Module III: Assured Software Analytics
Lecture 1: Design by Contract

Jüri Vain, Leonidas Tsiopoulos

2017 fall

Slides are based on Tevfik Bultan lectures

Software Assurance

Application of technologies and processes to achieve a required level of confidence that software systems and services function in the intended manner, are free from accidental or intentional vulnerabilities, provide security capabilities appropriate to the threat environment, and recover from intrusions and failures.

Level of confidence

- The word "*confidence*" implies that there is a *basis* for the belief that software systems and services function in the intended manner.

- A key responsibility of software assurance is to create *auditable evidence* that supports achievement of assurance goals.

Assurance case

A documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a system's properties are adequately justified for a given application in a given environment.

Case review

- An assurance case developer is expected to justify through evidence that a set of claims have been met
  - Doubt the *claim*
  - Doubt the *argument*
  - Doubt the *evidence*

Software Assurance

Application of technologies and processes to achieve a required level of confidence that software systems and services function in the intended manner, are free from accidental or intentional vulnerabilities, provide security capabilities appropriate to the threat environment, and recover from intrusions and failures.

System functionality assurance

- Assured software must provide correct functionality
  - Correct and complete requirements
  - Reduce errors in software development

Design by Contract

- Design by Contract and the language that implements the Design by Contract principles (called Eiffel) was developed in Santa Barbara by Bertrand Meyer
- Bertrand Meyer won the 2006 ACM Software System Award for the Eiffel programming language!
  - Award citation: "*For designing and developing the Eiffel programming language, method and environment, embodying the Design by Contract approach to software development and other features that facilitate the construction of reliable, extendible and efficient software*."
- The company which supports the Eiffel language is located in Santa Barbara:
  - Eiffel Software (http://www.eiffel.com)
- The material in the following slides is mostly from the following paper:
  - "Applying Design by Contract," B. Meyer, IEEE Computer, pp. 40-51, October 1992.

Dependability and Object-Orientation

- An important aspect of object oriented design is reuse
  - For reusable components correctness is crucial since an error in a module can effect every other module that uses it

- Main goal of object oriented design and programming is to improve the quality of software
  - The most important quality of software is its **dependability**

- Design by contract presents a set of principles to produce dependable and robust OO software
  - Basic design by contract principles can be used in any OO programming language

What is a Contract?

- There are two parties:
  - Client which requests a service
  - Supplier which supplies the service

- Contract is the agreement between the client and the supplier

- Two major characteristics of a contract
  - Each party expects some *benefits* from the contract and is prepared to incur some *obligations* to obtain them
  - These benefits and obligations are documented in a contract document

- Benefit of the client is the obligation of the supplier, and vice versa.

What is a Contract?

- As an example let's think about the contract between a tenant and a landlord

Party	Obligations	Benefits
Tenant	Pay the rent at the beginning of the month.	Stay at the apartment.
Landlord	Keep the apartment in a habitable state.	Get the rent payment every month.

What is a Contract?

- A contract document between a client and a supplier protects both sides
  - It protects the client by specifying how much should be done to get the benefit. The client is entitled to receive a certain result.
  - It protects the supplier by specifying how little is acceptable. The supplier must not be liable for failing to carry out tasks outside of the specified scope.

- If a party fulfills its obligations it is entitled to its benefits
  - *No Hidden Clauses Rule*: no requirement other than the obligations written in the contract can be imposed on a party to obtain the benefits

How Do Contracts Relate to Software Design?

- You are not in law school, so what are we talking about?
- Here is the basic idea
  - One can think of pre and post conditions of a procedure as obligations and benefits of a contract between the client (the caller) and the supplier (the called procedure)
- Design by contract promotes using pre and post-conditions (written as assertions) as a part of module design
- Eiffel is an object oriented programming language that supports design by contract
  - In Eiffel the pre and post-conditions are written using require and ensure constructs, respectively

Contracts

- The pre and postconditions are assertions, i.e., they are expressions which evaluate to true or false
  - The precondition expresses the requirements that any call must satisfy
  - The postcondition expresses the properties that are ensured at the end of the procedure execution

- If there is no precondition or postcondition, then the precondition or postcondition is assumed to be true (which is equivalent to saying there is no pre or postcondition)

Assertion Violations

- What happens if a precondition or a postcondition fails (i.e., evaluates to false)
  - The assertions can be checked (i.e., monitored) dynamically at run-time to debug the software
  - A ***precondition violation*** would indicate a bug at the ***caller***
  - A ***postcondition violation*** would indicate a bug at the ***callee***

- Our goal is to prevent assertion violations from happening
  - The pre and postconditions are not supposed to fail if the software is correct
    - hence, they differ from exceptions and exception handling
  - By writing the contracts explicitly, we are trying to avoid contract violations, (i.e, failed pre and postconditions)

Defensive Programming vs. Design by Contract

- Defensive programming is an approach that promotes putting checks in every module to detect unexpected situations

- This results in redundant checks (for example, both caller and callee may check the same condition)
  - A lot of checks makes the software more complex and harder to maintain

- In Design by Contract the responsibility assignment is clear and it is part of the module interface
  - prevents redundant checks
  - easier to maintain
  - provides a (partial) specification of functionality

Design by Contract in Eiffel

In Eiffel procedures are written is in the following form:

```
procedure_name(argument declarations) is
    -- Header comment
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```

Design by Contract in Eiffel

An example:

```
put_child(new_child: NODE) is
    -- Add new to the children of current node
require
    new_child /= Void
do
    ... Insertion algorithm ...
ensure
    new_child.parent = Current;
    child_count = old child_count + 1
end  -- put_child
```

- `Current` refers to the current instance of the object (`this` in Java)
- `Old` keyword is used to denote the value of a variable on entry to the procedure
- Note that "=" is the equality operator (== in Java) and "`/=`" is the inequality operator (`!=` in Java)

The put_child Contract

- The put_child contract in English would be something like the table below.
  - Eiffel language enables the software developer to write this contract formally using require and ensure constructs

Party	Obligations	Benefits
Client	Use as argument a reference, say `new_child`, to an existing object of type `Node`.	Get an updated tree where the `Current` node has one more child than before; `new_child` now has `Current` as its parent.
Supplier	Insert `new_child` as required.	No need to check if the argument actually points to an object.

Class Invariants

- A class invariant is an assertion that holds for all instances (objects) of the class
  - A class invariant must be satisfied after creation of every instance of the class
  - The invariant must be preserved by every method of the class, i.e., if we assume that the invariant holds at the method entry it should hold at the method exit
  - We can think of the class invariant as conjunction added to the precondition and postcondition of each method in the class

- For example, a class invariant for a binary tree could be (in Eiffel notation)

```
invariant
  left /= Void implies (left.parent = Current)
  right /=Void implies (right.parent = Current)
```

Design by Contract and Inheritance

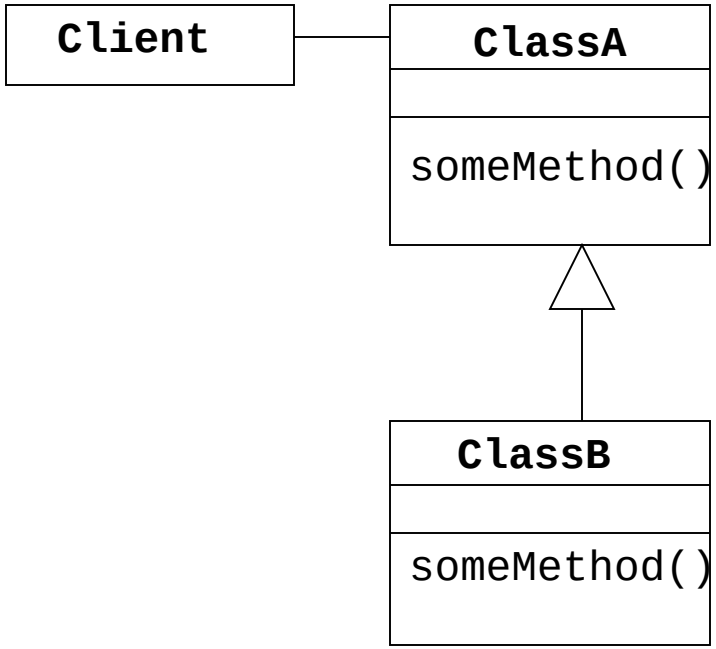
- Inheritance enables declaration of subclasses which can redeclare some of the methods of the parent class, or provide an implementation for the abstract methods of the parent class

- Polymorphism and dynamic binding combined with inheritance are powerful programming tools provided by object oriented languages
  - How can the Design by Contract can be extended to handle these concepts?

Inheritance: Preconditions

- If the **precondition** of the `ClassB.someMethod` is **stronger** than the precondition of the `ClassA.someMethod`, then this is not fair to the `Client`

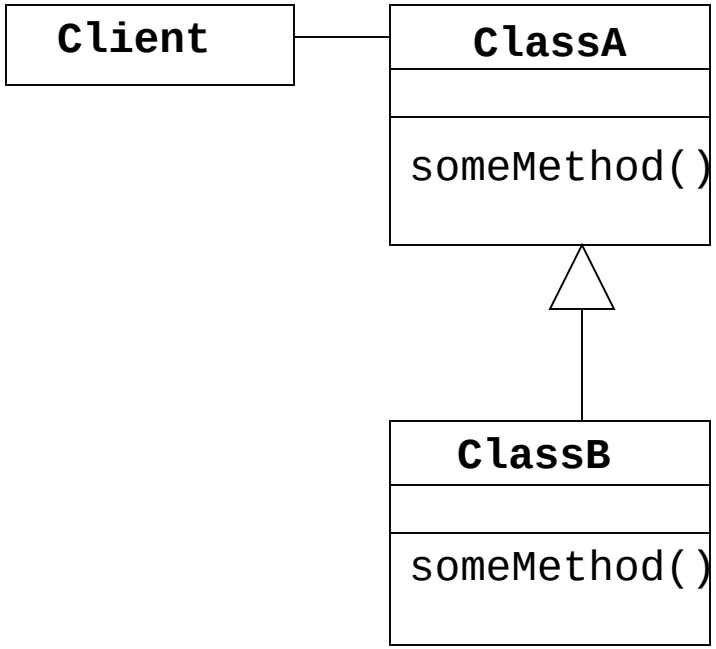
- The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`



Inheritance: Postconditions

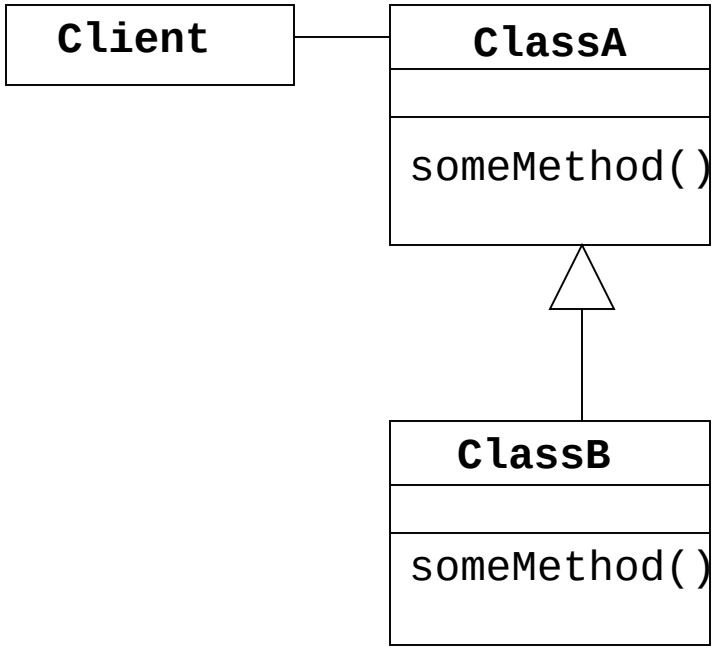
- If the **postcondition** of the `ClassB.someMethod` is **weaker** than the postcondition of the `ClassA.someMethod`, then this is not fair to the `Client`

- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA.someMethod`



Inheritance: Invariants

- If the class **invariant** for the `ClassB` is **weaker** than the class invariant for the `ClassA`, then this is not fair to the `Client`
- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA`



Behavioral Subtyping

- These inheritance rules in design-by-contract are related to the concept of *behavioral subtyping*
  - Given a program that has a type T, and a type S where S is a subtype of T, if you change the type of objects with type T in the program to the type S, **then the behavior of the program should not change**

- This is not enforced in object-oriented programming languages
  - In general it would be undecidable to check if a program conforms to behavioral subtyping

- The inheritance rules in design-by-contract ensure that the contracts follow the behavioral subtyping principle

Inheritance in Eiffel

- Eiffel enforces the following
  - the precondition of a derived method to be weaker
  - the postcondition of a derived method to be stronger

- In Eiffel when a method overwrites another method the new declared precondition is combined with previous precondition using disjunction

- When a method overwrites another method the new declared postcondition is combined with previous postcondition using conjunction

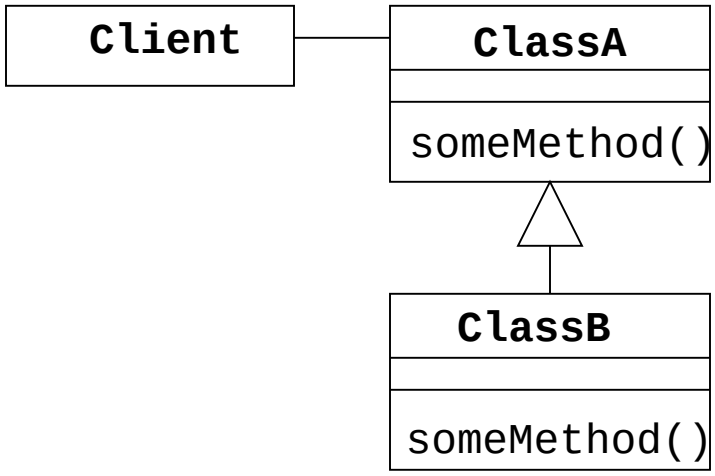
- Also, the invariants of the parent class are passed to the derived classes
  - invariants are combined using conjunction

In `ClassA`:

```
invariant
    classInvariant
someMethod() is
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```

In `ClassB` which is derived from `ClassA`:

```
invariant
    newClassInvariant
someMethod() is
require
    newPrecondition
do
    Procedure body
ensure
    newPostcondition
end
```



The precondition of `ClassB.aMethod` is defined as:

```
newPrecondition or Precondition
```

The postcondition of `ClassB.aMethod` is defined as:

```
newPostcondition and Postcondition
```

The invariant of `ClassB` is

```
classInvariant and newClassInvariant
```

Dynamic Design-by-Contract Monitoring

- Enforce contracts at run-time
- A contract
  - Preconditions of modules
    - What conditions the module requests from the clients
  - Postconditions of modules
    - What guarantees the module gives to clients
  - Invariants of the objects
- Precondition violation, the client is to blame
  - Generate an error message blaming the client (caller)
- Postcondition violation, the server is to blame
  - Generate an error message blaming the server (callee)
- Eiffel compiler supports dynamic design-by-contract monitoring. You can run the program with design-by-contract monitoring on, and it will report any contract violations are runtime

Design-by-Contract Java

- There are dynamic design-by-contract monitoring tools for Java
  - preconditions, postconditions and class invariants are written as Java predicates (Java methods with no side effects, that return a boolean result)
  - Tool: JContractor (http://jcontractor.sourceforge.net/) developed by Murat Karaorman from UCSB

- Given the precondition, postcondition and class invariant methods, dynamic design-by-contract monitoring tools instrument the program to track contract violations and report any contract violations at runtime

- A different approach to writing design-by-contract specifications is to use an annotation language
  - An annotation language is a language which has a formal syntax and semantics but written as a part of the comments in a program
    - So it does not interfere with the program execution and can be completely ignored during compilation
    - However, specialized compilers and tools can interpret the annotations

Java Modeling Language (JML)

- JML is a behavioral interface specification language
- The Application Programming Interface (API) in a typical programming language (for example consider the API of a set of Java classes) provides very little information
  - The method names and return types, argument names and types
- This type of API information is not sufficient for figuring out what a component does
- JML is a specification language that allows specification of the behavior of an API
  - not just its syntax, but its semantics
- JML specifications are written as *annotations*
  - As far as Java compiler is concerned they are comments but a JML compiler can interpret them

JML Project(s) and Materials

- Information about JML and JML based projects are available at Gary Leavens' website:
  - http://www.cs.ucf.edu/~leavens/JML/

- These lecture notes are based on:
  - Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005
  - Slides by Yoonsik Cheon
  - JML tutorials by Joe Kiniry

JML

- One goal of JML is to make it easily understandable and usable by Java programmers, so it stays close to the Java syntax and semantics whenever possible

- JML supports design by contract style specifications with
  - Pre-conditions
  - Post-conditions
  - Class invariants

- JML supports quantification (`\forall`, `\exists`), and specification-only fields and methods
  - Due to these features JML specifications are more expressive than Eiffel contracts and can be made more precise and complete compared to Eiffel contracts

JMLAnnotations

- JML assertions are added as comments to the Java source code
  - either between `/*@ . . . @*/`
  - or after `//@`
    - These are ***annotations*** and they are ignored by the Java compiler

- In JML properties are specified as Java boolean expressions
  - JML provides operators to support design by contract style specifications such as `\old` and `\result`
  - JML also provides quantification operators (`\forall`, `\exists`)

- JML also has additional keywords such as
  - `requires, ensures, signals, assignable, pure, invariant, non null, . . .`

Design by Contract in JML

- In JML constracts:
  - Preconditions are written as a `requires` clauses
  - Postconditions are written as `ensures` clauses
  - Invariants are written as `invariant` clauses

JML assertions

- JML assertions are written as Java expressions, but:
  - Cannot have side effects
    - No use of =, ++, `--`, etc., and
    - Can only call *pure* methods (i.e., methods with no side effects)

- JML extentions to Java expression syntax:

Syntax	**Meaning**
`\result`	the return value for the method call
`\old(E)`	value of `E` just before the method call
`a ==> b`	a implies b
`a <== b`	b implies a
`a <==> b`	a if and only if b
`a <=!=> b`	`!(a <==> b)`

JML quantifiers

- JML supports several forms of quantifiers
  - Universal and existential (`\forall` and `\exists`)
  - General quantifiers (`\sum`, `\product`, `\min`, `\max`)
  - Numeric quantifier (`\num_of`)

```
(\forall Student s; class272.contains(s);
                    s.getProject() != null)
```

```
(\forall Student s; class272.contains(s) ==>
                    s.getProject() != null)
```

- Without quantifiers, we would need to write loops to specify these types of constraints

JML Quantifiers

- Quantifier expressions
  - Start with a decleration that is local to the quantifier expression

```
(\forall Student s;
```

  - Followed by an optional range predicate

```
class272.contains(s);
```

  - Followed by the body of the quantifier

```
s.getProject() != null)
```

JML Quantifiers

- `\sum`, `\product`, `\min`, `\max` return the sum, product, min and max of the values of their body expression when the quantified variables satisfy the given range expression

- For example,

  `(\sum int x; 1 <= x && x <= 5; x)` denotes the sum of values between 1 and 5 inclusive

- The numerical quantifier, `\num_of`, returns the number of values for quantified variables for which the range and the body predicate are true

JML Example: Purse

```
public class Purse {

  final int MAX_BALANCE;
  int balance;
  //@ invariant 0 <= balance && balance <= MAX_BALANCE;

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

  . . .

}
```

JML Invariants

- Invariants (i.e., class invariants) must be maintained by all the methods of the class
  - Invariants must be preserved even when an exception is thrown
- Invariants are implicitly included in all pre and post-conditions
  - For constructors, invariants are only included in the post-condition not in the pre-condition. So, the constructors ensure the invariants but they do not require them.
- Invariants document design decision and makes understanding the code easier

Invariants for non-null references

- Many invariants, pre- and post-conditions are about references not being null.
  - The `non_null` keyword 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(){
  ...
}
```

JML Example: Purse, Cont'd

```
/*@ requires amount >= 0;
  @ assignable balance;
  @ ensures balance == \old(balance) - amount
  @                 && \result == balance;
  @ signals (PurseException) balance == \old(balance);
@*/
int debit(int amount) throws PurseException {
  if (amount <= balance) { balance -= amount; return balance; }
  else { throw new PurseException("overdrawn by " + amount); }
}
```

- The `assignable` clause indicates that `balance` is the only field that will be assigned
  - This type of information is very useful for analysis and verification tools
  - The default assignable clause is: `assignable \everything`

JML post conditions

- The keyword `\old` can be used to refer to the value of a field just before the execution of the method
- The keyword `\result` can be used to refer to the return value of the method
- Both of these keywords are necessary and useful tools for specifying post conditions

Exceptions in JML

- In addition to normal post-conditions, JML also supports exceptional posticonditions
  - Exceptional postconditions are written as `signals` clauses

- Exceptions mentioned in throws clause are allowed by default, i.e. the default signals clause is

  ```
  signals (Exception) true;
  ```

  - To rule them out, you can add an explicit

    ```
    signals (Exception) false;
    ```

  - or use the keyword `normal_behavior`

    ```
    /*@ normal_behavior
      @ requires ...
      @ ensures ...
      @*/
    ```

JML Example: Purse, Cont'd

```
/*@ requires p != null && p.length >= 4;
  @ assignable \nothing;
  @ ensures \result <==> (\forall int i; 0 <= i && i < 4;
  @                                      pin[i] == p[i]);
  @*/
boolean checkPin(byte[] p) {
  boolean res = true;
  for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
  return res;
}
```

JML Example: Purse, Cont'd

```
/*@ requires 0 < mb && 0 <= b && b <= mb
  @          && p != null && p.length == 4
  @          && (\forall int i; 0 <= i && i < 4;
  @                              0 <= p[i] && p[i] <= 9);
  @ assignable MAX_BALANCE, balance, pin;
  @ ensures MAX_BALANCE == mb && balance == b
  @         && (\forall int i; 0 <= i && i < 4; p[i] == pin[i]);
  @*/
Purse(int mb, int b, byte[] p) {
  MAX_BALANCE = mb; balance = b; pin = (byte[]) p.clone();
}
```

Model variables

- In JML one can declare and use variables that are only part of the specification and are not part of the implementation

- For example, instead of a `Purse` assume that we want to specify a `PurseInterface`
  - We could introduce a model variable called `balance` in order to specify the behavioral interface of a Purse
  - Then, a class implementing the `PurseInterface` would identify how its representation of the balance relates to this model variable

JML Libraries

- JML has an extensive library that supports concepts such as sets, sequences, and relations.

- These can be used in JML assertions directly without needing to re-specify these mathematical concepts

JML & Side-effects

- The semantics of JML forbids side-effects in assertions.
  - This both allows assertion checks to be used safely during debugging and supports mathematical reasoning about assertions.

- A method can be used in assertions only if it is declared as `pure`, meaning the method does not have any side-effects and does not perform any input or output.

- For example, if there is a method `getBalance()` that is declared as

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

  then this method can be used in the specification instead of the field balance.

- Note that for pure methods, the assignable clause is implicitly

```
assignable \nothing
```

Assert clauses

- The `requires` clauses are used to specify conditions that should hold just before a method execution, i.e., preconditions
- The `ensures` clauses are used to specify conditions that should hold just after a method execution, i.e., postconditions
- An `assert` clause can be used to specify a condition that should hold at some point in the code (rather than just before or right after a method execution)

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

Assert in JML

- Although assert is also a part of Java language now, assert in JML is more expressive

```
for (n = 0; n < a.length; n++)
   if (a[n]==null) break;
/*@ assert (\forall int i; 0 <= i && i < n;
  @                          a[i] != null);
  @*/
```

JML Tools

- There are tools for parsing and type-checking Java programs and their JML annotations
  - JML compiler (**`jmlc`**)
- There are tools for supporting documentation with JML
  - HTML generator (**jmldoc**)
- There are tools for runtime assertion checking:
  - Test for violations of assertions (pre, postconditions, invariants) during execution
  - Tool: **jmlrac**
- There are testing tools based on JML
  - JML/JUnit unit test tool: **jmlunit**
- Automated verification:
  - Automatically prove that contracts are never violated at any execution
  - Automatic verification is done statically (i.e., at compile time) using theorem proving
  - Tool: **ESC/Java**
- Automatically inferring specifications:
  - Monitor the program execution and infer specifications
  - Tool: **Daikon**