Métodos Formais em Engenharia de Software

Design by Contract and JML: concepts and tools

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

MI/MEI 2008/2009

Talk Outline

- Design by Contract (DBC)
  - contracts in software
  - contract the design
- Java Modelling Language (JML)
  - basic usage
  - tool support
- References

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

Design by Contract

- Introduced by Bertrand Meyer [Mey97] for Eiffel...
- ... as a systematic approach to specifying and implementing object-oriented software components
- Interaction between these components is based on precisely defined specifications of their mutual obligations – the contracts
- Contracts allow for:
  - recording details of method responsibilities and assumptions
  - document intention (specification) of software components (object invariants; pre- and post-conditions of methods; etc.)
  - avoiding constantly checking arguments
  - assigning blame across interfaces

Contracts in Software (I)

- Software Requirements Specification...

```
O diagrama de use cases para a configuração dos parâmetros pode ser visualizado
na Figura 11 e os requisitos pretendidos são os seguintes:
  * Possuir opção para ...
  * ...
```

- Several methodologies targeted to different abstraction levels... (often with an imprecise semantics).
- Purpose: refine them in order to reach clear and unambiguous specifications for each component (contracts)

Input: the DL domain parameters q, r and g associated with the keys s and w'.
Assumptions: private key s, DL domain parameters q, r and g, and public key w' are valid.
Output: the derived shared secret value, which is a nonzero field element $z \in GF(q)$
Operation. The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Compute a field element $z = \exp(w', s)$.
2. Output z as the shared secret value.

Contracts in Software (II)

- Contracts are certainly needed to inform the programmer what are the requirements during the coding process...
- ... but are equally valuable for documenting purposes:

```
/** Calcula valor que, quando multiplicado por ele próprio, se aproxima
  *   do argumento passado à função.
  * @param x : argumento
  * @return valor calculado
  *//
public static double sqrt(double x)
{ … }
```

  - records the specification of the function;
  - ...and details of the API usage.
- Utility depends heavily on the pertinence/quality of descriptions:
  - What does it mean "close to"???
  - Does it works with negative arguments?
  - ...

Contracts in Software (III)

- Ideally, we expect a description language that has:
  - Enough expressive power;
  - Precise meaning;

```
/*@ 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

Advantages of DBC

- Contracts are:
  - more abstract than code (e.g. `sqrt` might be implemented using linear search, Newton's method, …)
  - not necessarily checkable (e.g. quantified over infinite types, or just textual strings…)
  - …but in most cases it is possible to automatically generate verification code for the tests
  - can always be up-to-date with implementations during development
- Allow blame assignment. Who is to blame if:
  - Pre-condition doesn't hold?
  - Post-condition doesn't hold?
- Avoids inefficient defensive checks:

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


More Advantages…

- Modularity of Reasoning:

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

…in order to understand this code we shall:

  - read the methods contracts…
  - instead of looking at "all" the code…

- Evaluate system quality through rigourous testing (specification-driven code) or through formal verification of key subsystems

- Refine design by refining contracts

- (reverse DBC) Can be used to understand/document/improve/maintain an existing code base

Java Modelling Language (JML)

- A Behaviour Interface Specification Language for Java (Gary T. Leavens et al. [BCC+05])
- It permits to:
  - specify behaviour of Java classes
  - record design & implementation decisions
- ...by adding assertions to Java source code
- JML syntax is well integrated with Java:
  - JML assertions are added as comments in .java files, 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 Post-Conditions

- 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 their conjunction;
  - \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 change the internal 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 binding operators: \forall, \exists, \sum, \product, \max, \min, ...

E.g. (\forall int i ; 0<=i && i<N ; a[i]==null)

Expressions and their Meaning (non-exhaustive list)

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)

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

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. pre-conditions or methods post-conditions).
- In short, it specifies who should be blamed if the property does not hold.

DBC and JML

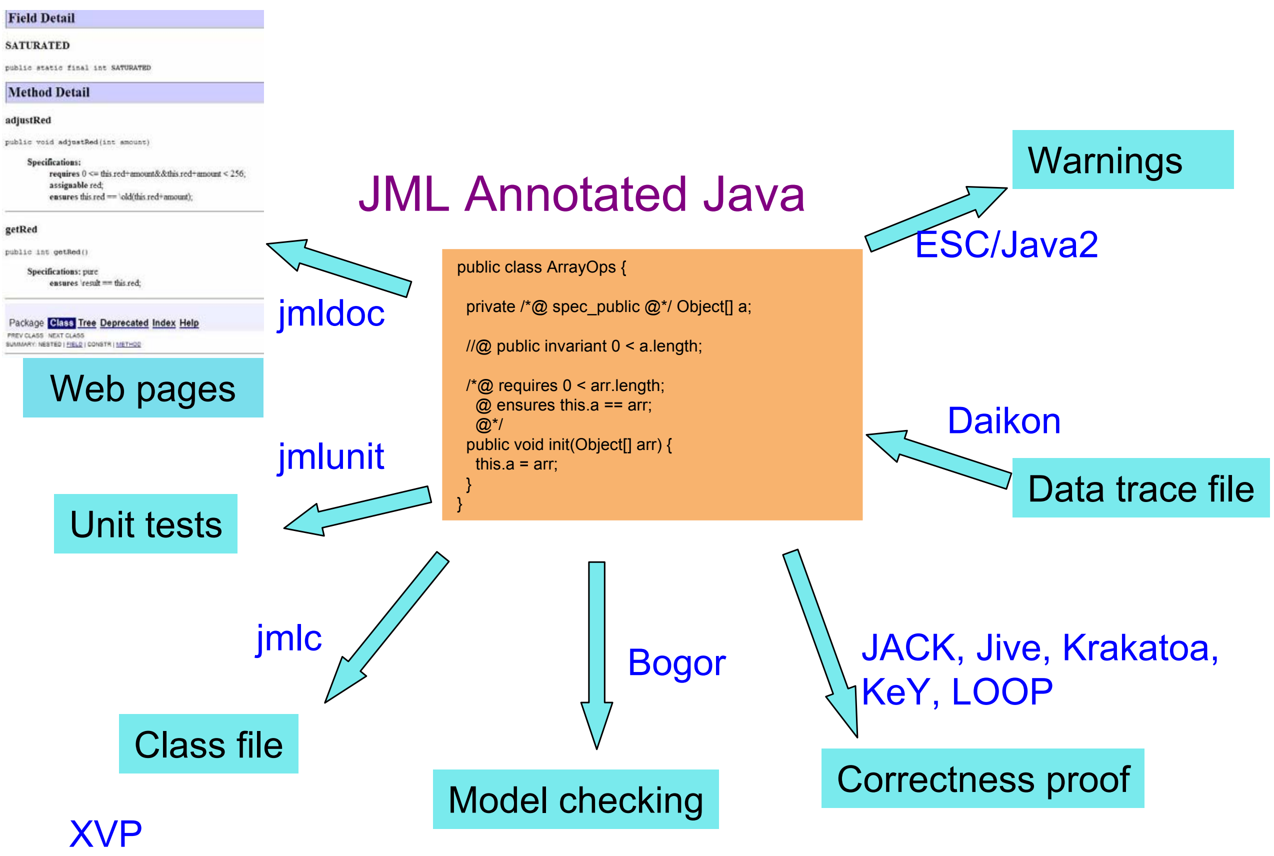
- DBC can roughly be seen as an expansion of pre- and post-conditions as assert and assume clauses.

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

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

- ...expanded into... (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;
...
```


JML Tool Universe



Runtime Assertion Checking (jmlc/jmlrac/jmlunit)

- jmlrac compiler by Gary Leavens, Yoonsik Cheon, et al. (Iowa State Univ.)
- Explore the fact that JML assertions are essentially Java boolean expressions.
- Translates JML assertions into runtime checks
  - performed during execution;
  - all assertions (occurring on the execution path) are tested
  - any violation of an assertion produces an (informative) error
- Checks binding expressions (with finite domains)
- Generates complicated test-code for free (abnormal behaviour, inherited contracts, etc.)
- Particularly powerful when combined with unit testing (jmlunit):
  - 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

Using JML-tools (JML2)

- JML-tools offer replacements to the standard Java compiler and runtime:
  - jmlc --- compiles an instrumented version of the code where JML-assertions are explicitly checked. Replaces javac command;
  - jmlrac --- environment for the execution of jmlc compiled programs (actually, a short script that adds jmlruntime.jar to the class path. Replaces java command.

```
$ jmlc -Q -e Prog.java
$ jmlrac Prog
...
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError: by method
Prog.myMethod
    at Prog.main(Prog.java:1284)
```

- and also:
  - jmldoc
  - jmlunit
  - ...

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 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://sourceforge.net/projects/jmlspecs/
- 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

References

- Meyer, B. - Applying "Design by Contract" - IEEE Computer (1992), 25(10): 40-51
- Leavens, G.; Poll, E.; Clifton, C.; Cheon, Y.; Ruby, C.; Cok, D.; Mueler, P.; Kiniry, J. & Chalin, P. - JML Reference Manual - (Draft), Nov. 2007
- Burdy, L.; Cheon, Y.; Cok, D.; Ernst, M.; Kiniry, J.; Leavens, G.; Leino, K. & Poll, E. - An overview of JML tools and applications - International Journal on Software Tools for Technology Transfer (STTT), Springer, 2005, 7, 212-232.
- Chalin, P.; Kiniry, J.; Leavens, G. & Poll, E. - Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2 - Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05), Springer, 2005, 342-363

Demo...

Exercises:

http://www.cs.ru.nl/~erikpoll/Teaching/JML/bagamount.html