jContractor: A Reflective Java Library to Support Design By Contract

Murat Karaorman[1,2], Urs Hölzle[2], John Bruno[2]

[1]Texas Instruments Inc.,
315 Bollay Drive, Santa Barbara, California 93117
muratk@ti.com

[2]Department of Computer Science,
University of California, Santa Barbara, CA 93106
{murat,urs,bruno}@cs.ucsb.edu

Abstract. *jContractor* is a purely library 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 providing contract methods following *jContractor* naming conventions. *jContractor* uses Java Reflection to synthesize an instrumented version of a Java class by incorporating code that enforces the present *jContractor* contract specifications. Programmers enable the run-time enforcement of contracts by either engaging the *jContractor* class loader or by explicitly instantiating objects using 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. Since *jContractor* is purely library-based, it 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++ and 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 formal parameters, the type of 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 [6]. 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 specification constructs which 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 purely library-based system and a set of naming conventions to support *Design By Contract* in Java. The *jContractor* system does not require 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 practice *Design By Contract* by using the *jContractor* library and by following a simple and intuitive set of conventions.

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine (JVM) `class` file format [10]. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. *jContractor* utilizes the meta-level information encoded in the standard Java class files to instrument the bytecodes on-the-fly during class loading. During the instrumentation process *jContractor* parses each Java class file and discovers the *jContractor* contract information by analyzing the class meta-data.

The *jContractor* design 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, to check method results inside postconditions using standard Java syntax; and how to check and enforce contracts at runtime.

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

1. Programmers add contract code to a class in the form of methods following *jContractor*'s naming conventions: *contract patterns*. The *jContractor* class loader recognizes these patterns and rewrites the code to reflect the presence of contracts.

2. Contract patterns can be inserted either directly into the class or they can be written separately as a *contract class* where the contract class' name is derived from the target class using *jContractor* naming conventions. The separate contract class approach can also be used to specify contracts for interfaces.

3. The *jContractor* library instruments the classes that contain *contract 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.

4. *jContractor* uses an intuitive naming convention for adding *preconditions*, *postconditions*, *class invariants*, *recovery* and *exception handling* in the form of `protected` methods. Contract code is hence distinguished from the functional code. The name and signature of each contract method determines the actual method with which the contract is associated.

5. Postconditions and exception handlers can access the *old* value of any attribute by using a special object reference, *`OLD`*. For example *`OLD.count`* returns the value of the attribute *`count`* just prior to the execution of the method. *jContractor* emulates this behavior by transparently rewriting class methods during class loading so that the entry values of *`OLD`* references are saved and then made available to the postcondition and exception handling code.

6. *jContractor* provides a class called *`RESULT`* with a `static boolean` method, *`Compare.`* Inside a method's postcondition it is possible to check the *result* associated with the method's execution by calling *`RESULT.Compare(<expression>)`*. The call returns `true` or `false` by comparing the value of the *`<expression>`* with the current result.

This paper presents an extension to Java to support Design By Contract by introducing *jContractor* as a pure library-based approach which utilizes the meta-level information found in Java class files and takes advantage of dynamic class loading in order to perform "reflective", on-the-fly bytecode modification.

2 *jContractor* Library and Contract Patterns

jContractor is a purely library-based approach to support Design By Contract constructs using standard Java. Table 1 contains a summary of key Design By Contract constructs and the corresponding *jContractor* patterns. One of the key contributions of *jContractor* is that it supports all Design By Contract principles using

a *pure-Java, library-based* 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 *jContractor* classes.

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

Construct	***jContractor Pattern***	***Description***

Precondition
(Client's obligation)
	`protected boolean`
	`methodName_PreCondition(`
	`<arg-list>)`
		Evaluated just before `methodName` with matching signature is executed. If the precondition fails the method throws a `PreConditionException` without executing the method.

Postcondition
(Implementor's promise)
	`protected boolean`
	`methodName_PostCondition(`
	`<arg-list>)`
		Evaluated just before `methodName` returns (i.e. *normal termination*). If the postcondition fails, a `PostConditionException` gets thrown.

Exception Handler
(Implementor's attempt)
	`protected Object`
	`methodName_OnException(`
	`Exception e)`
	`throws Exception`
		Called when `methodName's` execution ends in *abnormal termination*, throwing an Exception. The exception handler provides an opportunity to do recovery by restoring invariants, resetting state, etc.,

Class invariant
(Implementor's promise)
	`protected boolean`
	`className_ClassInvariant(`
	`)`
		Evaluated once before each invocation of a public method, `m`, *and* once before `m` is about to return -- *normal termination.* If class invariant fails, a `ClassInvariantException` is thrown instead of returning the result.

old	`OLD.attr` `OLD.foo( )`	Expression evaluates to *value* of `attr` on method entry. `OLD` methods can only be used inside postcondition and exception handler methods; `attr` can be any non-private class attribute.
Result	`RESULT.compare(expr)`	Evaluates true/false depending on if current result matches the expr. RESULT class is part of *jContractor* distribution.

```
class Dictionary  … {
    protected Dictionary OLD;

    Object put(Object x, String key)
    {
          putBody();
    }
    protected boolean put_PreCondition(Object x,
                                       String key)
    {
          return ( (count <= capacity)
                &&  !(key.length()==0)  );
    }
    protected boolean put_PostCondition(Object x,
                                        String key)
    {
          return (   (has (x))
                  &&  (item (key) == x)
                  &&  (count == OLD.count + 1) )
    }
    protected Object put_OnException(Exception e)
                      throws Exception
    {
          count =  OLD.count;
          throw e;         //rethrow exception.
    }
    protected boolean Dictionary_ClassInvariant()
    {
          return (count >= 0);
    }
 …}
```

Figure 1-a. Dictionary Class Implementing Contract for *put* Method

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-a shows a *jContractor* based implementation of the put method for the *Dictionary* class. An alternative approach is to provide a separate *contract class*, *Dictionary_CONTRACT*, as shown in Figure 1-b, which

```
class Dictionary_CONTRACT extends Dictionary …
{
    protected boolean put_PostCondition(Object   x,
                                        String key)
    {
          return (   (has (x))
                  && (item (key) == x)
                  && (count == OLD. count + 1) )
    }

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

Figure 1-b. Separate Contract Class for *Dictionary*

contains the contract code using the same naming conventions. The contract class can (optionally) extend the target class for which the contracts are being written, which is the case in our example. For every class or interface *X* that the *jContractor ClassLoader* loads, it also looks for a separate contract class, *X_CONTRACT*, and uses contract specifications from both *X* and *X_CONTRACT* (if present) when performing its instrumentation. The details of the class loading and instrumentation will be presented in subsequent sections.

Table 2 shows the informal contract specifications 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.

Table 2. Contract Specification for Inserting Element to Dictionary

	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.

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) by using the *jContractor class loader* which instruments all classes containing contracts during class loading; (2) by using a factory style instantiation using the *jContractor* library.

The simplest and the 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
                             //  jContractor classloader being 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 the *jContractor* factory to instantiate an instrumented `Dictionary` object, *`dict`*:

```
dict = (Dictionary) jContractor.New("Dictionary");
                                  // instruments  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 method, say `foo`, of class `X`, if there is no `boolean` method by the name, `foo_PreCondition` with the same argument signature, in either `X`, `X_CONTRACT` or one of their descendants then the default precondition for the `foo` method is "true". The same "default" rule applies to the postconditions and class invariants.

2.2 Naming Conventions for Preconditions, Postconditions and Class Invariants

The following naming conventions constitute the *jContractor* patterns for pre- and postconditions and class invariants:

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 by declaring a pseudo object, `OLD`, with the same class type and using the `OLD` object to access the values.

2.3 Exception Handling

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's "*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 uses a clean and safe instrumentation "trick" to mimic the Eiffel keyword, *old*, and 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. The trick involves using the "syntax notation/convention", *OLD.x* to mean the value that x had when method body was entered. Same notation is also used for method references as well, e.g., *OLD.foo()* is used to refer to the result of calling the member *foo()* when entering the method. We will later explain how the *jContractor* instrumentation process rewrites expressions involving *OLD* to achieve the desired effect. First we illustrate its usage from the example in Figure 1. The class Dictionary first declares *OLD*:

```
private Dictionary OLD;
```

Then, in the postcondition of the `put` method taking *<Object x, String key>* arguments, the following subexpression is used

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

to specify that the execution of the corresponding put method increases the value of the object's *count* by 1. Here *OLD.count* refers to the value of *count* at the point just before the *put*-method began to execute.

jContractor implements this behavior using the following instrumentation logic. When loading the Dictionary class, *jContractor* scans the postconditions and exception handlers for *OLD* usage. So, when it sees the *OLD.count* in *put_PostCondition* it inserts code to the beginning of the `put` method to allocate a unique temporary and to save count to this temporary. Then it rewrites the expression in the postcondition replacing the *OLD.value* subexpression with an access to the temporary. In summary, the value of the expression *OLD.expr* (where *expr* is an arbitrary sequence of field dereferences or method calls) is simply the value of *expr* on entry to the method.

It is also possible for an exception handler or postcondition method to revert the state of `attr` to its old value by using the `OLD` construct. 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 within an exception-handler. For example,

```
attr = OLD.attr;
```

or

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

The first example restores the object reference for `attr` to be restored, and the second example restores the object state for `attr` (by cloning the object when entering the method, and then attaching the object reference to the cloned copy.)

2.5 Separate Contract Classes

jContractor allows contract specifications for a particular class to be externally provided as a separate class, adhering to certain naming conventions. For example, consider a class, *X*, which may or may not contain *jContractor* contract specifications. *jContractor* will associate the class name, `X_CONTRACT`, with the class *X*, as a potential place to find contract specifications for *X*. `X_CONTRACT` must extend class *X* and use the same naming conventions and notations developed earlier in this paper to specify pre- and postconditions, exception handlers or class invariants for the methods in class *X*.

If the implementation class *X* also specifies a precondition for the same method, that precondition is *logical-AND*'ed with the one in `X_CONTRACT` during instrumentation. Similarly, postconditions, and class invariants are also combined using *logical-AND*. Exception handlers in the contract class override the ones inherited from *X*.

The ability to write separate contract classes is useful when specifying contracts for legacy or third party classes, and when modifying existing source code is not possible or viable. It can be used as a technique for debugging and testing system or third-party libraries.

2.6 Contract Specifications for Interfaces

Separate contract classes also allow contracts to be added to interfaces. 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`. Methods defined in the contract class are used to instrument the class "implementing" the interface.

```
interface IX {

      int foo(<args>);
}

class IX_CONTRACT  {

  protected boolean foo_PreCondition(<args>) { … }
  protected boolean foo_PostCondition(<args>){ … }
}
```

Contracts for interface classes can only include pre- and postconditions, and can only express constraints using expressions involving the method's arguments or interface method calls, without any references to a particular object state. If the implementation class also specifies a precondition for the same method, the conditions are *logical-AND*'ed during instrumentation. Similarly, postconditions are also combined using *logical-AND*.

3 Design and Implementation of *jContractor*

The *jContractor* package uses *Java Reflection* 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 by 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 bytecodes from the *jContractor instrumentor* while loading 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 contains contract methods, the instrumentor makes a copy of the class bytecodes, modifying the public methods with wrapper code to check contract violations, and returns the modified bytecodes 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<Instrumented>* object in the diagram, to highlight the fact that it is instrumented, but syntactically it is a *Foo* object.

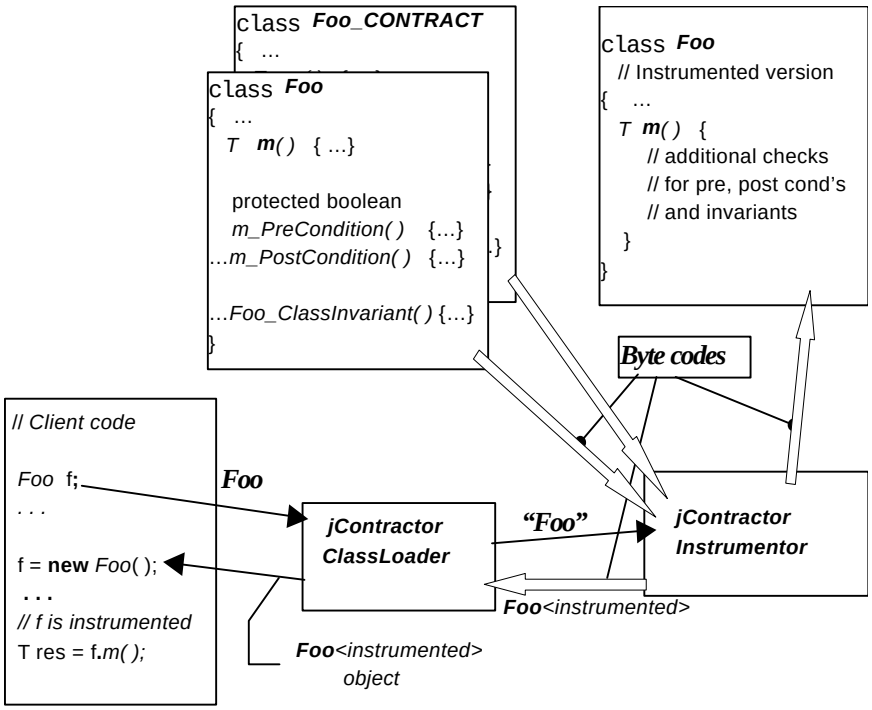


Figure 2. ***jContractor*** **Class Loader based Instrumentation**

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.

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

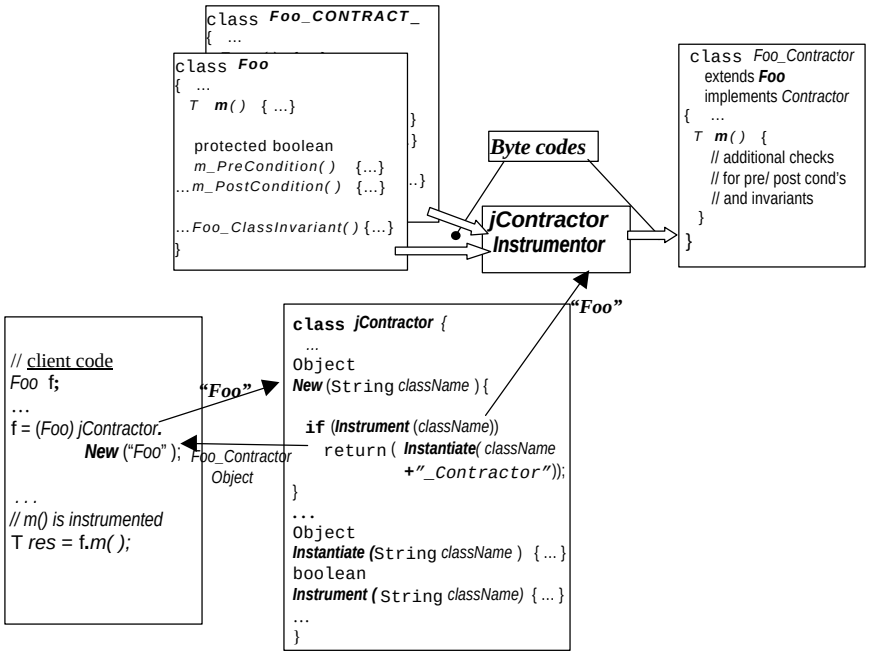


Figure 3 ***jContractor*** **Factory Style Instrumentation and Instantiation**

invokes methods of the returned object as a `Foo` object, it calls the instrumented methods in `Foo_Contractor` due to the polymorphic assignment and dynamic binding.

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. *jContractor*'s key instrumentation policy is to inline the contract code for each method within the target method's body, to avoid any extra function call. Two basic transformations are applied to the original method's body. First, `return` statements are replaced by an assignment 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` object reference 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.

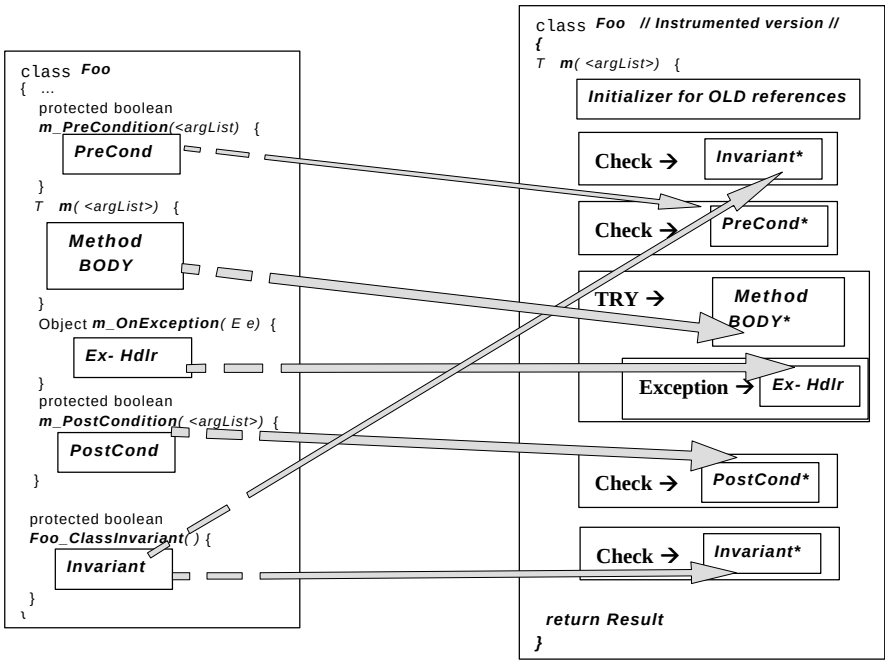


Figure 4. *jContractor* Instrumentation Overview

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 shows a concrete example of the instrumented code that is generated.

3.2 Instrumentation of OLD References

jContractor takes the following actions for each unique *OLD-expression* inside a method's postcondition or exception handler code. Say the method's name is `m()` and the expression is `OLD.attr`, and `attr` has type T, then *jContractor* incorporates the following code while rewriting `m():`

```
T  $OLD_$attr = this.attr;
```

```
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.length()==0) );

      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.

The effect of this code is to allocate a temporary, *$OLD_$attr*, and record the value

of the expression, *attr*, when the method code is entered. The code rewriting logic then replaces all occurrences of *OLD.attr* inside the contract code with the temporary variable *$OLD_$attr* whose value has been initialized once at the beginning of the method's execution.

3.3 Instrumentation of RESULT References

jContractor allows the following syntax expression inside a method's postcondition method to refer to the result of the current computation that led to its evaluation:

```
RESULT.Compare(expression)
```

RESULT is provided as part of the *jContractor* library package, to facilitate this syntax expression. It exports a single *static boolean* method, *Compare()*, taking a single argument with one variant for each built-in Java primitive type and one variant for the *Object* type. These methods never get invoked in reality, and the sole purpose of having them (like the *OLD* declarations discussed earlier) is to allow the Java compiler to legally accept the syntax, and then rely on the instrumentation logic to supply the right execution semantics.

During instrumentation, for each method declaration, *T m()*, a temporary variable *$m_$Result* is internally declared with the same type, *T*, and used to store the result of the current computation. Then the postcondition expression shown above is rewritten as:

```
($m_$Result == (T)(expression))
```

3.4 Use of Reflection

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine `class` file format. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. During the instrumentation process *jContractor* parses and analyzes the meta-information encoded in the class byte-codes in order to discover the *jContractor* contract patterns. When the class contains or inherits contracts, *jContractor* instrumentor modifies the class bytecodes on the fly and then passes it to the class loader. The class name and its inheritance hierarchy; the method names, signatures and code for each class method; the attribute names found and referenced in class methods constitute the necessary and available meta information found in the standard Java class files. The presence of this meta information in standard Java class byte codes and the capability to do dynamic class loading are essential to our way building a pure-library based *jContractor* implementation.

Core Java classes include the `java.lang.reflect` package which provides reflection capabilities that could be used for parsing the class information, but using this package would require prior loading of the class files into the JVM. Since jContractor needs to do its instrumentation *before* loading the class files, it cannot use core reflection classes directly and instead uses its own class file parser.

4 Discussion

4.1 Interaction Of Contracts With Inheritance And Polymorphism

Contracts are essentially specifications checked at run-time. They are not part of the functional implementation code, and a "correct" program's execution should not depend on the presence or enabling of the contract methods. Additionally, the exceptions that may be thrown due to runtime contract violations are not checked exceptions, therefore, they are not required to be part of a method's signature and do not require clients' code to handle these specification as exceptions. In the rest of this section we discuss the contravariance and covariance issues arising from the way contracts are inherited.

The inheritance of preconditions from a parent class follows *contravariance*: as a subclass provides a more specialized implementation, it should weaken, not strengthen, the preconditions of its methods. Any method that is redefined in the subclass should be able to at least handle the cases that were being handled by the parent, and in addition handle some other cases due to its specialization. Otherwise, polymorphic substitution would no longer be possible. A client of *X* is bound by the contractual obligations of meeting the precondition specifications of *X*. If during runtime an object of a more specialized instance, say of class *Y* (a subclass of *X*) is passed, the client's code should not be expected to satisfy any stricter preconditions than it already satisfies for *X*, irrespective of the runtime type of the object.

jContractor supports contravariance by evaluating the a *logical-OR* of the precondition expression specified in the subclass with the preconditions inherited from its parents. For example, consider the following client code snippet:

```
// assume that class Y extends class X
X x;
Y y = new Y();      // Y object instantiated
x = y;              // x is polymorphically attached
                    // to a Y object
int i = 5; …
x.foo(i);           // only PreCondition(X,[foo,int i]) should be met
```

When executing `x.foo()`, due to dynamic binding in Java, the *foo()* method that is found in class Y gets called, since the dynamic type of the instance is Y. If *jContractor* is enabled this results in the evaluation of the following precondition expression:

PreCondition(X,[foo,int i]) ∨ *PreCondition(Y,[foo,int i])*

This ensures that no matter how strict *PreCondition(Y,foo)* might be, as long as the *PreCondition(X,foo)* holds true, `x.foo()` will not raise a precondition exception.

While we are satisfied with this behavior from a theoretical standpoint, in practice a programmer could violate contravariance. For example, consider the following precondition specifications for the `foo()` method defined both in X and Y, still using the example code snippet above:

PreCondition(X, [foo,int a]) :	*a > 0*	*(I)*
PreCondition(Y, [foo,int a]) :	*a > 10*	*(II)*

From a specification point of view *(II)* is stricter than *(I)*, since for values of a*: 0<a<=10, (II)* will fail, while *(I)* will succeed, and for all other values of a, *(I)* and *(II)* will return identical results. Following these specifications, the call of previous example:

```
x.foo(i);     // where i is 5
```

does not raise an exception since it meets *PreCondition(X,foo,int a).* However, there is a problem from an implementation view, that Y's method `foo(int a)` effectively gets called even though its own precondition specification, *(II),* is violated. The problem here is one of a design error in the contract specification. Theoretically, this error can be diagnosed from the specification code using formal verification and by validating whether following *logical-implication* holds for each redefined method `m()`:

PreCondition(ParentClass,m) Þ *PreCondition(SubClass,m)*

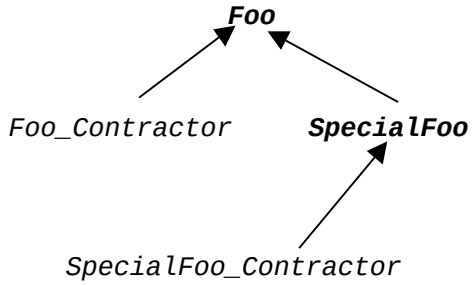
For the previous example, it is easy to prove that *(I)* does not logically-imply *(II).* It is beyond the scope of *jContractor* to do formal verification for logical inference of specification anomalies. *jContractor* does, however, diagnose and report these types of design anomalies, where any one of the *logical-OR*'ed precondition expressions evaluates to *false*. In the above example, *jContractor* would throw an exception to report that the precondition has been illegally strengthened in the subclass, thus forcing the programmer to correct the precondition.

A similar specification anomaly could also occur when a subclass strengthens the parent class's invariants, since *jContractor* checks the class invariants when preconditions are evaluated. The subclass' invariant's runtime violation is caught by *jContractor* instrumented code as an exception, with the correct diagnostic explanation.

The inheritance of postconditions is similar: as a subclass provides a more specialized implementation, it should strengthen, not weaken the postconditions of its interface methods. Any method that is redefined in the subclass should be able to guarantee at least as much as its parent's implementation, and then perhaps some more, due to its specialization. *jContractor* evaluates the *logical-AND* of the postcondition expression found in the subclass with the ones inherited from its parents. Similar anomalies as discussed above for preconditions can also appear in postcondition specifications due to programming errors. *jContractor* will detect these anomalies should they manifest during runtime execution of their respective methods.

4.2 Factory Style Instrumentation Issues

When factory style instrumentation is used, *jContractor* constructs a contractor 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:



jContractor allows for the polymorphic substitution of either `SpecialFoo` objects or the instrumented `SpecialFoo_Contractor` objects with `Foo` objects.

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 [2] and others who worked in the field of program correctness. The idea of extending an object-oriented language using only libraries and naming conventions appeared in [3]. The notion of compiling assertions into runtime checks first appeared in the *Eiffel* language [7].

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 to 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 single *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, which forestalls throwing the exception at all. *jContractor'* support for *old* supports cloning semantics where references are involved, while Eiffel does not.

Duncan & Hölzle introduced Handshake[1], 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 bytecodes) using externally provided specification code containing directives to alter the pre-compiled semantics [5]. The flexibility of the approach allows Handshake to add contracts to classes declared `final`; to system classes; and to interfaces as well as classes. Some of the shortcomings of the approach are that contract specifications are written externally using a 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 [4]. 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 [8], 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* [9].

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 naming conventions 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* naming conventions. *jContractor* checks for these patterns in class definitions and rewrites those classes on the fly by instrumenting their methods to check contract violations at runtime.

The greatest advantage of *jContractor* over existing approaches 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.

The *jContractor* library instruments the classes that contain *contract 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. *jContractor* object factory provides an alternative mechanism that does not require engaging the *jContractor* ClassLoader to instantiate instrumented objects. Clients can instantiate objects directly from the *jContractor* factory, which can use any standard class loader and does not require a command line switch. Either way, clients 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 when instrumented object instances are used.

We also describe a novel instrumentation technique that allows accessing the *old* values of variables when writing postconditions and exception handling methods. For example, `OLD.count` returns the value of the attribute `count` at method entry. The instrumentation arranges for the attribute values or expressions accessed through the `OLD` reference to be recorded at method entry and replaces the `OLD` expressions with automatically allocated unique identifiers to access the recorded value.

References

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

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

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

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

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

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

7. Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

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

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

10. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.