# *jContractor:* A Reflective Java Library to Support Design By Contract

Murat Karaorman Urs Hölzle John Bruno
Department of Computer Science,
University of California,
Santa Barbara, CA 93106
{murat,urs,bruno}@cs.ucsb.edu
http://www.cs.ucsb/{~murat,~urs,~bruno}

Technical Report TRCS98-31
8 December 1998

## Abstract

*`jContractor` is a purely library and design-pattern based approach to support Design By Contract specifications such as preconditions, postconditions, class invariants, and recovery and exception handling in Java. `jContractor` uses an intuitive naming convention, and standard Java syntax to instrument Java classes and enforce Design By Contract constructs. The designer of a class specifies a contract by defining protected methods which conform to the `jContractor` design patterns. `jContractor` uses Java Reflection to synthesize an instrumented version of a Java class containing `jContractor` contract specifications. The instrumented version contains code which enforces the Design By Contract specifications. Programmers enable the run-time enforcement of contracts by either incorporating the `jContractor` class loader or by instantiating objects directly from the instrumented subclass through the `jContractor` factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not. Since `jContractor` is purely library-based, it works with any Java implementation and requires no special tools such as modified compilers, modified JVMs, or pre-processors.*

## 1. Introduction

One of the shortcomings of mainstream object-oriented languages such as C++ or Java is that class or interface definitions provide only a signature-based application interface, much like the APIs specified for libraries in procedural languages. Method signatures provide limited information about the method: the types of the formal parameters, the type of the returned value, and the types of exceptions that may be thrown. While type information is useful, signatures by themselves do not capture the essential semantic information about what the method does and promises to deliver, or what conditions must be met in order to use the method successfully. To acquire this information, the programmer must either analyze the source code (if available) or rely on some externally communicated specification or documentation, none of which is automatically checked at compile or runtime.

A programmer needs semantic information to correctly design or use a class. Meyer introduced *Design By Contract,* as a way to specify the essential semantic information and constraints that govern the design and correct use of a class [M92]. This information includes assertions about the state of the object that hold before and after each method call; these assertions are called *class invariants*, and apply to the public interface of the class. The information also includes the set of constraints that must be satisfied by a client in order to invoke a particular method. These constraints are specific to each method, and are called *preconditions* of the method. Each precondition specifies conditions on the state of the object and the argument values that must hold prior to invoking the method. Finally, the programmer needs assertions regarding the state of the object after the execution of a method and the relationship of this state to the state of the object just prior to the method invocation. These assertions are called the *postconditions* of a method. The assertions governing the implementation and the use of a class are collectively called a *contract*. Contracts are not necessarily part of the implementation code of a class, however, a runtime monitor could check whether contracts are being honored.

In this paper we introduce *jContractor*, a pure-Java library-based system which supports contracts in Java using a design-pattern [GHJV95] approach. *jContractor* supports Design By Contract without requiring any special tools such as modified compilers, runtime systems, modified JVMs, or pre-processors, and works with any pure Java implementation. Therefore, a programmer can take advantage of the Design By Contract approach by using the *jContractor* library and by following a simple and intuitive set of conventions.

The *jContractor* library implementation addresses three key issues which arise when adding contracts to Java: how to express preconditions, postconditions and class invariants and incorporate them into a standard Java class definition; how to reference entry values of attributes inside postconditions using standard Java syntax; and how to enforce contracts and report their violations at runtime.

An overview of the *jContractor* approach to solving these problems is given below:

1. Programmers add contract code to a class in the form of methods "contract design patterns." The *jContractor* class loader looks for these patterns and rewrites the code to reflect the presence of contracts.
2. The *jContractor* library instruments the classes that contain *contract design patterns* on the fly during class loading or object instantiation. Programmers enable the run-time enforcement of contracts either by engaging the *jContractor* class loader or by explicitly instantiating objects from the *jContractor* object factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not.
3. *jContractor* uses an intuitive naming convention (contract design patterns) for adding *preconditions, postconditions, class invariants, recovery* and *exception handling* in the form of `protected` methods. This allows the contracts to be distinguished from the functional code. The name and signature of each contract method determines the actual method with which the contract is associated.

4. Postconditions and exception handlers can access the old value of any attribute by using a static method of a *jContractor* class, `OLD`. For example `OLD.value(count)` returns the value of the attribute count at method entry. The code instrumentation arranges for attribute values accessed through the `OLD` interface to be recorded at function entry.

## 2. jContractor Library and Contract Design Patterns

*jContractor* is a purely library-based approach to support Design By Contract constructs in standard Java environments. Table 1 contains a summary of Design By Contract constructs and their corresponding design patterns supported by *jContractor*. One of the key contributions of this work is that *jContractor* supports all Design By Contract principles using a *pure-Java*, *library-only* approach. Therefore, any Java developer can immediately start using Design By Contract without making any changes to the test, development, and deployment environment after obtaining a copy of the *jContractor* classes.

| Design By Contract Construct | Design Pattern | Description |
|---|---|---|
| ***Precondition***<br>*(Client's obligation)* | protected boolean ***methodName_PreCondition(*** *methodSignature)* | Evaluated just before *methodName* with matching signature is executed. If precondition fails the method throws *PreConditionException* without executing the method body. |
| ***Postcondition***<br>*(Implementor's promise)* | protected boolean ***methodName_PostCondition(*** *methodSignature)* | Evaluated just before *methodName* returns (*normal termination*). If the postcondition fails, the method throws *PostConditionException* instead of returning the result. |
| ***Exception Handler***<br>*(Implementor's attempt)* | protected Object ***methodName_OnException(*** *Exception e) throws Exception* | Invoked just after *methodName* throws an Exception (*abnormal termination*). The exception handler provides an opportunity to do recovery by restoring invariants, resetting state, etc., |
| ***Class invariant***<br>*(Implementor's promise)* | protected boolean ***className_ClassInvariant( )*** | For each invocation of a public method, *m*, the class invariant is evaluated once before *m* is executed <u>*and*</u> once before *m* is about to return -- *normal termination.* If class invariant fails, a *ClassInvariantException* is thrown instead of returning the result. |
| ***OLD*** | OLD.***value***(*attr*) → | Expr evaluates to *value* of *attr* on method entry.OLD methods can only be used inside, postcondition and exception handler methods; *attr* can be any class attribute or formal argument. |

**Table 1. Summary of `jContractor` Design By Contract Constructs**

A programmer writes a contract by taking a class or method name, say `put`, then appending a suffix depending on the type of constraint, say `_PreCondition`, to write the `put_PreCondition.` Then the programmer writes the method body describing the precondition. The method can access both the arguments of the `put` method with the identical signature, and the attributes of the class. When *jContractor* instrumentation is engaged at runtime, the precondition gets checked each time the `put` method is called, and the call throws an exception if the precondition fails.

The code fragment in Figure 1 shows a *jContractor*-based implementation of the `put` method for the `Dictionary` class. Contract specifications are shown in Table 2 for inserting an element into the *dictionary*, a table of bounded capacity where each element is identified by a certain character string used as key.

```
class Dictionary  . . {

            Object put(Object x, String key)
            {
                  putBody();
            }
            protected boolean put_PreCondition(Object x, String key)
            {
                  return ( (count <= capacity)
                        && (! key.empty)  );
            }
            protected boolean put_PostCondition(Object x, String key)
            {
                  return (   (has (x))
                         &&   (item (key) == x)
                         &&   (count == OLD.value(count) + 1) )
            }
            protected Object put_OnException(Exception e)
                        throws Exception
            {
                  OLD.restore(count);       //
                  throw e;                  // rethrow exception.
            }

            protected boolean Dictionary_ClassInvariant() {
                  return (count >= 0);
            }
      . . .
}
```

**Figure 1. Dictionary Class Implementing Contract for `put` Method.**

| | Obligations | Benefits |
|---|---|---|
| **Client** | *(Must ensure precondition)* Make sure table is not full & key is a non-empty string | *(May benefit from postcondition)* Get updated table where the given element now appears, associated with the given key. |
| **Supplier** | *(Must ensure postcondition)* Record given element in table, associated with given key. | (*May assume precondition)* No need to do anything if table given is full, or key is empty string. |

**Table 2. Contract Specification for Inserting Element to Dictionary**

## 2.1 Enabling Contracts During Method Invocation

In order to enforce contract specifications at run-time, the contractor object must be instantiated from an instrumented class. This can be accomplished in two possible ways: (1) use the *jContractor class loader* which instruments all classes containing contracts during class loading; (2) use a factory style instantiation using the *jContractor* library.

The simplest and preferred method is to use the *jContractor class loader*, since this requires no changes to a client's code. The following code segment shows how a client declares, instantiates, and then uses a `Dictionary` object, *`dict`*. The client's code remains unchanged whether *jContractor* runtime instrumentation is used or not:

```
Dictionary  dict;                 // Dictionary (Figure-1) defines contracts.

dict = new Dictionary();          // instantiates dict from instrumented or non- instrumented
                                  // class depending on whether jContractor classloader is engaged.
dict.put(obj1, "name1");          // If jContractor is enabled, put-contracts are enforced, i.e.,
                                  // contract  violations result in an exception being thrown.
```

The second approach uses the *jContractor* object factory, by invoking its *`New`* method. The factory instantiation can be used when the client's application must use a custom (or third party) class loader and cannot use *jContractor class loader*. This approach also gives more explicit control to the client over *when* and *which* objects to instrument. Following code segment shows the client's code using *jContractor* factory to instantiate an instrumented `Dictionary` object, *`dict`*:

```
dict = (Dictionary) jContractor.New("Dictionary");
                                  // dict always instantiated from instrumented Dictionary
dict.put(obj1, "name1");          // put-contracts are enforced
```

Syntactically, any class containing *jContractor* design-pattern constructs is still a pure Java class. From a client's perspective, both instrumented and non-instrumented instantiations are still `Dictionary` objects and they can be used interchangeably, since they both provide the same interface and functionality. The only semantic difference in their behavior is that the execution of instrumented methods results in evaluating the contract assertions, (e.g., *put_PreCondition*) and throwing a Java runtime exception if the assertion fails.

Java allows method overloading. *jContractor* supports this feature by associating each method variant with the pre- and postcondition functions with the matching argument signatures.

For any class method, say `foo`, if the class does not define or inherit any boolean method by the name, `foo_PreCondition` with the same argument signature, then implicitly the precondition for the `foo` method is considered "`true`". The same "default" rule also applies to the postconditions and class invariants.

## 2.2 Design Patterns for Preconditions, Postconditions and Class Invariants

The following design patterns summarize the construction of name and signatures for contract methods:

*Precondition*: protected boolean ***methodName*** + ***"_PreCondition"*** + ***(*** *<arg-list>* ***)***
*Postcondition*: protected boolean ***methodName*** + ***"_PostCondition"*** + ***(*** *< arg-list >* ***)***
*ClassInvariant*: protected boolean ***className*** + "***_ClassInvariant***" + ***( )***

Each construct's method body evaluates a boolean result and may contain references to the object's internal state with the same scope and access rules as the original method. Pre- and postcondition methods can also use the original method's formal arguments in expressions. Additionally, postcondition expressions can refer to the old values of object's attributes using static accessor methods of *OLD* class as explained further below.

*jContractor* constructs the precondition expression for a method by logical-OR'ing it with all preconditions for that method that are inherited from the parent class. This prevents a subclass from strengthening its parent preconditions.

Similarly, postconditions (and class invariants), are evaluated as the logical-AND of the current class's postconditions (class invariant) with those that are inherited from the parent classes. This prevents the weakening of parent's postconditions or invariants by a subclass.

## 2.3 Exception Contracts

The postcondition for a method describes the contractual obligations of the contractor object only when the method terminates successfully. When a method terminates abnormally due to some exception, it is not required for the contractor to ensure that the postcondition holds. It is very desirable, however, for the contracting (supplier) objects to be able to specify what conditions must still hold true in these situations, and to get a chance to restore the state to reflect this.

*jContractor* supports the specification of general or specialized exception handling code for methods. The instrumented method contains wrapper code to catch exceptions thrown inside the original method body. If the contracts include an exception-handler method for the type of exception caught by the wrapper, the exception handler code gets executed.

If exception handlers are defined for a particular method, each exception handler must either re-throw the handled exception or compute and return a valid result. If the exception is re-thrown no further evaluation of the postconditions or class-invariants is carried out. If the handler is able to recover by generating a new result, the postcondition and class-invariant checks are performed before the result is returned, as if the method had terminated successfully.

The exception handler method's name is obtained by appending the suffix, "`_OnException`", to the method's name. The method takes a single argument whose type belongs to either one of the exceptions that may be thrown by the original method, or to a more general exception class. The body of the exception handler can include arbitrary Java statements and refer to the object's internal state using the same scope and access rules as the original method itself. The *jContractor* approach is more flexible than the Eiffel "rescue" mechanism because separate handlers can be written for different types of exceptions and more information can be made available to the handler code using the exception object which is passed to the handler method.

## 2.4 Supporting *OLD* Values and Recovery

*jContractor* is able to support Design By Contract style postcondition expressions in which one can refer to the "*old*" state of the object just prior to the method's invocation using a clean and safe instrumentation "trick". We introduce a new class, ***OLD***, with a static method, ***value***, which allows us to mimic the Eiffel keyword, ***old***. In the postcondition of the `put` method, the subexpression:

```
(count == OLD.value(count) + 1)
```

illustrates the use of the OLD class interface to access the "old" value of the object's "`count`" attribute. Here the old value refers to the value of the variable, `count`, at the point just before the put-method began to execute.

To emulate this behavior, *jContractor* generates code to save all attributes accessed in the method's postconditions or exception handlers using *OLD*. *jContractor* then replaces all `OLD.value(attr)` expressions by simple variable expressions referring to the temporary variables used to record the method entry value of the attribute.

It is also possible for an exception handler or postcondition method to revert the state of *attr* to its old value by:

```
attr = OLD.value(attr);
```

This may be used as a basic recovery mechanism to restore the state of the object when an invariant or postcondition is found to be violated or within an exception-handler.

## 2.5 Contract Specifications for Interfaces.

Java `interfaces` can only be used to declare method names and signatures. Since *jContractor* contracts are written as method declarations, it is impossible to include them "inside" an interface declaration. Nevertheless, it is desirable to write contracts for an interface. *jContractor* allows interface contracts to be externally provided as a separate class, adhering to certain naming conventions and design patterns. For example, consider the `interface` *`IX`* and the class *`C`* which implements this interface. The class *`IX_CONTRACT`*, contains the pre- and postconditions for the methods in *`IX`*. *jContractor* finds contract implementation classes for interfaces using the naming convention of appending "*`_CONTRACT`*" to the interface name. Methods defined in the contract class are used to instrument the class "implementing" the interface.

Contracts for interface classes can only include pre- and postconditions, and can only express constraints about the method arguments, without any references to the object state. If the implementation class also specifies a precondition for the same method, the conditions are *logical-OR*'ed during instrumentation (similar to the inherited conditions.) Similarly, postconditions are combined using *logical-AND*.

```
interface IX {
      int foo(<args>);
}


class IX_CONTRACT{

      protected boolean foo_PreCondition(<args>)  { . . . }
      protected boolean foo_PostCondition(<args>) { . . . }
}

class C  . . .
      implements IX;
{
      . . .
      int foo(<args>) { . . . }
      . . .
}
```

The following is a client code snippet where an IX contractor object is instantiated using the *jContractor* and the implementing class, ***C***:

```
IX ixObj =  (IX) new C();
. . .
ixObj.foo(…);    // pre & postcondition contracts
                 // defined by IX_CONTRACT are enforced
```

## 3. Design and Implementation of *jContractor*

The *jContractor* package uses the *Java Reflection* mechanism to detect Design By Contract patterns during object instantiation or class loading. Classes containing contract patterns are instrumented on the fly using the *jContractor* library. We begin with explaining how instrumentation of a class is done using the two different mechanisms explained in section 2.1. The rest of this section explains the details of the instrumentation algorithm.

The primary instrumentation technique uses the *jContractor class loader* to transparently instrument classes during class loading. The scenario depicted in Figure 2 illustrates how the *jContractor Class Loader* obtains instrumented class byte-codes from the *jContractor instrumentor* while loading the class, *Foo*. The *jContractor class loader* is engaged when launching the Java application. The instrumentor is passed the name of the class by the class loader and in return it searches the compiled class, *Foo*, for *jContractor* contract patterns. If the class implements contract methods the instrumentor makes a copy of the class byte-codes, modifying the public methods with wrapper code to check contract violations, and returns the modified byte-codes to the class loader. Otherwise, it returns the original class without any modification. The object instantiated from the instrumented class is shown as the *Foo** object in the diagram, to highlight the fact that it is instrumented, but syntactically it is a *Foo* object.

If the command line argument for `jContractor` is not present when starting up the application, the user's own (or the default) class loader is used, which effectively turns off the *jContractor* instrumentation. Since contract methods are separate from the public methods, the program's behavior remains exactly the same except for the runtime checking of contract violations. This is the preferred technique since the client's code is essentially unchanged and all that the supplier has to do is to add the *jContractor* contract methods to the class.

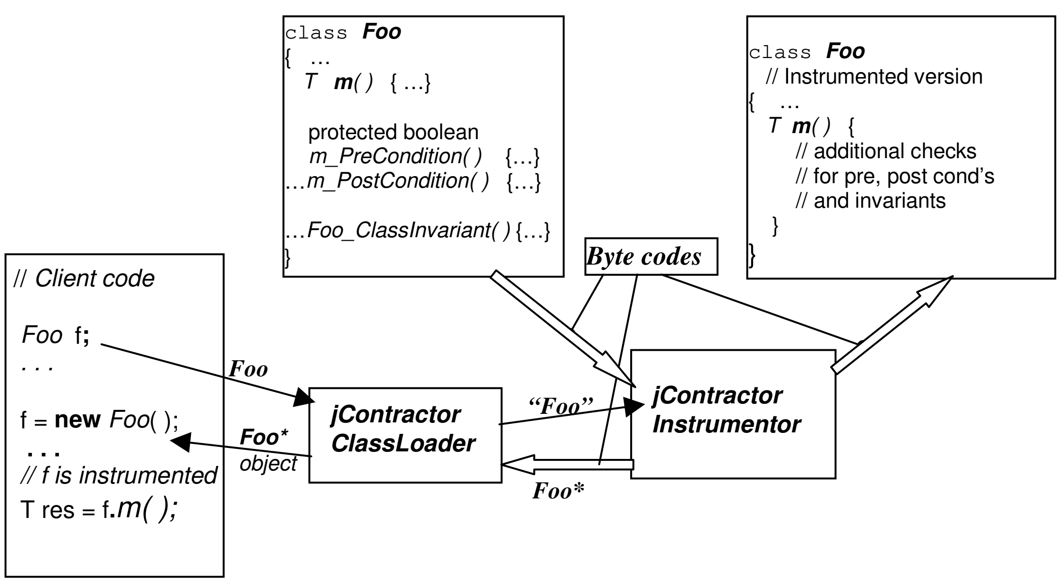



**Figure 2.** ***jContractor Class Loader*** **based Instrumentation**

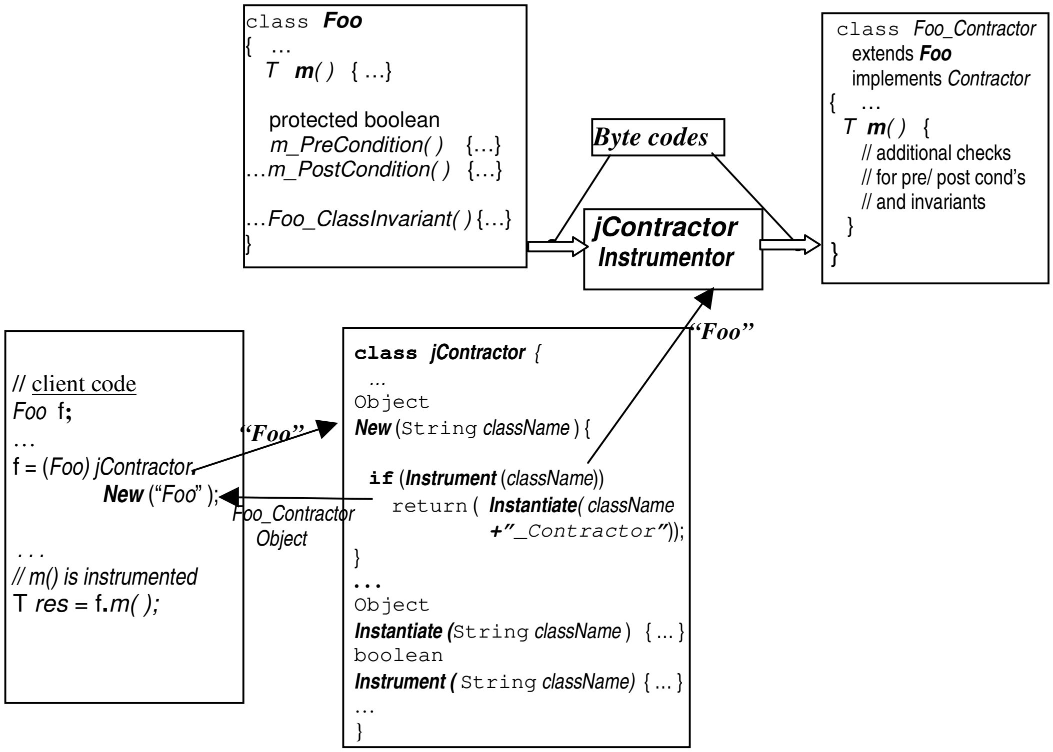



**Figure 3** ***jContractor*** **Factory Style Instrumentation and Instantiation**

The alternative technique is a factory style object instantiation using the *jContractor* library's `New` method. `New` takes a class name as argument and returns an instrumented object conforming to the type of requested class. Using this approach the client explicitly instructs *jContractor* to instrument a class and return an instrumented instance. The factory approach does not require engaging the *jContractor* class loader and is safe to use with any pure-Java class loader. The example in Figure 3 illustrates the factory style instrumentation and instantiation using the class *Foo*. The client invokes *jContractor*.`New()` with the name of the class, "*Foo*". The `New` method uses the *jContractor Instrumentor* to create a subclass of *Foo*, with the name, *Foo_Contractor* which now contains the instrumented version of *Foo*. `New` instantiates and returns a *Foo_Contractor* object to the client. When the client invokes methods of the returned object as a *Foo* object, the instrumented methods in *Foo_Contractor* get called due to the polymorphic assignment and the dynamic binding of methods in Java.

The remainder of this section contains details of the instrumentation algorithm for individual *jContractor* constructs.

## 3.1 Method Instrumentation

*jContractor* instruments contractor objects using a simple code rewriting technique. Figure 4 illustrates the high level view of how code segments get copied from original class methods into the target instrumented version. Two basic transformations are applied to the original method body. First, `return` statements are replaced by an assignment

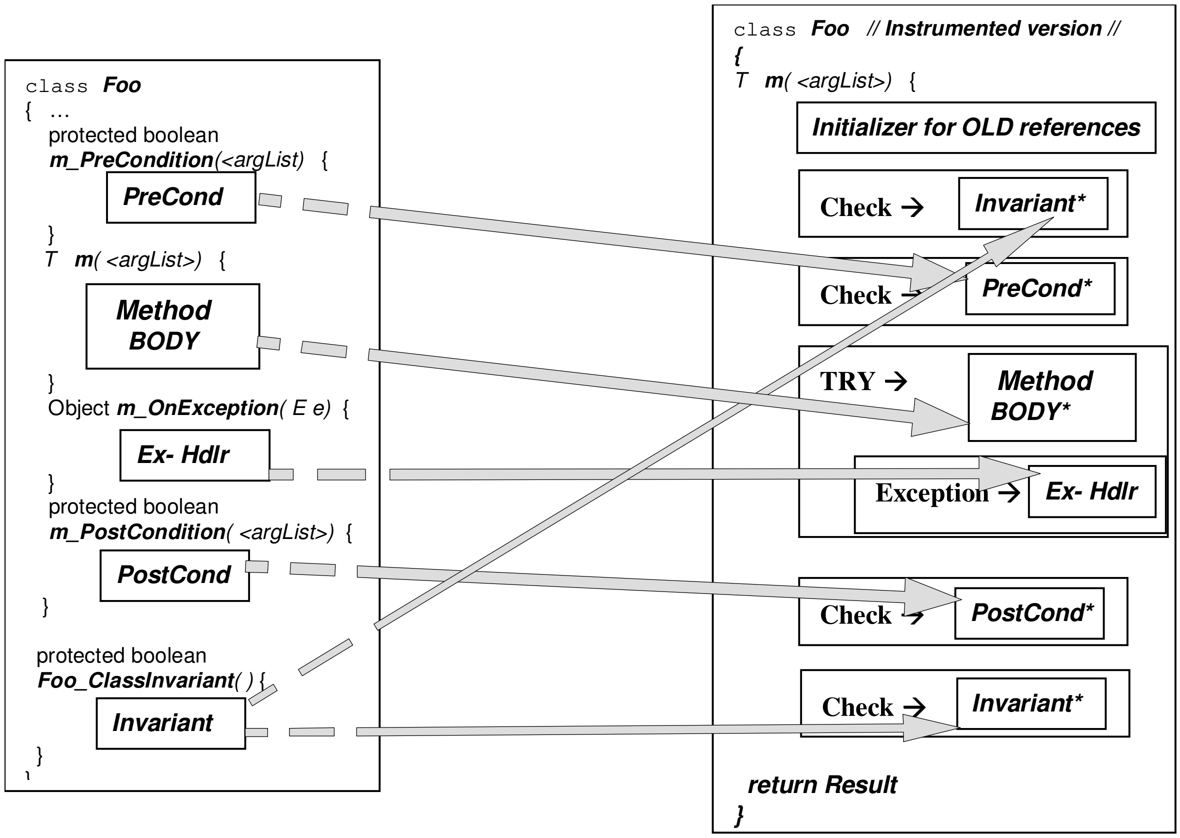



**Figure 4** ***jContractor*** **Instrumentation Overview**

statement – storing the result in a method-scoped temporary – followed by a labeled `break`, to exit out of the method body. Second, references to "old" values, using the OLD class' static methods are replaced by a single variable – this is explained in more detail in a subsection. After these transformations, the entire method block is placed inside a wrapper code as shown in the instrumented code in Figure 4.

A *check wrapper* checks the boolean result computed by the wrapped block and throws an exception if the result is `false`. A *TRY wrapper* executes the wrapped code inside a try-catch block, and associates each exception handler that the contract specifies with a catch phrase inside an exception wrapper. *Exception wrappers* are simple code blocks that are inserted inside the catch clause of a try-catch block with the matching `Exception` type. Typically, exception handlers re-throw the exception, which causes the instrumented method to terminate with the thrown exception. It is possible, however, for the exception handler to recover from the exception condition and generate a result.

Figure 5 illustrates a concrete example referring to the `Dictionary` class shown in Figure 1. This example shows the instrumented code corresponding to the `put` method of the `Dictionary` class.

### 3.2 Instrumentation of OLD References

*jContractor* allocates temporaries and records the values at method entry for all attributes accessed in the method's postconditions or exception handlers using OLD interface. The code rewriting logic then replaces all occurrences of "`OLD.value(attr)`" with a reference to the temporary variable associated with the attribute, `attr`.

### 3.3 Performance Considerations and Implementation Status

The overhead of performing the runtime checks only affects the instrumented method

```
class Dictionary_Contractor  extends Dictionary … {
      . . .
      Object put(Object x, String key)
      {
            Object   $put$Result;
            boolean  $put_PreCondition,
                     $put_PostCondition,
                     $ClassInvariant;
            int  $OLD_$count = this.count;

            $put_PreCondition = (  (count <= capacity)
                                   && (! key.empty) );
            if (!$put_PreCondition) {
                  throw new PreConditionException();
            }
            $ClassInvariant = (count >= 0);
            if (!$ClassInvariant) {
                     throw new ClassInvariantException();
            }

            try {
                  $put$Result = putBody();
            }
            catch (Exception e) {      // put_OnException
                  count = $OLD_$count; //restore(count)
                  throw e;
            }

            $put_PostCondition =((has(x)) &&
                                 (item (key) == x) &&
                                 (count == $OLD_$count + 1));
            if (!$put_PostCondition) {
                     throw new PostConditionException();
            }
            $ClassInvariant = (count >= 0);
            if (!$ClassInvariant) {
                     throw new ClassInvariantException();
            }
            return $put$Result;
      }
   } . . .
```

**Figure 5. Factory Instrumented Dictionary Subclass.**

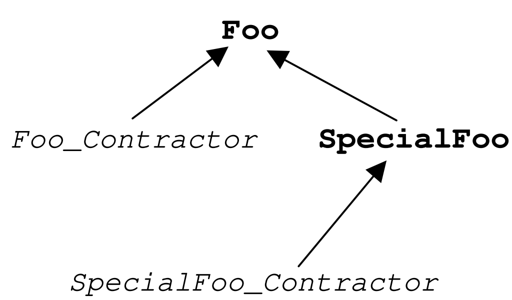
calls involving objects instantiated from a class containing contracts. The cost of evaluating the pre- and postconditions and class invariants during a method call can be significant, since a boolean expression needs to be evaluated for each condition. However, during instrumentation *jContractor* inlines the code for the contract methods within the method body, and thus avoids the function call overhead that would otherwise be present in a user-enforced contract implementation.

We have not completed the implementation of *jContractor* libraries at the time of submitting this paper, however, we plan to make it available in the near future.

## 4. Discussion

### 4.1 Interaction with Inheritance and Polymorphism

When factory style instrumentation is used, *jContractor* constructs a `Contractable` subclass as a direct descendant of the original base class. Therefore it is possible to pass objects instantiated using the instrumented subclass to any client expecting an instance of the base class. Other than enforcing the contract specifics, an instrumented subclass, say *`Foo_Contractor`*, has the same interface as the base class, `Foo`, and type-wise conforms to `Foo`. This design allows the contractor subclasses to be used with any polymorphic substitution involving the base class. Consider the following class hierarchy where the `SpecialFoo` class extends the base class `Foo` and therefore is a more specialized version of `Foo`. *jContractor'*s factory instrumentation method can be used with both `SpecialFoo` and `Foo` to yield the sub classes: `Foo_Contractor` and `SpecialFoo_Contractor` into the class hierarchy as shown below:




*jContractor* allows for the polymorphic substitution of either `SpecialFoo` objects or the instrumented, `SpecialFoo_Contractor` objects, with `Foo` objects. In a polymorphic assignment, a `SpecialFoo` object, *`sf`*, can be passed to a client expecting a `Foo` type object. The client, in this case is expected to only know of `Foo` objects, and interact with the object, *`sf`*, based on the contractual agreements with `Foo` Class. The evaluation of the pre and postconditions and throwing of the related exceptions work as expected with the `Foo` methods inherited by the `SpecialFoo` object.

The contravariance problem that arises in the inheritance of contracts when a subclass strengthens the precondition, say by adding new constraints or weakens the postcondition is avoided in *jContractor* by the following technique. The preconditions are always evaluated as the logical-OR of all preconditions that are inherited from parent classes. This prevents the subclass from strengthening parents' precondition. Similarly, postcondition are logical-AND'ed with those that are inherited.

### 4.2 Limitations

The *jContractor* factory subclasses the base contract class during instrumentation so factory style instrumentation fails for classes that are `final`. Client application must use the *jContractor* class loader to enforce contracts for final classes.

## 5. Related Work

The idea of associating boolean expressions (assertions) with code as a means to argue the code's correctness can be traced back to Hoare [H69] and others who worked in the field of program correctness. The idea of extending an object-oriented language using only libraries and design patterns appeared in [KB93]. The notion of compiling assertions into runtime checks first appeared in the *Eiffel* language [M92b].

*Eiffel* is an elegant language with built-in language and runtime support for Design By Contract. *Eiffel* integrates preconditions (*require-clause*), postconditions (*ensure-clause*), class invariants, **old** and **rescue/retry** constructs into the definition of methods and classes. *jContractor* is able provide all of the contract support found in *Eiffel*, with the following differences: *jContractor* supports exception-handling with finer exception resolution – as opposed to a singe **rescue** clause; *JContractor* does not support the **retry** construct of *Eiffel*. We believe that if such recovery from an exception condition is possible, it is better to incorporate this handler into the implementation of the method itself, and not throw the exception in the first place.

Duncan & Hölzle introduced Handshake[DH98], which allows a programmer to write external contract specifications for Java classes and interfaces without changing the classes themselves. Handshake is implemented as a dynamically linked library and works by intercepting the JVM's file accesses and instrumenting the classes on the fly using a mechanism called, binary component adaptation (BCA). BCA is developed for on the fly modification of pre-compiled Java components (class byte-codes) using externally provided specification code containing directives to alter the pre-compiled semantics [KH98]. The flexibility of the approach allows Handshake to add contracts to classes declared `final`; system classes; and interfaces as well as classes. Some of the shortcomings of the approach are that contract specifications are written externally using special syntax; and that Handshake Library is a non-Java system that has to be ported to and supported on different platforms.

Kramer's *iContract* is a tool designed for specifying and enforcing contracts in Java [K98]. Using *iContract*, pre-, postconditions and class invariants can be annotated in the Java source code as "comments" with tags such as: @pre, @post. The *iContract* tool acts as a pre-processor which translates these assertions and generates modified versions of the Java source code. *iContract* uses its own specification language for expressing the boolean conditions.

Mannion and Philips have proposed an extension to the Java language to support Design By Contract [MM98], employing *Eiffel*-like keyword and expressions which become part of a method's signature. Mannion's request that Design By Contract be directly supported in the language standard is reportedly the most popular "non-bug" request in the Java Developer Connection Home Page (bug number 4071460).

Porat and Fertig propose an extension to C++ class declarations to permit specification of pre- and postconditions and invariants using an assertion-like semantics to support Design By Contract [PF95].

## 6. Conclusion

We have introduced *jContractor,* a purely library-based solution to write Design By Contract specifications and to enforce them at runtime using Java. The *jContractor* library and design patterns can be used to specify the following Design By Contract constructs: pre- and postconditions, class invariants, exception handlers, and **old** references. Programmers can write contracts using standard Java syntax and an intuitive naming convention. Contracts are specified in the form of protected methods in a class definition where the method names and signatures constitute the *jContractor* design patterns. *jContractor* checks for these patterns in class definitions and rewrites on the fly instrumented versions of these classes that checks contracts violations at runtime.

The greatest advantage of *jContractor* over the other is the ease of deployment. Since *jContractor* is purely library-based it does not require any special tools such as modified compilers, runtime systems, pre-processors or JVMs, and works with any pure Java implementation.

*jContractor* library instruments the classes that contain *contract design patterns* during class loading or object instantiation. Programmers enable the run-time enforcement of contracts by using a command line switch at start-up which installs the *jContractor* instrumenting class loader. An alternative mechanism is to instantiate objects using the *jContractor* object factory. The *jContractor* factory instantiates object from a new class it creates as a subclass of the base class. The new class redefines the methods in its interface with instrumented versions. The *jContractor* factory can use any class loader and no command line switch is needed to enable it. Either way, programmers can use exactly the same syntax for invoking methods or passing object references regardless of whether contracts are present or not. Contract violations result in the method throwing proper runtime exceptions.

We also describe an interesting instrumentation technique which allows accessing the *old* value of variables when writing postcondition and invariant methods. For example, `OLD.value(count)` returns the value of `count` at method entry. The instrumentation arranges for values accessed through the *OLD* interface to be recorded at method entry and rewrites the "*OLD*.value*(count)*" expression as a single variable to access the recorded value.

*jContractor* delivers all the benefits of Design By Contract to Java in a clean, easily deployable and efficient way.

## 7. References


[DH98] Andrew Duncan and Urs Hölzle. *Adding Contracts to Java with Handshake*. Technical Report TRC98-32, University of California, Santa Barbara, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns – Elements of Reusable Software*, Addison-Wesley, 1995.

[H69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

[KB93] Murat Karaorman and John Bruno. Introducing Concurrency to a Sequential Langauge. *Communications of the ACM*. Vol.36, No.9, September 1993, pp.103-116.

[K98] Reto Kramer. *iContract – The Java Design by Contract Tool*. Proc. of TOOLS '98, Santa Barbara, CA August 1998. Copyright IEEE 1998.

[KH98] Ralph Keller and Urs Hölzle. *Binary Component Adaptation*. Proc. of ECOOP '98, Lecture Notes in Computer Science, Springer Verlag, July 1998.

[M92] Bertrand Meyer. *Applying Design by Contract*. In Computer (IEEE), vol. 25, no. 10, October 1992, pages 40-51.

[M92b] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

[MM98] Mike Mannion and Roy Phillips. *Prevention is Better than a Cure*. Java Report, Sept.1998.

[PF95] S.Porat and P.Fertig. *Class Assertions in C++*. Journal of Object Oriented Programming, 8(2):30-37, May 1995.