An Assertion Checking Wrapper Design for Java

Roy Patrick Tan
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
rtan@vt.edu

Stephen H. Edwards
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
edwards@cs.vt.edu

ABSTRACT

The Java Modeling Language allows one to write formal behavioral specifications for Java classes in structured comments within the source code, and then automatically generate run-time assertion checks based on such a specification. Instead of placing the generated assertion checking code directly in the underlyling class bytecode file, placing it is a separate wrapper component offers many advantages. Checks can be distributed in binary form alongside the compiled class, and clients can selectively include or exclude checks on a per-class basis without recompilation. In this approach, when checks are excluded the underlying code is just as efficient as if assertions were "compiled out." In addition, a flexible mechanism for enabling or disabling assertion execution on a per-class or per-package basis is also included. This paper presents a design for separating assertion checking code into wrapper classes and discusses the issues arising from this design.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.3.2 [**Programming Languages**]: Language Classifications—*JML*

General Terms

Languages

Keywords

JML, run-time checking, design by contract, interface violation

SAVCBS '03 Helsinki, Finland

1. INTRODUCTION

The Java Modeling language (JML) [6] is a behavioral specification language for Java that allows programmers to add model-based specifications to their code. Specifications, including preconditions, postconditions, and invariants, are placed in specially-formatted, structured comments. The JML tool set allows run-time checks to be generated from such specifications and embedded directly in the generated class file, to be checked at run-time [1]. JML's design-by-contract support provides specific syntactic slots that clearly separate the implementation details from the assertion checks. Its support for model-only fields and methods cleanly supports reasoning about a component's abstract state [2].

The benefits of checking design-by-contract assertions are well known [3, 7]. However, due to performance concerns, it is current practice to include run-time assertion checks during testing, but then remove them when distributing production components. This benefits the original implementor, but does little for the clients of that component. As commercial components become more prevalent, and new designs more frequently make use of classes and subsystems purchased from other sources, it is important to consider how such assertion checks can be of use to component clients, as well as how they might add value to a component being offered for sale.

JML-based assertion checks, like those produced by most other techniques, can be left in the compiled, binary version of a class that is distributed to customers. As with other techniques, execution of these checks at run-time can be controlled through a global switch. However, even when checks are not being executed, the resulting code still suffers a performance penalty, both due to the code bloat imposed by the inactive checks and to the cost of constantly looking up whether or not to perform each check.

This paper discusses ongoing work that will address these issues. The goals of this work include:

- Allowing binary distribution of compiled checks alongside the underlying class, so that checks can be included or excluded without source code access or recompilation.
- Imposing no additional overhead when code is run directly, without including the assertion checking wrappers.
- Supporting per-class or per-package run-time enabling or disabling of assertion check execution.

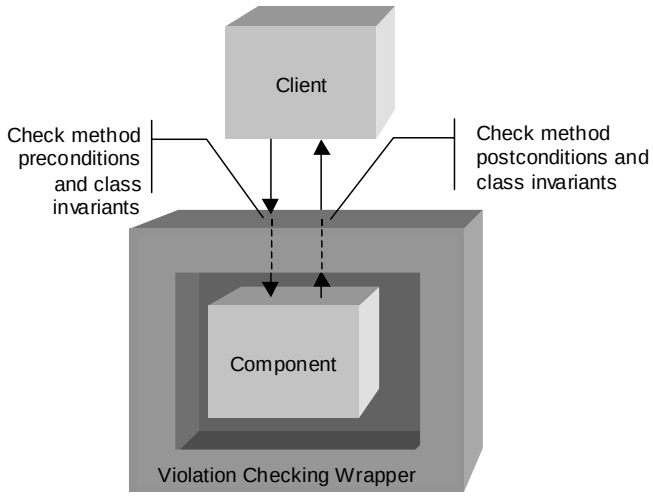

Figure 1: The wrapper implements assertion checks and delegates other work to the wrapped component.

- Maintaining transparent compatibility–JML users will not have to change any existing JML source code or the way they compile their code.

2. A DESIGN FOR ASSERTION CHECKING WRAPPERS

Our approach adopts a wrapper design [4] that begins with a simple idea: move assertion checking code to a separate class, such that we now have two classes that have the same externally visible features: an unwrapped original class, and the wrapper class that performs the checks but delegates actual computation to the wrapped component. Figure 1 illustrates this approach.

In principle, the concept of using a wrapper to separate assertion checks from the underlying component is simple. Following the decorator pattern [5], the wrapper provides the same external interface as the underlying component, and just adds extra features transparently. One can then extract a common interface capturing the publicly visible features of the underlying component, and set up the wrapper and the original component as two alternative implementations of this interface. By using a factory method [5],

```
public class List
{
    private /*@ spec_public @*/ int elementCount;

    //@ requires elementCount > 0;
    public Object removeFirst() {
        // implementation here ...
    }

    // ...
}
```

Figure 2: A `List` class, abbreviated for simplicity.

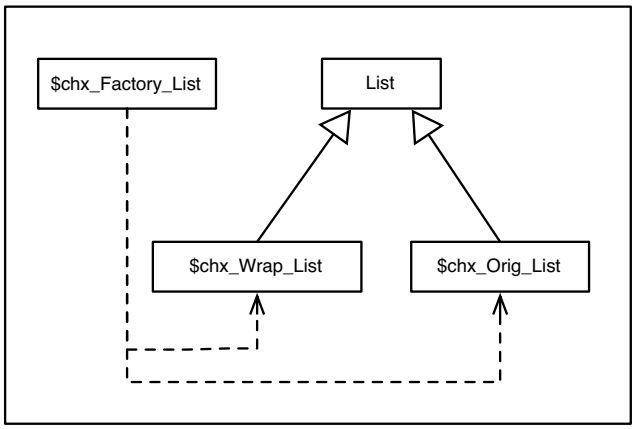

Figure 3: The original list class will be transformed into four components.

the decision of which implementation object to instantiate at any given point can be separated from the object requesting the instance. Using a factory shifts control over which instance to create into another component. With the right support, this also allows users to enable or disable assertion checks on a per class basis, at run-time.

The implementation details are probably best illustrated with an example. Figure 2, shows a snippet of code for a simple `List` class. For this paper, we are not interested in how the methods (such as `removeFirst`) are implemented, so we do not show implementation code. What we are interested in is how to separate the assertions (as illustrated by the `requires` clause) from the implementation code.

As shown in Figure 3, the wrapper-based design involves automatically generating four different class files from the source shown in Figure 2:

- The original class that contains the actual implementation.
- The wrapper class that contains the assertion checks.
- An interface that both the original and wrapper classes implement.
- A factory class that is called to create an instance of `List`.

```
public interface List
{
    // ...
    public int $chx_get_elementCount();

    public void $chx_set_elementCount(int count);

    public Object removeFirst();
}
```

Figure 4: Instead of a class, there is now a `List` interface that both the wrapper and nonwrapper classes implement.

```
public class Wrapper
{
    public Wrappable wrappedObject = null;
    public static CheckingPrefs isEnabled = null;
}

public class $chx_Wrap_List extends    Wrapper
                            implements List
{
   // ...
   public int $chx_get_elementCount() {
      return wrappedObject.elementCount();
   }

   public Object removeFirst() {

      // ...

      if ( isEnabled.precondition() ) {
         // checkPre performs the actual
         //precondition check.
         checkPre$RemoveFirst$List();
      }

      return (($chx_Orig_List)wrappedObject)
                  .removeFirst();
   }
}
```

Figure 5: The `Wrapper` class, from which all wrappers inherit, and the generated wrapper class for `List`.

Both the wrapper and the original class perform the same essential operations—both export the same (behavioral) interface. In our design, we make this explicit by making `List` an interface and having both the wrapper class and the original class implement it. Figure 4 shows a snippet of this automatically generated `List` interface. The `List` interface redeclares the public methods of the original class. Also, accessor method declarations for public fields are added so that the fields accessible in the original class are also accessible through the interface.

The wrapper class "wraps" or decorates an instance of the original component, but adds checking code before and after every method. To do this, every wrapper component has a `wrappedObject` field to hold a reference to the wrapped instance of the original component. This is achieved by having every wrapper component be a subclass of `Wrapper`. Then, each method defined in the original component is also defined in the wrapper, where it is implemented by performing any pre-state checks, delegating to the wrapped object for the core behavior, and then performing any post-state checks.

Figure 5 shows the `Wrapper` class from which all wrappers inherit. It contains just two fields, `wrappedObject` and `isEnabled`. The `isEnabled` member can be queried to determine whether or not particular checks should be performed at run-time. Both members are initialized by the factory method that creates instances of the corresponding wrapper class.

Figure 5 also illustrates the basic structure of the wrapper

```
public class $chx_Statics_List
{
   public static CheckingPrefs isEnabled = null;
   public static List newObject() {
      List      result    = new $chx_Orig_List();
      Wrappable wrappable = (Wrappable)result;
      if ( isEnabled != null && isEnabled.wrap() )
      {
         result = new
             $chx_Wrap_List( result, isEnabled );
      }
      wrappable.$chx_this = result;
      return result;
  }
}
```

Figure 6: A factory is used to create `List` instances.

class for `List`; it shows the output of our wrapper generator tool for JML, simplified for brevity. Before every assertion check is performed, the `isEnabled` field is queried by calling the appropriate method. In Figure 5 for example, `isEnabled.precondition()` is tested and the precondition is checked only if the method returns `true`. An exception is thrown if an executed check fails.

The factory method that is invoked to create new `List`s is shown in Figure 6. For every constructor in the original class, there is a corresponding factory method in the factory class. If every `List` object is instantiated using the factory instead of a call to `new`, we can transfer the decision of whether to create a wrapped or unwrapped version of the object to a separate component. In this case, the factory method queries the static member `isEnabled` to decide whether to return a wrapped or unwrapped object.

So far, our discussion has dealt with transforming only one class. So how is inheritance addressed in this technique? For example, if `List` inherits from `Bag`, what relationship(s) exist between the generated wrapper classes, generated interfaces, and original classes? The solution is straightforward: let each of `List`'s interface, wrapper, and original class extend the corresponding interface or class from `Bag`. That is, the `List` wrapper extends the `Bag` wrapper, the `List` interface extends the `Bag` interface, and the original `List` class extends the original `Bag` class. Figure 7 illustrates this idea.

At the highest level of the inheritance hierarchy, instead of the wrapper class inheriting directly from `Object`, the wrapper class inherits from `Wrapper`. Similarly, the non-wrapper class inherits from `Wrappable`. A practical implication of this design is that if a class has JML specifications, all of its superclasses must be transformed by our tool regardless of whether they have specs or not. This process can be handled automatically by extending the JML tool set. Even if source files for the superclass(es) without specifications are unavailable, we can obtain the needed signature of the class through reflection or inspection of the bytecode.

3. ELIMINATING THE CHANGES NEEDED IN CLIENT CODE

Placing assertion checks in wrappers provides several advantages: assertion checks can be selectively included or excluded without requiring recompilation, and when they are

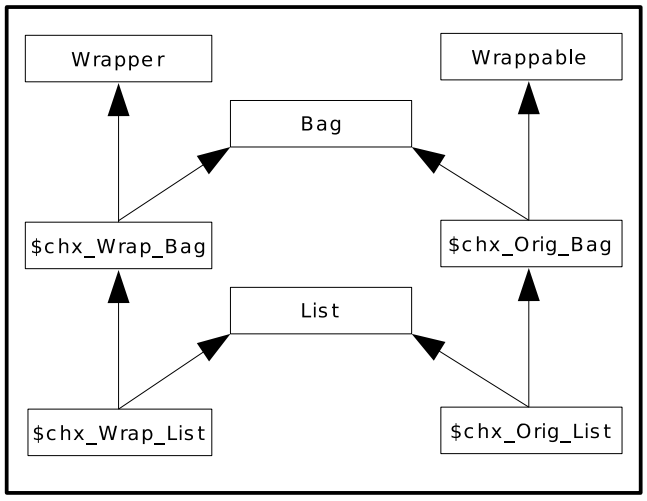

Figure 7: Dealing with inheritance: `List` inherits from `Bag`.

excluded there is no additional overhead imposed. On the other hand, the design presented in Section 2 requires some changes to basic coding practices in both the client code and in component being checked.

3.1 Changes to Client Code

The changes needed in client code fall into three areas: object creation, data member access, and static method invocation. Rather than calling `new` to create a new object, client code must now call the corresponding factory method. For example:

```
List l = new List();
```

now should be phrased this way

```
List l = $chx_Statics_List.newObject();
```

In addition, the way public fields are accessed changes. Current Java design practices discourage the use of publicly accessible data members. However, for code that violates such practices, within this framework there can never be direct access to such fields. Instead, automatically generated accessor methods must be used:

```
a = l.length;
```

now should be phrased this way

```
a = l.$chx_get_length();
```

Finally, static method calls to the class being checked must also be transformed. The `$chx_Statics_List` class generated by the tool set will also contain a dispatch stub for each static method in the class being wrapped. The stub will, depending on whether or not wrapper usage is enabled, forward the call to a corresponding static method in either the wrapper or in the underlying class.

```
public class $chx_Orig_List extends Wrappable
{
    private int elementCount;

    public int $chx_get_elementCount() {
        return elementCount;
    }

    // ...

    public Object removeFirst() {
        // implementation here ...
    }

    // ...
}
```

Figure 8: The original class is modified.

3.2 Changes to the Original Class

The original class also needs modification to work within this framework. First, since we've appropriated the name `List` for the interface, we rename the original `List` class to `$chx_Orig_List`. Second, private methods within the class are promoted to package level access so that the wrapper can have access to these methods (to adding checks to them). Third, accessor methods for all data members must be generated. Figure 8 illustrates these modifications on the original `List` class.

One problem with this approach is that when the unwrapped object calls another method of its own, that call will not go through the wrapper so its behavior will not be checked. That is, if `removeFirst` calls a method belonging to `this`, the method must be checked for contract violations as well. The solution is for the original class to inherit from an object called `Wrappable`, which contains one field: `$chx_this`. The `$chx_this` field is a reference to either the associated wrapper if the non-wrapper object is contained inside one, or to `this` if it is not wrapped. Each time the original class invokes one of its own methods, instead of using `this` (e.g. `this.m()`), the modified version of the original class uses `$chx_this` (e.g. `$chx_this.m()`).

A related problem is calling non-public methods. If a method has a call to some private method `this.p()`, the call would be translated to `$chx_this.p()` where `$chx_this` might be a wrapper object. However, to perform the actual computation, the wrapper object must also be able to access the wrapped object's original method. Thus, private methods must be elevated to at least package level access. A similar problem exists for protected methods, but they must be promoted to public access, since superclasses may not be in the same package as the subclass. This means that certain access control violations may not be caught at runtime. Violations should still be detectable at the compile phase, however.

3.3 Removing the Need for Source-Level Modifications

All of these modifications, both to the client code and to the component being checked, add clutter and complexity. Further, if we wish for wrapper-based objects to be used by

existing code, perhaps code distributed in binary-only form, then how can we impose stylistic modification requirements on that code? Finally, changes to both the client code and the component code to adopt this framework will necessarily impose additional overhead, even when check execution is disabled.

To address these concerns, we are designing a custom class loader for the JML tool set. Rather than requiring the client code and the underlying component to have modifications transformed into them at compile time, instead the class loader can dynamically transform the original bytecode sequences at load time if the wrapper framework is being used.

In essence, the JML compiler generates bytcode for the original `List` class in the file `List.class`, just as if no assertions were being used. The bytecode for the three other wrapper support classes are generated in `*.chx` files. One can run the resulting Java program normally using the `java` command, which has the effect of completely ignoring all of the wrapper-related files and running the original unmodified bytecode. Alternatively, by adding the assertion checking class loader at the start of the command line, the necessary modifications to both client and component code are made on-the-fly at load time. This class loader knows about the special file name extension used by the wrapper support classes so that it can detect and load wrapper-enabled classes differently than code without JML assertions. This approach allows the wrappers to be distributed in binary form along with the original component, but still maintain zero additional overhead when wrappers are not being used.

We have designed such a class loader and are in the process of implementing it. After exploring the critical security issues, it also appears possible to use this technique to retroactively add wrapper-based assertion checking features even within protected packages, such as `java.util`.

4. RUN-TIME CONTROL OF ASSERTION BEHAVIOR

As with most competing techniques, JML allows embedded assertion checks to be enabled or disabled when the code is run. Effectively, JML uses a single global switch for each family of assertions—so postconditions can be disabled independently of preconditions, for example. Some tools provide more fine-grained control. iContract provides a graphical interface for selectively enabling or disabling the generation and inclusion of assertion checks on a per-class basis at build time.

With the wrapper approach, inclusion or exclusion of assertion checks is deferred until load time. As a result, it would be more preferable to allow fine-grained control over which wrapper-enabled classes should use wrappers, as well as which families of assertion checks should actually be executed at program startup or even during run-time. We have devised an approach that allows this control at the individual class level, as well as at the Java package level.

As discussed in Section 2, every wrapper is given a reference to a `CheckingPrefs` object called `isEnabled`. Rather than using a single global object for this purpose, there is one such `CheckingPrefs` object for every wrappable class. Further, there is a similar object for each Java package, and the package-level and class-level preferences objects are linked together into a tree structure that mirrors the Java package nesting structure. In this tree, the preferences objects for individual classes are the leaves. The custom class loader takes care of incrementally constructing this tree as classes are loaded. Note that when an individual wrapper checks a preference setting via `isEnabled`, it retrieves the setting directly from that object and no tree traversal is needed.

Using a graphic control panel at program startup, one can use a collapsible/expandable tree view to set preferences about what whether or not wrappers should be used on wrappable classes, and if wrappers are used, which assertions should be executed and which should be skipped. Tree nodes in this graphical view map directly to the tree-structured network of preferences objects. If one changes an option, that change is stored in the corresponding preferences object and also propagated down through its children all the way to the leaves. Thus, tree traversal happens when the user makes an option setting, rather than when settings are looked up inside each assertion test.

In addition to making such changes at program startup, the same control panel could also be used to modify preferences during run-time by executing it in a separate thread. Further, preference settings could also be saved to or read from properties files. The result is a flexible approach to fine-grained control of assertion checking options that scales well.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have outlined a strategy for extending JML assertion checking using wrappers. Using wrappers allows checking code to be distributed in compiled form alongside the original code, eliminates the associated overhead of checking code when it is unused, and allows run-time control over contract checking on a per-class basis.

This approach is not without challenges, however. Two of the more troublesome are dealing with inline assertion checks and dealing with `super` calls. The wrapper approach deals only with assertions that can be checked before and after a method is called. Assertions within methods need to be handled in a different manner. One possible solution is to place additional functionality into the custom classloader such that it can inject the appropriate assertion checks into the bytecode when the class is being loaded.

The problem with `super` calls is that when within a component it calls a method of its superclass, there is not an easy way to call the superclass of the wrapper component instead, and so in our current design these calls go unchecked. JML currently solves a similar problem by renaming methods and using reflection to call the superclass methods. The disadvantage of this approach is the high performance overhead of reflection in Java.

We are in the process of completing the three parts needed to make this framework viable. First, the JML compiler has been extended to take in JML-annotated Java source code and produce the four corresponding files described in Section 2. The original class's bytecode should be the same as if it were compiled with `javac`. Second, the custom classloader for incorporating load-time modifications to client and component code has been designed and must be completed. Third, a controller that allows users to manage assertion execution preferences in a convenient way has been prototyped and must be completed.

Acknowlegements

We gratefully acknowledge the financial support from the National Science Foundation under the grant CCR-0113181. Any opinions, conclustions or recommendations expressed in this paper do not necessarily reflect the views of the NSF.

6. REFERENCES

[1] Y. Cheon. A runtime assertion checker for the java modeling language. Technical Report 03–09, Department of Computer Science, Iowa State University, April 2003.

[2] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. H. Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03–10, Department of Computer Science, Iowa State University, March 2003.

[3] S. H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Software Testing, Verification & Reliability*, 10(4):249–262, 2000.

[4] S. H. Edwards. Making the case for assertion checking wrappers. In *Proceedings of the RESOLVE Workshop 2002*, pages 95–104. Dept. of Computer Science, Virginia Tech, 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[7] J. M. Voas. Quality time: How assertions can increase test effectiveness. *IEEE Software*, 14(2):118–119, 1997.