Computación y Sistemas Vol. 2 No. 1 pp. 35-48
© 1998, CIC - IPN. *ISSN 1405-5546* Impreso en México

JavaA: Inclusión de Aserciones en Java

Miguel Katrib y Damián Fernández
Dpto. de Ciencia de la Computación
Universidad de la Habana
teléfono (537)708690 fax (537)335774
mkm@matcom.uh.cu, damian@comuh.uh.cu

Artículo recibido el 22 de mayo de 1998; Aceptado el 10 de julio de 1998

Resumen

Las aserciones son cláusulas lógicas que se integran a los lenguajes de programación para mejorar la especificación y la verificación. El lenguaje Java no tiene recursos de este tipo. En este artículo se presenta una propuesta de integración de aserciones al lenguaje Java. Se mejoran, con la inclusión de cuantificadores, las posibilidades de estas aserciones para el diseño y especificación. Se analizan las premisas que debe satisfacer una implementación de esta integración y se muestra un esquema general de implementación así como que se analizan las limitaciones del propio Java para la misma.

Palabras Clave

Java, aserciones, diseño por contratos, programación orientada a objetos.

Introducción

Las aserciones [1] son cláusulas lógicas que se utilizan para la especificación y verificación de programas. Las aserciones se integran al modelo orientado a objetos a través de la metáfora de Diseño por Contratos [Meyer, 1992] en el lenguaje Eiffel [Meyer, 1992a]. La aplicación de esta metáfora a la construcción de software orientado a objetos (OO) a través de un lenguaje de programación concreto aporta ventajas desde dos puntos de vista:

- En el proceso de análisis, diseño y programación porque facilita el modelado y la documentación.
- En la confiabilidad y seguridad si pueden ser monitoreadas en ejecución.

Esta metáfora de Diseño por Contrato se basa en la relación que se establece entre una clase proveedora y una clase cliente y que es ejecutada por los objetos instancias de la clase proveedora y por los objetos instancias de la clase cliente. Las aserciones se dividen en tres categorías: precondiciones, postcondiciones e invariantes. Cada servicio (método en términos de OO) ofrecido por una clase y ejecutado a través de sus objetos exige el cumplimiento de una condición (precondición) por parte del cliente y a cambio garantiza una condición (postcondición) que se cumplirá al terminar la ejecución del servicio. Además de estos "contratos particulares" entre cada servicio y la clase cliente, la clase proveedora asegura un invariante o condición que siempre se cumplirá por los objetos de la clase proveedora cada vez "que den la

[1] Del inglés *assertions* también denominadas en castellano asertos o aseveraciones.

cara" a la clase cliente (es decir, antes y después de brindarle cada servicio).

Lamentablemente Java es un lenguaje de programación OO que no tiene integrado recursos de este tipo. El papel de las pre y postcondiciones en Java ha sido considerado en [Papurt, 1997] pero sólo en el aspecto metodológico, sin dar propuestas concretas de integración y sin ninguna mención a lenguajes concretos que si lo han hecho como es el caso de Eiffel. La falta de aserciones en un lenguaje como Java tiene mayor significación debido a que por la portabilidad, difusión y popularidad que ha alcanzado Java, se distribuyen muchas "componentes Java" de software a través de la WWW para las que por consiguiente sería muy deseable disponer de recursos de especificación que potencien la reusabilidad gracias al mejor diseño y a la mayor seguridad en la corrección de estas componentes que pueden aportar las aserciones.

El objetivo del presente trabajo es proponer cómo se pudiera dar una integración de aserciones estilo Eiffel en el lenguaje de programación Java, analizar cuáles serían las restricciones de esta integración y esbozar las bases de una traducción que haría factible la implementación de esta ampliación de Java (en lo adelante JavaA).

Premisas de la Integración de las Aserciones

Una de los inconvenientes al realizar extensiones a un lenguaje de programación es la "impedancia" entre el lenguaje base de partida (Java en este caso) y el lenguaje ampliado (JavaA). Por esta razón esta ampliación debe satisfacer las siguientes premisas:

1. La sintaxis de JavaA debe ser similar a la de Java (a la sazón similar a la de C++). Si al texto fuente de una clase en JavaA se le eliminasen las aserciones, debería quedar un texto sintácticamente correcto en Java.
2. Debe darse un esquema de cuál sería el equivalente en Java de las construcciones de JavaA.
3. Debe procurarse una herramienta de traducción capaz de a partir de una clase en JavaA generar una clase Java equivalente de la que se pueda generar a su vez el archivo con extensión `".class"` (en lo adelante archivo `.class`), de los *byte code* de la clase. Este traductor no deberá depender de información que no sea capaz de representarse en un archivo `.class` ni deberá a su vez generar información que no pueda representarse en un `.class`.
4. Un programador de Java debe poder utilizar una componente JavaA (el archivo `.class` generado para la misma) sin estar obligado a conocer JavaA. Es decir, como si no existiesen aserciones integradas en esta componente (Java tradicional). Sin embargo pudiera también aprovecharse de la existencia de aserciones en la componente `.class` al programar una clase Java cliente aún cuando no disponga del traductor (3) ya que:
   - Las precondiciones lo protegerían de errores en la programación de la clase cliente.
   - Las postcondiciones y los invariantes lo protegerían de errores en la clase proveedora (la programada en JavaA).

   Claro está, la captura o no de las excepciones, que lleguen a la nueva clase producto de violaciones de esta metáfora de Diseño por Contrato, quedarán a responsabilidad del programador.
5. Sólo el programador de una clase cliente que quiera introducir sus propias aserciones en la clase, o un programador de una clase que herede de una clase JavaA (extends en la terminología Java), y que quiera aprovecharse de la existencia de aserciones en la clase base (ancestro), tendría que usar la herramienta (3).
6. Para facilitar la documentación y para evitar tener que ofrecer los textos fuentes de toda clase JavaA, se deberá disponer de otra herramienta (JAI[2]) que "reconstruya" (haga la ingeniería reversa) a partir de este .class una interfaz de clase JavaA (que incluya las aserciones).
7. Las restricciones e imposiciones metodológicas de esta integración (que como se verá más adelante algunas son debidas a las propias limitaciones de Java) deben ser mínimas. El uso correcto de las mismas debe intentar garantizarse automáticamente por el traductor (3).
8. No podrá garantizarse un comportamiento coherente (de hecho tampoco lo hace Java) cuando se utilicen aquellos archivos .class que correspondan a clases en JavaA y que hayan sido modificados manualmente o con alguna otra herramienta.

Aserciones Tipo EIFFEL en Java

Precondiciones

Una precondición en JavaA tiene la forma

```
f(...)
  require {<cláusulas lógicas>}
{ ...
```

y se ubica entre el encabezado y el cuerpo de un método. Las cláusulas lógicas deben evaluar verdadero (`true`) para que se ejecute el cuerpo del método.

[2] Java Assertion Interface.

Si es en la redefinición de un método `f` entonces deberá tener la forma

```
f(...)
  require else {<cláusulas lógicas>}
{ ...
```

En este caso la evaluación de la precondición de `f` ocurre de la forma

```
<cláusulas lógicas> or else <cláusulas lógicas del
                                              ancestro>
```

es decir, el método `f` puede ejecutarse si se cumple su precondición o si se cumple la precondición del método en el padre (que a su vez hará lo mismo con la de su padre y así sucesivamente). En este caso se dice que la precondición es más débil que la del padre porque exige menos (evalúa a `true`) en todos los casos en los que la precondición del padre evalúa a `true`.

Las cláusulas lógicas son expresiones lógicas de Java separadas por ';' (punto y coma) que significa una operación lógica de conjunción (&& de Java). En el caso de una precondición en estas cláusulas lógicas pueden aparecer los parámetros del método y variables y llamados a funciones de la clase.

Postcondiciones

Una postcondición se ubica después del cuerpo de un método.

```
f(...)
  { ...
   }
ensure {<cláusulas lógicas>}
```

Que las cláusulas lógicas evalúen verdadero es la garantía que da el método para indicar que ha cumplido con su parte en el contrato. Por tanto estas cláusulas se evaluarán si la ejecución del método terminó normalmente (no fue abortada por ninguna excepción).

En el caso de las postcondiciones las cláusulas lógicas son, al igual que en las precondiciones, expresiones lógicas en Java en las que se pueden usar además dos construcciones especiales que se verán más adelante en el ejemplo de una clase pila.

Si es en una redefinición del método `f` entonces la postcondición debe tener la forma

```
f(...)
  { ...
   }
ensure then{<cláusulas lógicas>}
```

En cuyo caso la evaluación de la postcondición de `f` ocurre de la forma

```
<cláusulas lógicas> and then <cláusulas lógicas del
                                              ancestro>
```

es decir, después de ejecutarse `f` se tiene que cumplir su postcondición y la del método respectivo en la clase padre (que a su vez hará lo mismo). Se dice en este caso que la precondición es más fuerte que la del padre porque si ella evalúa a `true` la del padre tiene que evaluar a `true`.

Invariantes

Un invariante tiene la forma

```
invariant {<cláusulas lógicas>}
```

y se ubica como última sentencia en el cuerpo de una clase. Las clases ancestro en la jerarquía de herencia pueden también tener invariantes, en este caso todos los invariantes deben evaluar verdadero.

Las reglas gramaticales completas para la inclusión de estas construcciones en Java puede verse en el Anexo.

Ejemplo de una clase STACK

Consideremos como ejemplo una síntesis de una clase STACK[3] para pilas acotadas

```
class STACK {

    ...recursos privados internos de la implementación de pila

    public STACK(int max)
      require {max > 0;} //max es el tamaño máximo de la pila
      { ... cuerpo del constructor }
      ensure {empty();} // después de crear la pila se asegura que está vacía

    public boolean empty(){ ... }ensure {result == (total()==0);}
    public boolean full(){ ... }ensure {result == (total()==max());}
    public int total(){ ... }
```

[3] Para mantener uniformidad con la literatura los nombres utilizados en este ejemplo están en inglés.

```
  public int max(){ ... } //cantidad máxima posible de elementos a almacenar

  public void push(Object x)
    require { !full();}
    { ... cuerpo de push ...}
    ensure { x==top(); !empty(); total()== old total() + 1; }

  public Object top()
    require { !empty();}
    { ... cuerpo de top ... }
    ensure {total() == old total();}

  public void pop()
    require { !empty();}
    { ... cuerpo de pop ... }
    ensure { !full(); total == old total()-1; }

  invariant { total()>=0; total()<= max(); }
}
```

Note que si se eliminan las aserciones queda un texto de clase sintácticamente correcto en Java. Cada aserción no tiene dependencias sintácticas con el resto de las aserciones de la clase, por lo que pueden quitarse algunas o todas las aserciones y la clase seguirá siendo sintácticamente correcta en JavaA.

Construcciones result y old

En el ejemplo anterior han sido utilizadas dos nuevas construcciones no utilizadas en las expresiones lógicas de Java: la variable `result` y el operador `old`.

La variable `result` es definida internamente por JavaA (el programador no tiene que declararla). Esta sólo puede ser utilizada dentro de postcondiciones de funciones (es decir métodos que retornen un valor). La variable `result` dentro de la postcondición toma el mismo valor que retorna la función. Note que esto no violenta la semántica de Java porque esta variable sólo puede utilizarse dentro de una postcondición y no dentro de código normal del método.

El operador `old` también sólo puede ser utilizado dentro de una postcondición. Debe preceder a un término que sea una variable (componente) de la clase o la llamada a una función (método no `void`) de la clase. El término `old f()` donde `f` es una función de la clase (`old total()` en el ejemplo anterior) tendrá el valor de `f()` antes de aplicar el método que precede a la postcondición; el término `old a` , donde `a` es una componente (variable de instancia) denota al valor de la componente `a` antes de ejecutar el método que precede a la postcondición. Dentro de una expresión lógica el operador `old` tiene mayor prioridad que los restantes operadores de Java.

Visibilidad de los recursos utilizados en las aserciones

Según la metáfora de Diseño por Contratos el cliente de una clase `C` debe garantizar, antes de llamar a un método `f` de dicha clase, que cumple con la precondición de `f`. La clase `C` debe entonces garantizar que los recursos (métodos o variables)[4] que utilice en una precondición sean visibles al menos a los mismos clientes para los que es visible el método. De esta manera un cliente de `C`, para el que es visible `f`, tiene la opción antes de llamar a `f` de protegerse con una pregunta de la forma

```
if <precondición de f>{ x.f(); ...
```

La violación de este requisito se reporta como error por el traductor de JavaA-Java. Para el caso del ejemplo `STACK` si el método `full` no tuviese la especificación `public` se reportaría un error ya que el método `push` es público y utiliza a `full` en su precondición `require { !full();}`. Un cliente de `push` no tendría posibilidad de protegerse de garantizar lo que ésta le exige con su precondición.

La tabla a continuación nos indica, según la visibilidad del método para el que se define una precondición, cuál debe ser la visibilidad de los métodos y variables (atributos) de la clase para que se puedan usar en la precondición.

No pasa lo mismo con las postcondiciones ya que es responsabilidad del método el garantizar que se cumpla la postcondición, de modo que en la postcondición sí pueden aparecer campos o métodos privados, protegidos o con especificación por defecto (privado fuera del paquete).

[4] Aunque recuerde que en un buen estilo de programación Java una clase no debe tener variables públicas (para que no puedan ser modificadas sino es bajo el control de la clase) y por tanto no habría que usar variables de la clase en las precondiciones de un método sino sólo funciones.

Visibilidad del método donde se define la precondición.	Visibilidad de los métodos y variables que pueden aparecer en la precondición
public	public
protected	public , protected
sin especificación (visible sólo a las clases del mismo package)	public, protected, sin especificación de visibilidad (visibles en el mismo package)
private	public, protected, sin especificación, private

Inclusión de Cuantificadores en las Aserciones

Las posibilidades de especificación de las aserciones se ve limitada por la ausencia de cuantificadores lógicos. Algunos intentos de inclusión de cuantificadores se han realizado para el lenguaje Eiffel [Katrib and Martínez , 1993], [Katrib and Coira, 1997], [Walden and Nerson, 1995].

Como cláusulas lógicas de las aserciones de JavaA se pueden utilizar el cuantificador universal `forall` y el cuantificador existencial `exist`.

El problema para usar cuantificadores es decidir cuál es el dominio al que se aplica el cuantificador. En la biblioteca estándar Java existe una `interface Enumeration`. Se considera entonces que si `coleccion` es un objeto de una clase `C` que implemente `Enumeration` se puede tener

```
forall x in coleccion: (ExpresionLógica)
```

la cual evalúa a `true` si para todo elemento de `coleccion`, que se asociarán automáticamente a `x` en una iteración sobre `coleccion`, la `ExpresionLógica` evalúa `true`, de lo contrario el `forall` evalúa `false`.

El cuantificador existencial tiene la forma:

```
exist x in coleccion: (ExpresionLógica)
```

la cual evalúa a `true` si para al menos un elemento de `colección`, que se asociará automáticamente a `x` en una iteración sobre `coleccion`, la `ExpresionLógica` evalúa `true`, de lo contrario evalúa false.

La `interface java.util.Enumeration` define dos métodos básicos para "enumerar", o "iterar" a través de un conjunto de objetos.

```
public  interface  java.util.Enumeration
{
      // Methods
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```

La semántica de la construcción `forall` sería equivalente a la de

```
while  (coleccion.hasMoreElements())
 {
   x = coleccion.nextElement();
   if <ExpresionLógica> return false;
    // se supone que x debe aparecer dentro de
                                    la expresión
 }
 return true;
```

y la semántica de la construcción `exists` sería equivalente al código Java

```
while  (coleccion.hasMoreElements())
 {
   x = coleccion.nextElement();
   if <ExpresionLógica> return true;
   // se supone que x debe aparecer dentro de
                                    la expresión
 }
 return false;
```

Ver más adelante el esquema completo de generación de código Java equivalente.

Desafortunada, e increíblemente, el lenguaje Java no tiene genericidad, por esa razón la función `nextElement` de la `interface Enumeration` devuelve un valor de tipo `Object`. En `JavaA` la variable cuantificada, `x` en este ejemplo, es entonces implícitamente de tipo `Object`. Es responsabilidad del programador hacer las operaciones necesarias dentro de `<ExpresionLogica>` (por ejemplo un *cast*) para interpretar el objeto de tipo `Object` como del tipo que realmente sea según la clase que implemente `Enumeration` y que sea la del tipo de los elementos guardados en `coleccion`.

Considérese por ejemplo una clase

```
class EMPLEADO {
  ...
  public EMPRESA trabaja_para(){ ... }
  public int edad(){ ... }
}
```

y las clases

```
class CONJUNTO implements Enumeration {
  ...//implementación de los métodos de
                                Enumeration
  ...//otros métodos para poner y quitar
       elementos
}

class EMPRESA {
  ...
  public CONJUNTO empleados(); { ... }
}
```

La función `empleados` devuelve un objeto de tipo `CONJUNTO` que es una clase que implementa a `Enumeration` y que contiene a todos los trabajadores de la empresa. Si `mi_empresa` es un objeto de tipo `EMPRESA` entonces la siguiente aserción expresa que todos los trabajadores de `mi_empresa` son menores de `50` años.

```
forall x in mi_empresa.empleados():
                              (EMPLEADO)x.edad()<=50
```

Note el uso del *cast* `(EMPLEADO)` para poder aplicar a `x` el método `edad`

El uso de estos cuantificadores en los invariantes de las clases `EMPLEADO` y `EMPRESA` nos permite expresar relaciones muchos-a-uno y uno-a-muchos entre objetos de dichas clases

```
class EMPLEADO {
  ...
  public EMPRESA trabaja_para(){ ... }
  public int edad(){ ... }
  ...
  invariant {exists e in
                trabaja_para().empleados():
                    (EMPLEADO)e == this;
}

class EMPRESA {
  ...
  public CONJUNTO empleados(); { ... }
  ...
  invariant {forall e in empleados():
        (EMPLEADO)e.trabaja_para() == this}
}
```

Ver en [Katrib and Coira , 1997] más ejemplos que ilustran la utilidad de la utilización de cuantificadores en las aserciones.

¿Qué Pasa Cuando se Viola una Aserción?

JavaA da la opción de controlar el cumplimiento de las aserciones en tiempo de ejecución. En este caso si no se cumple una aserción (evalúa `false`) JavaA dispara automáticamente una excepción, a diferencia del estilo Java el programador no es quien tiene que preguntar por el cumplimiento de la aserción para disparar una excepción (con una instrucción `throw` de Java).

Para representar las diferentes violaciones se incluyen las clases `PreconditionException`, `PostconditionException` e `InvariantException` que heredan de `AssertionException` que hereda a su vez de la clase Java `RuntimeException`.

¿Dónde ubicar el recipiente de la excepción, es decir dónde capturar la excepción?. Siguiendo la semántica de la metáfora de Diseño por Contrato la violación de una precondición es responsabilidad del método que llama (quien no verificó previamente antes de llamar) y la violación de un invariante o de una postcondición es responsabilidad en primera instancia del método llamado. Para facilitar expresar esta semántica JavaA permite escribir secciones `catch` después del cuerpo de un método `f` por ejemplo en la forma

```
f()
  require{ ... }{
  ...cuerpo del método
}
  ensure{ ... }
  catch ...
```

En esta sección `catch` se podrán capturar las excepciones que se produzcan por violación de la postcondición de `f` o de un invariante de la clase a la que pertenece `f` si se escribiese por ejemplo

```
catch (PostconditionException pe) o
                catch (InvariantException ie)
```

También se puede capturar aquí una excepción producida por la violación de una precondición de un método al que `f` llame, si se incluye

```
catch (PreconditionException pe)
```

Nótese que la violación de la precondición de `f` no caerá en esta sección `catch` sino en la sección `catch` correspondiente a `PreconditionException` de la rutina que llamó a `f`.

Se podrán detectar también aquí otras excepciones disparadas explícitamente dentro del código de `f`, o disparadas por la máquina virtual de Java, si se escriben explícitamente las secciones `catch` respectivas

Para no entrar en contradicción con Java no se prohibe la escritura de posibles secciones `try /catch` dentro del cuerpo de `f` aunque si se cumple metodológicamente con la metáfora del Diseño por Contrato (en el que el programador no tiene por qué disparar explícitamente excepciones) y del método OO (en el que los métodos deben ser lo suficientemente pequeños) esto no debiera ser necesario.

Sobre la Implementación

Por cada clase JavaA se generará una clase Java (a partir de la cual se genera el archivo `.class`) con las siguientes características

Precondiciones

Por cada método `f` de una clase `C` y con la precondición

```
f (...)
   requiere { precond}
  {..cuerpo de f..}
```

se generará un método de la forma

```
final protected void f_C_require
                                (...parámetros...){
  if (!precond)
    throw new  Internal_PreconditionException
               ( ..parámetros de la excepción ..)
}
```

Así para la precondición del método `push` de la clase `STACK` anterior se generaría

```
final protected void push_STACK_require(){
  if (! full()) throw new
           Internal_PreconditionException( ... )
```

En este caso no se requieren parámetros porque en la precondición de `push` no se utiliza el parámetro de `push`.

Ver más abajo por qué se dispara `Internal_PreconditionException` y no `PreconditionException`.

En el caso en que el método esté sobrecargado (hay otra definición del método con el mismo nombre pero con parámetros diferentes) entonces en el llamado a la función *require* correspondiente a la precondición se ponen todos los mismos parámetros que en el método. De este modo no hay problemas de ambigüedad al sobrecargar a la función *require* para la otra definición del método.

Por cada precondición de una redefinición del método `f` de la forma

```
f(...)
 require else { precond }
 {..cuerpo de f..}
```

en una clase `C1` que extiende a `C`, se generará un método de la forma:

```
final protected void f_C1_require(
                                  ..parámetros..){
  if ( !precond ) f_C_require( ..parámetros..)
  // si es true no evalúa la del padre
```

Si se esta redefiniendo `f` pero no se añaden precondiciones entonces se generará:

```
final protected void f_C1_require(
                                  ..parámetros..){
 f_C_require( ..parámetros..)
}
```

Postcondiciones

Por cada postcondición

```
f(...)
  {..cuerpo de f..}
ensure { postcond }
```

de un método `f` en una clase `C` se generará un método de la forma:

```
final protected void f_C_ensure(..parámetros..){
  if ( !postcond ) throw new
        PostconditionException(..parámetros de la
                                  excepción..);
}
```

En este caso los `parámetros` en `f_C_ensure` son los correspondientes a las variables locales de `f` que se usen en la postcondición. Por cada término de la forma `old h` que se use en la postcondición se generará una instrucción `h_local = h` al inicio del cuerpo del método y es esta variable local `h_local` la que se pasará como parámetro.

Para la postcondición del método `push` del ejemplo de la clase `STACK` se genera

```
final protected void push_STACK_ensure(Object x,
                                       int p1){
  if !(!empty() && top()==x && total()== p1+1)
          throw new PostconditionException(...);
}
```

En una clase `C1` que herede de `C` y que incluya una redefinición de `f` con una postcondición de la forma:

```
ensure then { postcond }
```

se generará un método de la forma:

```
final protected void f_C1_ensure(..parámetros..){
  if { postcond ){ f_C_ensure(..parámetros..)
  else throw new PostconditionException
                (..parámetros de la excepción..);
}
```

Si en `C1` se redefine un método `f` sin añadir nuevas postcondiciones entonces se generará:

```
final protected void    f_C1_ensure
                                  (..parámetros..){
  f_C_ensure(..parámetros..)
}
```

Invariantes

Para un invariante de la clase `C`:

```
invariant { invcond }
```

se generará un método de la forma:

```
final protected void class_invariant() {
  super.class_invariant(); // si la
            superclase tiene invariante
   if ( !invcond ) throw new
   InvariantException(..parámetros de la
                          excepción..);
}
```

Recuerde que tienen que cumplirse los invariantes de todas sus clases ancestros. Para todas las clases en la jerarquía, a partir de la clase que introduzca un invariante, se genera el correspondiente método `class_invariant`. Por ello `super.class_invariant()` llamará al método de evaluar invariante generado para la superclase (que a su vez llamará al de su superclase y así sucesivamente hasta llegar a la primera clase en la jerarquía en la que se introdujo un invariante).

¿Dónde se sitúa el código para el control de cumplimiento de precondiciones, postcondiciones e invariantes?

Según la metáfora de Diseño por Contrato, antes de entrar a ejecutar un método deben cumplirse la precondición del método y el invariante de la clase de tal modo que por cada método `f` de una clase `C` se generará en el cuerpo de éste

```
f(...){
  class_invariant()
  f_C_require(..parametros de la
                      precondición..);
  ..cuerpo normal Java de f ..
}
```

los parámetros formales de `f_C_require( ...parametros...)` corresponden a aquellos parámetros de `f` que se utilizan dentro de la precondición `precond`. Note que como `f_C_require` está dentro de la misma clase que `f` entonces dentro del cuerpo de `f_C_require` se pueden usar todos los métodos, variables de instancia y variables de clase que aparezcan en `precond`.

La postcondición de un método y el invariante de la clase deben cumplirse al terminar el método. Luego hay que generar código para verificar este cumplimiento por cada punto en el que haya un `return` en el cuerpo del método.

[3] Recuerde que los compiladores de Java reportan error si encuentran que en un método hay código al que ningún flujo potencial de ejecución puede llegar.

Si `f` es un método función (es decir que devuelve un valor en la terminología Java-C++) entonces cada `return` como el siguiente

```
T f(){
  ...
    return expresión_de_tipo_T
  ...
}
```

se sustituye por

```
T f(){
  T f_result;
  ...
    f_result = expresión_de_tipo_T;
    f_C_ensure(...); //puede tener como
       parámetro a f_result si la variable
                  //result aparece en  la
                              postcondición
    class_invariant();
    return f_result;
  ...
}
```

En caso de que la postcondición utilice la variable especial `result` entonces `f_result` debe pasarse como parámetro en el llamado `f_C_ensure` quien a su vez la capturará en un parámetro formal del mismo nombre. Note que en Java una instrucción `return` puede estar dentro de una sección `catch`, esto no importa para la sustitución anterior que debe realizarse donde quiera que aparezca un `return`.

Si `f` es un método procedimiento (es `void` en la terminología Java-C++) entonces por cada `return` como el siguiente

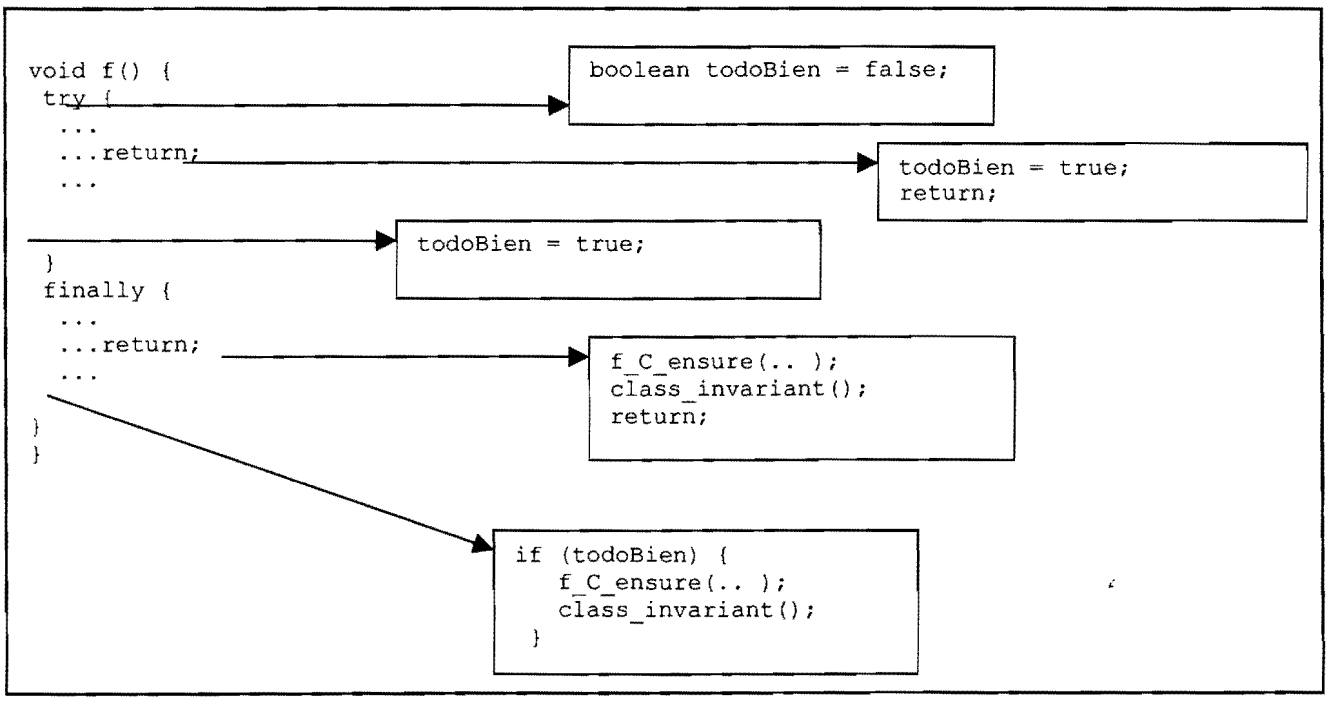
```
void f(){
  ...
    return
  ...
}
```

se sustituye por

```
void f(){
...
    f_C_ensure(..parámetros..);
    class_invariant();
    return;
  ...
}
```

Como en Java la salida de un método que sea un procedimiento (función `void`) no tiene que hacerse a través de un `return` explícito entonces deberá realizarse una sustitución similar al finalizar el cuerpo del método (que puede ser a su vez el de una sección `catch` que esté dentro del método) si un posible flujo de ejecución puede llegar al final del cuerpo del método.[5]

En Java todo método puede terminar con una sección `finally`. Esta sección de código se ejecuta después de salir del método con un `return` o porque se llegó a la última instrucción, o si se ha abortado la ejecución del método por una excepción. Es decir, la sección `finally` se ejecuta siempre, por tanto si un método tiene sección `finally` el control de la postcondición y del invariante debe hacerse dentro de la sección `finally` si es que la ejecución del método no fue abortada por una excepción. El esquema de generación de código si hay sección `finally` sería entonces;



El uso del indicador `todoBien` es para cuando se cae en la sección `finally` porque se abortó la ejecución del método por una excepción. En este caso después de ejecutar la sección `finally` no hay que evaluar postcondiciones ni invariantes ya que la excepción se sigue propagando.

Constructores e invariantes

Según la metáfora de Diseño por Contrato el invariante debe cumplirse luego de haberse creado un objeto. Aplicar esto a JavaA significa que el invariante debe cumplirse después de aplicar un constructor.

El control de postcondiciones e invariantes dentro del código de cada constructor se generará de manera similar a la anterior a la que se analizó anteriormente para un método.

Sin embargo, Java permite definir una clase sin que tenga constructores explícitos. En este caso aplicará un constructor por defecto (que no tiene parámetros). A su vez si en el constructor de una clase no se invoca explícitamente al constructor de una clase padre Java aplica el constructor por defecto de la clase padre. Por tanto para toda clase `C`

```
class C {
  ...//no tiene constructor de la forma C(){...}
  ...
  class_invariant { invcond }
}
```

que no tiene un constructor sin parámetros, se generará un constructor por defecto de la forma

```
class C {
  ...
  C() {
  //se llama implícitamente al constructor por
                defecto de la superclase que a su
  //vez controlará el cumplimiento de su invariante
   class_invariant(); //evaluar el cumplimiento del
                                    invariante de C
  ...
 final protected void class_invariant(...){
                        ...evaluacion de invcond...}
}
```

Aserciones con cuantificadores

En una clase `C` por cada aserción con cuantificador universal

```
forall x in coleccion: (<ExpresiónLógica>)
```

se generará un llamado `C_forall_k(colección, ...otros parámetros según la aserción...)` a la siguiente función lógica con código Java

```
final boolean C_forall_k(Enumeration c,
    ..otros parámetros según la aserción..){
 Object x;
  while (c.hasMoreElements())
   {
    x = c.nextElement();
    if !<ExpresionLógica> return false;
     // se supone que x debe aparecer dentro
                              de la expresión
  }
  return true;
}
```

Recuerde que una clase pueden haber cualquier cantidad de aserciones con cuantificadores en postcondiciones, precondiciones e invariantes, en la generación estas se enumerarán de la forma `C_forall_k` y se llamarán por la función de evaluación de la aserción correspondiente.

Para el invariante de la clase `EMPRESA` vista anteriormente se generaría

```
final protected boolean EMPRESA_forall_1(Enumeration
                                                c){
Object x;
while (c.hasMoreElements())
  {
    x = c.nextElement();
   if (!((EMPLEADO)x.trabaja_para() == this)) return
                                              false;
  };
return true;
}

void class_invariant(){
  if ((EMPRESA_forall_1(empleados()))) throw ...
}
```

La generación es análoga para el cuantificador existencial.

Disparo y captura de excepciones por violación de aserciones

Para un método `f` de una clase `C` definido en JavaA en la forma siguiente

```
 f(...)
   require {...}
 {...cuerpo de f ...}
   ensure {...}
 catch {...//para capturar violación de
        postcondiciones, invariantes y otras
 ...
```

se generará

```
f(...){
 try{ class_invariant();
      f_C_require(...);
   ...cuerpo de f...
}
 catch (Internal_PreconditionException ipe) {
   throw new PreconditionExeption(...)
}
...secciones catch que haya escrito el programador
                                        en JavaA
```

Al evaluar `f_C_require` si no se cumple la precondición se dispara una excepción de tipo `Internal_PreconditionException` ésta será capturada por la sección `catch` anterior que es generada implícitamente por el traductor y que lo que hace es redisparar la excepción como de tipo `PreconditionException` para que sea capturada por la sección `catch (PreconditionException pe)` que el programador haya escrito para el método que llamó a `f` y quien es el responsable de cumplir con esa parte del contrato (y por tanto el primer responsable de tratar esta excepción).

Algunos Problemas de Implementación

Problema con la herencia y los invariantes

Supongamos que tenemos el siguiente caso:

```
class A
{
  public void f() { ...}
}

class B extends A
{
  ...
      class_invariant   {   ...   }
}

B b = new B();
b.f();
```

Según la semántica del Diseño por Contrato al llamarse al método `f` desde una instancia de `B` deberán verificarse los invariantes definidos en `B`. Sin embargo, `f` fue definida en `A` que no tenía invariantes y que incluso puede ser una clase Java no generada a partir de JavaA. En este caso según el modelo de implementación visto más arriba al evaluar `f` no se comprobaría el invariante.

La solución aplicada por el traductor JavaA-Java es que aunque una clase no redefina un método se insertará en la clase descendiente un método que verifique el invariante (al entrar y salir del método) y ejecute un llamado al método correspondiente en el ancestro.

Para la clase `B` del ejemplo de código anterior se generaría la clase Java correspondiente:

```
class B extends A {
  ...
  public void f(){
    class_invariant(); // se verifica el
                cumplimiento del invariante
    super.f();

    class_invariant(); // se verifica el
                cumplimiento del invariante
}

protected void class_invariant() { ...
}
```

En este caso si se tiene

```
B b = new B();
b.f();
```

al invocarse el método `f` en una instancia de `B`, se ejecutará el método `f` generado para `B` por el traductor JavaA-Java, que verificará el invariante correctamente e invocará al método `f` original en la clase `A`.

El problema se produce si el método `f` fue declarado `final` en `A`, entonces al traducir la clase `B` de JavaA (que introduce un invariante) a Java no se podría insertar una nueva definición de `f`.

Este problema no tiene solución si la clase `A` no fue generada por JavaA y no tenemos el código fuente de `A` o si lo tenemos pero no queremos recompilarlo con JavaA.

Una mejor solución a este problema, para el caso de una jerarquía desarrollada totalmente en JavaA, que evita esta generación de nuevos métodos por cada uno que no se redefina, es la siguiente.

Siempre que se traduzca una clase de JavaA a Java, aunque ésta no especifique un invariante se le generará uno que sólo invoque al del ancestro (si hay algún ancestro con invariante). Se generarán dentro de los métodos de dicha clase los llamados correspondientes al invariante (según el modelo de implementación visto anteriormente). De esta forma para el caso de una clase `A` que no tenga invariante

```
class A {
  ...final public void f()
  ...
} // A no tiene invariante
```

se genera de todos modos

```
class A {
  ...final public void f() {
       class_invariant(); // se chequea el
                    invariante al empezar
       ... //código de f
       class_invariant(); // se chequea el
                    invariante al terminar
    }
...
  protected void class_invariant() {
    ...//no se hace nada salvo invocar al
                  invariante del ancestro
      //si algún ancestro tiene invariante
  }
}
```

entonces para la clase `B` no habría que introducir una nueva definición de `f`

```
class B extends A {
  ...
//no se genera ninguna redefinición de f

  protected void class_invariant() {
   ...// se invoca al invariante de la
superclase
   ...// se evalua el invariante de esta
clase B
}
```

Con este modelo de generación no hay problema aún si el método `f` en la clase `A` fue declarado `final`. Al llamar a

```
b.f()
```

se aplicaría el `f final` de `A` que llamaría a `class_invariant` pero por polimorfismo (ya que estamos con una instancia de `B`) se invocaría al `class_invariant` redefinido en `B` (que es el que llamaría al del ancestro).

Una reflexión conceptual

Otra solución a este problema podría ser considerar que sólo tienen que garantizar el cumplimiento del invariante de una clase los métodos definidos en dicha clase, por ejemplo si se tiene:

```
class A{
  ...
  public void f() { ...}
}

class B extends A {
  ...
  public void g() {...}

   invariant {...}
}
```

y se hace

```
B b = new B();
b.f(); // no hay que garantizar el invariante de B
b.g(); // se tiene que garantizar el invariante
```

Puede pensarse que a fin de cuentas si `f` no fue redefinida en `B` entonces `f` no necesita, ni se aprovecha, de la noción de consistencia de un objeto de tipo `B` introducida por el invariante de `B` y por tanto no tiene que comprobarlo ni

garantizarlo. Si la aplicación de un método como `f` dejase a un objeto inconsistente desde el punto de vista de `B` entonces este error tendría sentido detectarlo sólo cuando se quiera usar al objeto bajo el punto de vista de `B`, es decir, a través de los métodos que éste redefine o que añade.

En contra de este enfoque puede decirse que retardaría la detección de errores de diseño y que además es probable que el error se reporte con inexactitud porque no lo reportaría el método que lo provocó (el que dejó al objeto en un estado inconsistente desde el punto de vista de la clase) sino la aplicación futura de otro método de la clase, todo lo cual entorpecería poder detectar dónde hay que hacer la corrección.

Problemas con las interfaces

El problema más lamentable de JavaA que no tiene solución es que no se pueden introducir aserciones dentro de una `interface` Java. El modelo de implementación estudiado anteriormente se basa en que toda clase JavaA se pueda llevar a una clase equivalente en Java de modo que un usuario de Java pueda seguir usando transparentemente la clase resultante pero con los beneficios de las aserciones. En la clase Java resultante el traductor tiene que introducir código para el control y evaluación de las aserciones de la clase y esto no sè puede hacer si es una `interface`.

Esto implica que a una `interface` no se le pueden poner aserciones porque la `interface` equivalente que se generaría tendría que tener código y en Java una `interface` no puede tener código. Esto de alguna manera es contradictorio con el concepto de *tipo puro absoluto* con el que algunos autores quieren "vender" las *interfaces* de Java. Las aserciones serían precisamente una posibilidad de poner un marco semántico a una `interface` que de lo contrario no sería más que una definición de tipo eminentemente sintáctica.

Puede pensarse que una `interface` con aserciones en JavaA se traduzca a una clase abstracta (con métodos `abstract`) a la que sí podrían ponérsele aserciones. Esto iría en contra de la premisa de equivalencia entre JavaA-Java e impediría la posibilidad de hacer herencia múltiple de esta clase abstracta resultante con alguna otra clase Java.

Para ser sinceros hay que decir que la verdadera causa de este problema está en el propio Java quien prohibe la inclusión de código en una `interface` para evitar con ello algunos de los conflictos de la herencia múltiple y las complicaciones que en el modelo interno de implementación de los objetos esto introduciría.

Conclusiones

La inclusión de aserciones en Java es conveniente por varias razones:

- Para facilitar el diseño, la especificación y la documentación, aumentando con ello las posibilidades de Java como lenguaje de programación para grandes sistemas. La inclusión además de cuantificadores en las aserciones mejora además su poder de especificación y modelado.
- Para mejorar la confiabilidad y las posibilidades de verificación al permitir controlar el cumplimiento de las aserciones en tiempo de ejecución, lo cual es aún de mayor utilidad en una concepción como la de Java que promueve la distributividad en el desarrollo de software y la reusabilidad de componentes realizadas por diferentes proveedores (la metáfora del Diseño por Contratos encaja de manera conveniente en un contexto donde puede ser mas frecuente que el proveedor y el cliente de una clase sean personas diferentes).
- Para facilitar la utilización de Java en la enseñanza de la metodología orientada a objetos propiciando con las aserciones desarrollar el pensamiento lógico y crear hábitos más productivos y seguros que integren las diferentes etapas del ciclo de desarrollo de software.

Un aspecto importante de esta integración de aserciones y Java no ha sido abordado: ¿Cómo relacionar las aserciones con la concurrencia? ¿Pueden servir como mecanismo de sincronización? ¿Se pueden seguir utilizando de la misma manera como recurso de verificación?. Los autores están desarrollando y explorando varias propuestas lo cual será objeto de un próximo trabajo.

Agradecimientos

Los autores agradecen a los colegas del grupo de Programación Orientada a Objetos del Dpto. de Ciencia de la Computación de la Universidad de la Habana y a Ernesto Pimentel de la Universidad de Málaga por sus comentarios y opiniones.

Anexo

Reglas para la inclusión de las aserciones en JavaA (en BNF extendida)

```
ClassDeclaration :
   ( ClassModifier )* "class" <IDENTIFIER> [ Super ] [ Interfaces]  ClassBody


ClassBody :
   "{" ( ClassBodyDeclaration )* InvariantDeclaration "}"


ClassBodyDeclaration:
   StaticInitializer | ConstructorDeclaration | MethodDeclaration |
   FieldDeclaration


InvariantDeclaration :
   "invariant" "{" ( Assertion )+ "}"


MethodDeclaration :
   ( MethodModifier )* ResultType MethodDeclarator [ Throws ] [
    MethodPrecondition ]
   ( Block | ";")
   [ MethodPostcondition ]
   ( "catch" "(" FormalParameter ")" Block )*


MethodPrecondition :
   "require" ["else"] "{" ( Assertion )+ "}"


MethodPostcondition :
   "ensure" ["then"] "{" ( Assertion )+ "}"


Assertion :
   [ <IDENTIFIER> ":" ] ( Expression | ForAll | Exist ) ";"


ForAll :
   "forall" <IDENTIFIER> "in" Expression ":" ( Expression | ForAll | Exist )


Exist :
   "exist" <IDENTIFIER> "in" Expression ":" ( Expression | ForAll | Exist )
```

Referencias

Katrib M., and Martínez L., "Collections and Iterators in Eiffel", *Journal of Object Oriented Programming*, Nov/Dec 1993.

Katrib M. and Coira J., "Improving Eiffel Assertions Using Quantified Iterators", *Journal of Object Oriented Programming*, Nov/Dec 1997.

Meyer B. 1992, "Design by Contract", *Advances in Object Oriented Software Engineering* (editor with Dino Mandrioli), Prentice-Hall 1992.

Meyer B. 1992a, *Eiffel: The Language*, Prentice-Hall Object Oriented Series, 1992.

Papurt DM. "The Sensible Use of Method Overriding: Satisfying Pre-and Postconditions Constraints", *Journal of Object Oriented Programming* (Design with Java column), Nov/Dec 1997.

Walden K. and Nerson JM., Seamless Object Oriented Software Architecture: Analysis and Design of Reliable Systems, Prentice Hall, 1995.

Miguel Katrib Mora *es Dr. en Computación y profesor titular del Depto. de Ciencia de la Computación de la Universidad de la Habana, jefe del grupo de lenguajes y orientación a objetos. Es vicepresidente de la Sociedad Cubana de Matemática y Computación. Miembro del proyecto iberoamericano de investigación IDEAS sobre Ingeniería y Ambientes de Software. Autor de 3 libros de texto y de numerosas publicaciones. Profesor invitado de muchas universidades de Iberoamérica.*