An introduction to formal specifications and JML

Advanced JML concerns

Yves Ledru
Université Grenoble-1
Laboratoire d'Informatique de Grenoble
Yves.Ledru@imag.fr
2013

© Yves.Ledru@imag.fr 2013

Offensive vs Defensive programming

© Yves.Ledru@imag.fr 2013

Preconditions vs exceptions

- Many applications adopt a defensive programming style:
  - The precondition is true, i.e. the operation may be called in any state with arbitrary parameter values
  - But checks are performed inside the code to prevent unauthorized use!
  - Exceptions are raised if the operation is called with illegal parameters or from an illegal state.

© Yves.Ledru@imag.fr 2013

Example: Integer Square Root

- The argument is first checked before computing its square root:

```
public static int isqrt2(int x){
  if(x<0){throw new IllegalArgumentException("Illegal"+x);}
  else { int res = 0;
         for (int i = 0; i*i <= x; i++){res = i; };
         return res;}
     }
```

- An exception is raised if the argument is negative.

© Yves.Ledru@imag.fr 2013

Defensive programming

- In defensive programming, the inputs are systematically checked.
- This makes the code more robust…
- … at the cost of (a) writing and (b) executing the checks.

- Robustness has a cost and slows down the program!

© Yves.Ledru@imag.fr 2013

Offensive programming

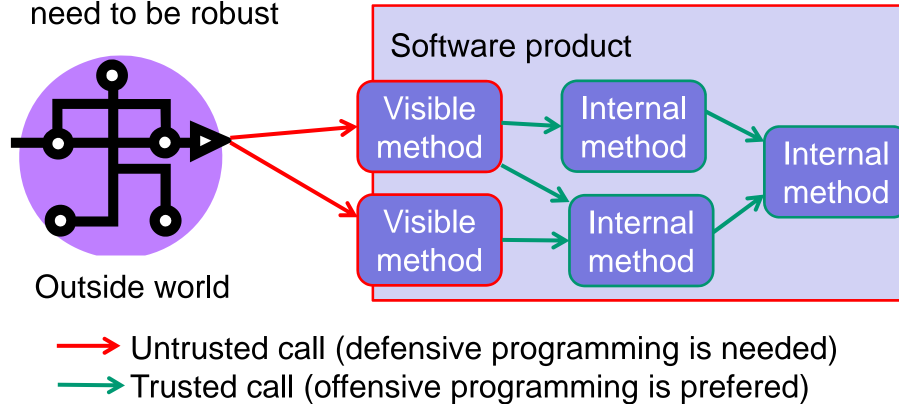
- In offensive programming, a contract is given for the use of the method.
- The contract may be defined in terms of JML assertions (pre, post, invariants)
- The method trusts the caller.
- It is the responsability of the caller to perform the checks, if needed.

- Less robust but potentially more efficient!

© Yves.Ledru@imag.fr 2013

In practice

- The borders of the software must be robust, because the outside world can't be trusted
- The internal calls come from trusted pieces of code and don't need to be robust



Untrusted call (defensive programming is needed)
Trusted call (offensive programming is prefered)

© Yves.Ledru@imag.fr 2013

In practice

- A software component is a mixture of defensive (outside) and offensive (inside) programming!*
- So JML must support both styles…

* Some software editors adopt a defensive style all over their software (outside and inside). Internal exceptions are used as oracle for structural tests.

© Yves.Ledru@imag.fr 2013

Defensive programming in JML

© Yves.Ledru@imag.fr 2013

Specifying exceptions in JML

- The offensive programming code

```
//@ requires x >= 0;
//@ ensures \result*\result <= x
//@ ensures (\result+1)*(\result+1)>x;
public static int isqrt(int x){…}
```

- becomes

```
//@ requires true;
//@ ensures \result*\result <= x ;
//@ ensures (\result+1)*(\result+1)>x;
//@ signals (IllegalArgumentException e) x < 0;
public static int isqrt2(int x){
if(x<0){throw new IllegalArgumentException("Illegal"+x);}
…}
```

© Yves.Ledru@imag.fr 2013

The signals clause

- The signals clause defines an exceptional post-condition
- `//@ signals (Exception e) condition;`
- Keyword **`exsures`** may be used instead of **`signals`**
- **`condition`** may involve the initial and the final states/parameters
- When the code raises exception **`e`**, the **`condition`** is checked by JML.
  - If the **`condition`** is **`false`**, a JML exception is raised denoting an incorrect implementation
  - If the **`condition`** is **`true`**, exception **`e`** is propagated to the caller.
- If another exception is raised, it is propagated to the caller

© Yves.Ledru@imag.fr 2013

Testing invariants and specifications

© Yves.Ledru@imag.fr 2013

Assertions also may be incorrect!

- Playing a test simply compares java code to its JML specification.
- When the test raises a JML exception, it simply means that the code and its executable specification disagree!
  - The code may be erroneous
  - The specification may be erroneous.
- Even if they agree, they may be both erroneous!

© Yves.Ledru@imag.fr 2013

Testing JML specifications

- It is strongly recommended to test JML specifications in order to increase confidence in the specification.
- Positive tests are nominal tests which should not raise a JML exception.
- Negative tests are robustness tests which should raise a JML exception.

© Yves.Ledru@imag.fr 2013

Testing invariants (1)

- It is recommended to cut the invariant into more elementary assertions on separate lines.

```
//@ invariant A && B && C;
```

Should be rewritten as

```
//@ invariant A;
//@ invariant B;
//@ invariant C;
```

- Because JMLexceptions include the line number corresponding to the violated invariant.

© Yves.Ledru@imag.fr 2013

Testing invariants (2)

- Use public variables, or setters without pre-condition to position the state variables.
- Force the evaluation of the invariant by calling `skip(){ }`

```
/*@ public invariant
    @((val != null) || (ltree == null && rtree == null));
    @*/
```

```
@Test
public void  test_0 () {
SetAsTree s=new SetAsTree();
}
```

Nominal test

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

Robustness test

© Yves.Ledru@imag.fr 2013

Testing invariants (3)

- The error message tells us that the exception was raised:
  - In the pre-state of skip()
  - Regarding specifications at line 7

```
junit.framework.AssertionFailedError:
        by method SetAsTree.skip@pre<File "SetAsTree.java", ...>
regarding specifications at
File "SetAsTree.java", line 7, character 26 when
        'val' is null
        'ltree' is SetAsTree@570522265
        'rtree' is null
        'this' is SetAsTree@570522265
```


Testing invariants (4)

- Make sure that you test all lines of the invariant:
  - A positive test passed through all lines
  - A negative test stops at the first failure

Testing pre-conditions

- Here the robustness test must raise a JMLEntryPreconditionError

```
//@ requires !emptySet();
public /*@ pure @*/ int max(){..}
```

```
@Test
public void test_2() {
SetAsTree s=new SetAsTree(5);
int m = s.max();
}
```

Nominal test

```
@Test
public void test_3() {
SetAsTree s=new SetAsTree();
int m = s.max();
}
```

Robustness test

© Yves.Ledru@imag.fr 2013

Testing post-conditions

- This kind of tests are more difficult to perform.
  - Provided you have an implementation, you can write nominal test the post-condition and the exceptional post-condition
  - In order to write robustness tests for the post-conditions, you need to seed errors in the code (more difficult to automate).

© Yves.Ledru@imag.fr 2013

History constraints

Invariants vs history constraints

- Invariants express properties that don't change.
- History constraints express dynamic properties.

- History constraints are post-conditions which apply to all methods of the class.
- History constraints express a property relating the initial and final state of each method.

Managing tickets in a queue

last_ticket
display
44
43
42
Customer Service
Serving nb:
42

© Yves.Ledru@imag.fr 2013

Class Ticket, invariant and history constraints

- The class has two variables, and an invariant:
  the display may not exceed the value of the last ticket issued.

```
public class Ticket {
  public int display;
  public int last_ticket;
  //@ public invariant  display <= last_ticket;
```

- History constraints express that these variables may only grow

```
//@ public constraint \old(display) <= display;
//@ public constraint \old(last_ticket) <= last_ticket;
```

- A more precise constraint may be expressed for last_ticket

```
/*@ public constraint \old(last_ticket) == last_ticket
  @                || \old(last_ticket)+1 == last_ticket;
  @*/
```

© Yves.Ledru@imag.fr 2013

Operation

Serving nb: 42

Serving nb: 43

- `serve_next()` modifies the display

```
//@ requires display < last_ticket;
//@ ensures display != \old(display);
public int serve_next(){
  display++;
  return display;
  }
```

- It implicitly includes the constraint that display (and last_ticket) may only grow or remain the same!
- The pre-condition guarantees invariant preservation.

© Yves.Ledru@imag.fr 2013

Another operation on the display

Serving nb: 42

Serving nb: 44

- `serve_all()` also modifies the display…

```
//@ requires true;
public int serve_all(){
  display=last_ticket;
  return display;
}
```

- No constraint prescribes `display` to increase by one unit!
- Here no pre-condition is needed to preserve the invariant.

© Yves.Ledru@imag.fr 2013

More on history constraints

- History constraints don’t apply to constructors (because state variables are not yet created in the initial state of the constructor).
- Like invariants, history constraints are written once and apply to all methods of the class.

© Yves.Ledru@imag.fr 2013