Assertions & Design-by-Contract using JML

Erik Poll

University of Nijmegen

Overview

- Assertions
- Design-by-Contract for Java using JML
- Contracts and Inheritance
- Tools for JML
- Demo ESC/Java2 static checker

Assertions

Assertions

An assertion expresses a property that should hold at some program point.

Assertions are a very basic software engineering tool for testting, debugging, and documentation.

Assertions occur in many guises in (test) code.

Assertions in C(++)

For example:

```
p = getDirectory(file);
#ifdef DEBUG
if (p==NULL){ print("file not in any directory"
              println("this should never happen
              exit; }
#endif
```

or

```
p = getDirectory(file);
assert(p!=NULL);
```

Assertions in C(++)

Standard practice at MicroSoft: definieer

```
#ifdef DEBUG
#define ASSERT(b,str)
  if (b) { } else {report(str); assert (false)}
#else #define ASSERT(b,str)
#endif
```

en dan

```
...
p = getDirectory(file);
ASSERT(p!=NULL, "file not in any directory");
```

Assertions

Assertions used in testing/debugging phase, but omitted in production code to avoid performance penalty.

(However, assertions sometimes left in production code to automate filing of bug reports.)

Assertions should have no side effects,

so that turning them on or off doesn't affect execution!

Assertions in Java

Java 1.4 introduces `assert` command.
Support for assertions was the most requested feature!

Usage

```
assert BooleanExpression;
```

or

```
assert BooleanExpression: AnyExpression;
```

Assertions are enabled/disabled by `java -ea/-da`

Code must be compiled with `javac -source 1.4`

Why/when use assertions ?

There are several standard situations where assertions are useful.

Eg consider

```
if (i <= 0 || j < 0) {
      ...
 } else if (j == 0 ) {
      ...
 } else {
      ...
 }
```

When use assertions ?

Comments can be used to explain this, eg.

```
if (i <= 0 || j < 0) {
      ...
 } else if (j == 0 ) {
      // here i > 0 and j is 0
      ...
 } else {
      // here i > 0 and j > 0
      ...
 }
```

Typical use: assert for control-flow

Use `assert`'s in complicated nested conditionals

```
if (i <= 0 || j < 0) {
      ...
} else if (j == 0 ) {
      assert i > 0 && j == 0;
      ...
} else {
      assert i > 0 && j > 0;
      ...
}
```

Better than comments!! (Why?)

Typical use: assert for control-flow

```
...
for (i=0; i<a.length; i++)
   if (a[i]==null) break;

...
```

Typical use: assert for control-flow

Use `assert`'s after repetitions with complicated exits

```
...
for (i=0; i<a.length; i++)
   if (a[i]==null) break;

assert i == a.length || a[i] == null;
...
```

Typical use: assert for control-flow

```
switch (day) {
    case Day.MAANDAG:
        ...
        break;
    ...
    case Day.ZONDAG:
        ...
        break;
}
...
```

Wat if day is not equal to any day ?

Typical use: assert for control-flow

Use `assert`'s in `switch` without `default` case

```
switch (day) {
    case Day.MAANDAG:
        ...
        break;
    ...
    case Day.ZONDAG:
        ...
        break;
    default:
        assert false;
}
...
```

Typical use: data consistency

In a class `Directory` with fields

```
private Directory[] child;
private Directory parent;
private int nr_of_children;
```

you can expect assertions such as

```
assert parent != null;
assert child != null;
assert 0 <= nr_of_children
         && nr_of_children <= child.length;
assert child[i] != null;
assert child[i].parent == this;
```

If these properties are invariants, they should be mentioned in the javadoc for the respective fields.

Asserts for postconditions

```
public class Directory {
private Directory[] child;
private Directory parent;

  public addDirectory(Directory d) {
    ...
     assert d.parent == this;
  }
}
```

This property is a postcondition of `addDirectory`. It could be mentioned in the javadoc for `addDirectory`.

Assert for locks or preconditions

```
private Object[] v;
public synchronized update(int i, String s){
    synchronized (v[i]) {
        updateTime(v[i]); helper(v[i],s);
} }
private void helper(Object o, String s) {
  assert Thread.holdsLock(this);
  assert Thread.holdsLock(o);
  ...
}
```

Holding these locks is a precondition for `helper`. It should be mentioned in the javadoc that `helper` assumes these locks to be held!

Assertions

Assertion serve to

- **document - making assumptions & design decisions of the programmer explicit**
- **improve readability**
- **help in testing**

Design by Contract for Java using JML

Design by Contract

Systematic use of assertions, introduced by Bertrand Meyer for the Eiffel language.

Idea: give assertions to express

- **preconditions and postconditions of individual methods and constructors**
- **class invariants**

JML (Java Modeling Langauge)

JML provides Design-by-Contract for Java.

- **More expressive than Eiffel**
- **JML extends the Java boolean expressions with a few operators, such as** `==>`, `\old`, `\result`, `\forall`, `\exists`, **....**
- **JML assertions are added as special comments in .java file, between** `/*@ ... @*/`, **or after** `//@`**.**
  Java compiler ignores these comments, but special JML tools can use them.
- **More tool support than just testing**

Pre- and postconditions

Pre- and post-conditions for methods, eg.

```
/*@ requires d != null;
  @ ensures  d.parent == this;
  @*/
public addDirectory(Directory d){
    ...
}
```

As usual, this is only an incomplete specification.

Design-by-Contract

Pre- and postcondition define a contract between a method and its clients:

- **Client must ensure precondition and may assume postcondition**
- **Method may assume precondition and must ensure postcondition**

Eg, in the example spec for `addDirectory(d)`***, it is the obligation of the client to ensure that*** `d` ***is not null.***

The `requires` ***clause makes this explicit*****.**

Pre- and postconditions

JML specs can be as strong or as weak as you want.

```
/*@ requires d != null;
  @ ensures  true;
  @*/
public addDirectory(Directory d){
    ...
}
```

This default postcondition "`ensures true`" can be omitted.

Exceptional postconditions

`signals` clauses specify when exceptions may be thrown.

```
/*@ requires d != null;
  @ ensures  d.parent == this;
  @ signals (DirectoryException e)
  @           e.getReason() == "Directory full"
  @           d.parent == \old(d.parent);
  @*/
public addDirectory(Directory d){
   ...
}
```

Invariants

Invariants (aka *class* invariants) are properties that "always" hold.

More precisely, invariants should be

- **established by constructors**
- **maintained by methods**

In other words, invariants are implicity included in

- **the postcondition of constructors**
- **both pre- and postcondition of methods**

Invariants: example

```
private Directory[] child;
private Directory parent;
private int nr_of_children;
/*@ invariant
  @   parent != null  &&
  @   children != null  &&
  @   0 <= nr_of_children &&
  @        nr_of_children <= child.length &&
  @   (\forall int i; 0 <= i && i < nr_of_children;
  @                child[i] != null &&
  @                chlid[i].parent == this) &&
  @   (\forall int i; nr_of_children <= i && i < child.le
  @                child[i] == null);
  @*/
```

Invariants & exceptions

NB: invariants should also be re-established if an exception is thrown! (Why?)

In other words, invariants also implicity included in exceptional postconditions.

Invariants

Invariants document design decisions.

Making them explicit helps in understanding, maintaining, and debugging the code.

For example, the invariant for **`Directory`** warns us that special care is needed with **`addDirectory(null)`**

requires vs. signals

There is often a trade-off between precondition and exceptional postcondition. Eg

```
/*@ requires d != null;
  @ ensures  d.parent == this;
  @*/
public addDirectory(Directory d) {
```

vs

```
/*@ requires true;
  @ ensures  d.parent == this;
  @ signals (NullPointerException) d == null;
  @*/
public addDirectory(Directory d) {
```

This is a design decision!

assert clauses

JML `assert` clause is now also supported by Java 1.4, but JML offers more expressivity. Eg,

```
...
for (i = 0; i < a.length; i++)
   if (a[i]==null) break;
//@ assert i != a.length  ==> a[i] == null;
/*@ assert (\forall int j; 0 <= j && j < i
                         ; a[i] != null);
  @*/
```

Contracts and inheritance

Contracts and inheritance

Contracts can be used to understand the consequences of inheritance, overriding, etc.

What should a subclass do about contracts of its parent class ?

Respect them! **Because:**

- **Breaking invariants in the subclass, by new or overriden methods, can break inherited code** (How?)
- **Breaking method contracts in the subclass, by overriden methods, can break inherited code** (How?) **and can break client code** (How?)

Contracts and inheritance

A subclass is called a behavioural subtype if it

- **introduce additional invariants**
- **give new pre- and postconditions for new methods**
- **weaken preconditions and strengthen postconditions of existing methods**

This garantees that no existing code, of the subclass itself and of clients of the parent, breaks.

NB all methods should maintain all invariants; in particular, the inherited parent methods should also maintain any additional invariants!

Contracts and inheritance

Inheritance provides two kinds of code reuse:

1. **child reusing the code of its parent**
2. **client code written for parent also works on children**

If you're only interested in 1 and not 2, then it is acceptable for overriden methods to break contracts of the parent. Still, you should check that this doesn't break the inherited parent code.

Eg., sometimes inheritance is used just to get access to **`protected`** fields, and not because a is-a relation holds.

Tools for JML

Tools for JML

- **Runtime assertion checking with jmlc** [Iowa Univ.]

  Special compiler inserts runtime tests for all JML assertions

  Any assertion violation results in a special exception.

- **Extended static checking with ESC/Java** [Compaq - KUN/Kodak]

  Automatically tries to prove simple JML assertions at compile time

- **Program verification with LOOP tool + PVS** [KUN]

  Interactively prove any JML assertions at compile time, using a theorem prover

Tools for JML

Runtime assertion checking

- low cost & effort
- easy to do as part of normal testing

Extended checking with ESC/Java

- higher cost & effort
- higher assurance: independent of any test suite
- But: not 100 % reliable

Program verification with LOOP tool + PVS

- much higher cost & effort
- much higher assurance
- 100 % reliable