Formal specification:
Design by Contract (DbC)
EECS 3311

Ilir Dema
demailir@yorku.ca

Course Project

- **Course Project**
  - Midterm project
  - Design vs Implementation
    - Problems of your initial design?
    - How to improve?

Review of Last Lecture

- What's DbC?
- DbC with Java
  - Without tool support
  - Lab 3: with tool support (basic)

Outline

- Java Modeling Language

Acknowledge

- Some of the covered materials are based on 22C:181@University of Iowa, 11LE13V-1210@University of Freiburg, and our previous EECS3311 offerings:
    - **Gary T. Leavens, Jochen Hoenicke, Alexander Nutz , Jonathan S. Ostroff, Jackie Wang,** Song W

JML – Java Modelling Language

Design by Contract with JML

Gary T. Leavens

Dept. of Computer Science
Iowa State University
Ames, IA 50010
leavens@cs.iastate.edu

Yoonsik Cheon

Dept. of Computer Science
University of Texas at El Paso
El Paso, TX 79968
cheon@cs.utep.edu

September 28, 2006

JML – Java Modelling Language

- formal specification language for sequential Java by Gary Leavens et. al.
  - – to specify behaviour of Java classes & interfaces
  - – to record detailed design decisions

- by adding annotations to Java source code in Design-By-Contract style, using eg. pre/postconditions and invariants

- Design goal: meant to be usable by any Java programmer

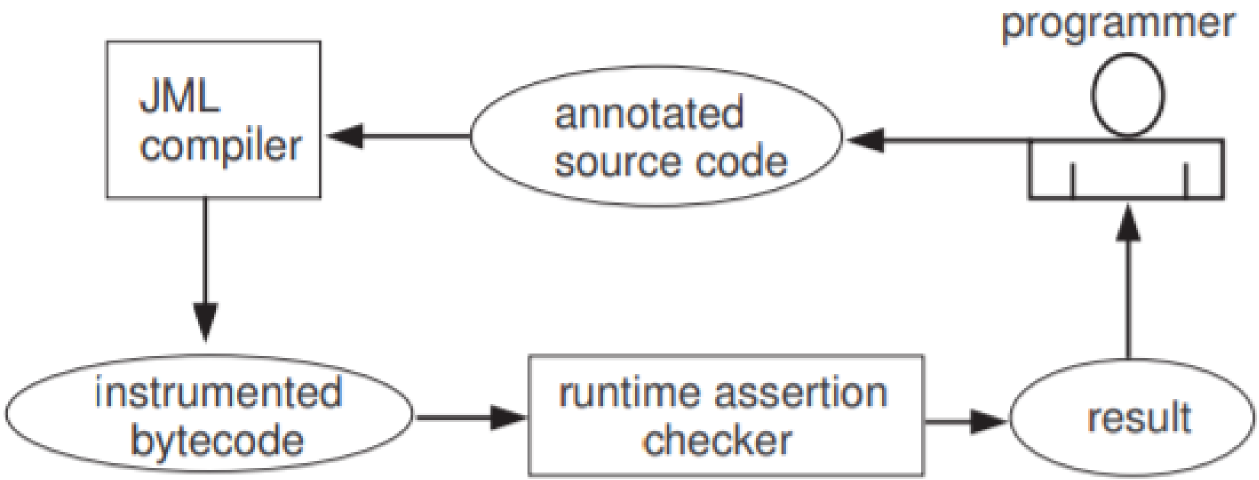
- Lots of info on http://www.jmlspecs.org

JML – Java Modelling Language

- JML is an Extension of Java for Design by Contract
  - http://www.openjml.org/

- Release can be downloaded:
  - http://www.openjml.org/downloads/

- JML compiler, runtime **assertion** checker



Types of Contracts Supported

- Pre-conditions (requires)
- Post-conditions (ensures)
- Loop Invariants/variants (loop_invariant/ decreases)
- Class Invariants (invariant)

Basic of JML

To make JML easy to use & understand:

- Properties specified as comments in .java source file, between `/*@ ... @*/`, or after `//@`
  (or in a separate file, if you don't have the source code, eg. of some API)

- Properties are specified in Java syntax, namely as Java boolean expressions,
  - extended with a few operators (\old, \forall, \result, . . .).
  - using a few keywords (`requires`, `ensures`, `invariant`, `pure`, `non_null`, . . .)

JML Syntax (Method specification)

- requires formula: The specification only applies if formula holds when function called.
  Otherwise behaviour of method is undefined.
- ensures formula: If the function exits normally formula has to hold.
- assignable variables: The function only changes values of variables
- signals (exception) formula: If the function signals exception then formula holds.
- signals_only exceptions: The function may only throw exceptions that are a subtype of one of the exceptions.
  If omitted function can signal only exceptions that appear in throws clause.
- diverges formula: The function may only diverge if formula holds.

JML Formula Syntax

A JML formula is a Java Boolean expression. The Java language is extended by some JML operators:

- \old(expression): The value of expression before the method was called (used in signal and ensures clause)
- \result: The return value (used in ensures clause).
- *F* ==> *G*: States that *F* implies *G*. This is an abbreviation for ! *F* || *G*.
- \forall *Type t*; condition; formula: States that formula holds for all *t* of type *Type* that satisfy condition.

JML Syntax (invariants)

- In JML class/loop invariants are also in /*@ . . . @*/
- Class
  - invariant formula: Whenever a method is called or returns, the invariant has to hold.
  - constraint formula: A relation between the pre-state and the post-state that has to hold for each method invokation.
- Loop
  - loop_invariant formula loop invariant always holds at during the iterations of the loop;
  - decreases expression It specifies an expression of type long or int that must be no less than 0 when the loop is executing, and must decrease by a given step each time around the loop.

Examples JML specification
pure

A method without side-effects is called pure.

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

Pure methods – and only pure methods – can be used *in* JML specifications.

Examples JML specification
requires

Pre-condition for method can be specified using requires:

```
/*@ requires amount >= 0;
  @*/
public int debit(int amount) {
   ...
}
```

Anyone calling debit has to guarantee the pre-condition.

Examples JML specification

ensures

Post-condition for method can be specified using ensures:

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

Anyone calling `debit` can assume postcondition (if method terminates normally, ie. does not throw exception)

`\old(...)` has obvious meaning

Examples JML specification
requires, ensures

Post-condition for method can be specified using ensures:

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

Anyone calling `debit` can assume postcondition (if method terminates normally, ie. does not throw exception)

`\old(...)` has obvious meaning

Examples JML specification
invariant

Invariants (aka *class* invariants) are properties that must be maintained by all methods, e.g.,

```
public class  BankAccount {
   final static int MAX_BAL = 1000;
   int balance;
    /*@ invariant 0 <= balance &&
                             balance <= MAX_BAL;
      @*/
   ...
```

Invariants are implicitly included in all pre- and postconditions.

Invariants must *also* be preserved if exception is thrown!

Examples JML specification
invariant

Another example, from an implementation of a file system:

```
public class Directory {
private File[] files;
/*@ invariant
    files != null
    &&
    (\forall int i; 0 <= i && i < files.length;
                    files[i] != null &&
                    files[i].getParent() == this)
  @*/
```

Examples JML specification
non_null

Many invariants, pre- and postconditions are about references not being `null`. non_null is a convenient short-hand for these.

```
public class Directory {

  private /*@ non_null @*/ File[] files;

  void createSubdir(/*@ non_null @*/ String name)
   ...
  Directory /*@ non_null @*/ getParent(){
   ...
```

Examples JML specification assert

An `assert` clause specifies a property that should hold at some point in the code, e.g.,

```
if (i <= 0 || j < 0) {
     ...
} else if (j < 5) {
     //@ assert i > 0 && 0 < j && j < 5;
     ...
} else {
     //@ assert i > 0 && j > 5;
     ...
}
```

Examples JML specification
signals

Exceptional postconditions can also be specified.

```
/*@ requires amount >= 0;
    ensures  true;
    signals (BankAccountException e)
               amount > balance           &&
               balance == \old(balance)  &&
               e.getReason()==AMOUNT_TOO_BIG;
  @*/
public int debit(int amount) { ...  }
```

Examples JML specification assignable

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

E.g., `debit` can *only* assign to the field `balance`.

Examples JML specification loop_invariant, decreasing

```java
public class BubbleSort {
    /*@
         requires arr != null;
         ensures \forall int k;0 <= k && k < arr.length-1;arr[k] >= arr[k+1];
     @*/
    public static void sort(int [] arr) {
        // bounds
        //@ loop_invariant 0 <= i <= arr.length;
        // elements up-to i are sorted
        //@ loop_invariant \forall int k; 0<= k < i; \forall int l; k < l < n; arr[k] >= arr[l];
        //@ decreasing n-i;
        for (int i = 0; i < arr.length; i++) {
            // bounds
            //@ loop_invariant
            // j-th element is always the largest
            //@ loop_invariant
            // elements up-to i remain sorted
            //@ loop_invariant
            //@ decreasing
            for (int j = arr.length-1; j > i; j--) {
                if (arr[j-1] < arr[j]) {
                    int tmp = arr[j];
                    arr[j] = arr[j-1];
                    arr[j-1] = tmp;
                }
            }
        }
    }
```

Examples JML specification
loop_invariant, decreasing

```
public class BubbleSort {
    /*@
         requires arr != null;
         ensures \forall int k;0 <= k && k < arr.length-1;arr[k] >= arr[k+1];
     @*/
    public static void sort(int [] arr) {
        // bounds
        //@ loop_invariant 0 <= i <= arr.length;
        // elements up-to i are sorted
        //@ loop_invariant \forall int k; 0<= k < i; \forall int l; k < l < n; arr[k] >= arr[l];
        //@ decreasing n-i;
        for (int i = 0; i < arr.length; i++) {
            // bounds
            //@ loop_invariant i <= j <= n-1;
            // j-th element is always the largest
            //@ loop_invariant \forall int k; j <= k < n; arr[j] >= arr[k];
            // elements up-to i remain sorted
            //@ loop_invariant \forall int k; 0 <= k < i; \forall int l; k < l < n; arr[k] >= arr[l];
            //@ decreasing j;
            for (int j = arr.length-1; j > i; j--) {
                if (arr[j-1] < arr[j]) {
                    int tmp = arr[j];
                    arr[j] = arr[j-1];
                    arr[j-1] = tmp;
                }
            }
        }
    }
```

LOTS of freedom in specifying

JML specs can be as strong or weak as you want

Eg for `debit(int amount)`

```
//@ ensures balance == \old(balance)-amount;
//@ ensures balance <= \old(balance);
//@ ensures true;
```

Good bottom-line spec to start: give minimal specs (`requires`, `invariants`) necessary to rule out (Runtime)Exceptions

JML specs can be low(er) level

```
//@ invariant f != null;
```

or high(er) level

```
//@ invariant child.parent == this;
```

Example: Account class

```
public class Account {
  private int bal;

  public Account(int amt) {
    bal = amt;
  }

  public Account(Account acc) {
    bal = acc.balance();
  }

  public void transfer(int amt, Account acc) {
    acc.withdraw(amt);
    deposit(amt);
  }

  public void withdraw(int amt) {
    bal -= amt;
  }

  public void deposit(int amt) {
    bal += amt;
  }

  public int balance() {
    return bal;
  }

   public static void main(String[] args) {
     Account acc = new Account(100);
     acc.withdraw(200);
     System.out.println("Balance after withdrawal: " + acc.balance());
   }
}
```

Example: Account class

```
public class Account {
  private /*@ spec_public @*/ int bal;
  //@ public invariant bal >= 0;

  /*@ requires amt >= 0;
    @ assignable bal;
    @ ensures bal == amt; @*/
  public Account(int amt) {
    bal = amt;
  }

  /*@ assignable bal;
    @ ensures bal == acc.bal; @*/
  public Account(Account acc) {
    bal = acc.balance();
  }

  /*@ requires amt > 0 && amt <= acc.balance();
    @ assignable bal, acc.bal;
    @ ensures bal == \old(bal) + amt
    @   && acc.bal == \old(acc.bal - amt); @*/
  public void transfer(int amt, Account acc) {
    acc.withdraw(amt);
    deposit(amt);
  }

  /*@ requires amt > 0 && amt <= bal;
    @ assignable bal;
    @ ensures bal == \old(bal) - amt; @*/
  public void withdraw(int amt) {
    bal -= amt;
  }

  /*@ requires amt > 0;
    @ assignable bal;
    @ ensures bal == \old(bal) + amt; @*/
  public void deposit(int amt) {
    bal += amt;
  }

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

    public static void main(String[] args) {
        Account acc = new Account(100);
        acc.withdraw(200);
        System.out.println("Balance after withdrawal: " + acc.balance());
    }
}
```

Exercise: JML specification for arraycopy

```
/*@ requires ... ;
     ensures ... ;
  @*/
static void arraycopy (int[] src,  int srcPos,
                       int[] dest, int destPos,
                       int len)
 throws NullPointerException,
        ArrayIndexOutOfBoundsException;
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Exercise: JML specification for arraycopy

```
/*@ requires src != null && dest != null &&
             0 <= srcPos && srcPos + len < src.length &&
             0 <= destPos && srcPos + len < dest.length;

    ensures (\forall int i; 0 <= i && i < len;
                 dest[dstPos+i] == \old(src[srcPos+i])) &&
               (* rest unchanged *)
  @*/
 static void arraycopy (int[] src,  int srcPos,
                        int[] dest, int destPos,
                        int len);
```

Forward, Reverse Implication Operators

- The operators ==> and <== work only on boolean-subexpressions. They compute forward and reverse implications, respectively.
  - raining ==> getsWet
  - getsWet <== raining

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Forward, Reverse Implication Operators

- The operators ==> and <== work only on boolean-subexpressions. They compute forward and reverse implications, respectively.

  - raining ==> getsWet
  - getsWet <== raining

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

  - These two operators are evaluated in short-circuit fashion, left to right. Thus, in a ==> b, if a is false, then the expression is true and b is not evaluated. Similarly, in a <== b, if a is true, the expression is true and b is not evaluated. In other words, a ==> b is equivalent to !a || b and a <== b is equivalent to a || !b.

Equivalence and Inequivalence Operators

- The operators <==> and <=!=> work only on boolean-subexpressions and have the same meaning as == and !=, respectively.

```
\result == (size == 0)
\result <==> size == 0
```

if and only if

Quantified expressions

- A quantified expression determines whether some or all of the items in a sequence meet a particular condition.

spec-quantified-expr ::= (*quantifier quantified-var-decls* ;
  [[*predicate*] ;]
  spec-expression)
quantifier ::= `\forall` | `\exists`
  | `\max` | `\min`
  | `\num_of` | `\product` | `\sum`
quantified-var-decls ::= [*bound-var-modifiers*] *type-spec quantified-var-declarator*
  [, *quantified-var-declarator*] ...
quantified-var-declarator ::= *ident* [*dims*]
spec-variable-declarators ::= *spec-variable-declarator*
  [, *spec-variable-declarator*] ...
spec-variable-declarator ::= *ident* [*dims*]
  [= *spec-initializer*]
spec-array-initializer ::= { [*spec-initializer*
  [, *spec-initializer*] ... [,]] }
spec-initializer ::= *spec-expression*
  | *spec-array-initializer*

Mathematically...

- (\forall T x; P; Q) $= \bigwedge\{Q \mid x \in T \wedge P\}$ Conjunction
- (\exists T x; P; Q) $= \bigvee\{Q \mid x \in T \wedge P\}$ Disjunction
- (\min T x; P; E) $= \min\{E \mid x \in T \wedge P\}$
- (\sum T x; P; E) $= \sum\{E \mid x \in T \wedge P\}$
- (\num_of T x; P; Q) $= \sum\{1 \mid x \in T \wedge P \wedge Q\}$

Universal and Existential Quantifiers

```
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

means ?

```
(\exists int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

means ?

The body of a universal or existential quantifier must be of type boolean.

Universal and Existential Quantifiers

```
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

means ?

```
(\exists int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

means ?

```
(\exists|forall Object x; someSet.has(x); ...)
```

```
(\forall Student s; juniors.contains(s); s.getAdvisor() != null)

(\exists Student s; juniors.contains(s); s.getAdvisor() != null)
```

Generalized Quantifiers

- The quantifiers **\max, \min, \product, and \sum**, are generalized quantifiers that return the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range.

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4

(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4

(\max int i; 0 <= i && i < 5; i) == 4

(\min int i; 0 <= i && i < 5; i-1) == -1
```

Numerical Quantifier

- The numerical quantifier, \num_of, returns the number of values for its variables for which the range and the expression in its body are true.

```
(\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1)
```

https://www.openjml.org/examples/

The following links show various working or tutorial examples of Java programs with OpenJML annotations,

- Binary search
- Invert injection
- Max by elimination
- Two Sum
- Sum-Max
- Selection Sort
- Merge Sort
- Heap Sort
- Bubble Sort
- VerifyThis 2019 #2A: Nearest Smaller Neighbor

BinarySearch Example

```java
public class BinarySearchGood {

    public static int search(int[] sortedArray, int value) {

        if (value < sortedArray[0]) return -1;
        if (value > sortedArray[sortedArray.length-1]) return -1;
        int lo = 0;
        int hi = sortedArray.length-1;

        while (lo <= hi) {
            int mid = lo + (hi-lo)/2;
            if (sortedArray[mid] == value) {
                return mid;
            } else if (sortedArray[mid] < value) {
                lo = mid+1;
            } else {
                hi = mid-1;
            }
        }
        return -1;
    }
}
```

Contracts for BinarySearch

- public static int search(int[] sortedArray, int value)

- Pre1: sortedArray is not null and the number of elements in sortedArray is small than MAX_Integer
- Pre2: sortedArray is sorted

- Post1: return value is in [0, sortedArray.length] if value is in sortedArray
- Post2 : return value is -1 if value is not in sortedArray

Contracts for BinarySearch

- public static int search(int[] sortedArray, int value)
- Pre1:
- Pre2:
- Post1:
- Post2:

Contracts for the While loop

- Inv1: both “lo” and “hi” should be in the range: [0, sortedArray.length]

- Inv2: if value is in sortedArray, value should in the range of [sortedArray[lo], sortedArray[hi]]

- Inv3: value should be larger than all the element before lo

- Inv4: value should be smaller than any element after hi

Contracts for the While loop

- Inv1:
- Inv2:
- Inv3:
- Inv4: