Java Modelling Language (JML) References

- www.jmlspecs.org
- G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06-rev28, July 2005.
- G. T. Leavens and Y. Cheon, Design by Contract with JML, August 2005.
- N. Cataño and M. Huisman, Formal specification of Gemplus' electronic purse case study using ESC/Java, In *Proceedings of Formal Methods Europe (FME 2002)*. Number 2391 of LNCS, pages 272-289. Springer-Verlag, 2002.
- C. Marché, C. Paulin-Mohring, and X. Urbain, The Krakatoa Tool for Cerification of Java/JavaCard Programs Annotated in JML, to appear in *Journal of Logic and Algebraic Programming*.

JML as a Design by Contract (DBC) Tool

- A "contract" between a program (Java class) and its clients.
- A *precondition* specifies the client's (user of the program) obligation (i.e., guarantees that must be met before calling a method).
- A *postcondition* specifies the implementor's obligation (i.e., guarantees properties that hold after execution).
- Contracts are "executable", i.e., can be checked by tools.

Example Java Class

```
public class Person {
  private String name;
  private int weight;

  public String toString() {
    return "Person(\"" + name + "\"," + weight + ")";
  }

  public int getWeight() { return weight; }

  public void addKgs(int kgs) {
    if (kgs >= 0) { weight += kgs; }
    else { throw new IllegalArgumentException(); }
  }

  public Person(String n) { name = n; weight = 0; }
}
```

Specifying the addKgs Method

```
/*@ requires kgs >= 0;
  @ requires weight + kgs >= 0;
  @ ensures weight == \old(weight) + kgs;
  @*/
public void addKgs(int kgs) { weight += kgs; }
```

Alternate Specification and Implementation of addKgs

```
/*@ requires weight + kgs >= 0;
  @ ensures kgs >=0 && weight == \old(weight) + kgs;
  @ signals_only IllegalArgumentException;
  @ signals (IllegalArgumentException) kgs < 0;
  @*/
public void addKgs(int kgs) {
  if (kgs >= 0) { weight += kgs; }
  else { throw new IllegalArgumentException(); }
}
```

Information Hiding and Invariants

```
public class Person {
  private /*@ spec_public non_null @*/
    String name;
  private /*@ spec_public @*/
    int weight;

  /*@ public invariant !name.equals("")
    @          && weight >= 0; @*/
```

Remaining Code and Specifications of the Person Class

```
//@ ensures \result != null;
public String toString() {
  return "Person(\"" + name + "\"," + weight + ")";
}

//@ ensures \result == weight;
public /*@ pure @*/ int getWeight() { return weight; }

/*@ requires n != null && !n.equals("");
  @ ensures n.equals(name)
  @   && weight == 0; @*/
public Person(String n) { name = n; weight = 0; }
```

Model Fields

If we want to change the code:

```
public class Person {
  private /*@ spec_public non_null @*/ String name;
```

to become:

```
public class Person {
  private /*@ non_null @*/ String fullName;
```

then we can add:

```
  //@ public model non_null String name;
  //@ private represents name <- fullName;
```

Dutch National Flag Example

```
public class Flag {

    public static final int BLUE = 1, WHITE = 2, RED = 3;

    //@ public normal_behavior
    //@   ensures \result <==>
    //@           (i == BLUE || i == WHITE || i == RED);
    public static /*@ pure @*/ boolean isColor(int i);

    public int t[];
    //@ public invariant  t != null &&
    //@    (\forall int k;
    //@       0 <= k && k < t.length; isColor(t[k]));
```

```
/*@ public normal_behavior
  @   requires 0 <= i && i <= j && j <= t.length ;
  @   ensures \result <==>
              (\forall int k; i <= k && k < j; t[k] == c);
  @*/
private /*@ pure @*/ boolean
  isMonochrome(int i, int j, int c);

/*@ public normal_behavior
  @   requires   0 <= i && i < t.length &&
                 0 <= j && j < t.length;
  @   modifiable t[i],t[j];
  @   ensures    t[i] == \old(t[j]) && t[j] == \old(t[i]);
  @*/
private void swap(int i, int j);
```

```
/*@ public normal_behavior
  @   modifiable t[*];
  @   ensures
  @     (\exists int b,r; isMonochrome(0,b,BLUE) &&
  @     isMonochrome(b,r,WHITE) &&
  @     isMonochrome(r,t.length,RED));
  @*/
public void flag()
```

```
{ int b = 0, i = 0, r = t.length;
  /*@ loop_invariant
    @   (\forall int k;
    @      0 <= k && k < t.length; isColor(t[k])) &&
    @   0 <= b && b <= i && i <= r && r <= t.length &&
    @   isMonochrome(0,b,BLUE) &&
    @   isMonochrome(b,i,WHITE) &&
    @   isMonochrome(r,t.length,RED);
    @ decreases r - i;
    @*/
  while (i < r) { switch (t[i]) {
   case BLUE:  swap(b++, i++); break;
   case WHITE: i++; break;
   case RED:   swap(--r, i); break;
      }}}
}
```

Case Study: Electronic Purse on Smart Card

- Debit, credit, and currency change operations were annotated, along with the methods they use. Key generation and authorization not considered.
- Specification was "lightweight". Functional specification describes behavior, but not of more complex code involving loops.
- 42 java classes involved; 432kB of Java (736kB with JML annotations); project length was 3 months.
- Some results:
  - Lightweight specification does not allow full verification, but nevertheless, some important errors in implementation were discovered.
  - Also, parts of code never reached were discovered (and could be removed).

Case Study Conclusions

- Lightweight formal verification (in this case using ESC/Java) can provide:
  - formal specifications of an application
  - checking of implementation by tools, increasing the confidence in the correctness of the implementation

- "The problems that we have found probably also would have been found by doing thorough testing, but using theorem proving techniques one is sure not to forget some cases, without having to put much effort in developing test scenarios."

- "Also, writing the formal specifications forces one to think very precisely about the intended behaviour of programs, which helps in finding errors."