Design by Contract with JML

Gary T. Leavens y Yoonsik Cheon

Diseño por Contrato con JML

Mi logo es mas lindo que el de Martín.. :P

"Java Modeling Language" combina:

- Diseño por Contrato de Eiffel (Meyer).
- Especificación basada en modelos (lenguajes de la familia Larch).
- Algunos elementos de calculo de refinamientos.

JML

El Design by Contract (DBC) de Meyer

"Object-Oriented Software Construction, 2nd Edition", Bertrand Meyer, 1997.

Ventajas:

- Se diseña software para que sea correcto.
- Ayuda con la documentación.
- Provee una base para testing y debugging sistemáticos:
  - Una violación de aserción en tiempo de ejecución es una manifestación de un bug en el software.

JML

El DBC de Meyer: Elementos

- Pre y post condiciones de métodos y constructores
- Invariantes de clase
- Invariantes y variantes de ciclos
- Aserciones de código

Semántica:

- Dada con ternas de Hoare.
- Existen propiedades de visibilidad.

JML

El DBC de Meyer: **require** y **ensure**

- Precondiciones:
  - Es la aplicación al software del concepto de función parcial. (en vez de funciones totales que dan valores esp. de error)
  - Regla de disponibilidad de precondición: todos los “features” mencionados deben ser visibles por los clientes que ven la rutina, así la pueden cumplir. No vale la regla para las post.
  - Violación de pre => bug en el cliente.
- Postcondiciones:
  - **old** *var* (valor al principio) y **Result** (valor devuelto).
  - Violación de post => bug en el proveedor.

JML

El DBC de Meyer: Herencia y DBC

- Si se redeclara (i.e., se redefine o da efecto a) un método:
  - La pre se da con **require else** y se hace “or” con la del padre.
  - La post se da con **ensure then** y se hace “and” con la del padre.
- Si se redeclara una función como atributo (se puede!):
  - La pre de la función deja de hacer falta (se relaja a False).
  - La post de la función pasa a ser parte del invariante.
    (es preferible dar la post de una función sin parámetros directamente en el invariante)

JML

El DBC de Meyer: **invariant**

- Los constructores deben asegurarlo.
- Los métodos exportados deben preservarlo.
- Los métodos privados no hace falta que lo preserven. (son “helpers” invocados desde el cuerpo de otro método)
- Herencia: Los invariantes de todos los “parents” de una clase se aplican a ella misma.

JML

El DBC de Meyer: Chequeo en tiempo de ejecución

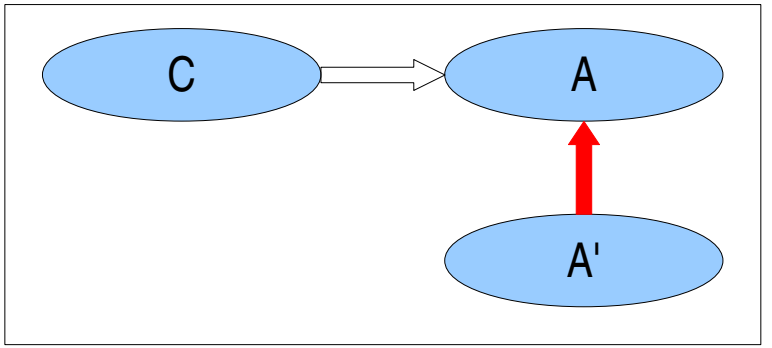
- Regla de evaluación de las aserciones: las llamadas a rutinas se deben ejecutar sin evaluar las aserciones asociadas, para evitar recursiones infinitas (son comunes).
- “Indirect Invariant Effect”: si un invariante habla del estado de otros objetos, se puede violar sin que el objeto se de cuenta. Por eso se chequea el invariante al inicio de las rutinas.

JML

El DBC de Meyer: Argumentos

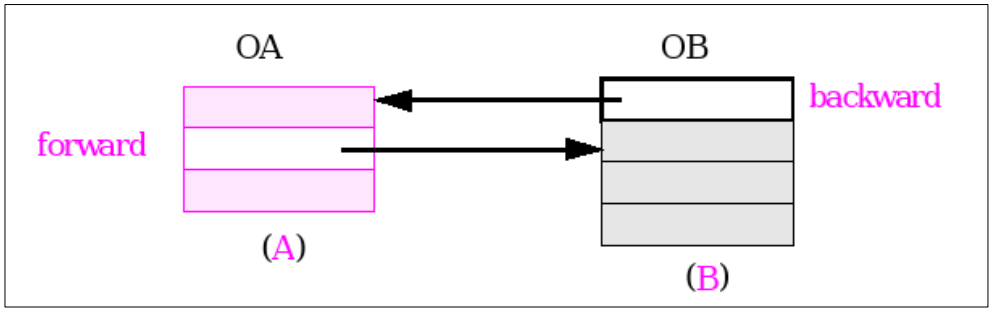
- Redeclaración de métodos:

```
some_routine_of_C (a1: A) is
        do
                ...; a1.r; ...
        end
```



- “Indirect Invariant Effect”:

```
class A ... feature forward: B ... end
class B ... feature backward: A ... end
```




```
round_trip: (forward /= Void) implies (forward.backward = Current)
```

JML

Ahora sí: Java Modeling Language

- "JML is a formal behavioral interface specification language for Java".
- Tiene todo lo que ofrece el DBC de Eiffel, y más:
  - Cuantificadores **\forall**, **\exists**, **\sum**, **\product**, **\min**, **\max**, **\num_of** (ejecutables o no dependiendo del rango).
  - Expresiones informales
  - Variables del modelo (**model**) y variables fantasmas (**ghost**).
  - "History constraints": aserciones sobre valores sucesivos.
  - Más cosas...

JML

JML Tools

- jml: Chequeador de sintaxis JML.
- jmlc: Compilador con chequeo en tiempo de ejecución.
- jmlunit: Generador inteligente de tests JUnit.
- jmldoc: Generador de Javadoc con especificaciones JML.
- Otras:
  - jml-launcher: GUI para usar las Tools.
  - jmlspec: Generador de esqueletos de especificacion.

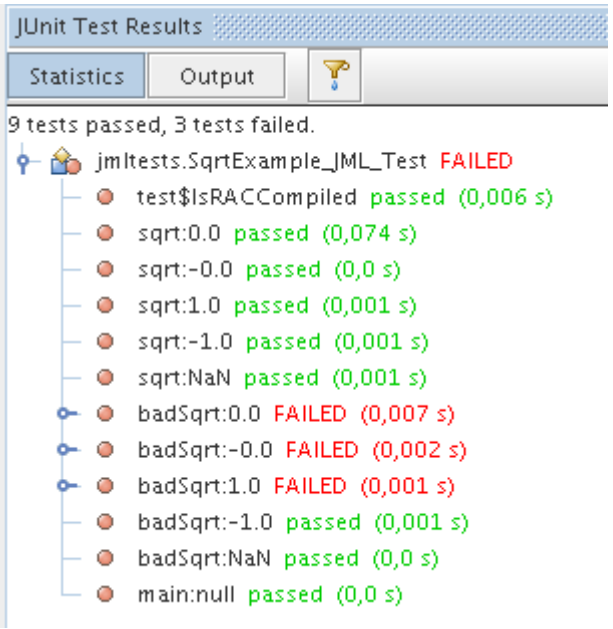
JML: Uso de las Tools

- Integro JML con NetBeans definiendo targets para Ant:
  - jmlc (compilar todos los sources con JML)
  - jmlunit-single (generar JUnit tests para las clases dadas)
- Me quedo sin Java 5.
- ¿Me cuido de que sea compilable sin JML?
- Sólo uso DBC en algunas partes del software.
- En general, diseñar en Java no es otra cosa que programar clases abstractas e interfaces.

```
/** SqrtExample.java: ojo, no compila */
package jmltests;

import org.jmlspecs.models.JMLDouble;

public class SqrtExample {
    public static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approxEqual(x,
                \result*\result, eps);
    public double srqt(double x) {
        return Math.sqrt(x);
    }

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approxEqual(x,
                \result*\result, eps);
    public double badSrqt(double x) {
        return Math.sqrt(x+1);
    }
}
```

Ejemplo simple

- Genero y ejecuto tal como esta el JMLUnit test:

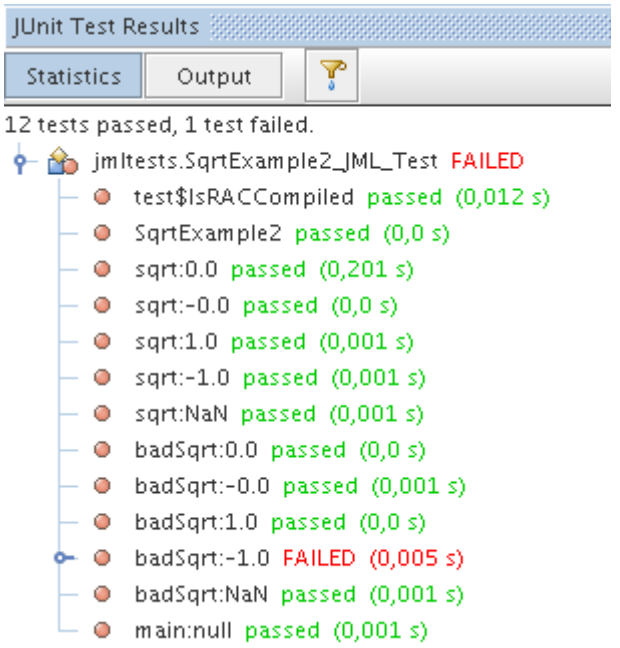


JML

```
/** SqrtExample2.java  */
package jmltests;

public class SqrtExample2 extends
        SqrtExample {
    /*@ also
      @ requires x < 0.0;
      @ ensures \result == -1.0;
      @*/
    public double srqt(double x) {
        return ((x >= 0.0) ?
                super.sqrt(x) : -1.0);
    }
    /*@ also
      @ requires x < 0.0;
      @ ensures \result == -1.0;
      @*/
    public double badSrqt(double x) {
        return ((x >= 0.0) ?
                super.sqrt(x) : -2.0);
    }
}
```

Ejemplo de overriding

- Genero y ejecuto tal como esta el JMLUnit test:



JML

Variables de modelo

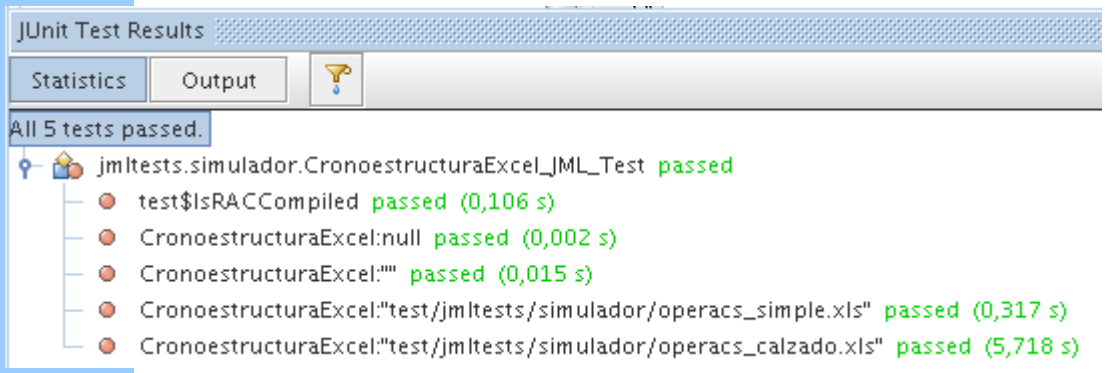
```
/** Cronoestructura.java: pseudocode. */
package jmltests.simulador;

public abstract class Cronoestructura {
  /*@ public model double[] egreso;
    @ public model double egresoTotal;
    @ public model int cant;
    @
    @ public invariant
    @   (egreso.length == cant) &&
    @   (egresoTotal == (\sum int i;
    @    0 <= i && i < cant; egreso[i]);
    @
    @ public represents
    @   egreso <- getEgreso(),
    @   egresoTotal <- getEgresoTotal(),
    @   cant <- getCant();
    @*/
  protected double[] _egreso;
  protected double _egresoTotal;
  public double[] getEgreso() { ... }
  public double getEgresoTotal() { ... }
  public int getCant() { ... }
```

```
/** CronoExcel.java: pseudocode. */
package jmltests.simulador;

import excel.Sheet;

public class CronoExcel
        extends Cronoestructura {
  public CronoExcel(String filename) {
    Sheet s = openSheet(filename);
    double e;
    int last = s.getLastRowNum();
    _egreso = new double(last);
    _egresoTotal = 0.0;
    for (int i = 0; i < last; i++) {
      e = sheet.getRow(i).getCell(1);
      _egreso[i] = e;
      _egresoTotal = _egresoTotal + e;
    }
  }
}
```

Personalizando los casos de test

```
/** CronoExcel_JML_TestData.java */
// This file was generated by jmlunit on
Thu Oct 05 22:39:47 GMT 2006.

package jmltests.simulador;
...
public class CronoExcel_JML_TestData
    extends junit.framework.TestCase {
...
  return new java.lang.String[] {
    // replace this comment with
    // test data if desired
    "test/operacs_simple.xls",
    "test/operacs_calzado.xls"
  };
...
}
```

- Genero test JMLUnit.
- Preparo planillas Excel de test.
- Modifico la TestData.
- Corro el test:



JML

Otras herramientas para JML

- Chequeo y verificación estáticos:
  - ESC/Java2: Detección automática de errores comunes.
  - LOOP: Traducción de JML a “proof obligations” para el probador de teoremas PVS.
  - JACK (del Everest): Plugins para Eclipse. Incluye calculo w.p. automático que genera “proof obligations” para PVS, Coq, etc.
- Generación de especificaciones:
  - Daikon: infiere invariantes probables observando ejecuciones.
  - Houdini: postula anotaciones y las chequea con ESC/Java2.

¡Fin!

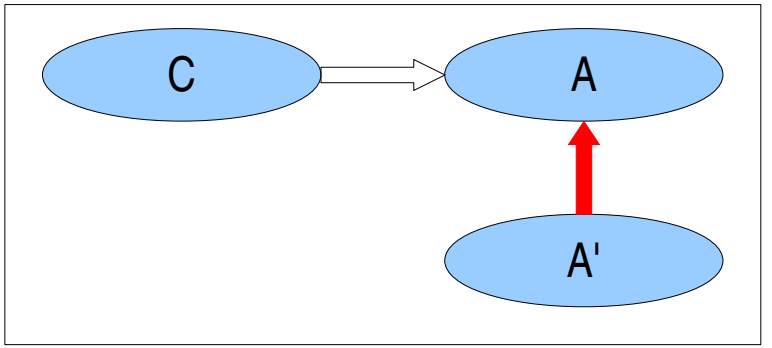
- Cosas que me quedaron afuera:
  - .jml, .jml-refines, .java-refines, y esas cosas...
  - que es Larch y calculo de refinamientos.
  - explicación sobre cómo genera JMLUnit los tests.

JML

El DBC de Meyer: Argumentos

- Redeclaración de métodos:

```
some_routine_of_C (a1: A) is
        do
                ...; a1.r; ...
        end
```



- "Indirect Invariant Effect":

```
class A feature
        forward: B
        attach (b1: B) is
                        -- Link b1 to current object.
                do
                        forward := b1
                                -- Update b1's backward reference for consistency:
                        if b1 /= Void then
                                b1.attach (Current)
                        end
                end
invariant
        round_trip: (forward /= Void) implies (forward.backward = Current)
end
```

```
class B feature
        backward: B
        attach (a1: A) is
                        -- Link a1 to current object.
                do
                        backward := a1
                end
end
```