Métodos Formais em Engenharia de Software

JML

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

MI, Braga 2008

Outline

- Design by Contract and JML
  - Design by Contract
  - Java Modeling Language
- Tool support
  - jmlc/jmlrac
  - ESC/Java2
- Small Demo
- Hands on... (exercises)

these slides were prepared by adopting/adapting "teaching material" from the JML and ESC/Java2 sites.

Design by Contract (DBC)

Design by Contract

- Introduced by Bertrand Meyer (Eiffel language)...
- (...influenced by VDM, Larch, ...)
- As a way of:
  - Recording details of method responsibilities and assumptions;
  - Document intention (specification) of software components (object invariants; methods; etc.);
  - Avoiding constantly checking arguments;
  - Assigning blame across interfaces.
- Goals:
  - work out application design by writing contracts rather than code;
  - express design at multiple levels (UML, JML, ...);
  - refine design by refining contracts;
  - write code once when architecture is stable.

Contract the Design (J. Kiniry)

- In practice, in many situations we need to address the opposite direction (e.g. software maintenance, certification, …)
- a body of code exists and must be annotated
  - the architecture is typically ill-specified;
  - the code is typically poorly documented;
  - the number and quality of unit tests is typically very poor;
  - the goal of annotation is typically unclear;
- Goals:
  - improve understanding of architecture with high-level specifications;
  - improve quality of subsystems with medium-level specifications;
  - realize and test against critical design constraints using specification-driven code and architecture evaluation;
  - evaluate system quality through rigorous testing or verification of key subsystems.

Contracts in Software

```
/*@ requires x >= 0.0;
  @ ensures Math.abs(\result*\result - x) < e;
  @*/
public static double sqrt(double x)
{ … }
```

	Obligations	Rights
Client	Passes non-negative number	Gets square root approximation
Implementor	Computes and returns square root	Assumes argument is non-negative

Pre and Postconditions

- A method's precondition says:
  - Implementor perspective: what is expected (assumed) from the environment (in particular, the method arguments);
  - Client perspective: what should be accomplished to "use" the method.
- A method's postcondition says:
  - Implementor perspective: what is intended with the method;
  - Client perspective: what is legitimate to assume from the method call.

Advantages of DBC

- Contracts are:
  - more abstract than code;
  - not necessarily constructive (e.g. quantified over infinite types);
  - but often machine checkable (so can help with debugging and testing);
  - and contracts can always be up-to-date .
- A contract can be satisfied in many ways. E.g. for square root:
  - Linear search
  - Binary search
  - Newton's method
  - ...
- These will have varying non-functional properties
  - Efficiency
  - Memory usage
- So, a contract abstracts from all these implementation details.

More advantages of DBC

- Blame assignment. Who is to blame if:
  - Precondition doesn't hold?
  - Postcondition doesn't hold?
- Avoids inefficient defensive checks

```
//@ requires a!=null && x!=null;
//@ requires (* a is sorted *);
public static int binarySearch(Thing[] a, Thing x)
{ … }
```

- Modularity of Reasoning

```
…
source.close();
dest.close();
getFile().setLastModified(loc.modTime().getTime());
…
```

- In order to understand this code…
  - read these methods contracts…
  - instead of look at "all" the code…

JML

Java Modeling Language (JML)

- A formal specification language for Java (Gary T. Leavens et al.)
  - to specify behaviour of Java classes
  - to record design&implementation decisions
- by adding assertions to Java source code, eg.
  - preconditions
  - postconditiions
  - invariants
- JML syntax is well integrated with Java:
  - JML assertions are added as comments in .java file, between /*@ … @*/, or after //@ ;
  - Properties are specified as Java boolean expressions, extended with some operators (\old, \forall, \result, …),
  - and some keywords (requires, ensures, signals, assignable, pure, invariant, non_null, …).

Pre and Postconditions

- Pre and postconditions for methods are established through the "requires" and "ensures" clauses:

```
/*@ requires amount >= 0;
  @ ensures balance == \old(balance)-amount;
  @ ensures \result == balance;
  @*/
public int debit(int amount) {
...
}
```

- where
  - \old(balance) refers to the value of balance before the execution of the method;
  - the multiple ensures clauses are equivalent to the conjunction of their properties;
  - \result refers to the outcome of the method (return value).

JML properties

- JML properties are boolean Java expressions…
- …with the proviso that their evaluation is "side-effect free" (i.e. does not changes the state).
- A method without side-effects is called pure. Programmers might signal methods as pure:

```
public /*@ pure @*/ int getBalance(){...}

Directory /*@ pure non_null @*/ getParent(){...}
```

- The non_null clause signals that the result of getParent() can't be null (can also be used in arguments and instance variables).
- JML property language is extended with the binding operators: \forall, \exists, \sum, \product, \max, \min, …
- E.g. (\forall int i ; 0<=i && i<N ; a[i]==null)

JML Expression	Meaning
`requires p ;`	p is a precondition for the call
`ensures p ;`	p is a postcondition for the call
`signals (E e) p;`	When exception type **E** is raised by the call, then p is a postcondition
`loop_invariant p;`	p is a loop invariant
`invariant p ;`	p is a class invariant (see next section)
`\result == e`	e is the result returned by the call
`\old(v)`	the value of v at entry to the call
`(\product int x ; p(x); e(x))`	$\prod_{x \in p(x)} e(x)$; i.e., the product of e(x)
`(\sum int x ; p(x); e(x))`	$\sum_{x \in p(x)} e(x)$; i.e., the sum of e(x)
`(\min int x ; p(x); e(x))`	$\min_{x \in p(x)} e(x)$; i.e., the minimum of e(x)
`(\max int x ; p(x); e(x))`	$\max_{x \in p(x)} e(x)$; i.e., the maximum of e(x)
`(\forall type x ; p(x) ; q(x))`	$\forall x \in p(x) : q(x)$
`(\exists type x ; p(x) ; q(x))`	$\exists x \in p(x) : q(x)$
`p ==> q`	$p \Rightarrow q$
`p <== q`	$q \Rightarrow p$
`p <==> q`	$p \Leftrightarrow q$
`p <=!=> q`	$\neg(p \Leftrightarrow q)$

Invariants

- Invariants (aka class invariants) are properties that must be maintained by all methods.

```
public class Wallet {
  public static final short MAX_BAL = 1000;
  private short /*@ spec_public @*/ balance;
  /*@ invariant 0 <= balance &&
                balance <= MAX_BAL;
    @*/
...
}
```

- spec_public turns visibility of balance public for specification purposes.
- Invariants are implicitly included in all pre- and postconditions.
- Invariants must also be preserved if an exception is thrown! (they must hold whenever the control is outside object's methods)
- Invariants allow you to define:
  - acceptable states of an object (helps in understand the code),
  - and consistency of an object's state (valuable for testing/ debugging).

Frame conditions

- Frame conditions (assignable clause) restrict possible side-effects of the methods (i.e. "where" the method is allowed to make changes)

```
/*@ requires amount >= 0;
    assignable balance;
    ensures balance == \old(balance)-amount;
  @*/
public void debit(int amount) {
  balance = balance - amount;
}
```

- They are a crucial ingredient when we are trying to reason about a program...

```
...
// let us assume that, at this point, name!=null;
debit(50);
// can we still be sure that name!=null ???
...
```

- Default assignable clause: assignable \everything.
- Pure method are implicitly assignable \nothing.

assert and assume clauses

- JML assert and assume clauses allow to attach a property to a given program location.

```
int x;
...
//@ assert x>=0;
x = f(x);
...
//@ assume x<0;
...
```

- The distinction is purely informative:
  - in an assert clauses, we take responsible for validating the property;
  - in assume, the property should follow from others guaranties (e.g. preconditions or methods postcontitions).
- In short, it specifies who should be blamed if the property does not hold.

DBC and JML

- DBC can roughly be seen as an expantion of pre and postconditions as assert and assume clauses.

```
//@ requires x >= 0.0;
//@ ensures Math.abs(\result*\result - x) < e;
public static double sqrt(double x)
{ … }

...
b = sqrt(a);
...
```

- Should be expanded into (performed by JML tools):

```
public static double sqrt(double x) {
  //@ assume x>=0.0;
  …
  //@ assert Math.abs(r*r - x) < e;
  return r;
}

...
//@ assert a>=0;
b = sqrt(a);
//@ assume Math.abs(b*b - a) < e;
...
```

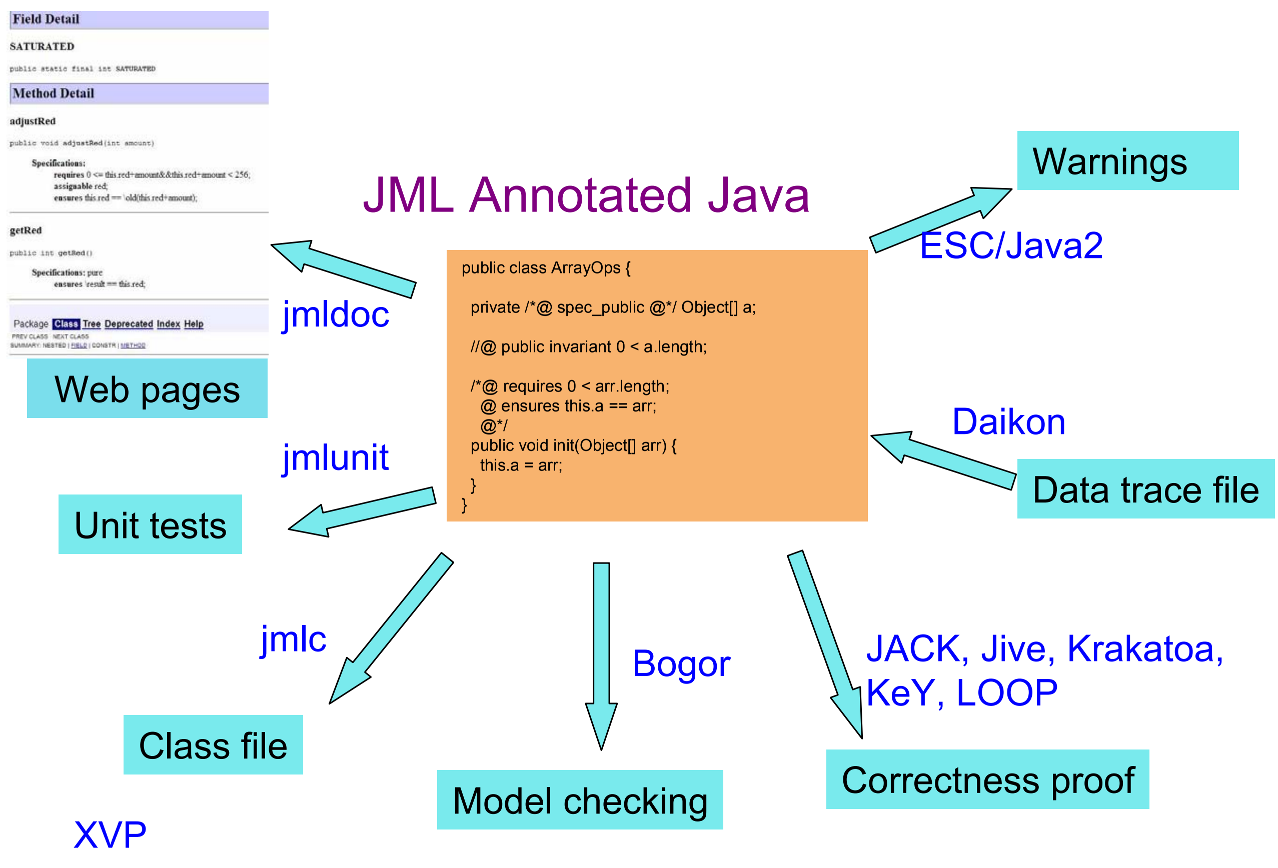
Loop Invariants

- When reasoning about cycles, we need to annotate them with invariants (to establish what is their outcome) and variants (to establish their termination).

```
int f = 1 ;
int i = 1 ;
/*@ loop_invariant i <= n &&
                   f == (\product int j ; 1 <= j && j <= i ; j ) ;
    decreases n-i;
  @*/
while ( i < n ) {
  i = i + 1 ;
  f = f  i ;
}
```

- A loop_invariant express a property that is valid when the control reaches the loop, and is preserved by it;
- The decreases clause expects an integer quantity (that decreses during the loop) --- the loop variant.

JML tools



Runtime Assertion Checking (jmlc/jmlrac)

- jmlrac compiler by Gary Leavens, Yoonsik Cheon, et al. (at Iowa State Univ.)
- translates JML assertions into runtime checks: during execution, all assertions are tested and any violation of an assertion produces an error.
- jmlrac even checks \forall if the domain of quantification is finite.
- jmlrac can generate complicated test-code for free.
- Usage:

```
$ jmlc -Q -e Prog.java
$ jmlrac Prog
```

- Very powerful when combined with unit testing...
  - cheap & easy to do as part of existing testing practice
  - better testing and better feedback, because more properties are tested, at more places in the code

Extended Static Checking (ESC/Java2)

- ESC/Java was originally developed by Rustan Leino (DEC SRC), and extended by David Cok and Joe Kirini (Eastman Kodak Company, University College Dublin).
- Extended static checking = fully automated program verification, with some compromises to achieve full automation.
- It verifies the code at compile time:
  - generates proof-obligations from the annotated code;
  - uses an automated prover (Simplify) to check if generated conditions are provable.
- But, since it is intended to be run in a fully automated manner, has some shortcomings:
  - it is not complete - ESC/Java may warn of errors that are impossible;
  - it is not sound - ESC/Java may miss an error that is actually present.
- ...but finds lots of potential bugs quickly (good at proving absence of runtime exceptions and verifying relatively simple properties).

Using ESC/Java2

- ESC/Java2 can be used:
  - as a stand-alone tool;

```
$ escjava2 Prog.java
...
Prog: Prog() ...
    [0.033 s 17264696 bytes]  passed
  [1.723 s 17264696 bytes total]
1 warning
```

  - as an eclipse plugin... (real-time verification)

- Possible problems detected during analysis are always referred as warnings --- the programmer should judge their pertinence (real problem, lack of capability to derive the property, ...)
- obs.: default loop treatment is very primitive... (escjava unfolds its definition a small number of times).

Static Checking vs. Runtime Checking

- ESC/Java2 checks specs at compile-time, jmlrac checks specs at run-time.
- ESC/Java2 proves correctness of specs, jml only tests correctness of specs. Hence:
  - ESC/Java2 is independent of any test suite, results of runtime testing are only as good as the test suite;
  - ESC/Java2 provides higher degree of confidence.
- But, as soon as we depend on complex properties, ESC/Java2 is no longer able to deal with them. Jmlrac can (maybe with a greater perfomance penalty, but that is something admissible in a testing phase).

Tool Download and Instalation

- Both tools are available for the major operating systems (macosx, linux, windows, ...)
- JML toolset:
  - http://www.jml-specs.org (Download section)
- ESC/Java2 standalone tool:
  - http://kind.ucd.ie/products/opensource/ESCJava2/
- ESC/Java2 Eclipse plugin (eclipse update site):
  - http://kind.ucd.ie/products/opensource/ESCJava2/escjava-eclipse/updates

Demo...