Design by Contract with JML

Joe Kiniry
IT University of Copenhagen
22 September 2010

rheinjug Java User Group

Introduction

The Java Modeling Language (JML)

- initiative of Prof. Gary Leavens [UCF]
- Behavioral Interface Specification Language (BISL) for Java
- annotations for Java programs expressing pre- and postconditions, invariants, etc.
- inspired by Eiffel's DBC and Larch
- primary design goal: easy to learn is a simple extension to Java's syntax

JML Overview

- formal language for expressing the behavior of your Java programs
- focuses on sequential Java
- describes functional behavior of APIs
- supports data and specification refinement
- includes mathematical models as Java library
- old tools: Java 1.4 — new tools: Java 1.6+

Current Work

- large array of tools supporting Java 1.6+
- detailed semantics
- multithreading
- temporal logic

JML's Goals

- practical and effective for describing detailed designs
- works with existing code
- you can use it where it works (not an all-or-nothing approach)
- wide range of tools

Detailed Design Specification

- JML handles
  - inter-module interfaces
  - classes and interfaces
  - data (fields)
  - methods
- JML does not handle
  - user interface
  - architecture
  - packages
  - dataflow
  - design patterns

A JML Example

A JML Example

```
private int balance;
final static int MAX_BALANCE;

/*@ invariant 0 <= MAX_BALANCE &&
              balance < MAX_BALANCE;
  @*/
```

A JML Example

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

A JML Example

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

A JML Example

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

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures     balance ==
                  \old(balance) - amount;
    signals (PurseException)
                  balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures     balance ==
                    \old(balance) - amount;
    signals (PurseException)
                    balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures     balance ==
                    \old(balance - amount);
    signals (PurseException)
                    balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
private byte[] pin;
private byte appletState;
/*@ invariant appletState == PERSONALIZED
              ==>
              pin != null && pin.length == 4 &&
              (\forall int i; 0 <= i && i < 4;
                              0 <= pin[i] && pin[i] <= 9);
  @*/
```

Design by Contract

Specification in Process

- "Contract the Design"
  - you are given an architecture with no specification, little documentation and you must somehow check the system is correct
- "Design by Contract"
  - you are designing and building a system yourself, relying upon existing components and frameworks

Contract the Design

- 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 of Contract the Design

- 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

A Process Outline for Contract the Design

- directly translate high-level architectural constraints into invariants
  - key constraints on data models, custom data structures, and legal requirements
- express medium-level design decisions with invariants and pre-conditions
- use JML models only where appropriate
- generate unit tests for all key data values

Design by Contract

- writing specifications first is difficult but very rewarding in the long-run
  - you *design* the system by writing *contracts*
- a refinement-centric process akin to early instruction in Dijkstra/Hoare approach
- ESC/Java2 works well for checking the consistency of formal designs
- resisting the urge to write code is *hard*

Goals of Design by Contract

- work out application design by writing contracts rather than code
- express design at multiple levels
  - BON/UML ⇨ JML ⇨ JML w/ privacy
- refine design by refining contracts
- write code *once* when architecture is stable

A Process Outline for Design by Contract

- outline architecture by realizing *classifiers* with *classes*
- capture system constraints with invariants
- use JML models only where appropriate
- focus on preconditions over postconditions
- develop test suite for your design by writing a data generator for your types

Assertions

- the **assert** statement is the fundamental construct used to specify the correct behavior of software
- the statement

```
assert S;
```

means

"S **must** be true at **this** point

in the program's execution"

Assertion Syntax in Java

- **all** modern programming languages have an **assert** statement
- beginning in Java 1.4, **assert** is a keyword
- the syntax of a Java assert statement is

```
assert <boolean>[: <String>]
```

- `boolean` is the predicate that **must** be true
- `String` is an optional message that will be printed if/when the assertion fails

Examples of Assertion Use

```
assert z != 0;
x = y/z;

assert (x > MIN_WIDTH);
my_window.setWidth(x);

assert p(x) : “p failed when x=” + x;
a_method_that_depends_upon_p(x);
```

Assertions vs. Logging

- if an assertion fails, the program **halts**
- thus, assertion failures are **critical** failures
- to assert something that is not critical, then a logging message is appropriate

```
if (Debug.DEBUG && !p(x))
  System.err.println("p("+x+") fails");
a_method_that_depends_upon_p(x);
```

Logging Frameworks

- it is **always** wiser to use a logging framework than to use embedded `println`s
- if a `println` must be used, guard it with a conditional on a constant boolean
  - setting the guard false eliminates all logging code (saves space and time)
- the premier logging frameworks are `java.util.logging`, log4J, and IDebug

Specifications

- specifications of software range in formality
  - informal - English documentation (e.g., “normal” comments)
  - semi-formal - structured English documentation (e.g., **Javadoc**)
  - formal - annotations and assertions (e.g,. **assert** statements and **contracts**)
- **contracts** are a **key concept** in robust software design and construction

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?
  - amount is bigger than the balance?
  - is the balanced changed when failure?

Semi-Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  *         <code>amount</code> must be
  *         non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

- many of the same questions arise even though the documentation is much clearer

Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  * @result the resulting balance.
  */
/*@ requires amount >= 0;
  @ ensures balance == \old(balance-amount) &&
  @                    \result == balance;
  @*/
  public int debit(int amount)
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
    return how much money is left */
  public int debit(int amount) {
    if (amount < 0) throw NBE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

HORRIBLE!

Calling Methods Correctly

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

if (debit_amount < 0)
  handle_bad_debit(debit_amount);
else
  resulting_balance = debit(debit_amount);
```

Design by Contract

- capture architectural, class-level decisions early as **constraints**
  - e.g., all `Citizens` have two parents
- realize constraints in software as **invariants**
  - an **invariant** is an assertion that must **always** be true whenever a method is called or exits
- capture contracts at method-level in medium-level design using English
  - realize contracts in code using **requires** and **ensures** statements

An Example Use of Design by Contract

CLASS	*CITIZEN*	**Part:** 1/1
TYPE OF OBJECT Person born or living in a country	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw	
Queries	Name, Sex, Age, Single, Spouse, Children, Parents, Impediment to marriage	
Commands	Marry. Divorce.	
Constraints	Each citizen has two parents. At most one spouse allowed. May not marry children or parents or person of same sex. Spouse s spouse must be this person. All children, if any, must have this person among their parents.	

Related Class Features

- queries
  - spouse? single?
- command
  - marry! divorce!
- constraints
  - at most one spouse is allowed
  - spouse's spouse must be this person

Class Sketch

```
Citizen my_spouse;
//@ invariant (my_spouse != null) ==>
//@           my_spouse.my_spouse == this;

Citizen spouse() { return spouse; }
boolean single() { return spouse == null; }
//@ requires single();
//@ ensures !single() && spouse() == new_spouse;
void marry(Citizen new_spouse)
  { my_spouse = new_spouse;
    new_spouse.my_spouse = this; }
//@ requires !single();
//@ ensures single();
void divorce() { my_spouse.my_spouse = null;
                 my_spouse = null; }
```

Impact of DBC

Impact on Design

- minimality and elegance becomes the natural order of design
  - fewer classes and methods
  - methods that have simpler purpose
  - implementations have lower complexity
- you understand what you are building before you build it!

Impact on Documentation

- less documentation written in English
- less ambiguity in documentation
- documentation is kept in-sync with implementation

Impact on Implementation

- fewer exceptions
- fewer try/catch blocks
- no manual parameter checking
- better error handling
- method bodies shrink dramatically
- Joe's four finger rule:
  - "If you can't cover your method body with four fingers, you probably don't understand it."

Impact on Testing

- specifications mean that no valid parameter testing is necessary in implementations
  - the precondition is **requiring** the client to fulfill their side of the contract for supplier
- when calling a method that has a specification, checking for errors, return values, etc. is no longer necessary
  - the supplier is **ensuring** (guaranteeing) their side of the contract to client

Unit Testing and Programming with Specs

- ~90% of your method-level unit tests are automatically generated
- ~25% **less code** is written because there is no need to test parameters values nor results of method calls for correctness
- code is not littered with `try/catch` blocks to catch exceptions

Tool Support

Analysis & Design and Specification Generation

- BONc = architecture specification in BON language (like mini-UML)
- Beetlz = refinement checker and generator from BON to JML
- Daikon = generate invariants by analyzing the heap at runtime as the system runs unit tests
- Houdini = statically generates simple contracts
- several tools do loop invariant derivation

Documentation and Embedding Specs

- jmldoc = JML + Javadoc
- Umbra/BMLlib = compile JML specs at the source level into BML specs in bytecode
- JMLEclipse = compile JML specs at the source level into JIR specs in bytecode

Runtime Checking

- JML runtime assertion checker
  - JML2 (“classic” JML tool suite) = Java 1.4
  - JML4c = JML on Eclipse JDT
  - OpenJML = JML on OpenJDK
  - JAJML = JML + JastAdd
  - AJML2 = JML + Aspect/J

Dynamic and Static Checking

- unit testing
  - JML-JUnit = JML + JUnit
  - JMLUnitNG = JML + TestNG
- static checking
  - ESC/Java2 = JML + ESC/Java for Java 1.4
  - JMLe = execute JML specs using CSP
  - JMLEclipse = JML + Eclipse JDT for Java 1.6+
  - OpenJML = JML + OpenJDK for Java 1.6+
  - Chase = frame condition checker

Verification

- LOOP tool = verification using HOL in PVS
- Jack = verification of JavaCard applets
- RCC = race condition checker
- Jive = verification using custom prover
- Bandera = verification via model checking
- Kiason = verification via symbolic interpretation
- Krakatoa = verification using Coq and Why
- KeY system = verification using dynamic logic
- Mobius Program Verification Environment (PVE) = verification using HOL and Coq with Proof-Carrying Code (PCC)

Case Studies

Industrial

- JavaCard smart card applets
- electronic voting systems in NL and IE
- VLSI CAD software
- mobile phone applets

- systems and customers I cannot talk about

Academic Pedagogy

- teaching at all levels (freshman to postgrad)
  - introductory programming, programming languages, software engineering, applied formal methods, semantics, etc.
- class projects
  - cellular automaton simulator, digital cash, petrol rationing, supermarket purchase tracking, high-level computer simulators
- student projects
  - The Guinness Simulator
  - verified video games

For More Information

- the JML home page
  - http://www.jmlspecs.org/
- Design by Contract
  - Meyer, B. Object-oriented Software Construction
- Mobius Program Verification Environment
  - http://mobius.ucd.ie/
- ESC/Java2, BONc, and other tools
  - http://kindsoftware.com/products/opensource/

Acknowledgements

K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, Jim Saxe, Raymie Stata, Cormac Flanagan (while at DEC SRC); David Cok (GrammaTech); Carl Pully (ACME-Labs); Cees-Bart Breunesse, Arnout Engelen, Christian Haack, Ichiro Hasuo, Engelbert Hubbers, Bart Jacobs, Martijn Oostdijk, Wolter Pieters, Erik Poll, Joachim van den Berg, Martijn Warnier (RUN); Gilles Barthe, Julien Charles, Benjamin Grigore, Marieke Huisman, Clément Hurlin, Mariela Pavlova, and Gustavo Petri (INRIA); Cesare Tinelli, Jeg Hagen, and Alex Fuches (Univ. of Iowa); Aleksey Schubert (Univ. of Warsaw); Michal Moskał (Wroclaw University); David Naumann (Stevens); Patrice Chalin, Perry James, George Karabotos, Frederic Rioux (Concordia University); Torben Amtoft, Anindya Banerjee, John Hatcliff, Venkatesh Prasad Ranganath, Robby, Edwin Rodríguez, Todd Wallenstein (KSU), Yoonsik Cheon (Univ. of Texas, El Paso), Dan Zimmerman (Univ. of Washington Tacoma), Curtis Clifton (Rose-Hulman), Gary Leavens (UCF), Todd Millstein (UCLA); Claudia Brauchli, Adam Darvas, Werner Dietl, Hermann Lehner, Ovidio Mallo, Peter Müller, Arsenii Rudich (ETHZ); Lorcan Coyle, Steve Neely, Graeme Stevenson, Dragan Stosic (UCD); and my present and past PhD and MSc students Alan Barrett, Dermot Cochran, Fintan Fairmichael, Robin Green, Radu Grigore, Alan Hicks, Ian Hull, Ralph Hyland, Mikoláš Janota, Josu Martinez, Alan Morkan, Ralph Skinner, and undergraduate students Eliott Bartley, Eva Darulova, Barry Denby, Jakub Dostal, Conor Gallagher, Patrick Tierney and the many students in my software engineering, programming, analysis, design, and architecture courses

Thanks!
Questions?