PROGRAMACIÓN ASERTIVA Y CONTRATOS

Programación asertiva

Ejemplos de situaciones que *no van a ocurrir nunca*:

- Con dos dígitos para el año basta
- Esta aplicación nunca va a usarse en el extranjero
- Este contador nunca va a ser negativo

> There is a luxury in self-reproach. When we blame ourselves we feel no one else has a right to blame us.
> --*Oscar Wilde, The Picture of Dorian Gray*

ASERCIONES

Añadir **aserciones** al código para chequear esas situaciones:

```
    void writeString(String s) {
      assert(s != null);
      ...
    }
    ...
    for (int i = 0; i < num_entries-1; i++) {
      assert(sorted[i] <= sorted[i+1]);
    }
```

Aserciones e invariantes

Las aserciones sirven para expresar **invariantes**.

Invariante — Condición que se puede considerar cierta durante la ejecución de un programa o de parte del mismo. Es un predicado lógico que se debe mantener siempre cierto durante una cierta fase de la ejecución.

Por ejemplo, una *invariante de bucle* es una condición que es cierta al principio y al final de cada ejecución de un bucle

Aserciones en Java

Forma 1:

```
assert Expression1;
```

Forma 2:

```
assert Expression1 : Expression2;
```

- `Expression1` es `boolean`
- `Expression2` devuelve un valor que es pasado al constructor de `AssertionError`, que usa una representación en forma de `String` del valor como detalle del mensaje

En versiones antiguas del JDK, notificar al compilador que las acepte:

```
javac -source 1.4 *.java
```

Las aserciones en Java imponen un alto coste en rendimiento y puede ser conveniente desabilitarlas en tiempo de ejecución:

```
java [ -enableassertions | -ea  ] [:<package name>"..." | :<class name> ]
java [ -disableassertions | -da ] [:<package name>"..." | :<class name> ]
```

¿Gestión de errores?

Las aserciones no son un mecanismo de tratamiento de errores:

```
try {
  BufferedReader in =
    new BufferedReader(new InputStreamReader(System.in));
  String input;
  System.out.print("Please Type Something here: ");
  input = in.readLine();
  assert((input.equalsIgnoreCase("Y") ||
          (input.equalsIgnoreCase("N"));   /* bad idea! */
  ...
} catch (Exception ex) {
  System.out.print("We've had an Exception: " + ex.getMessage());
}
```

Efectos colaterales

Cuidado con los efectos colaterales de las expresiones de una aserción:

```
while (Iterator i.hasNext() {
  assert(i.next() != null); /* side effect */
  Object obj = i.next();
  // ...
}

while (Iterator i.hasNext() {
  Object obj = i.next();
  assert(obj != null);
  // ...
}
```

¿Por qué? En Java, los objetos (y las estructuras de datos) no son inmutables.

Tipos de invariantes

Invariantes internas

Sustituir los comentarios que indicaban invariantes:

```
if (i % 3 == 0) {
  ...
} else if (i % 3 == 1) {
  ...
} else { // We know (i % 3 == 2)
  ...
}
```

Mejor con aserciones:

```
if (i % 3 == 0) {
  ...
} else if (i % 3 == 1) {
  ...
} else {
  assert i % 3 == 2 : i;
  ...
}
```

Invariantes de control de flujo

Para **selectivas**:

```
switch(suit) {
  case Suit.CLUBS:
    ...
    break;
  case Suit.DIAMONDS:
    ...
    break;
  case Suit.HEARTS:
    ...
    break;
  case Suit.SPADES:
    ...
}
```

Añadir:

```
default:
  assert false: suit;
```

o también (protección aunque se deshabiliten las aserciones, pero sin coste extra):

```
default:
  throw new AssertionError(suit);
```

Puntos **inalcanzables**:

```
void foo() {
  for (...) {
    if (...)
      return;
  }
  assert false; // Execution should never reach this point!!!
}
```

Invariantes de clase

Son un tipo de invariantes **internas** que se aplican a todas las instancias de una clase, en todos los momentos, excepto cuando una instancia está en transición de un estado consistente a otro.

Por ejemplo, en un árbol binario equilibrado, una invariante de clase puede indicar que está ordenado y equilibrado:

1. Añadir código en Java:

```
// Returns true if this tree is properly balanced
private boolean isBalanced() {
  ...
}
```

2. Todo constructor y método *público* debe llamar a `assert isBalanced();` antes del `return`.

Es recomendable incluir comprobaciones de invariantes de clase al principio de los métodos de clases cuyo estado es modificable por otras clases (v.g. *setters*).

Inmutabilidad e invariantes

La **inmutabilidad** en los objetos (y estructuras de datos) es una garantía de que se cumplan las invariantes de clase.

¿Por qué la inmutabilidad?

- Por rendimiento (v.g. `String` en Java): si es inmutable, para copiar un objeto basta con copiar la referencia (*interning*)
- Por *thread-safety* para código concurrente

Inmutabilidad en los lenguajes

- En C++, usar `const`
- En Java: ¿usar `final` para marcar un parámetro constante?
  - Las subclases podrían redefinir los parámetros y volver a hacerlos mutables
  - `final` se aplica a la referencia, no al objeto en sí
- Scala: diferencia entre `val` y `var`
- Muchos lenguajes funcionales (Lisp, Haskell, Erlang, Clojure, etc.) y algunos orientados a objetos (Eiffel) definen la inmutabilidad por defecto

CONTRATOS

Contrato

- Un contrato entre dos partes define derechos y responsabilidades por ambas partes
- Define las repercusiones por incumplimiento del contrato

Design By Contract (DBC)

- Desarrollado para lenguaje *Eiffel* por Bertrand Meyer
- Documentar y aceptar los derechos y responsabilidades de cada módulo de software para asegurar la correción de un programa
- Un programa correcto es aquél que hace nada más y nada menos que lo que dice hacer

Precondición

- Qué debe ser cierto antes de llamar a una rutina/método (sus requisitos)
- Una rutina jamás debe ser llamada si se violan sus precondiciones
- Es responsabilidad del que la llama hacer que se cumplan

Postcondición

- Qué garantiza la rutina: estado del mundo cuando la rutina/método termina
- Implica que la rutina debe finalizar: no puede haber bucles infinitos

Invariante de clase

- Condición que se cumple para todas las instancias de la clase, desde la perspectiva del llamador
- Durante el procesamiento interno, la invariante puede no cumplirse, pero sí cuando la rutina termina y se devuelve el control al llamador
- Una clase no puede dar permiso de escritura sin restricciones sobre las propiedades (*data members*) que participan en la definición de la invariante

Contratos en los lenguajes

Contratos en Java

- OpenJML, lenguaje de especificación que permite chequear programas en Java con anotaciones para especificar contratos: `@requires`, `@ensures`, `@loop_invariant`, `@old`, etc.
- *iContract*, inactiva, recuperada en Java Contract Suite o JContractS
- *Contracts for Java* o Cofoja
- Usar ApectJ Oval para programar contratos
- Etc.

Contratos en Scala

- Scala permite especificar aserciones (`assert`), precondiciones (`require`), postcondiciones (`ensuring`) e invariantes (`assume`).
- Ejemplo:

```scala
def divide(x: Int, y: Int): Int = {
  require(x > y, s"$x > $y")
  require(y > 0, s"$y > 0")
  x / y
} ensuring (_ * y == x)
```

- Ejercicio: Pedir a Copilot un tutorial de Design by Contract en Scala.

¿Hay contratos en C++?

Aún no hay contratos en C++17 ni en C++20 ni en C++23.

Para interesados:

- Leer el working paper sobre el estado de la propuesta de contratos en C++
- Ver el video de J. D. García sobre Contracts programming after C++17: Desde el minuto 4'10''

Ejemplo: Java + anotaciones

Java no permite especificar contratos (los *assert* no son lo mismo). Así que hay que utilizar extensiones al código Java con anotaciones, como *OpenJML* o *iContract*:

Ejemplo: Inserción en una lista ordenada

```
/**
  * @invariant forall Node n in elements() |
  *    n.prev() != null
  *      implies
  *         n.value().compareTo(n.prev().value()) > 0
  */
public class OrderedList {
  /**
    * @pre contains(aNode) == false
    * @post contains(aNode) == true
    */
    public void insertNode(final Node aNode) {
      // ...
    }
    // ...
}
```

- Una postcondición puede necesitar expresarse con parámetros pasados a un método para verificar un comportamiento correcto.
- Si el método puede cambiar el valor del parámetro pasado (parámetro mutable), el contrato puede incumplirse.
    - Opción: Usar `variable@pre` de *iContract*

Ejemplo: Java + OpenJML

El mismo contrato puede expresarse en JML para que OpenJML lo verifique:

```
public class OrderedList {
  /*@ public invariant
    @   (\forall int i; 1 <= i < elements.size();
    @      elements.get(i - 1).value().compareTo(elements.get(i).value()) < 0);
    @*/

  /*@ requires !contains(aNode);
    @ ensures contains(aNode);
    @ ensures (\forall int i; 1 <= i < elements.size();
    @      elements.get(i - 1).value().compareTo(elements.get(i).value()) < 0);
    @*/
  public void insertNode(final Node aNode) {
    // ...
  }
}
```

OpenJML: anotaciones

- `requires`: expresa precondiciones que deben cumplirse antes de invocar un método.
- `ensures`: expresa postcondiciones que deben cumplirse al terminar normalmente.
- `invariant`: propiedad que debe mantenerse siempre cierta para los objetos de la clase.
- `\result` y `\old(expr)`: permiten referirse al valor devuelto y al valor previo de una expresión.
- `\forall` indica que la propiedad debe cumplirse para todos los valores que satisfacen la condición indicada.

Dead programs tell no lies

El diseño y la programación basada en contratos son una forma de incrementar la **calidad** del código mediante *early crash*.

Hay otras técnicas, pero en general el principio básico es: cuando el código descubre que sucede algo que supuestamente es imposible o "no debería suceder", el programa ya no es viable: eutanasia.

- En Java se lanza una `RuntimeException` cuando sucede algo extraño en tiempo de ejecución.
- Se puede/debe hacer lo mismo con cualquier lenguaje

Aserciones versus contratos en Java

- No hay soporte para **propagar** aserciones por una jerarquía de herencia: si se redefine un método con contrato, las aserciones que implementan el contrato no serán llamadas correctamente (excepto si se duplican en el código)
- No hay soporte para valores **antiguos**: si se implementara un contrato mediante aserciones, habría que añadir código a la precondición para guardar la información que quiera usarse en la postcondición. (v.g. `variable@pre` en *iContract* versus `old expression` en Eiffel)
- El sistema de **runtime** y las **bibliotecas** no están diseñadas para dar soporte a contratos, así que estos **no se chequean**. Y es precisamente en la frontera entre el cliente y la biblioteca donde hay más problemas.

Código perezoso

- Se recomienda escribir código "perezoso" para los contratos: ser estricto en lo que se acepta al empezar y prometer lo menos posible al terminar.
- Si un contrato indica que se acepta cualquier cosa y promete la luna a cambio, ¡habrá que escribir un montón de código!

Redefinición de contratos

> A routine redeclaration [in a derivative] may only replace the original precondition by one equal or weaker, and the original post-condition by one equal or stronger
> --Bertrand Meyer

Los métodos de clase declaran *precondiciones* y *postcondiciones* al redefinir una operación en una subclase derivada.

- Las **precondiciones** sólo pueden sustituirse por otras más débiles/laxas. Los métodos pueden redefinirse con implementaciones que **aceptan un rango más amplio de entradas**.
- Las **postcondiciones** sólo pueden sustituirse por otras más fuertes/estrictas. Los métodos pueden redefinirse con implementaciones que **generan un rango más estrecho de salidas**.
- Las **invariantes** sólo pueden sustituirse por otras más fuertes/estrictas. Las clases e interfaces pueden *derivarse* para *restringir* el conjunto de estados válidos. Un objeto debe tener un estado *consistente* con cualquiera de sus superclases o interfaces.