Re-Engineering the Java Buffer Library with OpenJava

Michael Jarrett

December 11, 2003

Abstract

The Java New I/O APIs includes a powerful library for creating a variety of memory buffers optimized for input and output. Due to internal platform-dependency issues and heavy optimization, there are 80 classes that make up the library, despite there only being 14 such classes that are visible to the user. There is heavy code duplication in this library that makes it difficult to maintain.

OpenJava is a Java preprocessor that operates on the premise of ‘compile-time reflection’, where during processing the compiler can read (and modify) the classes being generated using an interface similar to that of the Java reflection package. The user specifies for each class a ‘metaclass’, which represents a Java class in OpenJava, and is capable of performing translations on the class using the reflective-style interfaces, or for Java source code, may also manipulate an abstract syntax tree.

Two types of metaclass were created to attempt to reduce the amount of duplicated code in the Java buffer library. The first was a metaclass that, when asked to translate its class, generates the read-only subclass of that class. The second was a metaclass that, when run on a buffer designed to store one primitive type, would generate buffer classes for the other primitive types, with the option to run translations of a ’chained’ metaclass. The result was the reduction of source-code lines by 50%, and number of classes by 56%, a significant improvement, but less than text-based approaches. However, this also reduced the code’s maintainability due to the dependencies created between the metaclass and the source classes. It is recommended that more generic metaclasses that implement syntax extensions are used to reduce the separation of dependent code and increase metaclass reuse, and that effort be extended to improving the OpenJava compiler.

1 Introduction

Metaprogramming is the term used to refer to a program that operates on a program. These programs are actually very commonplace; the classic example is a compiler, which translates source code into an executable binary file. Other metaprograms exist though, and increasingly popular is the idea of a metaprogram that translates between two human-readable source code formats. These can be used in many places, for example where the object language (the language of the output of a metaprogram generator) is inefficient for expressing a desired concept, or where it is necessary to repeat constructs or patterns in many places. Properly used, this style of metaprogramming can increase writeability, readability, and maintainability of project source.

As a demonstration, the Java buffer library included as part of the Java 1.4 SDK was re-engineered, using a tool called OpenJava. The goal was to reduce classes and lines of code in the source, and through this, increase the maintainability of the library.

2 Related Work

A case study[1] examined the use of the XML-based Variant Configuration Language (XVCL)[2] in the generation of the Java NIO buffer library included in Sun’s Java SDK 1.4.1. XVCL uses a technique referred to as frames. Their approach divided the classes into three levels - a level for the basic data types, a level for endianness and byte-ordering issues, and a level for read-only/read-write generation. The result was an implementation reducing the lines of code by 72%.

Work[3] by the authors of OpenJava explored the use of OpenJava to make cleaner and less error-prone the process of implementing design patterns. The *adapter* and *visitor* design patterns were implemented with the assistance of OpenJava metaclasses.

OpenJava itself was the offspring of a similar project known as OpenC++[4]. The project was originally written by the same author that now writes OpenJava, but has the advantage of a more popular community following.

The Polyglot[5] project is a Java preprocessor operating in the same way as OpenJava, but concentrating almost exclusively on the concept of language extensions by the translation of abstract syntax trees. While not providing the reflective interface to code that is the specialty of OpenJava, it has much stronger support and control of syntax extensions. Over a dozen practical language extensions have been successfully implemented.

3 Java Buffer Library

The "Java Buffer Library" refers to the classes within the package *java.nio*, within the Java class libraries. These classes are a part of Java New I/O APIs[6], introduced into the Java 1.4 specification by Java Specification Request 51 (JSR-51)[7], entitled "New I/O APIs for the Java^{TM} Platform." These buffer classes fulfill one of JSR-51's goals, namely, "An API for fast buffered binary I/O, including the ability to map files into memory when that is supported by the underlying platform."[7]

These new buffer libraries were designed to facilitate efficient manipulation of data in the form of Java's primitive data-types, with a focus on cases where such data would eventually be read or written by a data stream. This makes the Java buffer library ideal for implementing stream protocols over sockets, or reading and writing binary data in a file.

At the heart of the buffer library is the class *Buffer*, an abstract class representing a buffer. This class introduces several properties to the concept of a Java buffer.

- Capacity is a buffer's maximum size. This is fixed at buffer creation time and cannot change.
- Limit refers to the extent of where data may be read or written, and may be arbitrarily set between 0 and capacity.
- Position determines where relative reads and writes will retrieve or write data in the buffer, and cannot be larger than limit.
- Mark acts as a single user-defined bookmark within the buffer, and must not be greater than position.

Below this is a subclass for each one of the primitive types, *byte, char, short, int, float, long, and double.* The byte version of the buffer is special, having the ability to be allocated 'directly'. These direct buffers are given memory outside the Java garbage-collected heap. Since they are not garbage collected, they do not move around in memory and therefore can be read or written by the underlying platform directly. Each type of buffer also has the ability to allocate itself on the Java heap. All buffers may also choose to become a 'view' of an array or another buffer, meaning that the buffer will become a wrapper to read or write the underlying data structure. A special variant of a byte buffer, called a mapped byte buffer, is able to represent a file memory-mapped using the underlying platform's capabilities.

This description of buffers lends itself well to modeling as a system family. Figure 1 shows the static features of a buffer, excluding the universally-provided runtime features described above.

While representing the user's view of buffer features, figure 1 does not convey much of the complexity that is needed to internally represent these buffers. Most of the instantiatiable

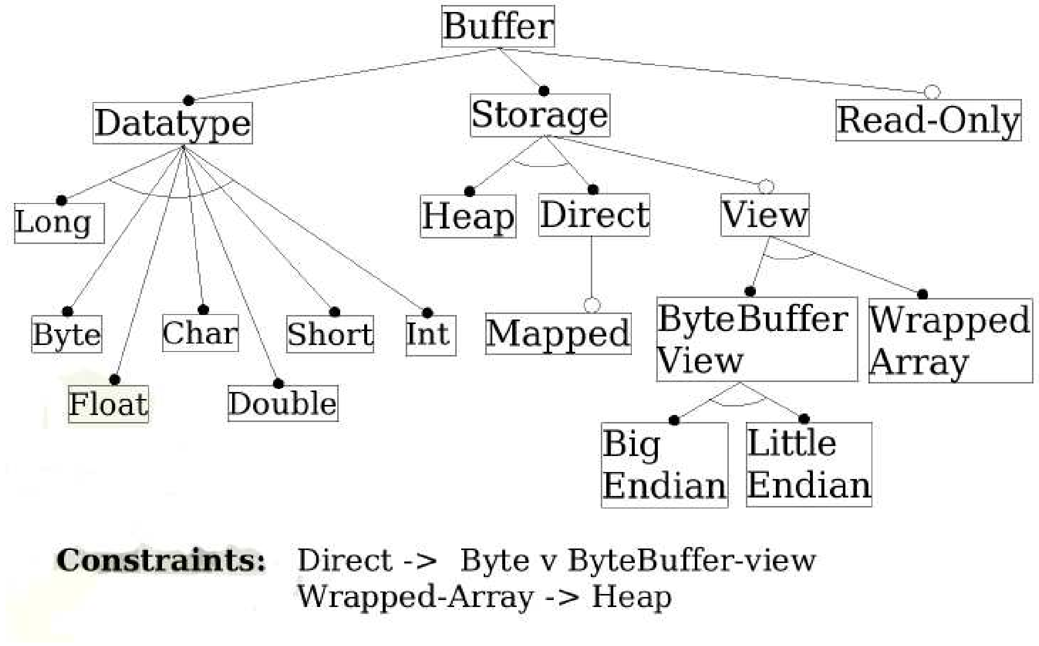

Figure 1: Feature Diagram of the Java Buffer Library

implementation classes are package-protected and handle the internals of actually manipulating the data structures that back the buffers. Each combination of attributes, and each variant in internal implementation of these attributes, each leads to a new class.

- Direct buffers, where the underlying buffer is correctly aligned for the platform becomes `ByteBufferAs[type]Buffer[B/L]` classes, where [type] is a primitive type other than *byte*. The B and L variants represent buffers where the buffer's byte order is big endian or little endian respectively.

- Direct buffers which are not correctly aligned in memory become `Direct[type]Buffer[S/U]`, where [type] is a primitive type (including *byte*, but without the S/U designator since a byte buffer cannot be made to not be aligned). U and S refer to buffers where the byte order of the platform matches that of the buffer, or not, respectively.

- Heap-allocated buffers or buffers made as a view of an array on the heap become `Heap[type]Buffer` classes, where [type] is a primitive type, including *byte*.

Each of these classes has a subclass suffixed with the letter R, which causes each of the buffers' mutator methods to throw a *ReadOnlyException*. This allows the use of the Java virtual method system to throw a read-only exception for read-only buffers without taking the runtime penalty of a flag test inside every function call.

With the exception of helper and exception classes, this leads to a total of 9 public classes, backed by 64 package classes. A summary of this is shown in figure 2, where the red line represents the division between public abstract classes and package-internal implementation classes.

The Java source for java.nio as produced by Sun Microsystems for Java 1.4.2 was investigated. There is a total of 36018 lines of Java source, however, many fragments are repeated in several places with trivial changes, and several files have large numbers of blank lines.

There are signs that much of this code may be auto-generated, since comments within the source files seem to indicate that a generic text-based templating solution was used. Unfortunately, no documentation was locatable describing exactly how the source files were generated internally at Sun. Since we do not have access to the code that generated these classes, the generated classes are treated as if they are the 'original' source code.

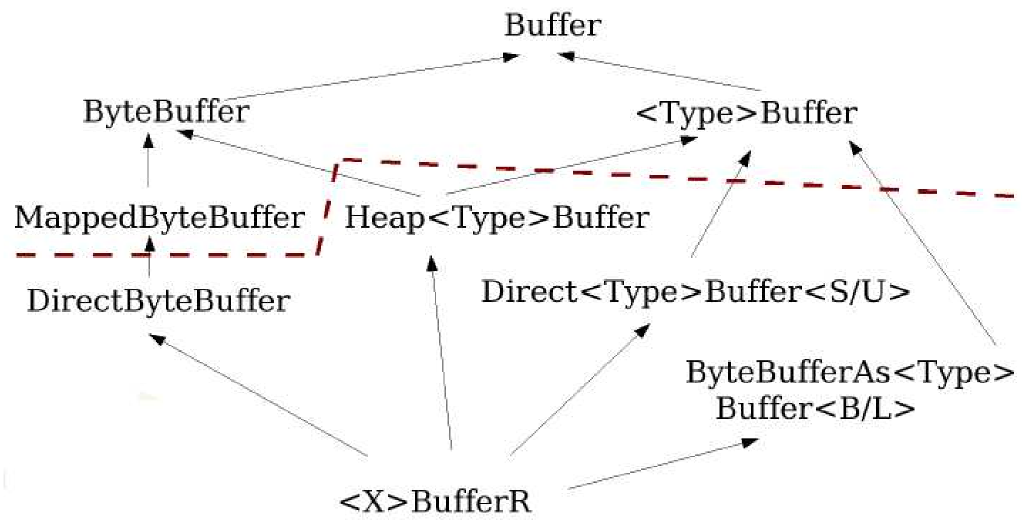

Figure 2: Java Buffer Library Classes

4 OpenJava

OpenJava[8] is a preprocessor for Java. It takes as input a set of source files, in a format that is based on Java with an extended syntax or modifications specified by exactly one 'meta-class' per loaded source file.

In metaprogramming terminology, OpenJava has the following properties:

- **Static:** All code is generated at compile time. initial papers hinted at the possibility of using the intersession and metaclass extension capabilities at runtime, recent papers and the implementation have stuck strictly to compile-time translation.

- **Generator:** Generates code as output at compile-time. Though it is still quite possible and even convenient due to the reflection interface to perform analysis on the code as well, there is no way to suppress code generation.

- **Two-stage:** Since metaclasses are written in Java, it is possible to generate these with OpenJava as well. However, there is no built-in support for automatically performing multiple stages of translation, so for all practical purposes, it is two-stage. Even the meta-code for the final stage needs to be compiled separately.

- **AST/reflection representation:** There is a dual representation of code in OpenJava. Backing every input file is an abstract syntax tree representing this code. Every class, included loaded classes, are represented by a reflection-style interface. This interface is the aspect that differentiates OpenJava from other Java translating solutions.

- **Untyped:** The meta-program is not type-safe in respect to generated code, and the code output is not type-checked until compiled by the Java compiler.

- **Manual Staging:** Meta-code and object code are physically separated into different files. Any code generated within the meta-code is explicitly manipulated as data.

It is unclear as to whether or not OpenJava is homogeneous. While both the meta-language and source language is Java, metaclasses may specify extended syntax, meaning source classes as input may not be pure Java. The lack of multi-stage support and strong typing eliminates many of the advantages of a homogeneous language; however, one does gain the benefit of only needing to understand Java syntax and whatever syntax extensions one chooses to introduce.

4.1 Language Representation

OpenJava actually has two language representations that work in parallel. The top layer of representation is a 'compile-time reflection' interface, which operates almost identically to Java's reflection interfaces. The key difference between OpenJava's reflection and Java's built-in reflection is that the instances of these reflective objects have mutators, allowing one to modify properties in a way that's guaranteed to generate correct Java code. At runtime, this is often referred to as 'intersession'. For example, there is simple API calls to change a method's access level, or to change the type of a field to represent a new class (represented by an instance of a metaclass).

Each reflective object is backed by some underlying information source, which can determine how much information is available and whether or not that information can be modified. Input files are backed by an abstract syntax tree. This allows the class, as well as the bodies of methods to be manipulated, either directly, or through special visitor classes. Compiled classes are backed by a Java *Class* reflection object. Classes loaded in this way cannot be modified. The system is (internally in OpenJava, not in metaclasses) easily extensible, allowing for the possibility of other backing information sources.

4.2 Syntax Extensions

A metaclass is allowed to extend the syntax parsed within object code in a limited way. It can introduce two concepts: keywords, and modifiers. A keyword is very generic, and can appear just about anywhere; it is up to the metaclass's callee-side translation to make sense of the keyword and enforce where it can be used. Modifiers can only be used in places where modifiers are traditionally used in Java declarations (fields, methods, classes, etc).

Both new keywords and new modifiers are able to specify a 'suffix', which represents a set of rules for building an AST underneath the keyword. The parser uses it to build the AST under the keyword, but otherwise does not attempt to interpret it in any way. It is assumed that the metaclass will translate the subtrees generated by the syntax extension and translate it into valid Java syntax. If the metaclass does not translate the syntax, it is stripped from the generated output.

The ability to extend syntax is rather limited, and it is difficult (or even impossible) to make certain forms of extension where type resolution is affected. These are likely more implementation issues than a fundamental design issue.

4.3 Translation Process

The OpenJava preprocessor parses the input files into abstract syntax trees (ASTs) and then creates an instance of each class's metaclass based on the AST. Any other classes that are depended on by the input classes are also loaded, either with source, or by loading the compiled classfile. An instance of the appropriate metaclass for each of these loaded classes is created, or if the metaclass is unknown, a generic metaclass is loaded.

After loading and parsing, OpenJava begins the "callee translation" phase. The method `translateDefinition` is called on the instance of the metaclass corresponding to each input file, giving the metaclass for each class a chance to perform translations.

Once each class has been given a chance to translate itself, the "caller translation" phase is entered. In this phase, each input is scanned for references to other classes. The metaclass for the referenced class is given an opportunity to translate the reference in the input class. Each kind of reference to a class that is valid in Java is allowed to be translated, and is generally forgiving of type resolution errors. However, syntax extensions cannot be defined on the reference.

Once all translations are complete, OpenJava writes Java[1] code out for each input class. Also, any new classes that were generated during translation are also written out, though the metaclasses for the generated classes (and the classes they refer to) are not given a chance

[1]In theory, this should be correct correct Java code, however it is possible for a metaclass to construct code that does not compile correctly. It is unknown if this is by design, or a bug in OpenJava's APIs.

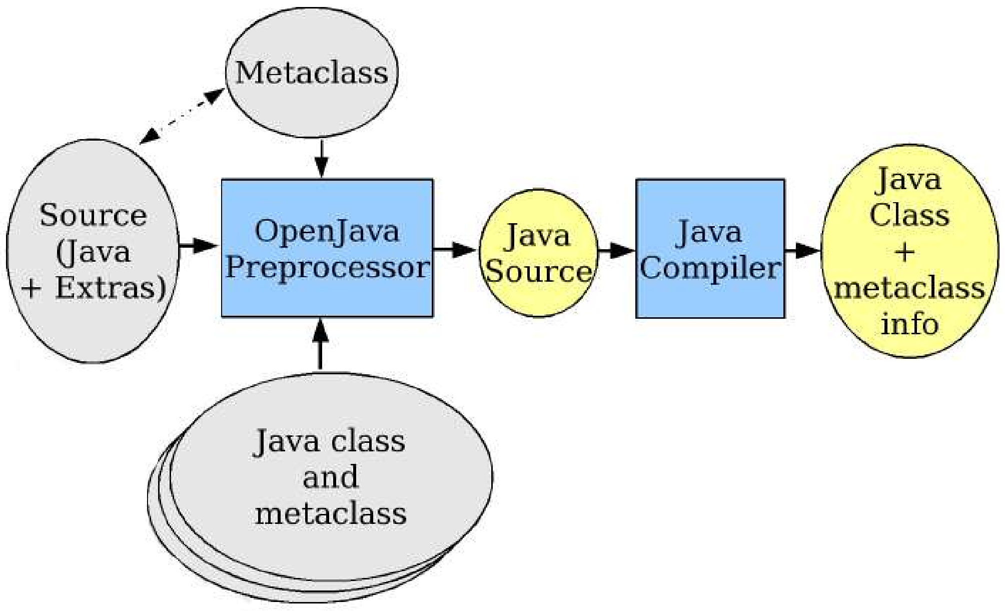

Figure 3: OpenJava Process

to translate them. Finally, OpenJava invokes the system's Java compiler, which compiles the translated Java code into class files. This process is outlines in figure 3

One other method of translation available to metaclasses is overriding the compile-time reflection interfaces themselves. By doing this, other classes that use the reflective interface to access the class can be given false information. However, it is unclear as to whether or not this ability would actually be useful in practice.

5 Applying OpenJava to the Java Buffer Library

The Java buffer library has a large volume of duplicated code. One study[1] found that 72% of the code in the Java buffer library was redundant, in that it recurred in identical or slightly modified form multiple times in the source. Such duplication negatively affects maintainability, especially in code as sensitive as a core Java library.

Observations showed that a large amount of code duplication was occurring due to the exponential number of classes introduced with the number of features. In fact, many of these features resulted in a set of almost identical subclasses being generated inheriting from each previously-existing class. This implies that these features could be generated by a metaclass, which could effectively reduce the amount of code that needs to be maintained. By generating features, it also collects in one place all changes related to that feature.

5.1 Restrictions

For the generated buffer to be useful, it must be binary-compatible with existing Java classes that use it. This has many implications. We cannot change any public interfaces in any way, nor make even subtle changes in how the code behaves functionally. This also implies that we cannot rely on code being recompiled at all, and certainly not with the OpenJava compiler. *This means that we cannot use caller-side translations that will affect the users of the buffer library.*

We also require that non-functional aspects of the code not be seriously affected. The most critical aspect of this is performance; the generated buffer library should not perform significantly worse than the original buffer library.

While not necessary, in the interest of ensuring compatibility without an extensive test suite, most of the code is designed to generate code that is equivalent to the original buffer code.

5.2 Pre-OpenJava Translations

Before working with the OpenJava compiler itself, several translations needed to be performed on the Java code before OpenJava would even accept it.

- OpenJava did not correctly handle classes that were not public, causing problems for the rather extensive set of package-protected classes. All classes had to therefore be made public.
- OpenJava did not understand the Java 1.4 assert statement, and refused to pass it on to the Java compiler. These statements had to be stripped from the source. Asserts are only enabled with a special virtual machine flag, so this is not a critical loss of functionality.
- The packages had to all be changed from java.nio to javam.nio. Java.nio is a system classpath, which meant most Java-based tools would eagerly try to load nio classes from this source instead of relying on generated classes.

The first two of these aspects are the result of bugs in the OpenJava compiler[2], and the third a result of the necessity for testing purposes.

While all three should have been capable of being handled by metaclasses, other bugs in OpenJava prevented this from happening. In particular, a metaclass was actually written to strip assert statements, but in certain cases, it would attempt to resolve the assert method on visiting the AST node for it, which would fail since there is no such method. The code for this class is included in appendix B. The other two aspects would be a perfect place to apply OpenJava, in fact it is where OpenJava's compile-time reflection interface shines through best, but both were thwarted by (different) bugs which affected the filename, directory, and class name of object source files.

To resolve these problems, a common UNIX stream parsing tool known as sed was used. This tool can do simple per-line textual substitutions. A single line of sed commands was all that was needed to perform all the required changes. The full source generation script is included in appendix C.

5.3 Read-Only Subclass

As shown in figure 2, the leaf of each class inheritance hierarchy is a 'read-only' subclass of each buffer type. As a subclass, the only methods it needs to override are methods for creating views (which need to return read-only buffers), and methods for mutating the buffer (which need to throw exceptions). Due to the strict 1-to-1 relationship between a buffer class and its read-only subclass, it would be ideal to have the metaclass for the parent class generate the 'read-only subclass'.

For this purpose, four metaclasses were written. "ReadOnlySubclass" is the base class for the remaining three; it generates the subclass itself, and adds several methods common to all variants of read-only subclasses, then calls a method "customReadOnlyMethods" to add type-specific methods (blank in this case).

Each of the three subclasses, handling the 'ByteAsX', 'HeapX' and 'DirectX' buffers respectively, implement `customReadOnlyMethods`, and add features specific to the read-only classes for those types of buffers. Several aspects of these different buffer types required custom handling on some methods. The most notable are the constructors, which due to the different data structures backing the buffers, varied wildly between heap-allocated, direct, and view buffers.

Each buffer class was instructed to instantiate the correct variant of ReadOnlySubclass, and then the read-only subclasses were removed from the source. Table 1 shows the result of this

[2]The first one was actually partially corrected later by the developer, after I provided details on the problem and instructions to correct it.

change. We have successfully reduced our lines of source by 29.1%, and reduced the number of classes by 26. However, these numbers do not accurately reflect the amount of code reduced, as the original Java buffer library had large amounts of blank space between methods. When looking at the word count (as generated by the UNIX tool `wc`), there is a reduction of only 8.5%. While we successfully reduced the number of classes, most of our line count gains seem to be in the clearing of white space.

Table 1: Source Before and After Applying ReadOnlySubclass

	Original	Read-Only
Lines of source	35844	24916
Lines of metacode	0	506
Words of source	64262	57331
Words of metacode	0	1485
Source classes	80	50
Metaclasses	0	4

Reductions in lines of code do not necessarily equal increase in maintainability. In this case, since the entirety of read-only functionality is contained within these subclasses, maintainability is increased. One downside is that the method bodies of the read-only subclasses are built using abstract syntax trees, which while reasonable for the very short code fragments found in read-only subclasses, are not easy in general to maintain. The APIs that are publicly accessible are fixed, so it would take a major overhaul of the buffer libraries as a whole before the read-only subclass generators needed any review. This implies that the complexity created by the read-only subclasses has been successfully factored out, and therefore that maintainability should be significantly improved.

The two places where maintainability has been hampered has been the fact that the constructors and mutators are hard-coded into the read-only subclass. A simple syntax extension in the base class allowing these methods to be explicitly flagged would have eliminated this, but would fail where mutators are inherited from the superclass of the buffer class.

5.4 Type Generators

Once the read-only generators were complete, a more ambitious class generation was attempted. In this case, it was decided that the greatest benefit would be attained if the data-type feature was generated, as it had the largest branching factor of any of the features.

It was decided that the best place for this generation interface was at height two in the inheritance tree, which is the level of classes directly inheriting the public buffer interfaces. The public buffer classes themselves were not generated since they were mostly JavaDoc comment with only minimal implementation; OpenJava provides no methods for generating or manipulating comments, so significant usability would be sacrificed by the loss of the API JavaDoc comments.

Several problems manifested themselves here that were not present in the read-only subclass case.

- Unlike read-only subclasses, some of these generated methods are relatively complex, and would be unreasonable to write ASTs for.
- There is no base class which could serve as a base for generation of data-type-specific subclasses in the class hierarchy.
- There is a level of generation that is specific to each storage type for the direct and byte-view buffer classes, to handle alignment and endianess cases. These are drastically different for each type of buffer, so must be handled separately.
- OpenJava doesn't apply metaclass translations to generated classes, which means that read-only subclass behaviour must be applied manually to each generated class.

It was decided to use a prototype approach. Two metaclasses were written, one heap buffers, and one for direct buffers. One buffer data-type for each variation of these buffers was chosen, and told to instantiate the appropriate metaclass. These metaclasses inherited from the appropriate read-only metaclass, allowing the read-only subclass to be generated for the prototype classes manually during the generation process.

These type-generating metaclasses would then generate a new class for each type, using the correct read-only subclass as the metaclass. The methods and fields of the prototype are copied to the new class, and then modified to correct type, datatype sizes, and other information. The entire problem of generating methods by AST is avoided by copying the prototype's version of the method and building on it. Finally, each generated class is told to translate itself, explicitly giving the read-only subclasses a chance to perform its callee-side translations.

The following classes were used as prototypes.

- DirectCharBufferS
- DirectCharBufferU
- HeapCharBuffer

The byte-view buffers were not templated in the interest of time constraints, and the extra complexity in converting many of the data-types across endianness changes. In general, there was more variation across data-types in the byte-view buffers than the other two types.

Table 2 shows the results after applying type generators, on top of the previous read-only subclass results. A reduction of 25% in source lines was achieved, and a reduction of 13 classes. It would be expected that a significant improvement on these numbers would be achieved if the byte-view buffers were included.

Table 2: Source Before and After Applying Type Generators

	Read-Only	Type-Generated
Lines of source	24916	17846
Lines of metacode	506	1139
Words of source	57331	48761
Words of metacode	1485	2992
Source classes	50	35
Metaclasses	4	6

Again, we must look at how the generation affects maintainability. In this case, unlike the read-only classes, the changes are much more fragile - since we depend on a prototype, any change to the prototype must be accompanied with a review to see the effects on the generated classes. In particular, assumptions regarding whether a method is local to the type prototyped, or meant to be propagated to generated classes are hidden in the metaclass. Here, the separation of the metacode from the object code has artificially separated tightly-related pieces of code, harming maintainability.

Many of the methods used to manipulate methods were also very fragile, for example, occurrences of binary shifts are assumed to be shifting by the size of the datatype, meaning shifts used in other ways would break. Even the choice of methods to modify involves matching on parameters to determine what translations to apply.

6 Conclusion

These results demonstrate that it is in fact quite possible to use OpenJava to generate large amounts of Java code in the Java buffer library. The total lines of code reduction is somewhat less than the previous effort using frames, but close enough to believe that further work could approach the results obtained in that project.

As to maintainability, as used here, maintainability did not improve a great deal. This is due to the separation of generation classes from the object code, which in practice were strongly related to each other. While this worked well for the read-only subclasses where the links between the two are very simple, the deficiencies in this approach shine through in the complex manipulations performed in the type-generator classes.

The use of a prototype as a means of generating classes proved to be a workable solution, but extremely ineffective due to the lack of annotations as to the relationship between prototype and other generated classes.

The lack of strong quoting or typing also hindered the use of OpenJava in this context, making it hard to add in new code fragments in a metaclass. Also, the lack of support for complex interactions of multiple metaclasses, especially in the case of generated classes, further adds to these dependency problems.

OpenJava has potential for being a strong tool for generation of code in an example like the Java buffer library, but needs several concept improvements, and drastic improvements in the prototype implementation before this can be realized.

7 Future Direction

The approach used to generate the Java buffer library could be improved by making metaclasses more generic. Using the syntax extension abilities of a metaclass, some of the decisions within the metaclasses could be delegated to annotations in the object code itself, making the semantics of the generation more explicit within the object code and leaving less coupling between the internals of the metaclass and the input files. While this improvement would result in more work in writing the metaclass, the result would likely eliminate many of the maintainability concerns encountered in this implementation.

One of the problems in using syntax extensions was that these abilities in OpenJava are not well-defined and unpredictable, and work should be done to make these more powerful and easier to use.

As an extreme version of this, entirely generic metaclasses could be used in a variety of contexts, allowing the programmer to introduce additional language constructs where they were convenient. An example of a generic metaclass that allows the duplication of methods for primitive types is included in appendix B, and was not used only due to the many problems with aggressive name resolution. For these generic classes to be effective, OpenJava needs better support for applying the effects of multiple metaclasses to a class.

The ability to apply the combined transformations of multiple metaclasses could significantly increase the utility of OpenJava. Either an approach within OpenJava itself to combine the properties of multiple metaclasses, or a simple metaclass that can load several 'transformation objects' could be used here. An example of the latter approach is shown in appendix B.

Support for generated classes in OpenJava needs to be greatly improved, most notably, each generated class should be translated according to its metaclass automatically, alternating between caller and callee translations until no more classes are generated. Generation of code in general should be an explicitly-controlled process, allowing classes to be created, removed, or generation prevented completely.

References

[1] "Case Study: eliminating redundant code in the Buffer library," *XVCL - Technology for Reuse,* http://fxvcl.sourceforge.net/buffer/index.htm (current Dec. 11, 2003).

[2] "XML-based Variant Configuration Language," *XVCL - Technology for Reuse,* http://fxvcl.sourceforge.net/ (current Dec. 11, 2003).

[3] M. Tatsubori and S. Chiba, "Programming Support of Design Patterns with Compile-time Reflection," Proc. OOPSLA'98 Workshop on Reflective Programming in C++ and Java, pp. 56-60, Vancouver, BC, 1998.

[4] “Welcome to OpenC++,” *OpenC++,* http://opencxx.sourceforge.net/index.shtml (current Dec. 11, 2003).

[5] N. Nystrom, M. Clarkson and A. Myers, “Polyglot: An Extensible Compiler Framework for Java,” Proc. 12th Intl. Conf. on Compiler Construction, April 2003.

[6] “New I/O APIs,” *Java Technology,* http://java.sun.com/j2se/1.4.2/docs/guide/nio/ (current Dec. 11, 2003).

[7] *Java Specification Request JSR-51, New I/O APIs for the Java^{TM} Platform,* Java Community Process, 2002.

[8] M. Tatsubori, “Welcome to OpenJava,” *OpenJava: An Extensible Java,* http://www.csg.is.titech.ac.jp/ mich/English/openjava/ (current Dec. 11, 2003).

A Metaclasses

```
package javam.nio.meta;
import openjava.mop.*;
import openjava.mop.edit.OJEditableClass;
import openjava.ptree.*;

/**
 * Generates an OJClass for a read-only buffer.
 * Subclasses of this fill in specific methods, and commonality is handled here.
 */
public class ReadOnlySubclass
    instantiates Metaclass
    extends OJClass
{
    /**
     * Visitor for subSequence since ojc doesn't like anonymous classes.
     */
    private class SubSequenceVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        OJClass myCl;

        public SubSequenceVisitor(Environment e, OJClass cl)
        {
            super(e);
            myCl= cl;
        }

        public TypeName evaluateDown(TypeName p)
        {
            if(p.getName().equals(getName()))
            {
                return new TypeName(myCl.getName());
            }
            return p;
        }
    }


    protected void customReadOnlyMethods(OJEditableClass cl)
        throws CannotAlterException, OJClassNotFoundException
    {
        // Subclasses should override this
    }

    protected void setupReadOnlyMethods(OJEditableClass cl)
        throws MOPException
    {
        // The original implementation declared:
        //   - order
        //   - isDirect
        // Neither is really needed. So we rely on inheritance to get them.
        // There may be a virtual function speed penalty, but this is Java,
        // if you can't take that, GO HOME!
        OJMethod m;
        StatementList sl;

        // First, determine the "public" buffer
        // In all these classes, it's just the read-only class's grandparent.
        OJClass pubBufClass= getSuperclass();

        // All read-only subclasses have the same asReadOnly, return duplicate();
        sl= new StatementList(new ReturnStatement(
                                  new MethodCall("duplicate", null)));
       m= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                        pubBufClass, "asReadOnlyBuffer", null, null, null, sl);
        cl.addMethod(m);

        // All read-only subclasses have the same isReadOnly(), return true;
        sl= new StatementList(new ReturnStatement(Literal.constantTrue()));
       m= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                        OJSystem.BOOLEAN, "isReadOnly", null, null,
                        null, sl);
        cl.addMethod(m);

        // Common exception setup for each method
        // TODO: makes package reference. Won't work in default package!
        sl= new StatementList(new ThrowStatement(new AllocationExpression(
                               OJClass.forName(getPackage()
                                + ".ReadOnlyBufferException"),
                               null)));

        // Perform same op for each "put" method in direct superclass,
        // plus "compact"
        // All read-only subclasses have public pubbuftype compact()
       m= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                        pubBufClass, "compact", null, null, null, sl);
        cl.addMethod(m);

        OJMethod[] protos= getDeclaredMethods();
        for(int i= 0; i < protos.length; i++)
        {
            if(!protos[i].getName().equals("put"))
                continue;
            OJMethod newmeth= OJMethod.makePrototype(protos[i]);
            newmeth.setBody((StatementList)sl.makeRecursiveCopy());
            cl.addMethod(newmeth);
        }


        // Special cases for the read-only char cases
        // toString can be inherited, but we need to change the returned type
        // on subSequence.
        // Since rewriting the method would suck, we just edit the old version :)
        if(pubBufClass.getSimpleName().startsWith("Char"))
        {
            OJMethod sample= getDeclaredMethod("subSequence",
                new OJClass[] {OJSystem.INT, OJSystem.INT} );

            OJMethod newMethod= new OJMethod(cl, sample.getModifiers(),
                sample.getReturnType(), sample.getName(),
                sample.getParameterTypes(), sample.getParameters(),
                sample.getExceptionTypes(),
                (StatementList)sample.getBody().makeRecursiveCopy());
            try{
                 newMethod.getBody().accept(new
                                  SubSequenceVisitor(getEnvironment(), cl));
            }catch(ParseTreeException e) { throw new MOPException(e.toString()); }

            cl.addMethod(newMethod);
        }

        customReadOnlyMethods(cl);
    }

    protected OJClass generateReadOnlyClass()
        throws MOPException
    {
        // Build the class name. If lower case, appends an R.
        // If the last letter is caps, it puts it before the last char.
        // Important to use getSimpleName to avoid full package in name.
        String className= getSimpleName();
        String lastChar= className.substring(className.length() - 1);
        if(lastChar.toLowerCase().equals(lastChar))
            className= className + "R";
        else
            className= className.substring(0, className.length() - 1) + "R"
                          + lastChar;

        MemberDeclarationList mdl= new MemberDeclarationList();

        // Class has to be public, or else we get an OJ_Unknown0.java
        // This is fine since there's no metainfo class, but it's a
        // bit counterintuitive, and we don't really care enough to protect it.
        // *
        // Some of the reader classes implement DirectBuffer.
        // Since their parent classes also implement DirectBuffer, we really
        // should not have to care about this. But if we do, I'll hack it in later.
        ClassDeclaration cd= new ClassDeclaration(
                                new ModifierList(ModifierList.PUBLIC), className,
                                new TypeName[] {new TypeName(getName())},
                                null, mdl);

        ClassDeclarationList cdl= new ClassDeclarationList(cd);
        CompilationUnit cu= new CompilationUnit(getPackage(), null, cdl);
        FileEnvironment e= new FileEnvironment(OJSystem.env, cu, className);

        // There's a bug that 'added' classes are not as rigorously checked
        // for package location. So we write to the package dir, and hope
        // to hell we are there.
        //e.setPackage(null);
        OJEditableClass c= new OJEditableClass(new OJClass(e, null, cd));

        setupReadOnlyMethods(c);

        return c;
    }

    public void translateDefinition()
        throws MOPException
    {
        OJSystem.addNewClass(generateReadOnlyClass());
    }


    /**
     * Hack to remove methods.
     */
    public void myRemoveMethod(OJMethod m)
        throws CannotAlterException
    {
        removeMethod(m);
    }

    /**
     * Hack to remove constructors.
     */
    public void myRemoveConstructor(OJConstructor c)
        throws CannotAlterException
    {
        removeConstructor(c);
    }

    /**
     * Hack to add constructors.
     */
    public void myAddConstructor(OJConstructor c)
        throws CannotAlterException
    {
        addConstructor(c);
    }

    /**
     * Hack to add methods.
     */
    public void myAddMethod(OJMethod m)
        throws CannotAlterException
    {
        addMethod(m);
    }
}
```

```
package javam.nio.meta;
```

```
import openjava.mop.*;
import openjava.ptree.*;
import openjava.mop.edit.OJEditableClass;


/**
 * ReadOnlySubclass for HeapXBuffer classes.
 */
public class HeapReadOnlySubclass
    instantiates Metaclass
    extends ReadOnlySubclass
{
    protected void customReadOnlyMethods(OJEditableClass cl)
        throws OJClassNotFoundException, CannotAlterException
    {
        // Find obj size
        String baseArray;
        String baseType= getSimpleName().substring(4);
        if(baseType.startsWith("Long"))
            baseArray= "long[]";
        else if(baseType.startsWith("Double"))
            baseArray= "double[]";
        else if(baseType.startsWith("Int"))
            baseArray= "int[]";
        else if(baseType.startsWith("Float"))
            baseArray= "float[]";
        else if(baseType.startsWith("Short"))
            baseArray= "short[]";
        else if(baseType.startsWith("Char"))
            baseArray= "char[]";
        else
            throw new OJClassNotFoundException("Don't_recognize_type_"
                                               + baseType + "!");

        // Add constructor
        OJClass[] params= {OJSystem.INT, OJSystem.INT
        };
        String[] paramNames= {
            "cap", "lim"
        };

        ExpressionList exprs= new ExpressionList();
        exprs.add(new Variable("cap"));
        exprs.add(new Variable("lim"));
        ConstructorInvocation ci= new ConstructorInvocation(exprs, null);
        StatementList sl= new StatementList(
            new ExpressionStatement(
                new AssignmentExpression(new Variable("isReadOnly"),
                AssignmentExpression.EQUALS,
                Literal.constantTrue()))
        );
        OJConstructor cons= new OJConstructor(cl, OJModifier.constantEmpty(),
            params, paramNames, null, ci, sl);
        cl.addConstructor(cons);

        // Constructor 2
        params= new OJClass[] {OJClass.forName(baseArray),
                               OJSystem.INT, OJSystem.INT};
        paramNames= new String[] {"buf", "off", "len"};
        exprs= new ExpressionList(new Variable("buf"), new Variable("off"),
                                  new Variable("len"));
        ci= new ConstructorInvocation(exprs, null);
        cons= new OJConstructor(cl, OJModifier.constantEmpty(),
            params, paramNames, null, ci, (StatementList)sl.makeRecursiveCopy());
        cl.addConstructor(cons);

        // Constructor 3
        params= new OJClass[] {OJClass.forName(baseArray), OJSystem.INT,
                               OJSystem.INT, OJSystem.INT, OJSystem.INT,
                               OJSystem.INT};
        paramNames= new String[] {"buf", "mark", "pos", "lim", "cap", "off"};
        exprs= new ExpressionList();
        exprs.add(new Variable("buf"));
        exprs.add(new Variable("mark"));
        exprs.add(new Variable("pos"));
        exprs.add(new Variable("lim"));
        exprs.add(new Variable("cap"));
        exprs.add(new Variable("off"));
        ci= new ConstructorInvocation(exprs, null);
        cons= new OJConstructor(cl, OJModifier.forModifier(OJModifier.PROTECTED),
                                params, paramNames, null, ci,
                                (StatementList)sl.makeRecursiveCopy());
        cl.addConstructor(cons);

        // Add slice
        ExpressionList el= new ExpressionList();
        el.add(new Variable("hb"));
        el.add(Literal.makeLiteral(-1));
        el.add(Literal.makeLiteral(0));
        el.add(new MethodCall("remaining", null));
        el.add(new MethodCall("remaining", null));
        el.add(new BinaryExpression(new MethodCall("position", null),
                                    BinaryExpression.PLUS,
                                    new Variable("offset")));

        sl= new StatementList(new ReturnStatement(
                              new AllocationExpression(cl, el)));
        OJMethod meth= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                                    getSuperclass(), "slice", null, null, null, sl);
        cl.addMethod(meth);


        // Add duplicate method
        el= new ExpressionList();
        el.add(new Variable("hb"));
        el.add(new MethodCall("markValue", null));
        el.add(new MethodCall("position", null));
        el.add(new MethodCall("limit", null));
        el.add(new MethodCall("capacity", null));
        el.add(new Variable("offset"));
        sl= new StatementList(new ReturnStatement(
            new AllocationExpression(cl, el)
        ));
        meth= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                             getSuperclass(), "duplicate", null, null, null, sl);
        cl.addMethod(meth);
    }
}
```

```
package javam.nio.meta;

import openjava.mop.*;
import openjava.ptree.*;
import openjava.mop.edit.OJEditableClass;


/**
 * A version of ReadOnlySubclass for DirectXBufferRY generation.
 */
public class DirectReadOnlySubclass
    instantiates Metaclass
    extends ReadOnlySubclass
{
    protected void customReadOnlyMethods(OJEditableClass cl)
        throws OJClassNotFoundException, CannotAlterException
    {
        // Find obj size
        int logSize= -1;
        String baseType= getSimpleName().substring(6);
        if(baseType.startsWith("Long") || baseType.startsWith("Double"))
            logSize= 3;
        else if(baseType.startsWith("Int") || baseType.startsWith("Float"))
            logSize= 2;
        else if(baseType.startsWith("Short") || baseType.startsWith("Char"))
            logSize= 1;
        else if(baseType.startsWith("Foom"))
            logSize=2;
        else
            throw new OJClassNotFoundException("Don't␣recognize␣type␣"
                                            + baseType + "!");

        // Add constructor
        OJClass[] params= {
            OJClass.forName("sun.nio.ch.DirectBuffer"),
            OJSystem.INT, OJSystem.INT, OJSystem.INT, OJSystem.INT, OJSystem.INT
        };
        String[] paramNames= {
            "db", "mark", "pos", "lim", "cap", "off"
        };

        ExpressionList exprs= new ExpressionList();
        exprs.add(new Variable("db"));
        exprs.add(new Variable("mark"));
        exprs.add(new Variable("pos"));
        exprs.add(new Variable("lim"));
        exprs.add(new Variable("cap"));
        exprs.add(new Variable("off"));
        ConstructorInvocation ci= new ConstructorInvocation(exprs,
            /*SelfAccess.constantSuper()*/null);
        OJConstructor cons= new OJConstructor(cl, OJModifier.constantEmpty(),
            params, paramNames, null, ci, null);
        cl.addConstructor(cons);


        // Add slice
        ExpressionList el= new ExpressionList();
        el.add(SelfAccess.constantThis());
        el.add(Literal.makeLiteral(-1));
        el.add(Literal.makeLiteral(0));
        el.add(new Variable("rem"));
        el.add(new Variable("rem"));
        el.add(new BinaryExpression(new Variable("pos"), BinaryExpression.SHIFT_L,
                                    Literal.makeLiteral(logSize)));

        StatementList sl= new StatementList();
        sl.add(new VariableDeclaration(new TypeName("int"), "pos",
                                       new MethodCall("position", null)));
        sl.add(new VariableDeclaration(new TypeName("int"), "lim",
                                       new MethodCall("limit", null)));
        sl.add(new VariableDeclaration(new TypeName("int"), "rem",
            new ConditionalExpression(
                new BinaryExpression(new Variable("pos"),
                                     BinaryExpression.LESSEQUAL,
                                     new Variable("lim")),
                new BinaryExpression(new Variable("lim"),
                                     BinaryExpression.MINUS,
                                     new Variable("pos")),
                Literal.makeLiteral(0))
        ));
        sl.add(new ReturnStatement(new AllocationExpression(cl, el)));

        OJMethod meth= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                                    getSuperclass(), "slice", null, null, null, sl);
        cl.addMethod(meth);


        // Add duplicate method
        el= new ExpressionList();
        el.add(SelfAccess.constantThis());
        el.add(new MethodCall("markValue", null));
        el.add(new MethodCall("position", null));
        el.add(new MethodCall("limit", null));
        el.add(new MethodCall("capacity", null));
        el.add(Literal.makeLiteral(0));
        sl= new StatementList(new ReturnStatement(
            new AllocationExpression(cl, el)
        ));
        meth= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                           getSuperclass(), "duplicate", null, null, null, sl);
        cl.addMethod(meth);
    }
}
```

```
package javam.nio.meta;

import openjava.mop.*;
import openjava.ptree.*;
import openjava.mop.edit.OJEditableClass;


/**
 * ReadOnlySubclass for ByteBufferAsXBufferY classes.
 */
public class ByteAsReadOnlySubclass
    instantiates Metaclass
    extends ReadOnlySubclass
{
    protected void customReadOnlyMethods(OJEditableClass cl)
        throws OJClassNotFoundException, CannotAlterException
    {
        // Find obj size
        int logSize= -1;
        String baseType= getSimpleName().substring(12);
        if(baseType.startsWith("Long") || baseType.startsWith("Double"))
            logSize= 3;
        else if(baseType.startsWith("Int") || baseType.startsWith("Float"))
            logSize= 2;
        else if(baseType.startsWith("Short") || baseType.startsWith("Char"))
            logSize= 1;
        else
            throw new OJClassNotFoundException("Don't␣recognize␣type␣"
                                               + baseType + "!");

        // Add constructor
        OJClass[] params= {
            OJClass.forName(getPackage() + ".ByteBuffer"),
            OJSystem.INT, OJSystem.INT, OJSystem.INT, OJSystem.INT, OJSystem.INT
        };
        String[] paramNames= {
            "bb", "mark", "pos", "lim", "cap", "off"
        };

        ExpressionList exprs= new ExpressionList();
        exprs.add(new Variable("bb"));
        exprs.add(new Variable("mark"));
        exprs.add(new Variable("pos"));
        exprs.add(new Variable("lim"));
        exprs.add(new Variable("cap"));
        exprs.add(new Variable("off"));
        ConstructorInvocation ci= new ConstructorInvocation(exprs,
            /*SelfAccess.constantSuper()*/null);
        OJConstructor cons= new OJConstructor(cl, OJModifier.constantEmpty(),
            params, paramNames, null, ci, null);
        cl.addConstructor(cons);

        params= new OJClass[] {OJClass.forName(getPackage() + ".ByteBuffer")};
        paramNames= new String[] {"bb"};
        exprs= new ExpressionList(new Variable("bb"));
        ci= new ConstructorInvocation(exprs,
            /*SelfAccess.constantSuper()*/null);
        cons= new OJConstructor(cl, OJModifier.constantEmpty(),
            params, paramNames, null, ci, null);
        cl.addConstructor(cons);

        // Add slice
        ExpressionList el= new ExpressionList();
        el.add(new Variable("bb"));
        el.add(Literal.makeLiteral(-1));
        el.add(Literal.makeLiteral(0));
        el.add(new Variable("rem"));
        el.add(new Variable("rem"));
        el.add(new BinaryExpression(new BinaryExpression(new Variable("pos"),
                                          BinaryExpression.SHIFT_L,
                                          Literal.makeLiteral(logSize)),
                                       BinaryExpression.PLUS, new Variable("offset")));

        StatementList sl= new StatementList();
        sl.add(new VariableDeclaration(new TypeName("int"), "pos",
                                       new MethodCall("position", null)));
        sl.add(new VariableDeclaration(new TypeName("int"), "lim",
                                       new MethodCall("limit", null)));
        sl.add(new VariableDeclaration(new TypeName("int"), "rem",
            new ConditionalExpression(
                new BinaryExpression(new Variable("pos"),
                                     BinaryExpression.LESSEQUAL,
                                     new Variable("lim")),
                new BinaryExpression(new Variable("lim"),
                                     BinaryExpression.MINUS,
                                     new Variable("pos")),
                Literal.makeLiteral(0))
        ));
        sl.add(new ReturnStatement(new AllocationExpression(cl, el)));

        OJMethod meth= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                                    getSuperclass(), "slice", null, null, null, sl);
        cl.addMethod(meth);


        // Add duplicate method
        el= new ExpressionList();
        el.add(new Variable("bb"));
        el.add(new MethodCall("markValue", null));
        el.add(new MethodCall("position", null));
        el.add(new MethodCall("limit", null));
        el.add(new MethodCall("capacity", null));
        el.add(new Variable("offset"));
        sl= new StatementList(new ReturnStatement(
            new AllocationExpression(cl, el)
        ));
        meth= new OJMethod(cl, OJModifier.forModifier(OJModifier.PUBLIC),
                           getSuperclass(), "duplicate", null, null, null, sl);
        cl.addMethod(meth);
    }
}
```

```
package javam.nio.meta;

import openjava.mop.*;
import openjava.ptree.*;
import openjava.mop.edit.OJEditableClass;

/**
 * Generates (in theory) all the direct classes.
 * One class takes the burden of being the 'main' class,
 * and all others are derived from information therein.
 *
 * This extends DirectReadOnlySubclass so we don't lose
 * the translation from that.
 */
public class DirectGenerator
    instantiates Metaclass
    extends DirectReadOnlySubclass
{
    /**
     * A visitor to change the log-size of each bitshift.
     */
    public class ShiftVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        private int logSize;

        public ShiftVisitor(int logSize_, Environment env)
        {
            super(env);
            logSize= logSize_;
        }

        public Expression evaluateUp(BinaryExpression b)
        {
            if(b.getOperator() == BinaryExpression.SHIFT_L)
                b.setRight(Literal.makeLiteral(logSize));
            return b;
        }
    }

    /**
     * A visitor to change the old types, as well as unsafe.
     */
    public class ReturnVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        private DirectType type;
        private String typeSrc, typeDest, typeRSrc, typeRDest;
        private OJClass unsafeType;

        public ReturnVisitor(DirectType type_, String letter, Environment env)
        {
            super(env);
            type= type_;
            typeSrc= getPackage() + ".Direct" + baseType.name + "Buffer" + letter;
            typeRSrc= "Direct" + baseType.name + "BufferR" + letter; // Gen'd no pkg
            typeDest= getPackage() + ".Direct" + type.name + "Buffer" + letter;
            typeRDest= getPackage()+".Direct" + type.name + "BufferR" + letter;
            try{
            unsafeType= OJClass.forName("Unsafe");
            } catch(OJClassNotFoundException exc) {}
        }

        public TypeName evaluateUp(TypeName n)
        {
            if(n.getName().equals(typeSrc))
                return new TypeName(typeDest);
            else if(n.getName().equals(typeRSrc))
                return new TypeName(typeRDest);
            else return n;
        }

        public Expression evaluateUp(MethodCall c)
        {
            if(c.getName().equals("copyTo" + baseType.name + "Array"))
                c.setName(c.getName().replaceFirst(baseType.name, type.copyType));
            else if(!(c.getReferenceExpr() instanceof Variable))
                ;
            else if(((Variable)c.getReferenceExpr()).equals("unsafe"))
                c.setName(c.getName().replaceFirst(baseType.name, type.name));
            return c;
        }
    }



    /**
     * Fixes FP-type code.
     */
    public class FixGetVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        private DirectType type;
        public FixGetVisitor(DirectType type_, Environment e)
        {
            super(e);
            type= type_;
        }

        public Expression evaluateUp(MethodCall c)
        {

            if(!(c.getReferenceExpr() instanceof Variable))
                ;
            else if(((Variable)c.getReferenceExpr()).equals("unsafe"))
                c.setName(c.getName().replaceFirst(type.name, type.copyType));
            return c;
        }

        public Statement evaluateUp(ReturnStatement r)
        {
            Expression old= r.getExpression();
            try{
            r.setExpression(new MethodCall(OJClass.forName("java.lang."+type.name),
                type.copyType.toLowerCase() + "BitsTo" + type.name,
                new ExpressionList(old)
            )); }catch(Exception e) { e.printStackTrace(); }
            return r;
        }
    }

    /**
     * Fixes FP-type code.
     */
    public class FixPutVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        private DirectType type;

        public FixPutVisitor(DirectType type_, Environment e)
        {
            super(e);
            type= type_;
        }

        public Expression evaluateUp(MethodCall c)
        {   try{
            if(c.getName().equals("swap"))
            {
                ExpressionList old= c.getArguments();
                MethodCall newCall= new MethodCall(
                    OJClass.forName("java.lang."+type.name),
                    type.name.toLowerCase() + "ToRaw" + type.copyType + "Bits",
                    old);
                c.setArguments(new ExpressionList(newCall));
            }
            else if(!(c.getReferenceExpr() instanceof Variable))
                ;
            else if(((Variable)c.getReferenceExpr()).equals("unsafe"))
                c.setName(c.getName().replaceFirst(type.name, type.copyType));

            }catch(Exception e) {e.printStackTrace();}
            return c;
        }
    }



    /**
     * A structure representing the classes we have to generate.
     */
    private static class DirectType
    {
        public String name;
        public String copyType;
        public int logSize;
        public OJClass primType;

        public DirectType(String name_, int logSize_, OJClass primType_,
                          String copyType_)
        {
            name= name_;
            logSize= logSize_;
            primType= primType_;
            copyType= copyType_;
        }
    }

    // TODO: Generate these from the src
    protected static DirectType[] genTypes= new DirectType[]
    {
        new DirectType("Short", 1, OJSystem.SHORT, "Short"),
        new DirectType("Int", 2, OJSystem.INT, "Int"),
        new DirectType("Float", 2, OJSystem.FLOAT, "Int"),
        new DirectType("Long", 3, OJSystem.LONG, "Long"),
        new DirectType("Double", 3, OJSystem.DOUBLE, "Long")
    };
    protected static DirectType baseType=
        new DirectType("Char", 1, OJSystem.CHAR, "Char");

    protected DirectReadOnlySubclass[] genClasses;


    /**
     * Generate a single generated class.
     */
    public DirectReadOnlySubclass generateTypeClass(DirectType type)
        throws MOPException
    {
        String letter=getSimpleName().substring(getSimpleName().length() - 1);
        ClassDeclaration cd= (ClassDeclaration)getSourceCode().makeRecursiveCopy();
        cd.setName("Direct" + type.name + "Buffer" + letter);
        cd.setMetaclass("DirectReadOnlySubclass");
        cd.setBaseclass(new TypeName(type.name + "Buffer"));
        ClassDeclarationList cdl= new ClassDeclarationList(cd);
        CompilationUnit cu= new CompilationUnit(getPackage(), null, cdl);
        FileEnvironment e= new FileEnvironment(OJSystem.env, cu, cd.getName());
        DirectReadOnlySubclass cl= new DirectReadOnlySubclass(e, null, cd);

        // If non-char, don't have subsequence or toString
        if(type.name != "Char" && baseType.name == "Char")
        {
            cl.myRemoveMethod(cl.getMethod("toString", new OJClass[]
                            {OJSystem.INT, OJSystem.INT}));
            cl.myRemoveMethod(cl.getMethod("subSequence", new OJClass[]
                            {OJSystem.INT, OJSystem.INT}));
        }

        OJClass superclass= cl.getSuperclass();

        // Fix constructors
        OJConstructor[] cons= cl.getDeclaredConstructors();
        for(int i= 0; i < cons.length; i++)
        {
            cons[i].getSourceCode().setName(cl.getSimpleName());
        }

        // Fix methods
        OJMethod[] meths= cl.getDeclaredMethods();
        for(int i= 0; i < meths.length; i++)
        {
            try
            {
                // Fix return types
                if(meths[i].getReturnType() == getSuperclass())
                {
                    meths[i].setReturnType(superclass);
                }
                else if(meths[i].getReturnType() == baseType.primType)
                {
                    meths[i].setReturnType(type.primType);
                }

                // Fix shift sizes
                meths[i].getBody().accept(new ShiftVisitor(type.logSize, e));
                // Fix returns and unsafe.get/put
                meths[i].getBody().accept(new ReturnVisitor(type, letter, e));

                // The S-classes have odd swap semantics for FP types
                if(letter.equals("S") && !type.name.equals(type.copyType))
                {
                    if(meths[i].getName().equals("get")
                       && (meths[i].getParameterTypes()).length < 2)
                    {
                        // unsafe.get<Type> -> unsafe.get<CopyType>
                        // return <Type>.<copyType>BitsTo<Type>(...)
                        meths[i].getBody().accept(new FixGetVisitor(type, e));
                    }
                    else if(meths[i].getName().equals("put"))
                    {
                        // First two only but only these have Bits.swap
                        // inside Bits.swap()
                        // x -> <Type>.<type>ToRaw<CopyType>Bits(x)
                        meths[i].getBody().accept(new FixPutVisitor(type, e));
                    }
                }

                // Special get/put parameters
                if(   meths[i].getName().equals("get")
                   && (meths[i].getParameterTypes()).length > 0
                   && (meths[i].getParameterTypes())[0].isArray())
                {
                    OJClass[] typez= meths[i].getParameterTypes();
                    typez[0]= OJClass.forName(type.name.toLowerCase()+"[]");
                    // Only way to change method params!
                    cl.myRemoveMethod(meths[i]);
                    cl.myAddMethod(new OJMethod(
                        cl, meths[i].getModifiers(), meths[i].getReturnType(),
                        meths[i].getName(), typez, meths[i].getParameters(),
                        meths[i].getExceptionTypes(), meths[i].getBody()
                    ));

                }
                else if(meths[i].getName().equals("put"))
                {
                    OJClass[] typez= meths[i].getParameterTypes();
                    if(typez[0] == getSuperclass())
                        typez[0]= superclass;
                    else if(typez.length > 1 && typez[1] == baseType.primType)
                        typez[1]= type.primType;
                    else if(typez[0] == baseType.primType)
                        typez[0]= type.primType;
                    else if(typez[0].isArray())
                        typez[0]= OJClass.forName(type.name.toLowerCase()+"[]");
                    // Only way to change method params!
                    cl.myRemoveMethod(meths[i]);
                    cl.myAddMethod(new OJMethod(
                        cl, meths[i].getModifiers(), meths[i].getReturnType(),
                        meths[i].getName(), typez, meths[i].getParameters(),
                        meths[i].getExceptionTypes(), meths[i].getBody()
                    ));
                }

            }
            catch(Exception ex) // Stupid ParseTreeException!
            {
                ex.printStackTrace();
            }

        }
```

```
        return cl;
    }


    /**
     * Entry to the wild generation action.
     *
     * We use our prototypes to generate a variety of new classes,
     * and once generated, we tell all of them to translate themselves,
     * finally, we translate ourself.
     */
    public void translateDefinition()
        throws MOPException
    {
        genClasses= new DirectReadOnlySubclass[genTypes.length];

        for(int i= 0; i < genTypes.length; i++)
        {
            genClasses[i]= generateTypeClass(genTypes[i]);
            OJSystem.addNewClass(genClasses[i]);
        }

        // Trigger each class's read-only generation
        for(int i= 0; i < genClasses.length; i++)
            genClasses[i].translateDefinition();

        // Finally, trigger our own read-only generation
        super.translateDefinition();
    }
}
```

```
package javam.nio.meta;

import openjava.mop.*;
import openjava.ptree.*;
import openjava.mop.edit.OJEditableClass;

/**
 * Generates the heap classes.
 * One class takes the burden of being the 'main' class,
 * and all others are derived from information therein.
 *
 * This extends HeapReadOnlySubclass so we don't lose
 * the translation from that.
 */
public class HeapGenerator
    instantiates Metaclass
    extends HeapReadOnlySubclass
{
    /**
     * A visitor to change the old types.
     */
    public class ReturnVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        private HeapType type;
        private String typeSrc, typeDest, typeRSrc, typeRDest;

        public ReturnVisitor(HeapType type_, Environment env)
        {
            super(env);
            type= type_;
            typeSrc= getPackage() + ".Heap" + baseType.name + "Buffer";
            typeRSrc= "Heap" + baseType.name + "BufferR"; // Gen'd no pkg
            typeDest= getPackage() + ".Heap" + type.name + "Buffer";
            typeRDest= getPackage()+".Heap" + type.name + "BufferR";
        }

        public TypeName evaluateUp(TypeName n)
        {
            if(n.getName().equals(typeSrc))
                return new TypeName(typeDest);
            else if(n.getName().equals(typeRSrc))
                return new TypeName(typeRDest);
            else return n;
        }
    }

    public class FixAllocVisitor
        extends openjava.ptree.util.EvaluationShuttle
    {
        String type;

        public FixAllocVisitor(String type_, Environment e)
        {
            super(e);
            type= type_;
        }

        public Expression evaluateUp(ArrayAllocationExpression a)
        {
            a.setTypeName(new TypeName(type.toLowerCase()));
            return a;
        }
    }


    /**
     * A structure representing the classes we have to generate.
     */
    private static class HeapType
    {
        public String name;
        public OJClass primType;

        public HeapType(String name_, OJClass primType_)
        {
            name= name_;
            primType= primType_;
        }
    }

    // TODO: Generate these from the src
    protected static HeapType[] genTypes= new HeapType[]
    {
        new HeapType("Short", OJSystem.SHORT),
        new HeapType("Int", OJSystem.INT),
        new HeapType("Float", OJSystem.FLOAT),
        new HeapType("Long", OJSystem.LONG),
        new HeapType("Double", OJSystem.DOUBLE)
    };
    protected static HeapType baseType=
        new HeapType("Char", OJSystem.CHAR);

    protected HeapReadOnlySubclass[] genClasses;


    /**
     * Generate a single generated class.
     */
    public HeapReadOnlySubclass generateTypeClass(HeapType type)
        throws MOPException
    {
        ClassDeclaration cd= (ClassDeclaration)getSourceCode().makeRecursiveCopy();
        cd.setName("Heap" + type.name + "Buffer");
        cd.setMetaclass("HeapReadOnlySubclass");
        cd.setBaseclass(new TypeName(type.name + "Buffer"));
        ClassDeclarationList cdl= new ClassDeclarationList(cd);
        CompilationUnit cu= new CompilationUnit(getPackage(), null, cdl);
        FileEnvironment e= new FileEnvironment(OJSystem.env, cu, cd.getName());
        HeapReadOnlySubclass cl= new HeapReadOnlySubclass(e, null, cd);

        // If non-char, don't have subsequence or toString
        if(type.name != "Char" && baseType.name == "Char")
        {
            // Unlike the direct buffers, toString here is package-private
            // so we must use getDeclaredMethod
            cl.myRemoveMethod(cl.getDeclaredMethod("toString", new OJClass[]
                            {OJSystem.INT, OJSystem.INT}));
            cl.myRemoveMethod(cl.getMethod("subSequence", new OJClass[]
                            {OJSystem.INT, OJSystem.INT}));
        }

        OJClass superclass= cl.getSuperclass();

        // Fix constructors
        OJConstructor[] cons= cl.getDeclaredConstructors();
        for(int i= 0; i < cons.length; i++)
        {
            try{
            cons[i].getSourceCode().setName(cl.getSimpleName());
            if((cons[i].getParameterTypes())[0].isArray())
            {
                OJClass[] params= cons[i].getParameterTypes();
                params[0]= OJClass.forName(type.name.toLowerCase()+"[]");
                OJConstructor myCons= new OJConstructor(cl,
                    cons[i].getModifiers(), params, cons[i].getParameters(),
                    cons[i].getExceptionTypes(), cons[i].getTransference(),
                    cons[i].getBody());
                cl.myRemoveConstructor(cons[i]);
                cl.myAddConstructor(myCons);
            }
            else
            {
                cons[i].getTransference().accept(new FixAllocVisitor(type.name,
                    cl.getEnvironment()));
            }
            } catch(Exception ex) { ex.printStackTrace(); }
        }

        // Fix field
        OJField[] fieldz= cl.getDeclaredFields();
        for(int i= 0; i < fieldz.length; i++)
        {
            if(fieldz[i].getName().equals("hb"))
            {
                fieldz[i].setType(OJClass.forName(type.name.toLowerCase() + "[]"));
            }
        }

        // Fix methods
        OJMethod[] meths= cl.getDeclaredMethods();
        for(int i= 0; i < meths.length; i++)
        {
            try
            {
                // Fix return types
                if(meths[i].getReturnType() == getSuperclass())
                {
                    meths[i].setReturnType(superclass);
                }
                else if(meths[i].getReturnType() == baseType.primType)
                {
                    meths[i].setReturnType(type.primType);
                }

                // Fix returns and unsafe.get/put
                meths[i].getBody().accept(new ReturnVisitor(type, e));

                // Special get/put parameters
                if(    meths[i].getName().equals("get")
                    && (meths[i].getParameterTypes()).length > 0
                    && (meths[i].getParameterTypes())[0].isArray())
                {
                    OJClass[] typez= meths[i].getParameterTypes();
                    typez[0]= OJClass.forName(type.name.toLowerCase()+"[]");
                    // Only way to change method params!
                    cl.myRemoveMethod(meths[i]);
                    cl.myAddMethod(new OJMethod(
                        cl, meths[i].getModifiers(), meths[i].getReturnType(),
                        meths[i].getName(), typez, meths[i].getParameters(),
                        meths[i].getExceptionTypes(), meths[i].getBody()
                    ));

                }
                else if(meths[i].getName().equals("put"))
                {
                    OJClass[] typez= meths[i].getParameterTypes();
                    if(typez[0] == getSuperclass())
                        typez[0]= superclass;
                    else if(typez.length > 1 && typez[1] == baseType.primType)
                        typez[1]= type.primType;
                    else if(typez[0] == baseType.primType)
                        typez[0]= type.primType;
                    else if(typez[0].isArray())
                        typez[0]= OJClass.forName(type.name.toLowerCase()+"[]");
                    // Only way to change method params!
                    cl.myRemoveMethod(meths[i]);
                    cl.myAddMethod(new OJMethod(
                        cl, meths[i].getModifiers(), meths[i].getReturnType(),
                        meths[i].getName(), typez, meths[i].getParameters(),
                        meths[i].getExceptionTypes(), meths[i].getBody()
                    ));
                }

            }
            catch(Exception ex) // Stupid ParseTreeException!
            {
                ex.printStackTrace();
            }

        }

        return cl;
    }


    /**
     * Entry to the wild generation action.
     *
     * We use our prototypes to generate a variety of new classes,
     * and once generated, we tell all of them to translate themselves,
     * finally, we translate ourself.
     */
    public void translateDefinition()
        throws MOPException
    {
        genClasses= new HeapReadOnlySubclass[genTypes.length];

        for(int i= 0; i < genTypes.length; i++)
        {
            genClasses[i]= generateTypeClass(genTypes[i]);
            OJSystem.addNewClass(genClasses[i]);
        }

        // Trigger each class's read-only generation
        for(int i= 0; i < genClasses.length; i++)
            genClasses[i].translateDefinition();

        // Finally, trigger our own read-only generation
        super.translateDefinition();
    }
}
```

B Generic Metaclasses

```
package ca.uwaterloo.ece750.meta;

import openjava.mop.*;
import openjava.ptree.*;
import openjava.ptree.util.*;

/**
 * A hack to handle the Java 1.4 "assert" keyword.
 * By default, it will strip them from the code, replacing them with ;
 */
public class AssertManager
    instantiates Metaclass
    extends OJClass
{
    /**
     * Does the actual translation work.
     * This is a recursive method. If any class exists that can truly can make
     * this a problem, THEN we'll make it non-recursive
     */
    public void modifyAsserts(OJClass cl)
    {
        OJMethod[] methods= cl.getAllMethods();
        for(int i= 0; i < methods.length; i++)
        {
            try
            {
                StatementList body= methods[i].getBody();
                if(null == body)
                        continue;
                body.accept(new EvaluationShuttle(getEnvironment())
                {
                    // While a bit of a hack, we will override visit
                    // instead of evaluateDown/evaluateUp.
                    // EvaluationShuttle is just used to avoid the
                    // stupid abstract methods.
                    public void visit(MethodCall mc)
                        throws ParseTreeException
                    {
                        // Parent is the ExpressionStatement
                        if(mc.getName().equals("assert"))
                        {
                            ExpressionStatement es= (ExpressionStatement)mc.getParent();
                            es.replace(new EmptyStatement());
                        }
                        // We're an anonymous visitor - who cares if we don't
                        // visit the subtree correctly?
                    }
                });
            }
            catch(CannotAlterException e)
            {
                continue;
            }
            catch(ParseTreeException e)
            {
                e.printStackTrace();
                continue;
            }
            catch(ClassCastException e)
            {
                System.out.println("ERROR! assert outside StatementList!");
            }
        }


        // Translate inner classes
        OJClass inner[]= cl.getAllClasses();
        for(int i= 0; i < inner.length; i++)
            modifyAsserts(inner[i]);
    }

    /**
     * Translates all method calls to handle asserts.
     * This also scans inner classes and translates them too.
     */
    public void translateDefinition()
    {
        modifyAsserts(this);
    }
}
```

```
package ca.uwaterloo.ece750.meta;

import openjava.mop.*;
import openjava.syntax.*;
import openjava.ptree.*;


/**
 * Generates copies of methods for stated primitive types.
 * Each method can specify the <B>templates</B> keyword, and give a
 * comma-separated list of primitive types. The method will be duplicated
 * for each of these types.
 *
 * Furthermore, the following constructs are provided.
 * <UL>
 * <LI>typename - Used as a parameter or return value, is replaced by
 *                the templated type.
 *                The method name will be scanned for TYPENAME Typename and
 *                typename, and swapped as well.
 * </UL>
 */
public class PrimitiveTemplate
```

```
    instantiates Metaclass
    extends OJClass
{
    public static final String TEMPLATES="templates";
    public static final String TYPENAME="typename";

    protected static final class TemplateType
    {
        public String name;
        public String camelName;
        public int bitSize;

        public TemplateType(String name, String camelName, int bitSize)
        {
            this.name= name;
            this.camelName= camelName;
            this.bitSize= bitSize;
        }
    }

    protected static final TemplateType[] templateTypes= new TemplateType[]
        {
            new TemplateType("byte", "Byte", 8),
            new TemplateType("char", "Char", 16),
            new TemplateType("short", "Short", 16),
            new TemplateType("int", "Int", 32),
            new TemplateType("long", "Long", 64),
            new TemplateType("float", "Float", 32),
            new TemplateType("double", "Double", 64)
        };

    protected static final TemplateType findType(String name)
    {
        for(int i= 0; i < templateTypes.length; i++)
            if(templateTypes[i].name.equals(name))
                return templateTypes[i];
        return null;
    }

    /**
     * Clone and translate methods that specify 'templates'.
     */
    public void translateDefinition()
        throws CannotAlterException, OJClassNotFoundException
    {
        OJMethod[] methods= getDeclaredMethods();
        for(int i= 0; i < methods.length; i++)
        {
            ParseTree templates= methods[i].getSuffix(TEMPLATES);
            if(null == templates)
            {
                System.out.println(methods[i].getName() + " is not templated.");
                continue;
            }
            else
            {
                System.out.println(methods[i].getName() + " templated!");
                // The original method must come out!
                OJMethod m= removeMethod(methods[i]);
                ObjectList ol= (ObjectList)templates;
                for(int j= 0; j < ol.size(); j++)
                {
                    TemplateType type= findType(((TypeName)ol.get(j)).getName());
                    if(null == type)
                    {
                        System.out.println("ERROR: Unknown template type!");
                        continue;
                    }
                    addMethod(createTemplatedMethod(m, type));
                }
            }
        }
    }


    /**
     * Translates an individual method to a specific type.
     * This will translate the method name, parameter types, and the return type.
     */
    protected static OJMethod createTemplatedMethod(OJMethod src, TemplateType type)
        throws CannotAlterException, OJClassNotFoundException
    {
        // makePrototype is brutal. It changes the parameter names. That's stupid.
        // So we make a new method. Manually.

        // Get attributes for the method
        String name= src.getName();
        OJClass returnType;
        OJModifier modifiers= src.getModifiers();
        ParameterList parameterList= (ParameterList)src.getSourceCode()
            .getParameters().makeRecursiveCopy(); // Copy to avoid sharing changes
        OJClass[] exceptions= src.getExceptionTypes();
        StatementList body= (StatementList)src.getBody().makeRecursiveCopy();

        // Mutate our attributes
        name= name.replaceAll("Typename", type.camelName);
        name= name.replaceAll("typename", type.name);
        if(src.getSourceCode().getReturnType().getName().equals(TYPENAME))
            returnType= OJClass.forName(type.name);
        else
            returnType= src.getReturnType();
        for(int i= 0; i < parameterList.size(); i++)
        {
            Parameter p= parameterList.get(i);
            if(p.getTypeSpecifier().getName().equals(TYPENAME))
                p.setTypeSpecifier(new TypeName(type.name));
            p.setVariable(p.getVariable().replaceAll("Typename", type.camelName).
                            replaceAll("typename", type.name));
        }
        return new OJMethod(src.getDeclaringClass(), modifiers, returnType, name,
                            parameterList, exceptions, body);
    }


    /**
     * Acknowledge <B>typename</B> and <B>templates</B> as keywords.
     */
    public static boolean isRegisteredKeyword(String keyword)
    {
        if(keyword.equals(TEMPLATES))
            return true;
        else if(keyword.equals(TYPENAME))
            return true;
        else
            return OJClass.isRegisteredKeyword(keyword);
    }


    /**
     * Define the behaviour of TEMPLATES keyword.
     */
    public static SyntaxRule getDeclSuffixRule(String keyword)
    {
        if(keyword.equals(TEMPLATES))
            return new DefaultListRule(new TypeNameRule(), TokenID.COMMA, false);
        return OJClass.getDeclSuffixRule(keyword);
    }
}
```

```
/*
 * CombinedClass.oj
 *
 * Oct 24, 2000  by Michiaki Tatsubori
 * Extended 2003 by Michael Jarrett
 */
package ca.uwaterloo.ece750.meta;


import openjava.mop.*;
import openjava.ptree.*;
import openjava.syntax.*;
import java.util.*;
import java.lang.reflect.*;


/**
 * The metaclass <code>CombinedClass</code> provides a class to be
 * bound to more than one metaclass.
 * <p>
 * An usage are:
 * <pre>
 * class MyPerson instantiates CombinedClass
 *     extends Object implements Runnable
 *     obeys PersistableClass, TransactionableClass
 * {
 *     ...
 * }
 * </pre>
 * <p>
 *
 * @author   Michiaki Tatsubori
 * @version  1.0
 * @see java.lang.OJClass
 */
public class CombinedClass instantiates Metaclass extends OJClass
{
    private static final String KEYWORD_OBEYS = "obeys";

    /**
     * Composing class metaobjects.
     */
    protected OJClass[] substances = null;

    public CombinedClass(Class javaclazz, MetaInfo metainfo) {
        super(javaclazz, metainfo);
        initializeSubstantialMetaclasses(javaclazz, metainfo);
    }

    /**
     * Initializes the substantial class metaobjects, which were specified
     * by the "obeys" phrase in the class declaration.
     */
    private void initializeSubstantialMetaclasses(Class javaclazz,
                                                  MetaInfo metainfo) {
        int cnum = 0;
        while (metainfo.get(KEYWORD_OBEYS + cnum) != null)  ++cnum;
        substances = new OJClass[cnum];
        for (int i = 0; i < substances.length; ++i) {
            String cname = metainfo.get(KEYWORD_OBEYS + i);
            /* Do not use OJClass.forClass() here.  It would changes
             * the bind of the metaclass to the class name.
             * The substantial class metaobjects must be created directly.
             */
            try {
                Class clazz = Class.forName(cname);
                Constructor cons = clazz.getConstructor(new Class[] {
                    Class.class, MetaInfo.class });
                substances[i] = (OJClass) cons.newInstance(new Object[] {
                    javaclazz, metainfo});
            } catch (Exception ex) {
                throw new Error(ex.toString());
            }
        }
    }
    public CombinedClass(Environment env, OJClass declaring,
                         ClassDeclaration cdecl) {
        super(env, declaring, cdecl);
    }

    /**
     * Translates the declaration part of base classes.
     */
    public void translateDefinition() throws MOPException {
        initializeSubstantialMetaclasses();

        System.out.println("The class " + getName() +
                           " obeys the specified substantial metaclass/es :");
        for (int i = 0; i < substances.length; ++i) {
            System.out.println(substances[i].getClass().getName());
            substances[i].translateDefinition();
        }
    }

    /**
     * Initializes the substantial class metaobjects, which are specified
     * by the "obeys" phrase.
     */
    private void initializeSubstantialMetaclasses() throws MOPException {
        String[] metaclazz = getSubstantialMetaclasses();

        for (int i = 0; i < metaclazz.length; ++i) {
            putMetaInfo(KEYWORD_OBEYS + i, metaclazz[i]);
        }

        substances = new OJClass[metaclazz.length];
        for (int i = 0; i < substances.length; ++i) {
            String cname = metaclazz[i];
            /* Do not use OJClass.forParseTree() here.  It would changes
             * the bind of the metaclass to the class name
             * The substantial class metaobjects must be created directly.
             */
            try {
                Class clazz = Class.forName(cname);
                Constructor cons = clazz.getConstructor(new Class[] {
                    Environment.class, OJClass.class,
                    ClassDeclaration.class });
                Environment env = getEnvironment();
                OJClass declaring = getDeclaringClass();
                ClassDeclaration cdecl = getSourceCode();
                substances[i] = (OJClass) cons.newInstance(new Object[] {
                    env, declaring, cdecl});
            } catch (Exception ex) {
                throw new Error(ex.toString());
            }
        }
    }

    private String[] getSubstantialMetaclasses() throws MOPException {
        ObjectList suffix = (ObjectList) getSuffix(KEYWORD_OBEYS);
        String[] result = new String[suffix.size()];
        for (int i = 0; i < result.length; ++i) {
            result[i] = suffix.get(i).toString();
        }
        return result;
    }

    private static Vector modifVec = new Vector();
    private static Hashtable keyTable = new Hashtable();

    /**
     * Metaclasses of the parsing class declaration.
     */
    static Class[] metaclasses = null;

    /* Extends syntax for custom modifiers */
    public static boolean isRegisteredModifier(String keyword) {
        if (metaclasses == null) {
            System.err.println("No consultant for modifiers!");
            return false;
        }

        for (int i = 0; i < metaclasses.length; ++i) {
            Class c = metaclasses[i];
            System.err.println("Consulting " + c +
                               " for the syntax of modifiers");
            try {
                Method m = c.getMethod("isRegisteredModifier",
                                       new Class[] {String.class});
                Boolean result
                    = (Boolean) m.invoke(null, new Object[] {keyword});
                if (result.booleanValue() == true)  return true;
            } catch (Exception ex) {
                System.err.println(ex);
            }
        }

        return false;
    }

    /* Extends syntax for custom suffix phrases */
    public static boolean isRegisteredKeyword(String keyword) {
        if (keyword.equals(KEYWORD_OBEYS))  return true;

        if (metaclasses == null) {
            System.err.println("No consultant for suffix phrases!");
            return false;
        }

        for (int i = 0; i < metaclasses.length; ++i) {
            Class c = metaclasses[i];
            System.err.println("Consulting " + c +
                               " for the syntax of suffix phrases");
            try {
```

```
            Method m = c.getMethod("isRegisteredKeyword",
                                   new Class[] {String.class});
            Boolean result
                = (Boolean) m.invoke(null, new Object[] {keyword});
            if (result.booleanValue() == true)  return true;
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }

    return false;
}

/* Extends syntax for custom suffix phrases */
public static SyntaxRule getDeclSuffixRule(String keyword) {
    if (keyword.equals(KEYWORD_OBEYS)) {
        /* The returned rule object is hacked to call back and register
         * type names them.
         */
        System.err.println("Consultants_dismissed");
        metaclasses = null;
        // Wow is this broken. I'll fix it.
        return new DefaultListRule(new HackedTypeNameRule(), TokenID.COMMA,
                                   false);
    }

    if (metaclasses == null) {
        System.err.println("No_consultant_for_suffix_phrases!");
        return null;
    }

    for (int i = 0; i < metaclasses.length; ++i) {
        Class c = metaclasses[i];
        System.err.println("Consulting_" + c +
                           "_for_the_syntax_of_suffix_phrases");
        try {
            Method m = c.getMethod("getDeclSuffixRule",
                                   new Class[] {String.class});
            SyntaxRule result
                = (SyntaxRule) m.invoke(null, new Object[] {keyword});
            if (result != null)  return result;
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }

    return null;
}

/* -- Caller-side expansions -- */

public Expression expandFieldRead(Environment env, FieldAccess expr) {
    Expression result;
    for (int i = 0; i < substances.length; ++i) {
        result = substances[i].expandFieldRead(env, expr);
        if (result != expr)  return result;
    }
    return super.expandFieldRead(env, expr);
}

public Expression expandFieldWrite(Environment env,
                                   AssignmentExpression expr) {
    Expression result;
    for (int i = 0; i < substances.length; ++i) {
        result = substances[i].expandFieldWrite(env, expr);
        if (result != expr)  return result;
    }
    return super.expandFieldWrite(env, expr);
}

public Expression expandMethodCall(Environment env,
                                   MethodCall expr) {
    Expression result;
    for (int i = 0; i < substances.length; ++i) {
        result = substances[i].expandMethodCall(env, expr);
        if (result != expr)  return result;
    }
    return super.expandMethodCall(env, expr);
}

public TypeName expandTypeName(Environment env, TypeName tname) {
    TypeName result;
    for (int i = 0; i < substances.length; ++i) {
        result = substances[i].expandTypeName(env, tname);
        if (result != tname)  return result;
    }
    return super.expandTypeName(env, tname);
}

public Expression expandAllocation(Environment env,
                                   AllocationExpression expr) {
    Expression result;
    for (int i = 0; i < substances.length; ++i) {
        result = substances[i].expandAllocation(env, expr);
        if (result != expr)  return result;
    }
    return super.expandAllocation(env, expr);
}

public Expression expandArrayAllocation(Environment env,
                                        ArrayAllocationExpression expr) {
    Expression result;
    for (int i = 0; i < substances.length; ++i) {
        result = substances[i].expandArrayAllocation(env, expr);
        if (result != expr)  return result;
    }
    return super.expandArrayAllocation(env, expr);
}
```

```
    public Expression expandArrayAccess(Environment env, ArrayAccess expr) {
        Expression result;
        for (int i = 0; i < substances.length; ++i) {
            result = substances[i].expandArrayAccess(env, expr);
            if (result != expr)  return result;
        }
        return super.expandArrayAccess(env, expr);
    }

    public Expression expandAssignmentExpression(Environment env,
                                                 AssignmentExpression expr) {
        Expression result;
        for (int i = 0; i < substances.length; ++i) {
            result = substances[i].expandAssignmentExpression(env, expr);
            if (result != expr)  return result;
        }
        return super.expandAssignmentExpression(env, expr);
    }

    public Expression expandExpression(Environment env,
                                       Expression expr) {
        Expression result;
        for (int i = 0; i < substances.length; ++i) {
            result = substances[i].expandExpression(env, expr);
            if (result != expr)  return result;
        }
        return super.expandExpression(env, expr);
    }

    public Statement expandVariableDeclaration(Environment env,
                                               VariableDeclaration decl) {
        Statement result;
        for (int i = 0; i < substances.length; ++i) {
            result = substances[i].expandVariableDeclaration(env, decl);
            if (result != decl)  return result;
        }
        return super.expandVariableDeclaration(env, decl);
    }

    public Expression expandCastExpression(Environment env,
                                           CastExpression expr) {
        Expression result;
        for (int i = 0; i < substances.length; ++i) {
            result = substances[i].expandCastExpression(env, expr);
            if (result != expr)  return result;
        }
        return super.expandCastExpression(env, expr);
    }

    public Expression expandCastedExpression(Environment env,
                                             CastExpression expr) {
        Expression result;
        for (int i = 0; i < substances.length; ++i) {
            result = substances[i].expandCastedExpression(env, expr);
            if (result != expr)  return result;
        }
        return super.expandCastedExpression(env, expr);
    }

    /* -- Error handlings -- */

    public OJField resolveException(NoSuchMemberException nmex, String name)
        throws NoSuchMemberException
    {
        for (int i = 0; i < substances.length; ++i) {
            try {
                return substances[i].resolveException(nmex, name);
            } catch (NoSuchMemberException ex) {}
        }
        return super.resolveException(nmex, name);
    }

    public OJMethod resolveException( NoSuchMemberException nmex,
                                      String name, OJClass[] argtypes)
        throws NoSuchMemberException
    {
        for (int i = 0; i < substances.length; ++i) {
            try {
                return substances[i].resolveException(nmex, name, argtypes);
            } catch (NoSuchMemberException ex) {}
        }
        return super.resolveException(nmex, name, argtypes);
    }

}

/* Super breaks if I make this inner class */
class HackedTypeNameRule
    extends TypeNameRule
{
    public TypeName consumeTypeName(TokenSource src)
        throws SyntaxException
    {
        TypeName name= super.consumeTypeName(src);
        String tname= name.getName();
        System.err.println("Found␣a␣consultant␣" + tname);

        Class[] old = CombinedClass.metaclasses;
        if (old == null) {
            CombinedClass.metaclasses = new Class[1];
        } else {
            CombinedClass.metaclasses = new Class[old.length + 1];
            for (int i = 0; i < old.length; ++i) {
                CombinedClass.metaclasses[i] = old[i];
            }
        }
        try {
            Class mc = Class.forName(tname);
            CombinedClass.metaclasses[CombinedClass.metaclasses.length - 1] = mc;
        } catch (ClassNotFoundException ex) {
            System.err.println(ex);
            CombinedClass.metaclasses = old;
        }

        return name;
    }
}
```

C SED Script

```
#!/bin/sh
# Translates files as follows:
#     Makes all package non-abstract classes public
#     Removes all assert statements
#     Changes the package to javam.nio
#     Gives the file a .oj extention

if [ "$1" == "" ]; then
    find . -name "*.java" -exec $0 \{\} \;
    rm *.java
else
    OJNAME=`echo $1 | sed -e "s/.java/.oj/"`
    sed -e "s/package␣java.nio/package␣javam.nio/;/assert/d;s/^class/public␣class/" $1 > $OJNAME
fi
```