

# A Toolset for Design By Contract For Java

Jing Wang, Lizhang Qin, Naveen Vemuri, Xiaoping Jia

DePaul University, School of Computer Science, Telecommunications and Information Systems  
243 So. Wabash Avenue, Chicago IL 60604

## Abstract

*Design By Contract (DBC) has been proved to greatly benefit software development. However, Java does not natively support DBC. The DBC toolset in our project provides a comprehensive solution for DBC and Java. The DocGen component generates documentation, including the contracts propagated from inherited classes. The ContractChecker component generates test code into the original Java code, thus enabling runtime validation. The StaticAnalyzer component statically checks for possible null pointer exceptions, array out of bounds exceptions, and contract adherence within methods.*

**Keywords** Design By Contract, Java, Documentation, Runtime Contract Checker, Static Analysis

## 1. Introduction

DBC was first introduced by Bertrand Meyer as part of the Eiffel Project. Under the DBC theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations – contracts [1]. DBC provides a formal way to incorporate specification information, which is obtained from comments, into the code. By doing this, the code’s implicit contracts are transformed into explicit requirements that must be satisfied.

The three key elements of DBC are method preconditions, method postconditions and class invariants. Method preconditions are conditions that the client must meet before a method is invoked. Method postconditions are conditions that a method must meet after its execution. Class invariants are consistency conditions of objects, which must hold for all instances.

DBC provides a systematic approach to building bug-free object oriented code. It provides a specific method for documenting software components and provides an effective framework for debugging testing, and quality assurance as well. It also provides better control over inheritance, as subclasses must satisfy the existing contracts of parent classes. All of these benefits of Design by Contract lead to more reliable, extensible code [2][3].

It is well known that Java does not support DBC, though the “assert” statement introduced in JDK 1.4 does represent a step in this direction. Assert provides a mean for programmers to provide runtime checking for their programs. Assert, however, does not provide contract information to indicate the functionalities of a method or class such as method preconditions, postconditions and class invariants, which are emphasized by DBC. The DBC toolset in this project is designed to incorporate DBC capability into the Java language, not only providing the conventions for DBC specification, but also consisting of three major components: *DocGen*,

*ContractChecker*, and *StaticAnalyzer*, which are intended to make use of the DBC information in the programs.

*DocGen* is a Java doclet developed on the basis of the standard doclet. It is able to extract and display any contract, including any inherited contract that is documented in a formal comment unit, as long as the documentation follows a certain format. *DocGen* has several major features that make it different from other similar tools: 1) It is based on the newest version of Java doclet, 2) It automatically processes all inherited contracts; 3) It allows some degree of flexibility in terms of where the contract should be documented.

*ContractChecker* does runtime checking for the contracts by generating new pieces of code and inserting them into the original code, so when executing the new code, any violations against the contract information programmer provide will be reported and avoid further severe exceptions.

*StaticAnalyzer* statically analyzes the program for potential null pointer and array out of bounds exceptions and also incorporates contract verification for each Java class and method. Contract verification indicates whether a specific method is able to satisfy the postconditions if the method's preconditions are assumed to be true. If the postconditions are not guaranteed given satisfied preconditions the contract is invalid.

## 2. Conventions to Specify Contract Information

The essential issue to incorporate DBC support to Java is to introduce the conventions to specify contract information. In our convention, any contract has to be commented according to a certain specification. Briefly, besides the Java documentation convention, the following rules must be followed:

- The contract should only be placed in a formal Java comment unit, which is wrapped by `/**` and `*/`. Any information in other comment form will be ignored
- Certain custom tags are used to express these contracts.
- Different contracts are placed at specific levels of the source code. Contracts that are placed in the wrong place will be ignored.

The advantage of denoting DBC information in comment form instead of within normal code (such as by creating a DBC API, for example) is that DBC enabled Java code with comment-based contract information is backwards-compatible with any Java compiler.

All contract information starts with a custom tag which indicates the type of this contract, such as method precondition, method postcondition, or class invariant. The custom tag is followed by a boolean expression which specifies the condition to be held.

### 2.1 Custom Tags

In the traditional DBC world, the three well known kinds of contracts are method precondition, method postcondition and class invariant. These custom tags and their allowed locations are specified in table 1. Method preconditions and postconditions are allowed only before the method to which they apply. A class invariant applies to a Java class as a whole, so it is allowed before a Java class. In practice, however, most class invariants contain information about the class fields. For programmers' convenience, an invariant tag is also allowed to appear before a field.

In addition to the standard tags shown in table 1, we expand the family of custom tags used to specify the contract information. In table 2, three more tags are introduced. These tags are @assert, @fact, and @loopInvariant.

### Basic Contract Tags

Tag	Description	Allowed Location
@pre	Method precondition	Before a method
@post	Method postcondition	Before a method
@invariant	Class invariant	Before a class; before a field of a class

Table 1 – custom tags which are also part of the typical DBC implementation.

### Extended Custom Tags

Tag	Description	Where allowed
@assert	Assertion for a statement. Equivalent to the Java “assert”	Before a statement
@fact	Fact that is guaranteed to hold for a statement.	Before a statement
@loopInvariant	Invariant that holds for each iteration of a loop.	Before a loop (for-loop, while-loop, or do-while-loop)

Table 2 – custom tags which are not part of the typical DBC implementation.

Those extended custom tags provide more capabilities for programmers to extend their contracts to the statement level. There are two specific advantages of these tags as implemented in this DBC tool. The first advantage is that *ContractChecker* will use the statement level contract information to do a more strict form of runtime checking, then can report the violations against contracts earlier and more accurately than simply checking the contract at the method or class level. Secondly, those extended custom tags provide useful logic condition information, which will greatly benefit the *StaticAnalyzer* in calculating the weakest precondition for statements in the method body.

## 2.2 Syntax for Expression

In our project, we are using an extended form of Java expression syntax to describe contract expressions. The only valid expression following with a custom tag is a boolean expression. To be able to express more complicated conditions, several new syntaxes are introduced to specify quantified expressions, the return value of a method, and the post-stage value of an object along with more operators.

### Operators In Expression

All normal Java operators are valid for describing contract conditions. Two additional operators are also allowed to add more capability for logical expression.

- ‘=>’ operator is used to specify logical implication between two predicates, such as  $P \Rightarrow Q$
- ‘<=>’ operator is used to specify logical equivalence between two predicates, such as  $P \Leftrightarrow Q$

## Quantified Expressions

Two kinds of quantified expressions are introduced: *forall* and *exists*. With the use of *forall* and *exists* the programmer can express properties that must be valid for all elements of and finite set of assertions for one element of this set. *forall* and *exists* expressions are produced through the following EBNF, where ‘(’, ‘)’, ‘\*’, and ‘?’ are meta-operators. ‘(’ and ‘)’ are used to separate terms, ‘\*’ means repeating zero or more times, and ‘?’ means optional:

```
forallExpression ::=  
  forall { variableDeclaration (; variableDeclaration)* (: expression)? @ expression }  
existsExpression ::=  
  exists { variableDeclaration (; variableDeclaration)* (: expression)? @ expression }  
where variableDeclaration refers to the quantified variable; the first expression refers to  
range of this variable; the expression after @ refers to the assertions of the variable.
```

The following is an example for a *forall* expression:

```
forall { int x : x>0 && x<10 @ x != y }
```

## Return Value of a Method

At times it is necessary to refer to the return value of a method as a type of method postcondition. We have a new syntax to express the return value given a method with name *m!*, where *m!* refers to the return value of method *m*.

## Post-stage Value of an Object

A mutable object may change its value within a method. It is also desirable to be able to express the post-stage value of an object. We use the following syntax:

*obj*'

where *obj* is the name of the object.

For example, *o'* is the post-stage value of object *o*. Note that the post-stage value can only be applied to an object. The correct way to specify the post-stage value of a field of a object is: *obj'.field*, where *obj* is the name of the object and *field* is the name of the field. See Appendix, example 1 for more information.

The following are examples of postconditions with quantified expressions, return value and post-stage value:

```
@post forall {int k : k>1 && k < size()-1 @ this'.element(k-1) == this.element(k) }  
@post element! == 10
```

Note that, if there is more than one contract expression of the same contract category within one comment block, such as more than one class invariant in the same comment block, the relationship between those contract expressions are always conjunctive

### 3. The *DocGen* Component

*DocGen* is an extension of the standard Java doclet and is capable of capturing DBC information. *DocGen* is able to generate the documentation using not only the direct class contract, but also all the inherited contract. This is important because in class hierarchies, the interface or superclass contract is implicitly inherited by all the subinterfaces or subclasses.

To remind the programmer that some contracts are inherited from other classes, *DocGen* will generate simple text before those contracts, to indicate which of them should be conjunctions and which of them should be disjunctions. Because documentation tools can only generate reports on package, class or method level, *DocGen* only handles the custom tags in table 1.

The Appendix contains some screenshots of the generated documentation in figure 2, figure 3, and figure 4. For illustration purpose, the contents of the tags do not follow the syntax described. Figure 2 shows the inherited class invariants. Figure 3 shows the inherited method preconditions and postconditions. Figure 4 shows the field level invariant, which is treated as a class invariant.

### 4. The *ContractChecker* Component

*ContractChecker* is a tool that checks the DBC contracts at runtime. It consists of two modules: *Contract Extraction Module* and *Code Generation Module*. By running *ContractChecker* on Java source code, modified code will be generated with the addition of the test code. The generated code is compiled and executed in a same way as the original code. If a contract is violated, exceptions will be raised and recorded in a trace file.

Unlike *DocGen*, *Contract Extraction Module* handles all of the extended custom tags. It extracts and parses the contract elements while *Code Generation Module* generates the test code. The generated code used to verify contracts are inserted at different levels of the original Java source code. The new code will be compiled and executed, and adds the ability to detect any contract violations.

#### 4.1 Custom Exceptions

In order to indicate a particular violation, four runtime exceptions have been defined.

- *PreconditionException* are thrown when preconditions of a method are violated.
- *PostconditionException* are thrown when postconditions of a method are violated.
- *InvariantException* are thrown when invariants of a class are violated.
- *ContractCannotBeCheckedException* are thrown when elements in a contract can not be tested. For example, if an object is not cloneable, it cannot be preserved. Thus the postcondition that contains a post-stage value of that object cannot be tested.

No specific exceptions are defined for violation of *fact*, *assert* or *loopInvariant*, since they are instrumented as *assert* statement.

#### 4.2 Check Point

It is important to know when the contract elements are checked. The contracts indicated by *@fact*, *@assert* and *@loopInvariant* are checked immediately before the corresponding statement. Table 3 shows how other contracts are tested.

Any code generated from the precondition for a constructor is placed right after a *super()* or *this()* call, possibly before the rest of the statements. The java compiler forces this convention.

In addition, the class invariant will not be enforced for any private methods, since a private method may temporarily modify the fields.

### 4.3 Contract Inheritance

Class hierarchies include class extension, interface implementation, interface extension and innerclasses. *ContractChecker* supports the propagation of class invariants, method preconditions and postconditions in class hierarchies. The class invariants of a sub-class are the conjunction of the class invariants of its super-classes or interfaces. The postconditions of a method in a subclass or subtype are the conjunction of the postconditions of the corresponding method in its superclass or interface. The preconditions of a method of a subclass, or subtype, are the disjunction of the preconditions of the corresponding method in its super-class or interface.

The same rule applies to an inner-class and its enclosing class. See Appendix, example 3 for further information on contract inheritance.

#### Check Points of Contracts

Contract Type Check Point	Class Invariant	Method Precondition	Method Postcondition
<b>Instance Method Entry</b>	Yes	Yes	No
<b>Constructor Entry</b>	No	Yes	No
<b>Private or Static Method Entry</b>	No	Yes	No
<b>Instance Method Exit</b>	Yes	No	Yes
<b>Constructor Exit</b>	Yes	No	Yes
<b>Private or Static Method Exit</b>	No	No	Yes

Table 3 – the check points for the standard contract types.

### 4.4 Instrumentation

Instrumentation is only performed for concrete classes or non-abstract methods in abstract classes. *@assert*, *@fact* and *@loopInvariant* are simply transformed into assert statements at their existing location. *@pre*, *@post* and *@invariant* are transformed into if-else statements. If the contract is violated, an exception will be thrown and the violation will be reported to the trace.txt file.

If a postcondition contains the pre-stage/ after-stage value of an object, a temporary variable is introduced to preserved the original value. However, if the object is not cloneable, a *ContractCannotBeCheckedException* will be thrown. If a postcondition contains the return value a temporary variable will also be introduced before the return statement to represent the return value. The return statement will simply return the value of that variable.

Table 4 shows the checking code patterns with their corresponding contracts.

*Entry point* for a method *m* in Table 4 refers to the point before the first statement in *m*, and *Exit point* for a method *m* refer to the point before *return* statement. So each method *m* only has one *entry point*, but may have several *exit points*.

### Rules For Generating Contract Checking Codes

Contract	Generated checking codes
@assert <i>exp</i>	<b>assert</b> <i>exp</i> ;
@fact <i>exp</i>	<b>assert</b> <i>exp</i> ;
@loopInvariant <i>exp</i>	<b>assert</b> <i>exp</i> ;
@invariant <i>exp</i> (For class <i>c</i> )	<pre>// Entry point for each public method in class c <b>if</b> (! <i>exp</i>) {     Contract contract = Contract.getInstance();     contract.reportViolation(<b>new</b> InvariantException(...)); }  // Exit point for each public method in class c <b>if</b> (! <i>exp</i>) {     Contract contract = Contract.getInstance();     contract.reportViolation(<b>new</b> InvariantException(...)); }</pre>
@pre <i>exp</i> (For method <i>m</i> )	<pre>// Entry point for method m <b>if</b> (! <i>exp</i>) {     Contract contract = Contract.getInstance();     contract.reportViolation(<b>new</b> PreconditionException(...)); }</pre>
@post <i>exp</i> (For method <i>m</i> , where assuming <i>exp</i> contains one post-state object <i>obj</i> ' with type <i>T</i> )  With exit point statement: <b>return</b> <i>rexp</i> ;	<pre>// Pre-stage value of obj is preserved at entry point of m T _pre\$\$_obj = null; <b>if</b> (<i>obj instanceof</i> Cloneable) {     <b>try</b> {         pre\$\$_obj = (T) <i>obj</i>.clone();     } <b>catch</b> (Exception e) {} } <b>else</b> {     Contract contract = Contract.getInstance();     contract.reportViolation(<b>new</b> ContractCannotBeCheckedException(...)); }  // Postcondition is tested at exit point // <i>exp'</i> is calculated by symbolic substitution of <i>m!</i> with <i>rexp</i>, <i>obj</i> // with _pre\$\$_obj, <i>obj'</i> with <i>obj</i>, in <i>exp</i> <b>if</b> (! <i>exp'</i>) {     Contract contract = Contract.getInstance();     contract.reportViolation(<b>new</b> PostconditionException(...)); }</pre>

Table 4 - examples of generated code for each contract type.

## 5. The *StaticAnalyzer* Component

The goal of static analysis and this DBC tool is to replace runtime analysis as much as possible. Static analysis, unlike software testing, does not depend on the comprehensibility and completeness of specific test cases. Software testing using test cases has two primary drawbacks, the first of which is that preparing test cases can be a time-consuming process, and the second of which is that a series of test cases can only show program correctness for those specific cases according to specific coverage criteria. Static analysis, which does not rely on test cases, goes

much further towards proving overall program correctness and adherence to a contract than software testing does. Unfortunately, static analysis cannot at this time completely replace software testing because in complex systems, static analysis becomes too inefficient.

The *StaticAnalyzer* component of this DBC toolset has three functionalities. The static contract verification functionality of *StaticAnalyzer* does utilize the DBC tags. This functionality will analyze specific methods and throw an `InvalidContract` exception if the implementation does not conform to the specification. The violation will be reported in the `trace.txt` file. The *StaticAnalyzer* makes use of the `@assert`, `@fact`, and `@loopInvariant` tags that may be included in the method body, but will attempt the analysis even without this information, the whole verification process can be conduct just using contractual interfaces like precondition, postconditions and invariants. Null pointer analysis and array out of bounds analysis are not specifically related to Design by Contract but are useful nonetheless.

## 6. Comparison with Other DBC Tools

Since the concept of DBC was first introduced, many DBC tools have been created. In this section we compare our tools with others according to several factors. Table 5 shows a general comparison between our DBC toolset, JDK 1.4 (although JDK is not a DBC tool, the `assert` statement provide a similar way to do runtime checking for conditions, so we also put it in the table) and commonly used DBC tools iContract, Eiffel, and JMSAssert.

**A General Comparison Among Several DBC Tools**

	<b>Expressiveness</b>	<b>Runtime check</b>	<b>Documentation</b>	<b>Designed for Static Analysis</b>
JDK 1.4	Low, only use valid java expression	Yes	No	No
iContract Eiffel JMSAssert	Using specified syntax with quantifier support	Yes	Yes - IContract No – Eiffel, JMSAssert	No
Our toolset	Using specified syntax with quantifier support	Yes	Yes	Yes

Table 5 – comparing functionality of DBC tools.

### Specification Expressiveness

The DBC toolset in our project brings a rich set of syntax. It supports not only the three key elements of DBC, but also extensions like the `@assert`, `@fact` and `@loopInvariant`. This is a highlighted feature compared with some DBC tools, such as JMSAssert [6]. Our toolset also has specific syntax for quantified expressions, which greatly improves the expressiveness.

### Class Inheritance Strategy

Almost all the DBC tools handle class inheritance issues using a similar strategy. Table 6 shows this comparison.

## Class Inheritance Strategy

	<b>Class Invariant</b>	<b>Precondition</b>	<b>Postcondition</b>
JDK 1.4	No	No	No
Eiffel	No	Disjunction with supertype	Conjunction with supertype
iContract, JMSAssert	Conjunction with supertype	Disjunction with supertype	Conjunction
Our toolset	Conjunction with supertype	Disjunction with supertype	Conjunction with supertype

Table 6 – comparing inheritance strategy of DBC tools.

### Class Inheritance Documentation

The DBC toolset in our project provides an extensive JavaDoc support: the *DocGen*. Other DBC tools, such as JASS [4] or iContract [5] may also have JavaDoc support, but do not capture the inherited DBC information. There are also separate doclets that support DBC tags, such as iDoclet, but most are developed based on JDK version 1.3 or lower. *DocGen*, on the other hand, fully supports the inheritance of the contracts. It can be used separately, as well as with the *ContractChecker*.

### Static Analysis

Static analysis, and particularly static contract verification, is perhaps the most important benefit of our DBC toolset when compared to other DBC products. iContract does not include static analysis while JMSAssert contains a utility called JStyle that does do limited static analysis, which serves as more of an automatic code review than an analysis of code correctness. JStyle helps to catch problems such as class or member naming conventions, class section ordering, and unsafe exception handling [10]. This can be a useful utility, but it does not provide as much value as static contract verification, which can statically analyze whether the implementation conforms to the specification or not.

## 7. Conclusions and Future Work

Design by Contract yields great benefits in software development. Our DBC toolset brings DBC into Java, makes it possible to formalize the DBC information, and to utilize DBC during for both runtime and static checking.

*DocGen*, with the support of JavaDoc, generates Java documentation that includes the information of any direct and inherited method preconditions, method postconditions and class invariants.

*ContractChecker* creates test code based on the DBC information. The test code is inserted into the original code. This process is independent of compiling and executing the code. At runtime, if a contract is violated, a particular exception is thrown and reported to alert the programmer.

*StaticAnalyzer* checks for null pointer and array out of bounds errors and utilizes Design by Contract information to check the correctness of a written contract. If a contract is not assured of fulfilled postconditions given satisfied preconditions *StaticAnalyzer* will identify this and alert the programmer of the problem.

Despite the advantages of our DBC toolset, it does has some limitations.

- Although the syntax of quantified expressions are specified, the toolset does not generate correct code for them. In the future, this can be achieved by introducing a loop statement through the range of the target variable.
- The test fails if a recursive call occurs during the check. This problem can be solved by using a register file or stack, to track the occurrence of checks, and make to sure that finite number of checks is performed.

This toolset is still under development; we are going to eliminate the above limitations in the future.

## References

- [1] B. Meyer, Applying "Design by Contract, in Computer", vol. 25, no. 10, pp. 40-51. October 1992
- [2] B. Meyer, Object-Oriented Software Construction, Prentice Hall, 1988.
- [3] T. Plessel, Design By Contract: A missing Link In The Quest For Quality Software, August 11, 1998.
- [4] C. Fischer, Combination and Implementation of Processes and Data: from CSP-OZ to Java, PhD Thesis, University of Oldenburg, 2000
- [5] R. Kramer, Examples of Design by Contract in Java, Object World Berlin '99, Berlin, May 1999
- [6] K. Rangaraajan, Invasive Testing of Java™ Classes, Man Machine Systems
- [7] X. Jia, S. Skevoulis, A Generic Approach of Static Analysis for Detecting Runtime Errors in Java Programs
- [8] S. Skevoulis, X. Jia, Generic Invariant-Based Static Analysis Tool for Detection of Runtime Errors in Java Programs
- [9] S. Skevoulis, A Light-weight Approach to Applying Formal Methods in Software Development, Technical Report, DePaul University, 1999
- [10] K. Rangaraajan, Critiquing Java Source Code

## Appendix

### A.1 Example 1 – A Java Class with Design by Contract information.

```

/**
 * @invariant x > 10
 */
public class A {
    int x;
    /**
     * @pre val > 20
     * @post this'.x == this.x + val
     */
    public void setX(int val){x = x + val;}
    /**
     * @post getX! == x
     */
    public int getX(){return x;}
}

```

Figure 1: An example of source code with precondition, postcondition, and invariant contracts

## A.2 Example 2 – The *DocGen* tool.

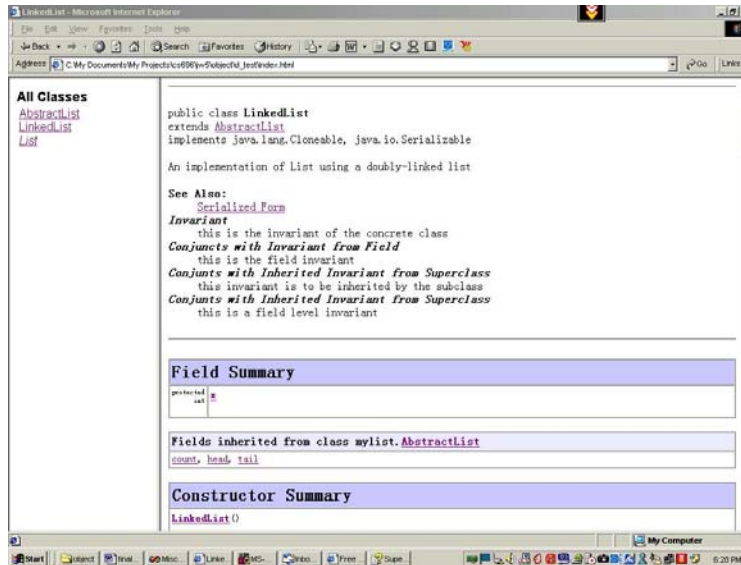


Figure 2: An Example of Class Invariants in *DocGen*

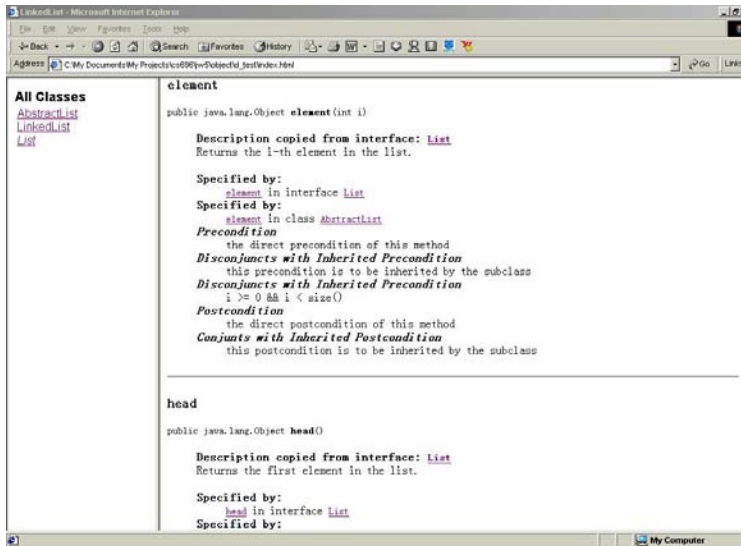


Figure 3: An Example of Method Preconditions and Postconditions in *DocGen*

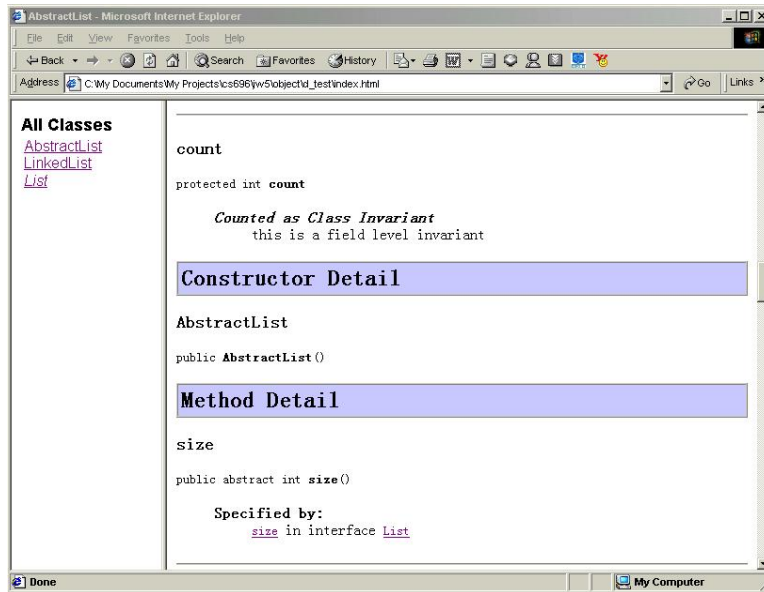


Figure 4: An Example of Field Level Invariant in *DocGen*

### A.3 Example 3 – Contract Inheritance.

The interface *Line* and a subclass *ShortLine* are declared as in the following table.

Super Type Declaration	Sub Type Declaration
<pre> /**  * @invariant length()&gt;0  */ public interface Line {   public int length();   /**    * @pre amount&gt;0    * @post cut! &gt; 0    */   public int cut(int amount); } </pre>	<pre> /**  * @invariant length()&lt;10  */ class ShortLine implements Line {   public int length() {...}   /**    * @pre amount&gt;-10    * @post cut! &lt; 50    */   public int cut(int amount) {...} } </pre>

Table 7 – An example of contract inheritance.

The code of *ShortLine*, when handled by *ContractChecker* after contract inheritance is as follows:

```

/**
 * @invariant length() >0 && length()<10
 */
class ShortLine implements Line {
  public int length() {...}

  /**
   * @pre amount>-10 || amount >0
   * @post cut! < 50 && cut! > 0
   */
  public int cut(int amount) {...}
}
}

```