Technical Memorandum UCB/ERL M03/44, UC Berkeley, November 24, 2003

Ptolemy II Coding Style

Authors: *Christopher Hylands Brooks, cxh@eecs.berkeley.edu*
Edward A. Lee, eal@eecs.berkeley.edu

1. Motivation

Collaborative software projects benefit when participants read code created by other participants. The objective of a coding style is to reduce the fatigue induced by *unimportant* formatting differences and differences in naming conventions. Although individual programmers will undoubtedly have preferences and habits that differ from the recommendations here, the benefits that flow from following these recommendations far outweigh the inconveniences. Published papers in journals are subject to similar stylistic and layout constraints, so such constraints are not new to the academic community.

Software written by the Ptolemy Project participants follows a variant of this style guide. Although many of these conventions are arbitrary, the resulting consistency makes reading the code much easier, once you get used to the conventions. We recommend that if you extend Ptolemy II in any way, that you follow these conventions. To be included in future versions of Ptolemy II, the code *must* follow the conventions.

A template that follows these rules can be found in `$PTII/doc/coding/templates/JavaTemplate.java`, where `$PTII` is the location of your Ptolemy II installation. In addition, several useful tools are provided in the directories under `$PTII/util/` to help enforce the standards.

- `lisp/ptjavastyle.el` is a lisp module for GNU Emacs that has appropriate indenting rules. This file works well with Emacs under both Unix and Windows.
- `testsuite/jindent` is a shell script that uses Emacs and the above module to properly indent many files at once. This script works best under Unix, but can work under Windows with Cygwin. To see how this script would all the Java files in a directory, run:
  `$PTII/util/testsuite/jindent -q *.java`
  To indent the files and check the changes in to CVS, remove the -q option.

This work is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), the Defense Advanced Research Projects Agency (DARPA), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from the State of California MICRO program, and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

- `testsuite/ptspell` is a shell script that checks Java code and prints out an alphabetical list of unrecognized spellings. It properly handles namesWithEmbeddedCapitalization and has a list of author names. This script works best under Unix. Under Windows, it would require the installation of the `spell` command. To run this script, type
  `$PTII/util/testsuite/ptspell *.java`
- `testsuite/chkjava` is a shell script for checking various other potentially bad things in Java code, such as debugging code, and FIXME's. This script works under both Unix and Windows. To run this script, type:
  `$PTII/util/testsuite/chkjava *.java`

2. Anatomy of a File

A Java file has the structure shown in figure 1. The key points to note about this organization are:

- The file is divided into sections with highly visible delimiters. The sections contain constructors, public variables (including ports and parameters for actor definitions), public methods, protected variables, protected members, private methods, and private variables, in that order. Note in particular that although it is customary in the Java community to list private variables at the beginning of a class definition, we put them at the end. They are not part of the public interface, and thus should not be the first thing you see.
- Within each section, methods appear in alphabetical order, in order to easily search for a particular method (in printouts, for example, finding a method can be very difficult if the order is arbitrary, and use of printouts during design and code reviews is very convenient). If you wish to group methods together, try to name them so that they have a common prefix. Static methods are generally mixed with non-static methods.

The key sections are explained below.

2.1 Copyright

The copyright used in Ptolemy II is shown in figure 2. This style of copyright is often referred to the community as a "BSD" copyright because it was used for the "Berkeley standard distribution" of Unix. It is much more liberal that the commonly used "GPL" or "Gnu Public License," which encumbers the software and derivative works with the requirement that they carry the source code and the same copyright agreement. The BSD copyright requires that the software and derivative work carry the identity of the copyright owner, as embodied in the lines:

```
Copyright (c) 1999-2003 The Regents of the University of California.
All rights reserved.
```

For contributions from other parties, "The Regents of the University of California" may be replaced with another institution name, and possibly augmented with contact information.

The copyright also requires that copies and derivative works include the disclaimer of liability in BOLD. It specifically *does not* require that copies of the software or derivative works carry the middle paragraph, so such copies and derivative works need not grant similarly liberal rights to users of the software.

The intent of the BSD copyright is to maximize the potential impact of the software by enabling uses of the software that are inconsistent with disclosing the source code or granting free redistribution

```
/* One line description of the class.

copyright notice

@ProposedRating color (email of proposer)
@AcceptedRating color (email of accepter)
*/

package name;

imports, in alphabetical order;

//////////////////////////////////////////////////////////////////////////
//// ClassName
/**
Class documentation.

@author Author Name
@version $Id$
*/
public class ClassName ... {

    constructors

    ///////////////////////////////////////////////////////////////////
    ////                         public variables                  ////

    public variables, in alphabetical order

    ///////////////////////////////////////////////////////////////////
    ////                         public methods                    ////

    public methods, in alphabetical order

    ///////////////////////////////////////////////////////////////////
    ////                         protected methods                 ////

    protected methods, in alphabetical order

    ///////////////////////////////////////////////////////////////////
    ////                         protected variables               ////

    protected variables, in alphabetical order

    ///////////////////////////////////////////////////////////////////
    ////                         private methods                   ////

    private methods, in alphabetical order

    ///////////////////////////////////////////////////////////////////
    ////                         private variables                 ////

    private variables, in alphabetical order
}
```

FIGURE 1. Anatomy of a Java file.

rights. For example, a commercial enterprise can extend the software, adding value, and sell the original software embodied with the extensions. Economic principles indicate that granting free redistribution rights may render the enterprise business model untenable, so many business enterprises avoid software with GPL licenses. Economic principles also indicate that, in theory, fair pricing of derivative works must be based on the value of the extensions, the packaging, or the associated services provided by the enterprise. The pricing cannot reflect the value of the free software, since an informed consumer will, in theory, obtain that free software from another source.

Software with a BSD license can also be more easily included in defense or national-security related applications, where free redistribution of source code and licenses may be inconsistent with the mission of the software.

Ptolemy II can include other software with copyrights that are different from the BSD copyright. The Ptolemy project generally does not, however, include GPL software, because provisions of the GPL license require that software with which GLP'd code is integrated also be encumbered by the GPL license. Nor do we include software with proprietary copyrights that do not permit redistribution of the software.

The date of the copyright for newly created files should be the current year:

```
Copyright (c) 2003 The Regents of the University of California.
All rights reserved.
```

If a file is a copy of a previously copyrighted file, then the start date of the new file should be the same as that of the original file:

```
Copyright (c) 1999-2003 The Regents of the University of California.
All rights reserved.
```

```
Copyright (c) 1999-2003 The Regents of the University of California.
All rights reserved.
Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

                                        PT_COPYRIGHT_VERSION_2
                                        COPYRIGHTENDKEY
```

FIGURE 2. Copyright notice used in Ptolemy II.

Ideally, files should have at most one copyright from one institution. Files with multiple copyrights are often in legal limbo if the copyrights conflict. If necessary, two institutions can share the same copyright, as in the following example:

```
Copyright (c) 2003 The Ptolemy Institute and The Regents of the
University of California.
All rights reserved.
```

Ptolemy II includes a copyright management system that will display the copyrights of packages that are included in Ptolemy II at runtime. The copyright management system is under development and likely to change. Currently, URLs such as `about:` and `about:copyright` are handled specially. If, within Ptolemy, the user clicks on a link with a target URL of `about:copyright`, then we eventually invoke code within $PTII/ptolemy/actor/gui/GenerateCopyrights.java. This class searches the runtime environment for particular packages and generates a web page with the links to the appropriate copyrights if certain packages are found.

2.2 Code rating

Following the copyright section is a pair of annotations labeled `@ProposedRating` and `@AcceptedRating`. These are followed by color (one of *red*, *green*, *green*, or *blue*) and the email address of the person responsible for the proposed or accepted rating level.

The intent of the code rating is to clearly identify to readers of the file the level of maturity of the contents. The Ptolemy Project encourages experimentation, and experimentation often involves creating immature code, or even “throw-away” code. Such code is *red*. We use a lightweight software engineering process documented in “Software Practice in the Ptolemy Project,”[1] to raise the code to higher ratings. That paper documents the ratings a:

• Red code is untrusted code. This means that the authors have no confidence in the design or implementation (if there is one) of this code or design, and that anyone that uses it can expect it to change substantially and without notice. All code starts at red.
• Yellow code is code with a trusted design. The authors have a reasonable degree of confidence in the design, and do not expect it to change in any substantial way. However, we do expect the API to shift around a little during development.
• Green code is code with a trusted implementation. The authors have confidence that the implementation is sound, based on test suites and practical application of the code. If possible, we try not to release important code unless it is green.
• Blue marks polished and complete code, and also represents a firm commitment to backwards-compatibility. Blue code is completely reviewed, tested, documented, and stressed in actual usage.

2.3 Imports

The imports section identifies the classes outside the current package on which this class depends.

1. J. Reekie, S. Neuendorffer, C. Hylands and E. A. Lee ``Software Practice in the Ptolemy Project,'' Technical Report Series, GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California, Berkeley, CA 94720, April 1999, http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII3.0/ptII3.0.2/doc/coding/sftwareprac/index.htm

The package structure of Ptolemy II is carefully constructed so that core packages do not depend on more elaborate packages. This limited dependencies makes it possible to create derivative works that leverage the core but drastically modify or replace the more advanced capabilities.

By convention, we list imports by full class name, as follows:

```
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.Entity;
import ptolemy.kernel.Port;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.Locatable;
import ptolemy.kernel.util.NameDuplicationException;
```

in particular, we do not use the wildcards supported by Java, as in:

```
import ptolemy.kernel.*;
import ptolemy.kernel.util.*;
```

The reason that we discourage wildcards is that the full class names in import statements makes it easier find classes that are referenced in the code. If you use an IDE such as Eclipse, it is trivially easy to generate the import list in this form, so there is no reason to not do it.

Imports are ordered alphabetically by package first, then by class name, as shown above.

3. Comment Structure

Good comments are essential to readable code. In Ptolemy II, comments fall into two categories, *Javadoc* comments, which become part of the generated documentation, and *code comments*, which do not. Javadoc comments are used to explain the interface to a class, and code comments are used to explain how it works.

Both Javadoc and code comments should be complete sentences and complete thoughts, capitalized at the beginning and with a period at the end. Spelling and grammar should be correct.

3.1 Javadoc and HTML

Javadoc is a program distributed with Java that generates HTML documentation files from Java source code files[2]. Javadoc comments begin with "`/**`" and end with "`*/`". The comment immediately preceding a method, member, or class documents that method, member, or class. Ptolemy II classes include Javadoc documentation for all classes and all public and protected members and methods. Members and methods should appear in alphabetical order within their protection category (public, protected etc.) so that it is easy to find them in the Javadoc output.

When writing Javadoc comments, pay special attention to the first sentence of each Javadoc comment. This first sentence is used as a summary in the Javadocs. It is extremely helpful if the first sentence is a cogent and complete summary.

Javadoc comments can include embedded HTML formatting. For example, by convention, in actor documentation, we set in italics the names of the ports and parameters using the syntax

2. See http://java.sun.com/j2se/javadoc/writingdoccomments/ for guidelines from Sun Microsystems on writing Javadoc comments.

```
/** In this actor, inputs are read from the <i>input</i> port ... */
```

The Javadoc program gives extensive diagnostics when run on a source file. Our policy is to format the comments until there are no Javadoc warnings. Private members and methods need not be documented by Javadoc comments.

3.2 Class documentation

The class documentation is the Javadoc comment that immediately precedes the class definition line. It is a particularly important part of the documentation. It should describe what the class does and how it is intended to be used. When writing it, put yourself in the mind of the user of your class. What does that person need to know? In particular, that person probably does not need to know *how* you accomplish what the class does. She only needs to know *what* you accomplish.

A class may be intended to be a base class that is extended by other programmers. In this case, there may be two distinct sections to the documentation. The first section should describe how a user of the class should use the class. The second section should describe how a programmer can meaningfully extend the class. Only the second section should reference protected members or methods. The first section has no use for them. Of course, if the class is abstract, it cannot be used directly and the first section can be omitted.

Comments should include honest information about the limitations of a class.

Each class comment should also include the following javadoc tags:

- @author
  The @author tag should list the authors and contributors of a class, for example:
  `@author Claudius Ptolemaus, Contributor: Tycho Brahe`
- @version
  The @version tag includes text that CVS automatically substitutes in the version. The @version tag starts out with:
  `@version $Id$`
  When the file is committed using CVS, the Id gets substituted, so the tag might look like:
  `@version $Id: NamedObj.java,v 1.213 2003/10/26 05:34:21 brahe Exp $`
- @since
  The @since tag refers the release that the class first appeared in. Usually, this is one decimal place after the current release. For example if the current release is 3.0.2, then the @since tag would read:
  `@since Ptolemy II 3.1`
  Adding an @since tag to a new class is optional, we usually update these tags programmatically when we do a release. However, authors should be aware of their meaning. Note that the @since tag can also be used when a method is added to an existing class, which will help users notice new features in older code.

3.3 Constructor documentation

Constructor documentation usually begins with the phrase “Construct an instance that ...” and goes on to give the properties of that instance. Note the use of the imperative case. A constructor is a command to construct an instance of a class. What it does is construct an instance.

3.4 Method documentation

Method documentation needs to state what the method does and how it should be used. For example:

```
/** Mark the object invalid, indicating that when a method
 *  is next called to get information from the object, that
 *  information needs to be reconstructed from the database.
 */
public void invalidate() {
    _valid = false;
}
```

By contrast, here is a poor method comment:

```
/** Set the variable _valid to false.
 */
public void invalidate() {
    _valid = false;
}
```

While this certainly describes what the method does from the perspective of the coder, it says nothing useful from the perspective of the user of the class, who cannot see the (presumably private) variable _valid nor how that variable is used. On closer examination, this comment describes *how* the method is accomplishing what it does, but it does not describe *what* it accomplishes.

Here is an even worse method comment:

```
/** Invalidate this object.
 */
public void invalidate() {
    _valid = false;
}
```

This says absolutely nothing.

Note the use of the imperative case in all of the above comments. It is common in the Java community to use the following style for documenting methods:

```
/** Sets the expression of this variable.
 *  @param expression The expression for this variable.
 */
public void setExpression(String expression) {
   ...
}
```

We use instead the imperative case, as in

```
/** Set the expression of this variable.
 *  @param expression The expression for this variable.
```

```
 */
public void setExpression(String expression) {
  ...
}
```

The reason we do this is that our sentence is a well-formed, grammatical English sentence, while the usual convention is not (it is missing the subject). Moreover, calling a method is a command "do this," so it seems reasonable that the documentation say "Do this." The use of imperative case has a large impact on how interfaces are documented, especially when using the listener design pattern. For instance, the java.awt.event.ItemListener interface has the method:

```
/** Invoked when an item has been selected or deselected.
 *  The code written for this method performs the operations
 *  that need to occur when an item is selected (or deselected).
 */
void itemStateChanged(ItemEvent e);
```

A naive attempt to rewrite this in imperative tense might result in:

```
/** Notify this object that an item has been selected or deselected.
 */
void itemStateChanged(ItemEvent e);
```

However, this sentence does not capture what the method does. The method may be called *in order to* notify the listener, but the *method* does not "notify this object". The correct way to concisely document this method in imperative case (and with meaningful names) is:

```
/** React to the selection or deselection of an item.
 */
void itemStateChanged(ItemEvent event);
```

The above is defining an interface (no implementation is given). To define the implementation, it is also necessary to describe what the method does:

```
/** React to the selection or deselection of an item by doing...
 */
void itemStateChanged(ItemEvent event) { ... implementation ... }
```

Comments for base class methods that are intended to be overridden should include information about what the method generally does, plus information that a programmer may need to override it. If the derived class uses the base class method (by calling `super.methodName()`), but then appends to its behavior, then the documentation in the derived class should describe *both* what the base class does and what the derived class does.

3.5 Referring to methods in comments

By convention, method names are set in the default font, but followed by empty parentheses, as in

```
/** The fire() method is called when ... */
```

The parentheses are empty even if the method takes arguments. The arguments are not shown. If the method is overloaded (has several versions with different argument sets), then the text of the documentation needs to distinguish which version is being used.

Other methods in the same class may be linked to with the {@link ...} Javadoc tag. For example, to link to a method Foo that takes a String:

```
* Unlike the {@link #Foo(String)} method, this method ...
```

Methods and members in the same package should have an octothorpe (# sign) prepended. Methods and members in other classes should use the fully qualified class name:

```
{@link ptolemy.util.StringUtilities.substitute(String, String,
String)}
```

Links to methods should include the types of the arguments.

To run Javadoc on the classes in the current directory, run `make docs`, which will create the HTML javadoc output in the doc/codeDoc subdirectory. To run Javadoc for all the common packages, run

```
cd $PTII/doc; make docs
```

The output will appear in $PTII/doc/codeDoc. Actor documentation can be viewed from within Vergil, right clicking on an actor and selecting View Documentation.

3.6 Tags in method documents

Methods should include Javadoc tags @param (one for each parameter), @return (unless the return type is void), and @exception (unless no exceptions are thrown). Note that we do not use the @throws tag, and that @returns is not a legitimate Javadoc tag, use @return instead.

The annotation for the arguments (the @param statement) need not be a complete sentence, since it is usually presented in tabular format. However, we do capitalize it and end it with a period.

Exceptions that are thrown by a method need to be identified in the Javadoc comment. An @exception tag should read like this:

```
* @exception MyException If such and such occurs.
```

Notice that the body always starts with "If", not "Thrown if", or anything else. Just look at the Javadoc output to see why. In the case of an interface or base class that does not throw the exception, use the following:

```
* @exception MyException Not thrown in this base class. Derived
*  classes may throw it if such and such happens.
```

The exception still has to be declared so that derived classes can throw it, so it needs to be documented as well.

3.7 FIXME annotations

We use the keyword “FIXME” in comments to mark places in the code with known problems. For example:

```
// FIXME: The following cast may not always be safe.
Foo foo = (Foo)bar;
```

To set up Eclipse to highlight FIXMEs, see the instructions in $PTII/doc/coding/eclipse.htm.

4. Code Structure

4.1 Names of classes and variables

In general, the names of classes, methods and members should consist of complete words separated using internal capitalization[3]. Class names, and only class names, have their first letter capitalized, as in AtomicActor. Method and member names are not capitalized, except at internal word boundaries, as in getContainer(). Protected or private members and methods are preceded by a leading underscore “_” as in _protectedMethod().

Static final constants should be in uppercase, with words separated by underscores, as in INFINITE_CAPACITY. A leading underscore should be used if the constant is protected or private.

Package names should be short and not capitalized, as in “de” for the discrete-event domain.

In Java, there is no limit to name sizes (as it should be). Do not hesitate to use long names.

4.2 Indentation and brackets

Nested statements should be indented by 4 characters, as in:

```
if (container != null) {
    Manager manager = container.getManager();
    if (manager != null) {
        manager.requestChange(change);
    }
}
```

Closing brackets should be on a line by themselves, aligned with the beginning of the line that contains the open bracket. Please avoid using the Tab character in source files. The reason for this is that code becomes unreadable when the Tab character is interpreted differently by different programs. Your text editor should be configured to react to the Tab key by inserting spaces rather than the tab character. To set up Emacs to follow the Ptolemy II indentation style, see $PTII/util/lisp/ptemacs.el. To set up Eclipse to follow the Ptolemy II indentation style, see the instructions in $PTII/doc/coding/eclipse.htm.

Long lines should be broken up into many small lines. The easiest places to break long lines are usually just before operators, with the operator appearing on the next line. Long strings can be broken

3. Yes, there are exceptions (NamedObj, CrossRefList, IOPort). Many discussions dealt with these names, and we still regret not making them complete words.

up using the + operator in Java, with the + starting the next line. Continuation lines are indented by 8 characters, as in the throws clause of the constructor in figure 1.

4.3 Spaces

Use a space after each comma:

```
Right: foo(a, b);
Wrong: foo(a,b);
```

Use spaces around operators such as plus, minus, multiply, divide or equals signs, and after semicolons:

```
Right: a = b + 1;
Wrong: a=b+1;
Right: for(i = 0; i < 10; i += 2)
Wrong: for(i=0 ;i<10;i+=2)
```

4.4 Exceptions

A number of exceptions are provided in the ptolemy.kernel.util package. Use these exceptions when possible because they provide convenient constructor arguments of type Nameable that identify the source of the exception by name in a consistent way.

A key decision you need to make is whether to use a compile-time exception or a run-time exception. A run-time exception is one that implements the RuntimeException interface. Run-time exceptions are more convenient in that they do not need to be explicitly declared by methods that throw them. However, this can have the effect of masking problems in the code.

The convention we follow is that a run-time exception is acceptable only if the cause of the exception can be tested for prior to calling the method. This is called a *testable precondition*. For example, if a particular method will fail if the argument is negative, and this fact is documented, then the method can throw a run-time exception if the argument is negative. On the other hand, consider a method that takes a string argument and evaluates it as an expression. The expression may be malformed, in which case an exception will be thrown. Can this be a run-time exception? No, because to determine whether the expression is malformed, you really need to invoke the evaluator. Making this a compile-time exception forces the caller to explicitly deal with the exception, or to declare that it too throws the same exception. In general, we prefer to use compile-time exceptions wherever possible.

When throwing an exception, the detail message should be a complete sentence that includes a string that fully describes what caused the exception. For example

```
throw IllegalActionException(this,
        "Cannot append an object of type: "
        + obj.getClass().getName() + " because "
        + "it does not implement Cloneable.");
```

Note that the exception not only gives a way to identify the objects that caused the exception, but also why the exception occurred. There is no need to include in the message an identification of the “this” object passed as the first argument to the exception constructor. That object will be identified when the exception is reported to the user.

If an exception is caught, be sure to use exception chaining to include the original exception. For example:

```
String fileName = foo();
try {
    // Try to open the file
} catch (IOException ex) {
    throw new IllegalActionException(this, ex,
        "Failed to open '" + fileName + "'");
}
```

5. Directory naming conventions

Individual demonstrations should be in directories under a `demo/` directory. The name of the directory, and the name of the model should match and both begin with capital letters. The demos should be capitalized so that it is possible to generate code for demonstrations. For example, the Butterfly demonstration is in `sdf/demo/Butterfly/Butterfly.xml`.

All other directories begin with lower case letters and most consist solely of lower case letters.