Software Engineering using Formal Methods

Java Modeling Language

Wolfgang Ahrendt & Josef Svenningsson & Meng Wang

2 October 2012

Road-map

first half of the course:

Modelling of distributed and concurrent systems

second half of course:

Deductive Verification of JAVA source code

1. *specifying* JAVA programs
2. proving JAVA programs correct

What kind of Specifications

system level specifications
(requirements analysis, GUI, use cases)
important, but
not subject of this course

instead:

unit specification – *contracts among implementers* on various levels:

- application level – application level
- application level – library level
- library level – library level

Unit Specifications

In the object-oriented setting:

units to be specified are interfaces, classes, and their methods

first focus on methods

methods specified by *potentially* referring to:

- result value,
- initial values of formal parameters,
- pre-state and post-stateaccessible part of pre/post-state

Specifications as Contracts

to stress the different roles – obligations – responsibilities in a specification:

widely used analogy of the specification as a *contract*

"Design by Contract" methodology

contract between *caller* and *callee* of method

callee guarantees certain outcome *provided* caller guarantees prerequisites

Running Example: ATM.java

```java
public class ATM {

    // fields:
    private BankCard insertedCard = null;
    private int wrongPINCounter = 0;
    private boolean customerAuthenticated = false;

    // methods:
    public void insertCard (BankCard card) { ... }
    public void enterPIN (int pin) { ... }
    public int accountBalance () { ... }
    public int withdraw (int amount) { ... }
    public void ejectCard () { ... }

}
```

Informal Specification

very informal Specification of '`enterPIN (`**`int`**` pin)`':

> *Enter the PIN that belongs to the currently inserted bank card into the ATM. If a wrong PIN is entered three times in a row, the card is confiscated. After having entered the correct PIN, the customer is regarded as authenticated.*

Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

precondition	card is inserted, user not yet authenticated, pin is correct
postcondition	user is authenticated

precondition	card is inserted, user not yet authenticated, `wrongPINCounter < 2` and pin is incorrect
postcondition	`wrongPINCounter` is increased by 1 user is not authenticated

precondition	card is inserted, user not yet authenticated, `wrongPINCounter >= 2` and pin is incorrect
postcondition	card is confiscated user is not authenticated

Meaning of Pre/Post-condition pairs

Definition

A **pre/post-condition** pair for a method `m` is **satisfied by the implementation** of `m` if:

> *When* `m` *is called in any state that satisfies the precondition then in any terminating state of* `m` *the postcondition is true.*

1. No guarantees are given when the precondition is not satisfied.
2. Termination may or may not be guaranteed.
3. Terminating state may be reached by normal or by abrupt termination (cf. exceptions).

non-termination and abrupt termination $\Rightarrow$ next lecture

What kind of Specifications

Natural language specs are very important.

but this course's focus:

"formal" specifications:

Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision
- eventually: *automation* of program analysis of various kinds:
  - static checking
  - program verification

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

in one single language.

$\Rightarrow$ JML is not external to JAVA

JML
is
JAVA + FO Logic + pre/post-conditions, invariants + more...

JML Annotations

JML extends Java by annotations.

JML annotations include:

- ✔ preconditions
- ✔ postconditions
- ✔ class invariants
- ✔ additional modifiers
- ✘ 'specification-only' fields
- ✘ 'specification-only' methods
- ✔ loop invariants
- ✔ ...
- ✘ ...

✔: in this course, ✘: not in this course

JML/Java integration

JML annotations are attached to Java programs
by
writing them directly into the Java source code files

not to confuse Java compiler:

JML annotations live in in special comments,
ignored by Java, recognized by JML tools.

JML by Example

from the file `ATM.java`

```
⋮

/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...

⋮
```

Everything between `/*` and `*/` is invisible for JAVA.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

But:

A JAVA comment with '@' as its first character
it is *not* a comment for JML tools.

JML annotations appear in JAVA comments starting with @.

How about "//" comments?

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated; @*/
```

equivalent to:

```
//@ public normal_behavior
//@ requires !customerAuthenticated;
//@ requires pin == insertedCard.correctPIN;
//@ ensures customerAuthenticated;
```

The easiest way to comment out JML? I.e. comment out the comment:

```
/*␣@ public normal_behavior ... @*/
```

```
//␣@ public normal_behavior
//␣@ requires !customerAuthenticated;
...
```

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

What about the intermediate '@'s?

Within a JML annotation, a '@' is ignored:

- if it is the first (non-white) character in the line
- if it is the last character before '*/'.

⇒ The blue '@'s are not *required*, but it's a *convention* to use them.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

This is a `public` specification case:

1. it is accessible from all classes and interfaces
2. it can only mention `public` fields/methods of this class

2. Can be a problem. Solution later in the lecture.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

Each keyword ending with **behavior** opens a ‘specification case’.

normal_behavior Specification Case

The method guarantees to *not* throw any exception (on the top level), *if the caller guarantees all preconditions of this specification case*.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

This specification case has two preconditions (marked by **`requires`**)

1. `!customerAuthenticated`
2. `pin == insertedCard.correctPIN`

here:
preconditions are *boolean Java expressions*

in general:
preconditions are *boolean JML expressions* (see below)

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
```

specifies only the case where both preconditions are true in pre-state

the above is equivalent to:

```
/*@ public normal_behavior
  @ requires (    !customerAuthenticated
  @              && pin == insertedCard.correctPIN );
  @ ensures customerAuthenticated;
  @*/
```

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

This specification case has one postcondition (marked by **ensures**)

- customerAuthenticated

here:
postcondition is *boolean Java expressions*

in general:
postconditions are *boolean JML expressions* (see below)

JML by Example

different specification cases are connected by '**also**'.

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

JML by Example

```
/*@ <spec-case1> also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) { ...
```

for the first time, JML expression not a JAVA expression

`\old(`*E*`)` means: *E* evaluated in the pre-state of `enterPIN`.

E can be any (arbitrarily complex) (JML) expression.

JML by Example

```
/*@ <spec-case1> also <spec-case2> also
  @
  @ public normal_behavior
  @ requires insertedCard != null;
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter >= 2;
  @ ensures insertedCard == null;
  @ ensures \old(insertedCard).invalid;
  @*/
public void enterPIN (int pin) { ...
```

two postconditions state that:

'Given the above preconditions, `enterPIN` guarantees:

`insertedCard == null` and `\old(insertedCard).invalid`'

JML by Example

Question:

could it be

```
  @ ensures \old(insertedCard.invalid);
```

instead of

```
  @ ensures \old(insertedCard).invalid;
```

??

Specification Cases Complete?

consider spec-case-1:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
```

what does spec-case-1 *not* tell about post-state?

recall: fields of class `ATM`:

```
insertedCard
customerAuthenticated
wrongPINCounter
```

what happens with `insertCard` and `wrongPINCounter`?

Completing Specification Cases

completing spec-case-1:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ ensures insertedCard == \old(insertedCard);
@ ensures wrongPINCounter == \old(wrongPINCounter);
```

Completing Specification Cases

completing spec-case-2:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ ensures insertedCard == \old(insertedCard);
@ ensures customerAuthenticated
@          == \old(customerAuthenticated);
```

Completing Specification Cases

completing spec-case-3:

```
@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 2;
@ ensures insertedCard == null;
@ ensures \old(insertedCard).invalid;
@ ensures customerAuthenticated
@         == \old(customerAuthenticated);
@ ensures wrongPINCounter == \old(wrongPINCounter);
```

Assignable Clause

unsatisfactory to add

```
@ ensures loc == \old(loc);
```

for all locations *loc* which *do not* change

instead:
add assignable clause for all locations which *may* change

```
@ assignable loc1,...,locn;
```

Meaning: No location other than $loc_1, \ldots, loc_n$ can be assigned to.

Special cases:

No location may be changed:

```
@ assignable \nothing;
```

Unrestricted, method allowed to change anything:

```
@ assignable \everything;
```

Specification Cases with Assignable

completing spec-case-1:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ assignable customerAuthenticated;
```

Specification Cases with Assignable

completing spec-case-2:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ assignable wrongPINCounter;
```

Specification Cases with Assignable

completing spec-case-3:

```
@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 2;
@ ensures insertedCard == null;
@ ensures \old(insertedCard).invalid;
@ assignable insertedCard,
@            insertedCard.invalid,
```

Assignable Groups

You can specify groups of locations as assignable, using '$*$'.

example:

```
@ assignable o.*, a[*];
```

makes all fields of object o and all locations of array a assignable.

JML Modifiers

JML extends the JAVA modifiers by additional modifiers.

The most important ones are:

- `spec_public`
- `pure`

Aim: admitting more class elements to be used in JML expressions.

JML Modifiers: `spec_public`

in `enterPIN` example, pre/post-conditions made heavy use of class fields

But: **`public`** specifications can only talk about **`public`** fields.

Not desired: make all fields public.

one solution:

- keep the fields **`private/protected`**
- make those needed for specification **`spec_public`**

```
private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINCounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated
                                     = false;
```

different solution: use specification-only fields (not covered in this course)

JML Modifiers: `pure`

It can be handy to use method calls in JML annotations.
Examples:

- `o1.equals(o2)`
- `li.contains(elem)`
- `li1.max() < li2.min()`

allowed if, and only if method is guaranteed to have no side effects.

In JML, you can specify methods to be '**`pure`**':

```
public /*@ pure @*/ int max() { ...
```

'**`pure`**' puts obligation on implementer, not to cause side effects, but allows to use method in annotations

'**`pure`**' similar to '**`assignable \nothing;`**', but global to method

JML Expressions ≠ Java Expressions

`boolean` JML Expressions (to be completed)

- each side-effect free `boolean` Java expression is a `boolean` JML expression
- if `a` and `b` are `boolean` JML expressions, and `x` is a variable of type `t`, then the following are also `boolean` JML expressions:
  - `!a` ("not a")
  - `a && b` ("a and b")
  - `a || b` ("a or b")
  - `a ==> b` ("a implies b")
  - `a <==> b` ("a is equivalent to b")
  - ...
  - ...
  - ...
  - ...

Beyond `boolean` Java expressions

How to express the following?

- an array `arr` only holds values ≤ 2
- the variable `m` holds the maximum entry of array `arr`
- all `Account` objects in the array `accountProxies` are stored at the index corresponding to their respective `accountNumber` field
- all created instances of class `BankCard` have different `cardNumbers`

to be answered in the next lecture

Literature for this Lecture

essential reading:

in KeY Book A. Roth and Peter H. Schmitt: Formal Specification. Chapter 5 only sections 5.1, 5.3, In: B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, vol 4334 of *LNCS*. Springer, 2006. (e-version via Chalmers Library)

further reading, all available at
`http://www.eecs.ucf.edu/~leavens/JML/documentation.shtml`:

JML Reference Manual Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry.
JML Reference Manual

JML Tutorial Gary T. Leavens, Yoonsik Cheon.
Design by Contract with JML

JML Overview Gary T. Leavens, Albert L. Baker, and Clyde Ruby.
JML: A Notation for Detailed Design