Tecnología de la Programación

JML

Departamento de Computación

Facultad de Informática
Universidade da Coruña

Curso 2006/2007

JML

- Java Modeling Language.
  - A formal behavioral interface specification language for Java.
- Diseño por contrato en Java.
  - Precondiciones
  - Postcondiciones
  - Invariantes de clase
- `http://www.jmlspecs.org`
- Hasta Java 1.4.
- Los contratos se incrustan en el código en forma de anotaciones/aserciones:

  ```
  //@ keyword aserción

  /*@ ... @*/
  ```

- La aserción combina expresiones java (sin efectos colaterales) y términos lógicos.
- JML convierte los contratos en código java que los verifican en tiempo de ejecución.

JML

- Java Modeling Language.
  - A formal behavioral interface specification language for Java.
- Diseño por contrato en Java.
  - Precondiciones
  - Postcondiciones
  - Invariantes de clase
- `http://www.jmlspecs.org`
- Hasta Java 1.4.
- Los contratos se incrustan en el código en forma de anotaciones/aserciones:

  `//@` *keyword aserción*

  `/*@ ... @*/`

- La aserción combina expresiones java (sin efectos colaterales) y términos lógicos.
- JML convierte los contratos en código java que los verifican en tiempo de ejecución.

JML

- Java Modeling Language.
  - A formal behavioral interface specification language for Java.
- Diseño por contrato en Java.
  - Precondiciones
  - Postcondiciones
  - Invariantes de clase
- `http://www.jmlspecs.org`
- Hasta Java 1.4.
- Los contratos se incrustan en el código en forma de anotaciones/aserciones:

  `//@` *keyword aserción*

  `/*@ ... @*/`

- La aserción combina expresiones java (sin efectos colaterales) y términos lógicos.
- JML convierte los contratos en código java que los verifican en tiempo de ejecución.

Ejemplo

- Calcular el máximo de x e y.

```
if (x >= y)
    z = x;
else
    z = y;
//@ assert z >= x && z >= y;
```

Problema

- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: `z = x + y + 1` es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: `z = x*x + y*y`.
- Especificación correcta:

```
//@ assert z >= x && z >= y &&
(z == x || z == y);
```

Problema

- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: `z = x + y + 1` es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: `z = x*x + y*y`.
- Especificación correcta:

```
//@ assert z >= x && z >= y &&
(z == x || z == y);
```

Problema

- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: `z = x + y + 1` es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: `z = x*x + y*y`.
- Especificación correcta:

```
//@ assert z >= x && z >= y &&
(z == x || z == y);
```

Problema

- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: `z = x + y + 1` es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: `z = x*x + y*y`.
- Especificación correcta:

```
//@ assert z >= x && z >= y &&
(z == x || z == y);
```

Anotar métodos

requires = precondición

ensures = postcondición

```
public final static double eps = 0.0001;

/*@ requires x >= 0.0;
  @ ensures JMLDouble.approximatelyEqualTo(x,
  @                \result * \result, eps);
  @*/
public double sqrt(double x) { ... }
```

Invariantes de clase

- Afectan a todos los métodos.

```
public class CuentaCorriente {
  public static final int MAX_BAL = 100000;
  private int _balance;
  /*@ invariant 0 <= _balance &&
                 _balance <= MAX_BAL
   @*/
```

Especificaciones formales.

- Expresiones Java.
  - No puede tener efectos colaterales (`++`,`--`,...)
  - Sólo llamadas a *métodos puros*:
    - Sin efectos colaterales.
    - Anotados como tales: `/*@ pure */ public int foo() { ...}`
- Formulas lógicas:
  - Operadores lógicos ($\Longrightarrow$, $\Longleftrightarrow$, ...).
  - Cuantificadores ($\forall$, $\exists$, ...).
- Variables especiales:
  - `\result` Valor de retorno del método.
  - `\old(x)` Valor inicial de la variable `x`.

Especificaciones formales.

- Expresiones Java.
  - No puede tener efectos colaterales (`++`,`--`,...)
  - Sólo llamadas a *métodos puros*:
    - Sin efectos colaterales.
    - Anotados como tales: `/*@ pure */ public int foo() { ...}`
- Formulas lógicas:
  - Operadores lógicos ($\Longrightarrow$, $\Longleftrightarrow$, ...).
  - Cuantificadores ($\forall$, $\exists$, ...).
- Variables especiales:
  - `\result` Valor de retorno del método.
  - `\old(x)` Valor inicial de la variable `x`.

Especificaciones formales.

- Expresiones Java.
  - No puede tener efectos colaterales (`++`,`--`,...)
  - Sólo llamadas a *métodos puros*:
    - Sin efectos colaterales.
    - Anotados como tales: `/*@ pure */ public int foo() { ...}`
- Formulas lógicas:
  - Operadores lógicos ($\Longrightarrow$, $\Longleftrightarrow$, ...).
  - Cuantificadores ($\forall$, $\exists$, ...).
- Variables especiales:
  - `\result` Valor de retorno del método.
  - `\old(x)` Valor inicial de la variable `x`.

Especificaciones formales.

- Expresiones Java.
  - No puede tener efectos colaterales (`++`,`--`,...)
  - Sólo llamadas a *métodos puros*:
    - Sin efectos colaterales.
    - Anotados como tales: `/*@ pure */ public int foo() { ...}`
- Formulas lógicas:
  - Operadores lógicos ($\Longrightarrow$, $\Longleftrightarrow$, ...).
  - Cuantificadores ($\forall$, $\exists$, ...).
- Variables especiales:
  - `\result` Valor de retorno del método.
  - `\old(x)` Valor inicial de la variable `x`.

Ejemplo

- Método de búsqueda binaria

```
/*@ requires a ! = NULL &&
  @    && (\forall int i;
  @             0 < i && i < a.length;
  @             a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  ...
}
```

Uso de JML

- Usar `jmlc` en lugar de `javac`.
- Usar `jmlrac` en lugar de `java`.
- Usar `jmldoc` en lugar de `javadoc`.
- Usar `jmlunit` en lugar de `junit`.

Ejemplo

```
public class Persona {
  private /*@ spec_public non_null @*/  String _nombre;
  private /*@ spec_public @*/ int _peso;
  //@ public invariant !_nombre.equals("") && _peso >= 0;

  /*@ also
    @ ensures \result != null;
    @*/
  public String toString() { ... }

  //@ ensures \result = _peso
  public int getPeso() { ... }

  //@ ensures kgs >= 0 && _peso == \old(kgs * _peso);
  public void añadirKgs(int kgs) { ... }

  /*@ requires !n.equals("");
    @ ensures n.equals(_nombre) && _peso == 0;
    @*/
  public Persona(/*@ non_null @*/ String n) { ... }
```

Ejemplo (corregido)

```
public class Persona {
  private /*@ spec_public non_null @*/  String _nombre;
  private /*@ spec_public @*/ int _peso;
  //@ public invariant !_nombre.equals("") && _peso >= 0;

  /*@ also
    @ ensures \result != null;
    @*/
  public String toString() { ... }

  //@ ensures \result = _peso
  public int getPeso() { ... }

  //@ requires kgs >= 0;
  //@ ensures _peso == \old(kgs * _peso);
  public void añadirKgs(int kgs) { ... }

  /*@ requires !n.equals("");
    @ ensures n.equals(_nombre) && _peso == 0;
    @*/
```

Cuantificadores

Ejemplo ∀

```
(\forall Student s;
     juniors.contains(s) ==> s.getAdvisor() != null)
```

- \forall
- \exists
- \sum
- \product
- \max
- \min
- \num_of

Ejemplo de \sum

sumaArray

```
/* sumaArray()
   Devuelve la suma de todos los elementos de
   un array de enteros.
 */
/*@
  @ requires array != null;
  @ ensures \result == (\sum int I; 0 <= I &&
  @                         I < array.length; array[I]);
  @ ensures array == \old(array) &&
  @     (\forall int I; 0 <= I &&
  @                     I < array.length;
  @                     array[I] == \old(array[I]));
  @*/
public int sumaArray(int[] array) { ... }
```

Especificaciones informales

- Sintaxis: `(* ... *)`
- Semántica: expresión booleana.
- Siempre se evalua a cierto.
- Uso: comentarios de las especificaciones.

```
@ ensures (* devuelve la suma de los elementos
del array *)
```

Especificaciones informales

- Sintaxis: `(* ... *)`
- Semántica: expresión booleana.
- Siempre se evalua a cierto.
- Uso: comentarios de las especificaciones.

```
@ ensures (* devuelve la suma de los elementos
del array *)
```

Especificaciones informales

- Sintaxis: `(* ... *)`
- Semántica: expresión booleana.
- Siempre se evalua a cierto.
- Uso: comentarios de las especificaciones.

```
@ ensures (* devuelve la suma de los elementos
del array *)
```

Especificación de alternativas

- Ejemplo: método `divide` para números enteros. Tiene dos comportamientos, cuando el divisor es mayor que cero o cuando el divisor es cero.
- Podríamos hacer
  `requires` divisor $> 0 \vee$ divisor $= 0$
  `ensures` divisor $> 0 \implies \ldots \wedge$ divisor $= 0 \implies \ldots$
- Podemos usar `also`.
  Une diferentes comportamientos.

Especificación de alternativas

- Ejemplo: método `divide` para números enteros. Tiene dos comportamientos, cuando el divisor es mayor que cero o cuando el divisor es cero.
- Podríamos hacer
  `requires` divisor $> 0 \vee$ divisor $= 0$
  `ensures` divisor $> 0 \implies \ldots \wedge$ divisor $= 0 \implies \ldots$
- Podemos usar `also`.
  Une diferentes comportamientos.

Especificación de alternativas

- Ejemplo: método `divide` para números enteros. Tiene dos comportamientos, cuando el divisor es mayor que cero o cuando el divisor es cero.
- Podríamos hacer
  `requires` divisor $> 0 \vee$ divisor $= 0$
  `ensures` divisor $> 0 \implies \ldots \wedge$ divisor $= 0 \implies \ldots$
- Podemos usar `also`.
  Une diferentes comportamientos.

Especificación de alternativas (II)

```
/*@ public normal_behavior
  @   requires divisor > 0;
  @   ensures divisor * \result <= dividendo &&
  @           divisor * (\result+1) > dividendo;
  @
  @ also
  @ public exceptional_behavior
  @   requires divisor == 0;
  @   signals (ArithmeticException);
  @*/
public int divide(int dividendo, int divisor)
  throws ArithmeticException
{ ... }
```

Invariantes de bucle

- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
  - Numérica (int ó long)
  - Mayor que cero al comienzo
- La cota se decrementa en cada iteración.
- Asegura la terminación.

Invariantes de bucle

- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
  - Numérica (int ó long)
  - Mayor que cero al comienzo
- La cota se decrementa en cada iteración.
- Asegura la terminación.

Invariantes de bucle

- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
  - Numérica (int ó long)
  - Mayor que cero al comienzo
- La cota se decrementa en cada iteración.
- Asegura la terminación.

Invariantes de bucle

- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
  - Numérica (int ó long)
  - Mayor que cero al comienzo
- La cota se decrementa en cada iteración.
- Asegura la terminación.

Ejemplo de invariante de bucle

```
int i = n;
int s = 0;
//@ loop_invariant i+s == n
//@ decreases i;
while (i >= 0)
{
  i = i-1;
  s = s+1;
}
```

Herencia

- Las especificaciones tb. se heredan. (subcontratación)

```
class B extends A { ... }

A a;

a = new B();
```

- La especificación no se puede “relajar”.
- La precondición se puede hacer más débil (menos restrictiva).
- Las postcondición se puede hacer más fuerte (más restrictiva).

Herencia

- Las especificaciones tb. se heredan. (subcontratación)

```
class B extends A { ... }

A a;

a = new B();
```

- La especificación no se puede “relajar”.
- La precondición se puede hacer más débil (menos restrictiva).
- Las postcondición se puede hacer más fuerte (más restrictiva).

Herencia

- Las especificaciones tb. se heredan. (subcontratación)

```
class B extends A { ... }

A a;

a = new B();
```

- La especificación no se puede “relajar”.
- La precondición se puede hacer más débil (menos restrictiva).
- Las postcondición se puede hacer más fuerte (más restrictiva).

Herencia

- Las especificaciones tb. se heredan. (subcontratación)

```
class B extends A { ... }

A a;

a = new B();
```

- La especificación no se puede “relajar”.
- La precondición se puede hacer más débil (menos restrictiva).
- Las postcondición se puede hacer más fuerte (más restrictiva).

Herencia (II)

```
class Parent {
    ...
    //@ invariant invParent;
    ...
}

class Child extends Parent {
    ...
    //@ invariant invChild;
...
}
```

La invariante de la clase hijo es `invChild && invParent`

Herencia (II)

```
class Parent {
    ...
    //@ invariant invParent;
    ...
}

class Child extends Parent {
    ...
    //@ invariant invChild;
...
}
```

La invariante de la clase hijo es `invChild && invParent`

Herencia. Ejemplo

- Ejemplo *naive*.
- `also` nos permite extender la especificación heredada.

```
class Parent {
     //@ requires i >= 0;
     //@ ensures \result >= i;
     int m(int i){ ... }
 }
 class Child extends Parent {
     //@ also
     //@   requires i <= 0
     //@   ensures \result <= i;
     int m(int i){ ... }
   }
```

Herencia. Ejemplo (II)

- La especificación resultante del método m() en Child.

```
//@   requires i >= 0;
//@   ensures \result >= i;
//@ also
//@   requires i <= 0
//@   ensures \result <= i;
int m(int i){ ... }
```

- Equivalente a:

```
//@ requires i >= 0 || i <= 0;
//@ ensures \old(i) >= 0 ==> \result >= i;
//@ ensures \old(i) <= 0 ==> \result <= i;
```

Herencia. Ejemplo (II)

- La especificación resultante del método m() en Child.

```
//@   requires i >= 0;
//@   ensures \result >= i;
//@ also
//@   requires i <= 0
//@   ensures \result <= i;
int m(int i){ ... }
```

- Equivalente a:

```
//@ requires i >= 0 || i <= 0;
//@ ensures \old(i) >= 0 ==> \result >= i;
//@ ensures \old(i) <= 0 ==> \result <= i;
```

Ejemplo (I)

- Función de ordenación.

```
/*@ ensures
  @ (\forall int i; 0 <= i && i < \result.size();
  @       \result.itemAt(i) instanceof Integer)
  @ &&
  @ (\forall int i; 0 < i && i < \result.size();
  @       c.compare(\result.itemAt(i-1),
  @                 \result.itemAt(i)) <=0);
  @*/
public static List sort(List l, Comparator c) {
  ...
}
```

- ¿ Correcto ?
- (* \result contiene exactamente los mismos elementos que \old(l), posiblemente en otro orden *)

Ejemplo (I)

- Función de ordenación.

```
/*@ ensures
  @ (\forall int i; 0 <= i && i < \result.size();
  @       \result.itemAt(i) instanceof Integer)
  @ &&
  @ (\forall int i; 0 < i && i < \result.size();
  @       c.compare(\result.itemAt(i-1),
  @                 \result.itemAt(i)) <=0);
  @*/
public static List sort(List l, Comparator c) {
  ...
}
```

- ¿ Correcto ?
- `(* \result contiene exactamente los mismos elementos que \old(l), posiblemente en otro orden *)`

Ejemplo (I)

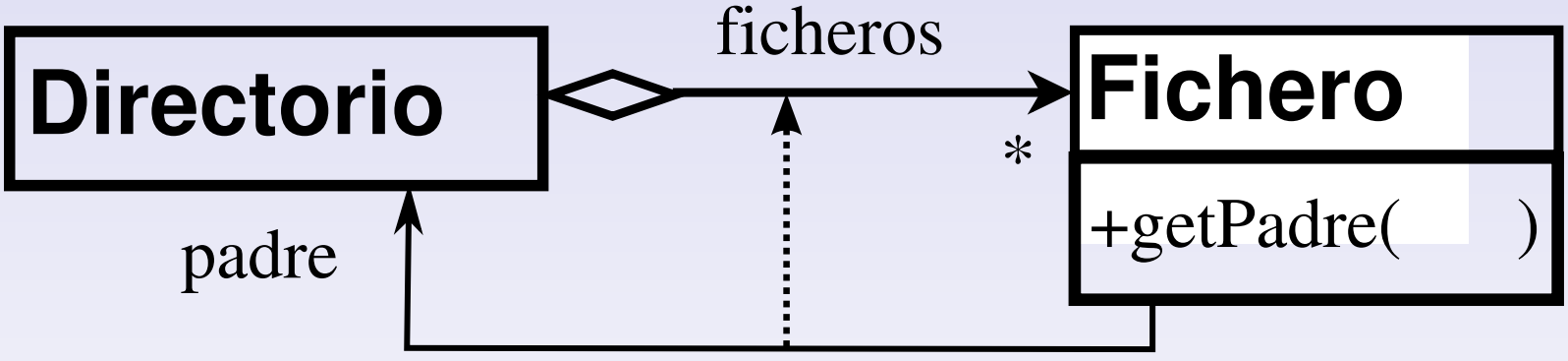
- Función de ordenación.

```
/*@ ensures
  @ (\forall int i; 0 <= i && i < \result.size();
  @       \result.itemAt(i) instanceof Integer)
  @ &&
  @ (\forall int i; 0 < i && i < \result.size();
  @       c.compare(\result.itemAt(i-1),
  @                 \result.itemAt(i)) <=0);
  @*/
public static List sort(List l, Comparator c) {
  ...
}
```

- ¿ Correcto ?
- `(* \result contiene exactamente los mismos elementos que \old(l), posiblemente en otro orden *)`

Ejemplo (II)

- Incluir restricciones del diseño.



Ejemplo (III)

```
public class Directorio {
  private Fichero[] _ficheros;

/*@ invariant

  _ficheros != null
  &&
  (\forall int i; 0 <= i && i < _ficheros.length;
                  _ficheros[i] != null &&
                  _ficheros[i].getParent() == this);
  @*/
```

Atributos fantasma (`ghost`)

- Sólo se pueden usar en las especificaciones.

```
class SimpleProtocol {
 //@ boolean ghost started;
 //@ requires !started;
 //@ assignable started;
 //@ ensures     started;
 void start() {
    ...
    //@ set started = true;
 }
 //@ requires    started;
 //@ assignable started;
 //@ ensures    !started;
 void stop() {
    ...
    //@ set started = false;
 }
```

Atributos fantasma (ghost) (II)

- Pero se pueden mezclar con otros atributos.

```
class SimpleProtocol {
 private Stack _stack;
 //@ boolean ghost started;
 //@ invariant started <==> (_stack != null);
 ...
```

- En el resto queda oculta la implementación de *started*.

```
 //@ requires !started;
 //@ assignable started;
 //@ ensures     started;
 void start() { ... }
...
```

Atributos fantasma (ghost) (II)

- Pero se pueden mezclar con otros atributos.

```
class SimpleProtocol {
 private Stack _stack;
 //@ boolean ghost started;
 //@ invariant started <==> (_stack != null);
 ...
```

- En el resto queda oculta la implementación de *started*.

```
 //@ requires !started;
 //@ assignable started;
 //@ ensures     started;
 void start() { ... }
...
```

Campos modelo

- Tb. son atributos sólo de la especificación.
- Se le asocia un valor.

```
class SimpleProtocol {
private ProtocolStack st;
//@ boolean model started;
//@ represents started <-- (st!=null);

//@ requires !started;
//@ assigable started;
//@ ensures   started;
void startProtocol() { ... }
...
```

Refines

- ¿ Si quiero extender/añadir un contrato sin modificar el fichero con el código fuente ?
- Solución: usar `refine`.

Persona.jml

```
package org.jmlspecs.samples.jmltutorial;

//@ refine "Persona.java";

public class Persona {
  private /*@ spec_public non_null @*/ String _nombre;
  private /*@ spec_public @*/ int _peso;

  /*@ public invariant !_nombre.equals("")
    @          && _peso >= 0; @*/

  //@ also
  //@ ensures \result != null;
  public String toString();
```

Refines

- ¿ Si quiero extender/añadir un contrato sin modificar el fichero con el código fuente ?
- Solución: usar `refine`.

Persona.jml

```
package org.jmlspecs.samples.jmltutorial;

//@ refine "Persona.java";

public class Persona {
  private /*@ spec_public non_null @*/ String _nombre;
  private /*@ spec_public @*/ int _peso;

  /*@ public invariant !_nombre.equals("")
    @          && _peso >= 0; @*/

  //@ also
  //@ ensures \result != null;
  public String toString();
```