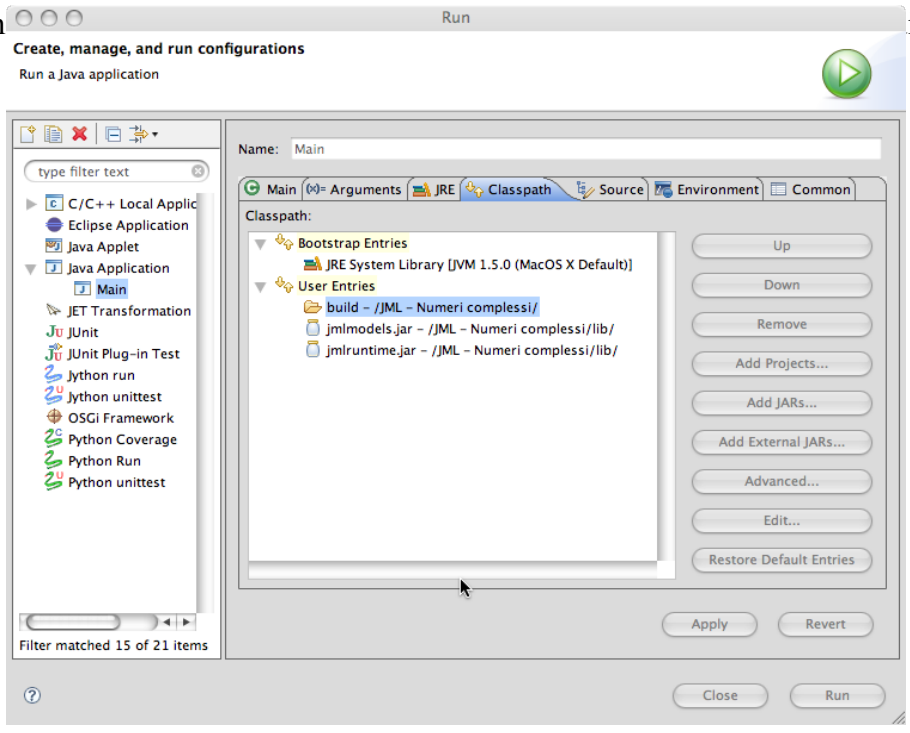
Utilizzare lo script è molto semplice: in Eclipse, infatti, basta scegliere la voce “Run as Ant build” dal menù contestuale oppure utilizzare la combinazione `Shift-Alt-X Q`.




## Esecuzione del programma da Eclipse con controllo dei contratti abilitato

Una volta compilato, il programma deve essere lanciato con le librerie adeguate per essere eseguito con il supporto di JML per permettere il controllo dei contratti a tempo di esecuzione. In particolare, è necessario che le librerie jmlmodels.jar e jmlruntime.jar siano presenti nel classpath di Java: anche questo è ottenibile con Eclipse a partire dalla finestra di dialogo “Run As” come mostrato nelle schermate seguenti.

Da notare l'in i al paragrafo precedente.



### Risoluzione di errori con JML

Grazie alla stesura dei contratti sono stati individuati e risolti diversi difetti nel programma. Ad esempio, nel metodo `getFase()` di `FormaAlgebrica`, che calcola l'angolo di fase di un numero complesso nella forma "a+ib", non era eseguita correttamente la riduzione al primo periodo dell'angolo.

Vengono riportati qui di seguito due esempi di stack trace con contratti violati.

Violazione di una postcondizione in `FormaEsponenziale`:

```
Exception in thread "AWT-EventQueue-0"
org.jmlspecs.jmlrac.runtime.JMLInvariantError: by method
FormaEsponenziale.<init>@post<File
"src/net/moioli/complessi/FormaEsponenziale.java", line 44, character 16>
      at
net.moioli.complessi.FormaEsponenziale.checkInv$instance$FormaEsponenziale(Forma
Esponenziale.java:346)
      at
net.moioli.complessi.FormaEsponenziale.<init>(FormaEsponenziale.java:172)
      at
net.moioli.complessi.Parser.internal$parseFormaEsponenziale(Parser.java:92)
      at net.moioli.complessi.Parser.parseFormaEsponenziale(Parser.java:753)
      at net.moioli.complessi.Parser.internal$parse(Parser.java:104)
      at net.moioli.complessi.Parser.parse(Parser.java:931)
      at net.moioli.complessi.Main.internal$jTextField2KeyPressed(Main.java:227)
      at net.moioli.complessi.Main.jTextField2KeyPressed(Main.java:1006)
      at
net.moioli.complessi.Main.internal$jTextField2KeyReleased(Main.java:216)
      at net.moioli.complessi.Main.jTextField2KeyReleased(Main.java:658)
      at net.moioli.complessi.Main.access$5(Main.java)
      at net.moioli.complessi.Main$3.keyReleased(Main.java:186)
      at java.awt.Component.processKeyEvent(Component.java:5515)
      at javax.swing.JComponent.processKeyEvent(JComponent.java:2713)
      at java.awt.Component.processEvent(Component.java:5331)
      at java.awt.Container.processEvent(Container.java:2010)
      at java.awt.Component.dispatchEventImpl(Component.java:4021)
      at java.awt.Container.dispatchEventImpl(Container.java:2068)
      at java.awt.Component.dispatchEvent(Component.java:3869)
      at
java.awt.KeyboardFocusManager.redispatchEvent(KeyboardFocusManager.java:1810)
      at
java.awt.DefaultKeyboardFocusManager.dispatchKeyEvent(DefaultKeyboardFocusManage
r.java:672)
      at
java.awt.DefaultKeyboardFocusManager.preDispatchKeyEvent(DefaultKeyboardFocusMan
ager.java:920)
      at
java.awt.DefaultKeyboardFocusManager.typeAheadAssertions(DefaultKeyboardFocusMan
ager.java:798)
```

```
	at
java.awt.DefaultKeyboardFocusManager.dispatchEvent(DefaultKeyboardFocusManager.j
ava:636)
	at java.awt.Component.dispatchEventImpl(Component.java:3907)
	at java.awt.Container.dispatchEventImpl(Container.java:2068)
	at java.awt.Window.dispatchEventImpl(Window.java:1774)
	at java.awt.Component.dispatchEvent(Component.java:3869)
	at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
	at
java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:2
69)
	at
java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:190
)
	at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:184)
	at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:176)
	at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
```

Violazione di una postcondizione in `Grafico`:

```
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError: by method
Grafico.add
	at net.moioli.grafici.Grafico.add(Grafico.java:3502)
	at net.moioli.grafici.Grafico.internal$$init$(Grafico.java:541)
	at net.moioli.grafici.Grafico.<init>(Grafico.java:60)
	at net.moioli.grafici.JGrafico.Block$(JGrafico.java:17)
	at net.moioli.grafici.JGrafico.<init>(JGrafico.java:20)
	at net.moioli.complessi.Main.Block$(Main.java:20)
	at net.moioli.complessi.Main.<init>(Main.java:33)
	at net.moioli.complessi.Main.internal$main(Main.java:303)
	at net.moioli.complessi.Main.main(Main.java:1696)
```