# iContract – The Java™ Design by Contract™ Tool

Reto Kramer
kramer@acm.org
Cambridge Technology Partners

**Abstract**

*Until today, the explicit specification of "software contracts" by means of class invariants and method pre- and post-conditions was only available to Eiffel developers. iContract is the first tool that provides the same thorough support for Java.*

*iContract is a freely available source-code pre-processor that instruments source-code with checks for class invariants as well as pre- and post-conditions that may be associated with methods in classes and interfaces. Special comment tags (@pre, @post, @invariant) are interpreted by iContract and converted into assertion check code that is inserted into the source-code. iContract supports universal and existential quantifiers in contract expressions. Contracts are propagated via all 4 Java type extension mechanisms (class extension, innerclasses, interface implementation and interface extension).*

*Due to the non-mandatory nature of the comment tags, source code that contains design by contract annotations remains fully compatible with Java and can thus be processed with standard Java compilers, enabling a risk-free adoption of the technique in your organisation.*

## 1: Introduction

In Eiffel [1] and the latest UML [2] there is native support for design by contract in which the interfaces among classes are viewed as contracts that specify benefits and obligations of both the client and the supplier. Obligations of clients are specified as preconditions of methods whereas the promises of the supplier are specified as postconditions which are guaranteed to hold provided that the preconditions are not violated. Class and interface invariants specify general object consistency properties.

Design by contract is a simple way to: (*i*) reduce test effort due to the separation of contract checks from regular application logic, (*ii*) save debugging effort due to the improved observability, where failure occurs close to the faults and (*iii*) ensure up-to-date and unambiguous documentation of interfaces.

Section 2 presents a simple annotated example class and the check code that iContract produces. Section 3 presents the code instrumentation rules for single, isolated classes whereas section 4 considers implications of Java class extension, interface implementation and interface extension mechanisms. Section 5 describes the current prototype implementation. Section 6 summarises related work, while section 7 and 8 outline directions for future work and conclude.

## 2: Simple example

To use iContract, Java sourcecode is annotated with three novel comment paragraph tags:

- `@invariant`, to specify class- and interface-invariants
- `@pre`, to specify preconditions on methods of classes and interfaces
- `@post`, to specify postconditions on methods of classes and interfaces

The tool is run as a preprocessor over a set of annotated sourcecode files. iContract instruments the methods in annotated files with assertion check code that enforces the specified pre-, postconditions and invariants. In the background it builds up a repository of contract related information about the classes, interfaces and methods. This information is used to support subcontracting which propagates pre-, postconditions and invariants along class-extension, interface-implementation, multiple interface-extension and innerclass relations.

The instrumented files are compiled instead of the originals, resulting in the same classes, interfaces and methods except that the pre-, postconditions and invariants are being enforced by means of automatically generated specification checks.

Despite being annotated with pre-, postconditions and invariants, the original files remain fully compliant to “standard” Java at all times due the annotations being a part of the (optional) comment paragraphs. This is an essential feature enabling organisations to adopt iContract in a risk-free manner.

The example `interface Person` contains method annotations with pre- and postconditions specifying an explicit contract between the implementers of `interface Person` and their clients that call the methods `setAge(int)` and `getAge()`. To clients, the interface method `setAge(int)` specifies the obligation that the age must be greater than zero. In return the benefit offered to clients of the `interface Person` is that they are promised to receive an age greater than zero, when calling `getAge()`.

```
FILE: Person.java

1:  interface Person {
2:
3:    /**
4:     * @post return > 0
5:     */
6:    int getAge();
7:
8:    /**
9:     * @pre age > 0
10:    */
11:   void setAge( int age );
12: }
```

```
FILE: Employee.java

1:  class Employee implements Person {
2:
3:    protected int age_;
4:
5:    public int getAge() {
6:      return age_;
7:    };
8:
9:    public void setAge( int age ) {
10:     age_ = age;
11:   };
12: }
```

The `class Employee` implements the `interface Person` by maintaining an instance variable to store the age of an Employee. The pre- and postconditions need not to be repeated in the implementation because iContract maintains contracts across class hierarchies as well as interface implementation and extension relations.

To instrument the `class Employee` such that the pre- and postconditions will be checked and enforced at runtime, iContract instruments the sourcecode with assertion statements.

```
class Employee implements Person {

  protected int age_;
```

```
  public int getAge() {
    //#*#---------------------------------------------------------------
    int __return_value_holder_;
    /* return age_; */
    __return_value_holder_ =  age_;
    if (!(__return_value_holder_ > 0))
      throw new RuntimeException ("Employee.java:5: error: postcondition "+
                                  "violated (Person.getAge()): "+
                                  "(/*return*/  age_) > 0");
    return __return_value_holder_;
    //---------------------------------------------------------------#*#
    };

  public void setAge( int age ) {
    //#*#---------------------------------------------------------------
    boolean __pre_passed = false; // true if pre-cond passed.
    // checking Person.setAge(int age)
    if (! __pre_passed ) {
    if (age > 0) __pre_passed = true; // Person.setAge(int age)
    }
    if (!__pre_passed) {
      throw new RuntimeException ("Employee.java:9: error: precondition "+
                                  "violated (Employee.setAge(int age)): "+
                                  "(/*Person.setAge(int age)*/ (age > 0)) "
    ); }
    //---------------------------------------------------------------#*#

    age_ = age;
  };
}
```

The following `main` code-section would result in a precondition violation, which leads to a `RuntimeException` being thrown:

```
...
public static void main(String argv[]) {
  Employee e = new Employee();
  e.setAge( 0 ); // violates precondition !
}

% java Employee
java.lang.RuntimeException: Employee.java:5: error: precondition violated
(Employee.setAge(int age)): (/*Person.setAge(int age)*/ (age > 0))
        at Employee.setAge(Employee_I.java:39)
        at Employee.main(Employee_I.java:5)
```

The error in the `main` causes the `Person.setAge(int)` precondition contract (`age > 0`) to be violated. This contract is propagated from the `interface Person` via the interface implementation relation to the `class Employee`. The error location indicated (`Employee.java:5:`) is the location of the contract violation in the original (non-instrumented) version of the `class Employee`.

## 3: Contracts on single classes

This section considers the instrumentation of invariant, pre- and postcondition check code for an isolated class only. It assumes that the class neither extends another class nor implements any interfaces. The implications of innerclasses, class extensions, interface implementation and interface extension will be considered in section 4.

Classes, interfaces and methods therein can be annotated with pre-, postconditions and invariants. Classes and interfaces are collectively referred to as types.

The @invariant tag applies to types, whereas the @pre and @post tags apply to methods in types. Multiple tags are interpreted as conjunctions. Tags may span multiple lines to enable the layout of large expressions.

### 3.1: Object lifecycle

During their lifecycle, objects generally undergo the three phases: (*i*) construction, (*ii*) operation and eventually (*iii*) destruction.

Construction of an object requires the preconditions of the particular constructor to hold. Upon successful construction the class invariants and postconditions must be established . Should the constructor fail due to an exception being thrown, the class invariants and the postconditions do not have to hold. They are thus not enforced in case of exceptional constructor exit.

Operation of an object must be split into two cases: (*i*) public, package and protected method calls being received (all referred to as "public") and (*ii*) private method calls. In both cases the preconditions of a method must hold, whereas only in the former case the invariants must hold as well. Invariants must not be enforced for private methods because they already were enforced upon reception of the "public" method that was preceding the private. In fact it is desirable to allow a sequence of private methods to violate class invariants temporarily as long as the invariants are established before the exit of the "public" method that triggered the chain of private calls. Should the method fail due to an exception being thrown, the class invariants must still hold (n.b. unlike constructors), whereas the postconditions do not have to hold and thus are not enforced in case of exceptional method exit.

The destruction of an object neither requires any prerequisites and invariants to hold, nor does it establish any postconditions and invariants (after all there is no object left to associate them with). Thus iContract will not instrument the finalize() method with invariant checks.

### 3.2: Contract-expressions and scope

The string following the @pre, @post and @invariant tag must be of the form:

```
<Contract-Expression> [#<ExceptionClassName>]
```

where <Contract-Expression> must evaluate to a boolean. The name #<ExceptionClassName> denotes an optional class that will be used to construct the exception that will be thrown in case the <Contract-Expression> evaluates to false. If the exception class in not mentioned, RuntimeException will be thrown.

```
/**
 * @pre i >= 0          #ArrayIndexOutOfBoundsException
 * @pre i < this.SIZE   #ArrayIndexOutOfBoundsException
 */
String getEntry( int i ) throws ArrayIndexOutOfBoundsException {
  // do the work here ..., no need to check index bounds manually
}
```

Within the same type $T$ (class or interface), multiple invariants, pre- and postconditions (e.g. $pre_1^{m_T}$ is (i>=0), $pre_2^{m_T}$ is (i<this.SIZE)) are conjuncted to form a single aggregated expression (e.g. $pre^{m_T}$ is (i>=0 && i<this.SIZE)) which will be used to instrument the method $m$ with appropriate check code:

$$pre^{m_T} = \bigwedge_j pre_j^{m_T} \tag{1}$$

$$post^{m_T} = \bigwedge_j post_j^{m_T} \tag{2}$$

$$invariant^{m_T} = \bigwedge_j invariant_j^T \tag{3}$$

If a class, interface, method or constructor is not annotated with invariants, pre- or postconditions, then the respective condition is assumed to (trivially) hold.

### 3.2.1: Contract-expressions

Contract-Expressions may be any Java expression that evaluates to a `boolean`. iContract complements the Java operators with the following additional ones:

- **forall** ($\forall$), supports universal-quantification over a set of elements.
  The syntax is:

  ```
  forall <Class> <var> in <Enum> | <Expr_var>
  ```

  where `<Class>` is the type of the bound variable `<var>`. `<Enum>` is an expression that must evaluate to an instance of `java.util.Enumeration` (containing elements of type `<Class>`[1]). `<Expr_var>` denotes a contract-expression which may contain references to the bound variable `<var>`.
  The `forall` expression evaluates to true, iff $\forall var \in Enum | Expr_{var}$. Where $Expr_{var}$ denotes expression $Expr$ evaluated using the value of $var$.
- **exists** ($\exists$), supports existential-quantification over a set of elements.
  The syntax is similar to the universal quantifier:

  ```
  exists <Class> <var> in <Enum> | <Expr_var>
  ```

  The `exists` expression evaluates to true, iff $\exists var \in Enum | Expr_{var}$. Where $Expr_{var}$ denotes expression $Expr$ evaluated using the value of $var$.
- **implies** ($\rightarrow$), `C implies I` is translated into the equivalent of "if C then check I". C and I must be Contract-Expressions.

The following examples illustrate the power of contract-expressions supported by iContract:

**Example invariant:** uses nested quantification expressions to specify "sanity" checks for a class representing an `Employer`. An `Employer` employs a number of `Employees`, each of which can be employed by a number of different `Employers` at the same time. The aim of the invariant specification is to express the requirement that all instances of `class Employee` must appear on the employment lists of their respective `Employers`. The example assumes that the instance variable `Employer.employees_` is of type `java.util.Vector` and that the method `Employee.getEmployers()` returns an instance of type `java.util.Enumeration`:

```
/**Each employee must be on the employment-list of all it's employers
 *
 * @invariant employees_ != null
 *              implies
 *                forall Employee e in employees_.elements() |
 *                  exists Employer c in e.getEmployers() |
 *                     c == this
 */
class Employer {
  protected Vector employees_; // of Employee
  ...
}
```

[1] The access to the elements will be casted: `(<Class>)(<Enum>.nextElement())`

### 3.2.2: Scope of variables in contract-expressions

Class invariants can access class- and instance-variables as well as methods of their associated class. Interface invariants do have the same scope as class invariants except that they can not access any instance variables.

Preconditions have access to all items that are in the scope of their associated method. Postconditions do have the same scope as preconditions except: (*i*) "return value" – in addition to the precondition scope, postcondition expression can access a pseudo-variable called return which denotes the result value of the method. (*ii*) "entry-value" – they have access to the values that expressions had at the start of the method. The syntax is compliant to UML/OCL [2] where the context-modifier @pre (previous) is appended to an expression in order to refer to its value at the entry of a method.

```
/**Append an element to the argument
 *
 * @post list.size() == list.size()@pre + 1;
 */
void append( Vector list, Object o );
```

If the type of an "entry-value" implements interface Cloneable (e.g. is a container, such as Vector), or is of type String, the value of <expr>@pre is a shallow copy of <expr>. Otherwise they are identical.

Invariants, pre- and postconditions have access to bound variables of quantification expression (e.g. the bound variable <var> in forall <Class> <var> ...).

If invariant-, pre- and postcondition expressions refer to instance variables, those variables must not be private, unless the class is final.

### 3.2.3: Discussion

In this section a summary of code instrumentation is presented. Issues that arise from these rules are illustrated with examples and solutions are suggested.

Generally preconditions are enforced when methods are entered whereas postconditions are enforced when methods return. Invariants are enforced at the entry of methods and at both, the normal return and exceptional termination of methods. Exceptions to these rules are summarised in table 1.

| | | Method | | Constructor |
|---|---|---|---|---|
| | | public, package, protected | private | |
| invariant | entry | √ | — | — |
| | exit | √ | — | √ |
| | exception | √ | — | — |
| precondition | entry | √ | √ | √ |
| | exit | n/a | n/a | n/a |
| | exception | n/a | n/a | n/a |
| postcondition | entry | n/a | n/a | n/a |
| | exit | √ | √ | (√) |
| | exception | — | — | — |

**Table 1. Instrumentation Rules for Invariants, Pre- and Postconditions**

Two rules are not captured in table 1:

- The destruction of objects is neither subject to any invariants nor to pre- and postconditions. Therefore the finalize() method is not instrumented for any of these.
- static method are not instrumented for invariant checks (public static methods are instrumented for pre- and postcondition checks though).

The naive approach to instrumentation contains the potential for recursive, non-terminating calls among invariant checks as illustrated in the following example:

```
1:  /**Example that demonstrates the automatic avoidance of
2:   * recursive, non-terminating invariant checks
3:   *
4:   * @invariant forall Employee employee
5:   *            in this.getEmployees().elements() |
6:   *            employee.getEmployer() == this
7:   */
8:  class Employer {
9:
10:  public static void main(String arv[]) {
11:    Employer company = new Employer();
12:    Employee george = new Employee();
13:    company.add( george );
14:  }
15:
16:  protected Vector employees_ = new Vector();
17:
18:  Enumeration getEmployees() {
19:    return employees_.elements();
20:  }
21:
22:  void add(Employee employee) {
23:    employee.setEmployer( this );
24:    employees_.addElement( employee );
25:  }
26: }
27: ...
```

The invariant on line 4 will be checked at the entry of method `Employer.add(Employee)`, defined on line 22 (called by main on line 13). To perform the check the invariant itself calls `getEmployees()` which – at its entry, checks the invariant ... – at this point the application is trapped in a non-terminating recursion that will overflow the stack.

To automatically avoid non-terminating recursion, iContract instruments check-code such that it keeps track of the call-chain at runtime in order to prevent recursive non-terminating checks. The mechanism is thread-safe.

As an example, the generated code that guards against recursive invariant checks at method entries is shown below. iContract generates a runtime bookkeeping table for each object that needs to obey an invariant (`Hashtable __icl_`). The table tracks the call-chain on a per thread basis, incrementing the recursion depth if a non-private method is entered. The recursion depth is decremented if the method is exit either via normal return or exceptional exit (code not shown here). Only if the recursion depth is 0, the invariant is evaluated. This mechanism

- guards against the non-terminating recursion problem
- ensures that for nested "public" calls (public, package, protected) to the same object (within the same thread) the invariant is not re-evaluation until the exit of the initial top level method
- prevents the evaluation of invariants, if "public" methods are called from within private ones (on the same object, in the same thread)

```
/**
 * @invariant age_ > 0
 */
public class Employee implements Person {
  //#*#---------------------------------------------------------------
  private java.util.Hashtable __icl_ = new java.util.Hashtable();
  private synchronized void __inv_check_at_entry__Employee(
```

```
    Thread thread, String loc) {
    if ( !__icl_.containsKey(thread) ) { // recursion depth 0
      __icl_.put(thread, new Integer(1));
      __check_invariant____Employee(loc); // evaluates the invariant
    }
    else // inc recursion depth
      __icl_.put(thread, new Integer(
        ((Integer)__icl_.get(thread)).intValue()+1));
}
//#*#---------------------------------------------------------------
```

## 4: Contracts on class hierarchies

Previously only single types were considered. This section explores implications of the four hierarchical type relations available in Java. Class extension, interface implementation, interface extension and innerclasses are collectively referred to as "type extension".

iContract analyses type extensions in order to support the "type substitution principle" which says that the formal method parameter of type $T$ may be substituted with actual parameters of type $T$ or any type-extension (interface or class). In order to preserve the semantics of the type $T$ (and its methods) in its descendants, the invariants on type $T$ as well as pre- and postconditions on methods of type $T$ must be propagated to all descendants of the type.

The actual means of propagation of invariants, pre- and postconditions varies:

- invariants are conjuncted across type extensions because the subtypes must also comply to all the invariants of their supertypes
- postconditions are conjuncted across type extensions because redefined methods in subtypes must offer at least the same service to their clients as the supertype method definition did (only then the types are substitutable). They may add additional promises (which impose further constraints) about the properties of their side effects or results though. Those additional properties must imply the supertype postconditions to ensure that they are preserved by the service extension. Conjuncting the postconditions of method $m$ of supertype $V$ ($post^{m_V}$) with the postconditions of method $m$ of subtype $W$ ($post^{m_W}$) guarantees this implication because: $post^{m_V} \wedge post^{m_W} \rightarrow post^{m_V}$
- preconditions are disjuncted across type extensions because redefined methods in subtypes must accept at least the same input arguments from their clients as the supertype method definition did (only then the types are substitutable). They may accept additional input, that would be rejected by the supertype methods though (under no circumstances however may they restrict the accepted input beyond the supertype restrictions because this would violate the type substitution principle). Disjunction of the preconditions of method $m$ of supertype $V$ ($pre^{m_V}$) with the preconditions of method $m$ of subtype $W$ ($pre^{m_W}$) guarantees this because if the precondition of the supertype holds, it implies that it still holds in the subtypes conjuncted precondition: $pre^{m_V} \rightarrow pre^{m_V} \vee pre^{m_W}$

To capture the type extension aspects of invariant, pre- and postcondition instrumentation formally, the following terms are introduced (see eq. 1, eq. 2 and eq. 3):

The total invariant that is instrumented for a method $m$ of type $T$ is denoted $invariant^{m_T^*}$ ($T^*$ is the supertype closure set which includes type $T$ and all its predecessor classes and interfaces). The total preconditions of method $m$ in type $T$ is denoted $pre^{m_T^*}$ and the total postcondition is denoted $post^{m_T^*}$.

In summary the totals are defined as:

$$pre^{m^*_T} = \bigwedge_{t \in T^*} pre^{m_t} \stackrel{eq.1}{=} \bigvee_{t \in T^*} \bigwedge_{i \in t} pre_i^{m_t} \tag{4}$$

$$post^{m^*_T} = \bigwedge_{t \in T^*} post^{m_t} \stackrel{eq.2}{=} \bigwedge_{t \in T^*} \bigwedge_{i \in t} post_i^{m_t} \tag{5}$$

$$invariant^{m^*_T} = \bigwedge_{t \in T^*} invariant^{m_t} \stackrel{eq.3}{=} \bigwedge_{t \in T^*} \bigwedge_{i \in t} invariant_i^t \tag{6}$$

### 4.1: Differences among mechanisms of type extension

The instrumentation code generation handles all four type extension mechanisms in a uniform way (class extension, interface implementation, interface extension and innerclasses).

The only special case is the delegation of subclass constructors to constructors of their superclasses (`super()`). If such a `super()`-delegation is used in constructor, Java compilers enforce the `super()` statement to be the first in the body. This forces iContract to insert the precondition check only after the `super()` call, whereas in methods the precondition- and invariant-check are the very first things to be executed.

This still preserves the semantics of preconditions because constructor overriding is not possible as each class has unique constructor names (because they are equal to the class name). Therefore there is no need to concern "type extension" implications for preconditions in constructors, as there is a unique mapping from constructor to its type (which is not true for methods that were overridden in multiple classes). Formally, in eq. 4 the superclass closure $T^*$ contains a single element – the class of the constructor – and therefore the disjunction operator can be omitted.

## 5: Implementation

The tool iContract is a freely available Java prototype implementation of the instrumentation techniques described in the previous sections. It pre-processes Java sourcecode (fig. 1). Special care was taken to ensure that the instrumented code is still easy to read. Intentionally the tool was not made to operate on bytecodes (`.class` level) because it was desirable that invariants, pre- and postconditions are subject to the same compile-time optimisations as the rest of the code.

Figure 1 gives an overview of the interaction among the major files and tools involved in using iContract.

- The goal is to transform sourcecode to executable Java applications (dashed arrow).
- Some of the code shall be instrumented while some of it must not suffer from any performance loss due to check code and thus is not instrumented
- Code that shall not be instrumented is compiled as usual (e.g. using javac)
- Code that shall be instrumented must be pre-processed with iContract. This creates two items:
  - an instrumented sourcecode file (e.g. `C_I.java`) that contains assertion-checks in methods
  - an internal repository (e.g. `__REP_C.class`) of invariant, pre- and post-condition information on the classes encountered in the input file. This

repository is subsequently accessed in order to propagate contracts along type extension relations.

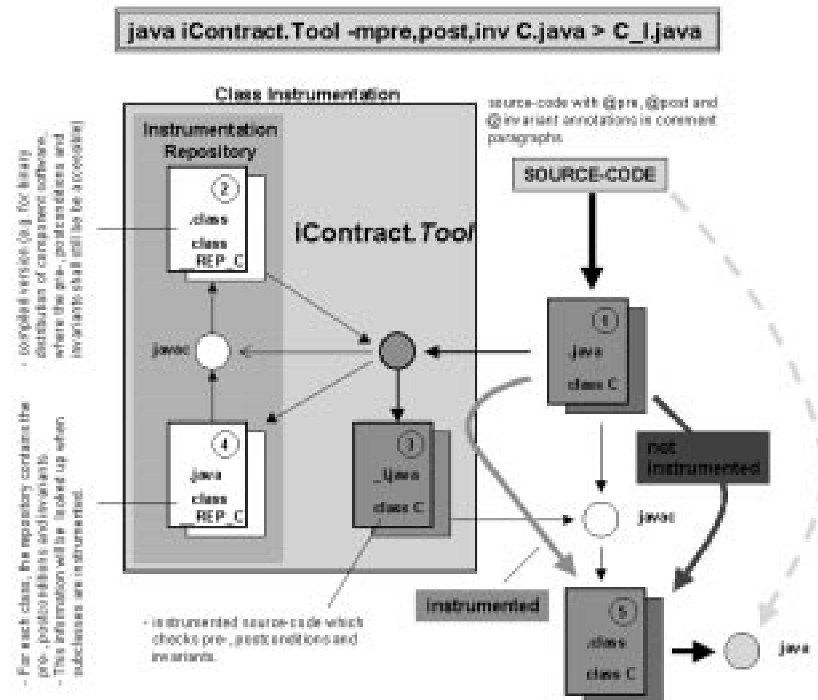


**Figure 1. Interaction among iContract Components**

### 5.1: Performance tuning

In order to let developers tune the performance of their applications, check code instrumentation must be configurable such that performance critical parts do not suffer from unacceptable overhead due to the check code evaluation. Invoking iContract with:

```
% java iContract.Tool -mpre,post,inv C.java > C_instr.java
```

creates an instrumented sourcecode file `C_instr.java` and an internal repository file `__REP_C.java` as well as its compiled version `__REP_C.class`. Methods in the sourcecode file are instrumented for invariants, pre- and postcondition checks. This however is configurable to allow performance tuning (e.g. `-mpre` only instruments precondition checks). iContract is invoked on a single file and thus allows instrumentation configuration down to the class level such that performance critical classes can be instrumented less extensively than non-critical parts of the system.

### 5.2: Binary contract repositories

Despite operating on sourcecode the tool is suitable to component vendors that refrain from distributing the sourcecode with their products. There are two levels of contract support: (*i*) vendors deliver a non-instrumented and an instrumented

version of the library, and/or (*ii*) vendors even deliver the “invariant, pre- and post-condition contracts” with their components by including the compiled repository files (e.g. `__REP_<classname>.class`) into their products (fig. 1).

If component vendors would adopt option (*ii*), user's would then copy the repository files to the same location as the component code itself. If extension of frameworks or components (e.g. implementation of interfaces) is required, iContract picks up the vendor repositories based on the class and package names such that the invariants, pre- and postconditions that the vendor product requires and provides will be enforced in user defined extensions.

### 5.3: Multithreaded applications

iContract changes the “structure” of the code by adding invariant check methods to classes (if the users requested it using option `-minv`). These methods are `synchronized` such that local variables (e.g the bound variables introduced by quantification expressions) are protected against concurrent access.

Users must review their concurrency handling, if quantifiers (`exists` and `forall`) are used in pre- or postconditions and if references to `return` or “entry-values” occur in postconditions. All these contract-expression features introduce automatically generated temporary variables to the scope of the methods. Those are subject to race conditions if not protected appropriately. Unfortunately iContract has no knowledge of the context in which it is being used and thus can not automate this reasoning.

### 5.4: Discussion

iContract is a prototype version which has helped to uncover a number of issues in providing design by contract to Java developers:

- In extreme cases contract expressions may have to be propagated to classes from four different types sources: class extension, interface implementation, interface extension and via innerclasses.
- If absent, default constructors (which check the class invariants) are added to classes explicitly.
- Type inference capabilities are required to support “old-value”- and “return value”-references in post-conditions (to enable storage of temporary values) as well as universal and existential quantifiers in invariants, pre- and post-conditions (to enable proper casting).
- A rule had to be established that instance variables, referred to in contract expressions, must not be private as this would prevent the evaluation of the supertype contracts propagated into subtypes.
- Prevent the non-terminating recursive invariant check problem using a thread-safe mechanism.
- Prevent public method invariants from being checked, if public methods are called from private ones.
- A suitable mechanism to store the contract information of types outside their source-code definition has been developed to allow the distribution of instrumented and extendable libraries without requiring the libraries source-code to be distributed (binary contract repository).

## 6: Related work

The industrial availability of design by contract was pioneered by the ISE Eiffel implementation which still is the only commercial OO language with full support for invariant, pre- and postcondition instrumentation of class hierarchies [1].

The latest UML version 1.1 [2] defines a language suitable to specify invariants, pre- and postconditions. Unlike Eiffel, UML supports quantification operators. It does however not address: (*i*) shallow-copies of "entry-values", (*ii*) the semantics of contracts on type extension relations and (*iii*) the potential of recursive contract checks.

## 7: Future work

iContract provides benefits to a wide range of system- and application-level developers. Further support for proposed Java capabilities such as "type parameterisation" (templates) [3], [4] is planned. Besides this extensions, four main areas of future work have been identified:

- integration of iContract support into common integrated development environments (IDEs).
- use the `javadoc` template features [7] to automatically create up-to-date html documentation of contract annotated Java sourcecode.
- extended support for specification of collection properties, in compliance to the UML Object Constraint Language [2].
- extensions towards support for specification of temporal system properties [5], [6] by introduction of temporal logic operators in contract-expressions of class invariants.

## 8: Conclusions

This paper describes design by contract extensions of the Java programming language and a prototype preprocessor implementation. Invariants, pre- and postconditions are specified as part of the documentation comments. Three new tags are introduced: `@invariant`, `@pre` and `@post` which are used to specify class- and interface invariants as well as method pre- and postconditions. Despite these language extensions, annotated programs remain fully compatible with Java compilers and the `javadoc` tool.

The prototype implementation supports design by contract over class hierarchies as well as interface-implementation, interface-extension and innerclass relations. In line with UML/OCL [2], invariant, pre- and postcondition expressions may include quantified expressions and postconditions may refer to method return- and "entry" values. The use of tagged paragraphs facilitates contract documentation using standard Java tools such as `javadoc` [7].

Developers can adopt design by contract using iContract as an incremental, risk-free approach to explicitly and unambiguously specify the assumptions and promises of their packages.

The prototype implementation is available free of charge through
http://www.promigos.ch/kramer.

## References


[1] MEYER BERTRAND. *Object Oriented Software Construction* . Prentice Hall, $2^{nd}$ edition, 1997.

[2] RATIONAL SOFTWARE CORP. *Unified Modeling Language, Object Constraint Language Specification.* available from http://www.rational.com/uml/, 1997.

[3] ODERSKY M., WADLER P. *The Pizza Compiler.* available from http://www.math.luc.edu/pizza/, February 1997

[4] AGESEN O., FREUND S., MITCHELL J. *Adding Type Parameterization to the Java Language.* OOPSLA 1997.

[5] NIERSTRASZ O., TSICHRITZIS D., Eds. *Object Oriented Software Composition.* Prentice Hall, 1995.

[6] MANNA Z., PNUELI AMIR. *The Temporal Logic of Reactive and Concurrent Systems, Specification* Springer Verlag, 1992.

[7] JAVASOFT INC. *Javadoc Templates.* JDK 1.2, Early Access. October 1997. available from http://www.sun.com/products/jdk/1.2/docs/