An introduction to formal specifications and JML

Are invariant properties always true?

Yves Ledru
Université Grenoble-1
Laboratoire d'Informatique de Grenoble
Yves.Ledru@imag.fr

2013

Are invariant properties always true?

- Invariant properties are not as stable as expected. They may become false, and this may remain undetected for a while…
- In this lesson, we will have a careful look at when the invariant is actually checked…

1. Inside a method

- The invariant is checked on entry and on exit of the methods of the class.
- It is not checked inside the method, unless the method calls another method of the class.

```
public class XandY {
  public int x = 0;
  public int y = 0;
  //@ public invariant y == 2*x;

  public void inc(){
    x+=1;
    y+=2;
  }
```

Pre-check

Here the invariant is false but this is not checked!

Post-check

Inside a method (2)

```
public void doubleInc(){
  x+=1;
  inc();
  y+=2;
}
```

Invariant will be checked when entering inc()

```
...JMLInvariantError: by method XandY.inc@pre... when
      'y' is 0
      'x' is 1
      'this' is XandY@24c98b07
      at XandY.checkInv$instance$XandY(XandY.java:114)
      at XandY.inc(XandY.java:323)
      at XandY.internal$doubleInc(XandY.java:18)
      at XandY.doubleInc(XandY.java:465)
```


2. Helper methods

Refactoring complex code…

- Sometimes the code of a method becomes too complex
- It would be easier to restructure the method and share common parts with other methods.
- Example: in the class of a balanced tree

```
public class BalancedTree{
      …
  //@ invariant IsBalanced();
public void ins(int v){…}
public void del(int v){…}
public void balance(){…}

public void insert(int v){ins(v); balance()}
public void delete(int v){del(v); balance()}
```

Problem: the invariant will be checked here!

Solution : private helper methods

- The internal methods are declared as « helper » and private.

```
public class BalancedTree{
      …
  //@ invariant IsBalanced();
private /*@ helper @*/ void ins(int v){…}
private /*@ helper @*/ void del(int v){…}
private /*@ helper @*/ void balance(){…}

public void insert(int v){ins(v); balance()}
public void delete(int v){ins(v); balance()}
```

- The invariant is not checked when entering or leaving an « helper » method.

Why helper methods are private?

- Because private methods can't be called outside the class.
- So there is no danger that an outside call will put the object in a visible state that does not satisfy the invariant.
- It is the responsibility of a public method which calls a private helper method to restore the invariant before returning.

A simpler example

```
public int x = 0;
public int y = 0;

//@ public invariant y == 2*x;

public void inc(){
  incX();
  incY();
  incY();
}

private /*@ helper @*/ void incX(){
  x+=1;
}
private /*@ helper @*/ void incY(){
  y+=1;
}
```

Invariant will not be checked when leaving **incX()** or entering **incY()**

3. Direct modification of public attributes of the class

Accessing public attributes

- External objects may directly modify the public variables of the class.
- The invariant may be invalidated and this will remain undetected until the next call to a method of the class
- We already experimented this when testing the invariant of the class

```
@Test
public void  test_1() {
SetAsTree s=new SetAsTree();
s.ltree = s;
s.skip();
}
```

The test class is external to SetAsTree.

Invariant is false at this point and will only be checked when entering `skip()`

Use private attributes!

- To avoid external unchecked modifications of the public variables, use private variables, associated to getters and setters.
- Declare these variables as /*@ spec_public @*/ to refer to these in the JML assertions.

Defining a property across objects.

Properties involving several classes/objects

- Sometimes, the invariant involves objects of several classes.
- The invariant is usually expressed in one of the classes.
- As a result, it is not checked on exit of the methods of the other class!

`class TwoCounters`
Invariant involving two SimpleCounters

`Class SimpleCounter`
Method **`setC()`**

A call to setC() may invalidate the invariant, and remain undetected!

In detail…

```
public class TwoCounters {
  private /*@ spec_public @*/ SimpleCounter sc1;
  private /*@ spec_public @*/ SimpleCounter sc2;
  //@ public invariant sc1.getC() <= sc2.getC();

public TwoCounters(SimpleCounter c1, SimpleCounter c2){
  sc1 = c1; sc2 = c2;}
public void skip(){ }
```

```
public class SimpleCounter {
  public int c;
  //@ public invariant c >= 0;

public SimpleCounter(){c = 0;}
public /*@ pure @*/ int getC(){return c;}

//@ requires v >= 0;
public void setC(int v) {c = v;}
```


An incorrect execution…

```
public class MisUseTwoCounters {
public static void main(String[] args) {
  SimpleCounter c1 = new SimpleCounter();
  SimpleCounter c2 = new SimpleCounter();
  TwoCounters tc = new TwoCounters(c1,c2);
  c1.setC(10);
  System.out.println("Counters: "+c1.getC()+" "+c2.getC());
  tc.skip();
}
```

The println was executed!
The error was found later…

Counters: 10 0
Exception ...JMLInvariantError: by method TwoCounters.skip@pre
<File "TwoCounters.java", line 15, character 15> regarding specifications at
File "TwoCounters.java", line 7, character 40 when
'this' is TwoCounters@3ee2cf81
at TwoCounters.checkInv$instance$TwoCounters(TwoCounters.java:273)
at TwoCounters.skip(TwoCounters.java:490)

Use a single point of entry!

- The problem here is that a third party (program MisUseTwoCounters) has kept a reference to **c1** and **c2** stored in the private fields of TwoCounters.

```
SimpleCounter c1 = new SimpleCounter();
SimpleCounter c2 = new SimpleCounter();
TwoCounters tc = new TwoCounters(c1,c2);
c1.setC(10);
```

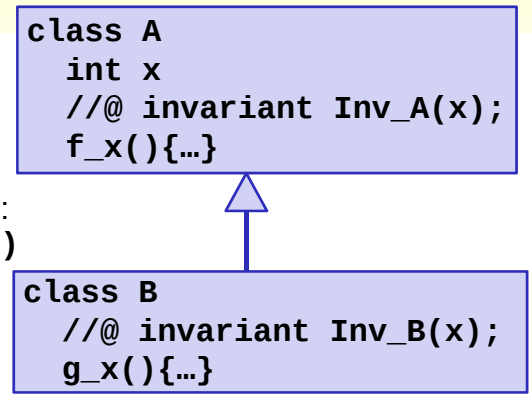
- A correct implementation would make TwoCounters as the single point of entry in the data structure.
- Calls to the setters of SimpleCounter should be wrapped into methods of TwoCounters.

```
public void setC1(int v){
  sc1.setC(v);
}
```


5. Case of inherited invariants

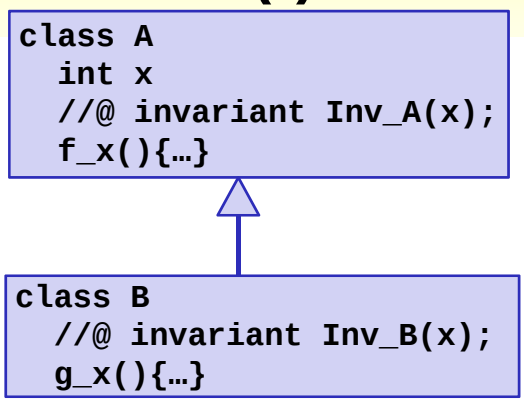
Invariant inheritance

- The invariant of A is inherited in B
- The invariant of class B is: **`Inv_A(x) && Inv_B(x)`**



- B also inherits method **`f_x()`**
- On exit of **`f_x()`**, only **`Inv_A(x)`** is checked
- On exit of **`g_x()`**, **`Inv_A(x)&& Inv_B(x)`** are checked

Invariant inheritance (2)



- As a result, the following program may break the invariant of B:

```
B b = new B();
// both invariants are checked here
b.f_x();
// only Inv_A(x) was checked at this point!
```


Trying it on a simple example…

```
public class EvenCounter extends SimpleCounter {
  //@ public invariant c%2 == 0;

public void inc2(){c+=2;}
public void skip(){ }

public static void main(String[] args) {
  EvenCounter ec = new EvenCounter();
  ec.inc2();
  ec.setC(7);
  ec.setC(9);
  ec.skip();
  }
}
```

The invariant of **EvenCounter** is not checked here

...JMLInvariantError: by method EvenCounter.skip@pre … regarding specifications at File "EvenCounter.java", … when
'c' is 9
…
at EvenCounter.skip(EvenCounter.java:474)

Single point of entry

- In order to avoid this problem, the methods of A should be systematically redefined in B.
- B becomes the single point of entry!

Conclusion

Conclusion

- Make sure that your invariant is actually checked!
  - Use private and protected variables to force the execution of associated methods
  - When the invariant involves several classes, make sure that there is a single point of entry
  - If you are not sure, you can always force the check by calling a method of the class (skip).
- Helper methods introduce a controlled flexibility in the enforcement of the invariant.

