Ptolemy Project Coding Style

October 3, 2010

Ptolemy Project Coding Style

Authors: Christopher X. Brooks, Edward A. Lee

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 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.

In general, we follow the Sun Java Style guide (http://java.sun.com/docs/codeconv/). We encourage new developers to use Eclipse (http://www.eclipse.org) as their development platform. Eclipse includes a Java Formatter, and we have found that the Java Conventions style is very close to our requirements. For information about setting up Eclipse to follow the Ptolemy II coding style, see http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/eclipse.htm, which is a copy of $PTII/doc/coding/eclipse.htm, where $PTII is the location of your Ptolemy II installation. A file template that follows these rules can be found in $PTII/doc/coding/templates/JavaTemplate.java In addition 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/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 `ispell` command as /usr/local/bin/ispell. 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
  ```

- adm/bin/fix-files is a shell script that fixes common problems in files. To run this script, type:

  ```
  $PTII/adm/bin/fix-files *.java
  ```

2 Anatomy of a File

A Java file has the structure shown in figures 1 and 2.

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, method 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 3.

Figure 1: Anatomy of a Java file, part1.

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

   copyright notice
*/
package MyPackageName;

// Imports go here, in alphabetical order, with no wildcards.

//////////////////////////////////////////////////////////////////////////
//// ClassName

/**
   Describe your class here, in complete sentences.
   What does it do?  What is its intended use?

   @author yourname
   @version $Id: codingStyle.tex,v 1.19 2010/09/30 22:26:37 cxh Exp $
   @see classname (refer to relevant classes, but not the base class)
   @since Ptolemy II x.x
   @Pt.ProposedRating Red (yourname)
   @Pt.AcceptedRating Red (reviewmoderator)
*/
public class ClassName {
    /** Create an instance with ... (describe the properties of the
     *  instance). Use the imperative case here.
     *  @param parameterName Description of the parameter.
     *  @exception ExceptionClass If ... (describe what
     *   causes the exception to be thrown).
     */
    public ClassName(ParameterClass parameterName) throws ExceptionClass {
    }

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

    /** Description of the variable. */
    public int variableName;

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

    /** Do something... (Use the imperative case here, such as:
     *  "Return the most recently recorded event.", not
     *  "Returns the most recently recorded event.")
     *  @param parameterName Description of the parameter.
     *  @return Description of the returned value.
     *  @exception ExceptionClass If ... (describe what
     *   causes the exception to be thrown).
     */
    public int publicMethodName(ParameterClass parameterName)
        throws ExceptionClass {
        return 1;
    }

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

    /** Describe your method, again using imperative case.
     *  @see RelevantClass#methodName()
     *  @param parameterName Description of the parameter.
     *  @return Description of the returned value.
     *  @exception ExceptionClass If ... (describe what
     *   causes the exception to be thrown).
     */
    protected int _protectedMethodName(ParameterClass parameterName)
        throws ExceptionClass {
        return 1;
    }

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

    /** Description of the variable. */
    protected int _aProtectedVariable;
```

Figure 2: Anatomy of a Java file, part2.

```
    ////////////////////////////////////////////////////////////////
    ////                         private methods                ////
    // Private methods need not have Javadoc comments, although it can
    // be more convenient if they do, since they may at some point
    // become protected methods.
    private int _privateMethodName() {
        return 1;
    }

    ////////////////////////////////////////////////////////////////
    ////                         private variables              ////
    // Private variables need not have Javadoc comments, although it can
    // be more convenient if they do, since they may at some point
    // become protected variables.
    private int _aPrivateVariable;
}
```

Figure 3: Copyright notice used in Ptolemy II.

```
Copyright (c) 1999-2010 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 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
```

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) 1999-2010 The Regents of the University of California.
All rights reserved.
```

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 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. In general, we do not include software with the GNU General Public License (GPL) license, because provisions of the GPL license require that software with which GLP'd code is integrated also be encumbered by the GPL license. In the past, we have made an exception for GPL'd code that is aggregated with Ptolemy II but not directly combined with Ptolemy II. For example cvs2cl.pl was shipped with Ptolemy II. This file is a GPL'd Perl script that access the CVS database and generates a ChangeLog file. This script is not directly called by Ptolemy II, and we include it as a "mere aggregation" and thus Ptolemy II does not fall under the GPL. Note that we do not include GPL'd Java files that are compiled and then called from Ptolemy II because this would combine Ptolemy II with the GPL'd code and thus encumber Ptolemy II with the GPL.

Another GNU license is the GNU Library General Public License now known as the GNU Lesser General Public License (LGPL). We try to avoid packages that have this license, but we on occasion we have included them with Ptolemy II. The LGPL license is less strict than the GPL - the LGPL permits linking with other packages without encumbering the other package. In general, it is best if you

avoid GNU code. If you are considering using code with the GPL or LGPL, we encourage you to carefully read the license and to also consult the GNU GPL FAQ at http://www.gnu.org/licenses/gpl-faq.html. We also avoid including 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) 2010 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) 1999-2010 The Regents of the University of California.
All rights reserved.
```

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:

```
Copyright (c) 2010 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. To see what packages are used in a particular Ptolemy configuration, do “Help”, then “About” and then ”Copyright“. 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 Imports

The imports section identifies the classes outside the current package on which this class depends. 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 [1]. 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

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

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:

```
/** 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. The doccheck tool from
http://java.sun.com/j2se/javadoc/doccheck/index.html gives even more extensive diagnostics in HTML format. We encourage developers to run doccheck and fix all warnings. The nightly build at http://chess.eecs.berkeley.edu/ptexternal/nightly/ includes a run of doccheck.

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
  ```

  If you are creating a new file that is based on an older file, move the authors of the older file towards the end:

  ```
  @author Copernicus, Based on Galileo.java by Claudius Ptolemaus, Contributor: Tycho Brahe
  ```

The general rule is that only people who actually contributed to the code should be listed as authors. So, in the case of a new file, the authors should only be people who edited the file. Note that all the authors should be listed on one line. Javadoc will not include authors listed on a separate line.

- @version The @version tag includes text that Subversion automatically substitutes in the version. The @version tag starts out with: `@version $Id$` When the file is committed using Subversion, the `@version $Id$` gets substituted, so the tag might look like: `@version $Id: makefile 43472 2006-08-21 23:16:56Z cxh $`

  Note that for Subversion keyword substitution to work properly, the file must have the svn:keyword attribute set. In addition, it is best if the svn:native property is set. Below is how to check the values for a file named README.txt:

```
bash-3.2$ svn proplist README.txt
Properties on 'README.txt':
  svn:keywords
  svn:eol-style
bash-3.2$ svn propget svn:keywords README.txt
Author Date Id Revision
bash-3.2$ svn propget svn:eol-style README.txt
native
```

  To set the properties on a file:

```
svn propset svn:keywords "Author Date Id Revision" filename
svn propset svn:eol-style native filename
```

  For details about properly configuring your Subversion environment, see http://chess.eecs.berkeley.edu/ptexternal/wiki/Main/Subversion#KeywordSubstitution

- @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 8.0.2, then the @since tag on a new file would read:

```
@since Ptolemy II 8.1
```

  Adding an @since tag to a new class is optional, we usually update these tags by running a script 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.

- @Pt.ProposedRating
- @Pt.AcceptedRating Code rating tags, discussed below.

3.3 Code rating

The Javadoc tags `@Pt.ProposedRating` and `@Pt.AcceptedRating` contain code rating information. Each tag includes the color (one of `red`, `yellow`, `green` or `blue`) and the Subversion login of the person responsible for the proposed or accepted rating level, for example:

```
@Pt.ProposedRating blue ptolemy
@Pt.AcceptedRating green ptolemy
```

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 we 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. We 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. We 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.

The Javadoc doclet at $PTII/doc/doclets/RatingTaglet.java adds the ratings to the Javadoc output.

3.4 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.

3.5 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.

3.6 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 foo() method 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.

3.7 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.

3.8 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;
```

By default, Eclipse will highlight FIXMEs.

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 [2]. 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.

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

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 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, after semicolons and after keywords like if, else, for, do, while, try, catch and throws:

```
Right: a = b + 1;
Wrong: a=b+1;

Right: for(i = 0; i < 10; i += 2)
Wrong: for (i=0 ;i<10;i+=2)

Right: if ( a == b) {
Wrong: if(a==b)
```

Note that the Eclipse clean up facility will fix these problems, see
http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/eclipse.htm.

4.4 Exceptions

A number of exceptions are provided in the 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 + "'");
}
```

4.5 Code Cleaning

Code cleaning is the act of homogenizing the coding style, looking for and repairing common problems. Fortunately, Eclipse includes a file formatter and a cleaner that fixes many common problems. Software that is to be formally released should be cleaned according to the guidelines set forth in $PTII/doc/coding/releasemgt.htm

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 should consist solely of lower case letters. Java package names with embedded upper case letters are not encouraged.

6 Makefiles

The Ptolemy tree uses makefiles to provide a manifest of what files should be shipped with the release and to break the system up into modules. The advantage of this system is that we ship only the files that are necessary.

There are a few different types of makefiles

- makefiles that have no subdirectories that contain source code. For example, $PTII/ptolemy/kernel/util contains Java files but has no subdirectories that contain Java source code.
- makefiles that have one or more subdirectories that contain source code. For example, $PTII/ptolemy/kernel contains Java files and ptolemy/kernel has subdirectories such as $PTII/ptolemy/kernel/util that contain Java source code.
- makefiles in directories that do not have source code in the current directory, but have subdirectories that contain source code. For example, the ptolemy directory does not contain Java files, but does contain subdirectories that contain Java files.

- makefiles that contain tests. For example, ptolemy/kernel/util/test contains Tcl tests.
- makefiles in $PTII/mk that are included by other makefiles.

6.1 An example makefile

Below are the sections of $PTII/ptolemy/kernel/util/makefile.

```
# Makefile for the Java classes used to implement the Ptolemy kernel
```

Each makefile has a one line description of the purpose of the makefile.

```
#
# @Authors: Christopher Hylands, based on a file by Thomas M. Parks
#
```

The authors for the makefile.

```
# @version $Id: makefile 56450 2009-12-06 06:54:56Z eal $
```

The version control version. We use `$Id$`, which gets expanded by Subversion to include the revision number, the date of the last revision and the login of the person who made the last revision.

```
# @Copyright (c) 1997-2010 The Regents of the University of California.
```

The copyright year should be the start with the year the makefile file was first created, so when creating a new file, use the current year, for example "Copyright (c) 2010 The Regents . . ."

```
# 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
```

The new BSD copyright appears in each makefile.

```
ME =  ptolemy/kernel/util
```

The makefile variable ME is set to the directory where that includes the makefile. Since this makefile is in ptII/ptolemy/kernel/util, ME is set to that directory. The ME variable is primarily used by make to print informational messages.

```
DIRS =  test
```

The DIRS makefile variable lists each subdirectory in which make is to be run. Any subdirectory that contains a makefile should be listed in DIRS.

```
# Root of the Ptolemy II directory
ROOT = ../../..
```

The ROOT makefile variable is a relative path to the $PTII directory. This variable is relative so as to avoid problems if the $PTII environment variable is not set or is set to a different version of Ptolemy II.

```
CLASSPATH =  \$(ROOT)
```

Set the Java classpath to the value of the ROOT makefile variable, which should be the same as the $PTII environment variable. Note that if this makefile contains Java files that require third party software contained in jar files not usually found in the Java classpath, then CLASSPATH would be set to include those jar files, for example CLASSPATH = $(ROOT)$(CLASSPATHSEPARATOR)$(DIVA_JAR) would include ptII/lib/diva.jar, where the DIVA_JAR makefile variable is defined in ptII.mk

```
# Get configuration info
CONFIG = $(ROOT)/mk/ptII.mk
include $(CONFIG)
```

The above includes $PTII/mk/ptII.mk The way the makefiles work is that the $PTII/configure script examines the environment, and then reads in the
$PTII/mk/ptII.mk.in file, substitutes in user specific values and creates $PTII/mk/ptII.mk. Each makefile refers to $PTII/mk/ptII.mk, which defines variable settings such as the location of the compilers.

```
# Flags to pass to javadoc. (Override value in ptII.mk)
JDOCFLAGS =  -author -version -public $(JDOCBREAKITERATOR) $(JDOCMEMORY) $(JDOCTAG)
```

Directory specific makefile variables appear here. This variable sets JDOCFLAGS, which is used if "make docs" is run in this directory. JDOCFLAGS is not often used, we include it here for completeness.

```
# Used to build jar files
PTPACKAGE =  util
PTCLASSJAR =  $(PTPACKAGE).jar
```

PTPACKAGE is the directory name of this directory. In this example, the makefile is in ptolemy/kernel/util, so PTPACKAGE is set to util. PTPACKAGE is used by PTCLASSJAR to name the jar file when make install is run. For this file, running `make install` will create util.jar. If a directory contains subdirectories that have source files, then PTCLASSJAR is not set and PTCLASSALLJAR and PTCLASSALLJARS is set, see below.

```
JSRCS = \
        AbstractSettableAttribute.java \
        Attribute.java \
        BasicModelErrorHandler.java \
```

And so on . . .

```
        ValueListener.java \
        Workspace.java
```

A list of all the .java files to be included. The reason that each Java file is listed separately is to avoid shipping test files and random kruft. Each file that is listed should follow this style guide.

```
EXTRA_SRCS =        $(JSRCS)
```

EXTRA_SRCS contains all the source files that should be present. If there are files such as icons or .xml files that should be included, then OTHER_FILES_TO_BE_JARED is set to include those files and the makefile would include:
EXTRA_SRCS = $(JSRCS) $(OTHER_FILES_TO_BE_JARED)

```
# Sources that may or may not be present, but if they are present, we don't
# want make checkjunk to barf on them.
# Don't include demo or DIRS here, or else 'make sources' will run 'make demo'
MISC_FILES =        $(DIRS)

# make checkjunk will not report OPTIONAL_FILES as trash
# make distclean removes OPTIONAL_FILES
OPTIONAL_FILES = \
        doc \
        'CrossRefList$$1.class' \
        'CrossRefList$$CrossRef.class' \
```

And so on . . .

```
        'Workspace$$ReadDepth.class' \
         $(PTCLASSJAR)
```

MISC_FILES and OPTIONAL_FILES are used by the make checkjunk command. The checkjunk target prints out the names of files that should not be present. We use checkjunk as part of the release process. MISC_FILES should *not* include the demo directory or else running make sources will invoke the demos. To determine the value of OPTIONAL_FILES, run `make checkjunk` and add the missing .class files. Since the inner classes have $ in their name, we need to use single quotes around the inner class name and repeat the $ to stop make from performing substitution.

```
JCLASS = $(JSRCS:%.java=%.class)
```

JCLASS uses a make macro to read the value of JSRCS and substitute in .class for .java. JCLASS is used to determine what .class files should be created when make is run.

```
all: jclass
install: jclass $(PTCLASSJAR)
```

The all rule is the first rule in the makefile, so if the command make is run with no arguments, then the all rule is run. The all rule runs the jclass rule, which compiles the java files. The install rule is run if make install is run. The install rule is like the all rule in that the java files are compiled. The install rule also depends on the value of PTCLASSJAR makefile variable, which means that make install also creates util.jar

```
# Get the rest of the rules
include $(ROOT)/mk/ptcommon.mk
```

The rest of the rules are defined in ptcommon.mk

6.2 jar files

If a directory contains subdirectories that contain sources or resources necessary at runtime, then the jar file in that directory should contain the contents of the jar files in the subdirectories. For example, $PTII/ptolemy/kernel.jar contains the .class files from $PTII/ptolemy/kernel/util and other subdirectories.

Using $PTII/ptolemy/kernel/makefile as an example, we discuss the lines that are different from the example above.

```
DIRS =                  util attributes undo test
```

DIRS contains each subdirectory in which make will be run

```
# Used to build jar files
PTPACKAGE =         kernel
PTCLASSJAR =
```

Note that in ptolemy/kernel/util, we set PTCLASSJAR, but here it is empty.

```
# Include the .class files from these jars in PTCLASSALLJAR
PTCLASSALLJARS = \
                 attributes/attributes.jar \
                 undo/undo.jar \
                 util/util.jar
```

PTCLASSALLJARS is set to include each jar file that is to be included in this jar file. Note that we don't include test/test.jar because the test directory contains the test harness and test suites and is not necessary at run time

```
PTCLASSALLJAR = $(PTPACKAGE).jar
```

PTCLASSALLJAR is set to the name of the jar file to be created, which in this case is kernel.jar.

```
install: jclass jars
```

The install rule depends on the jars target. The jars target is defined in ptcommon.mk. The jars target depends on PTCLASSALLJAR, so if PTCLASSALLJAR is set, then make unjars each jar file listed in PTCLASSALLJARS and creates the jar file named by PTCLASSALLJAR

7 Subversion Keywords

If you are checking files in to the Ptolemy II Subversion repository, then you must set two svn properties:

- svn:keywords must be set to "Author Date Id Revision"
- svn:eol-style must be set to "native"

To enable keyword substitution, such as Id being changed to

```
@version $Id: Foo.java 43472 2006-08-21 23:16:56Z cxh $,
```

you need to set up ~/.subversion/config so that each file extension has the appropriate settings. See http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/eclipse.htm#Subversive for details which involve adding $PTII/ptII/doc/coding/svn-config-auto-props.txt to ~/.subversion/config

Why is it necessary to add have a pattern for every file? The answer is that Subversion decides that everything is a binary file and that it is safer to check things in and not modify them. However, there should be a repository wide way to set up config instead of requiring each user to do so.

To test out keyword substitution on new files, follow the steps below. If you have read/write permission to the
source.eecs.berkeley.edu SVN repositories, then use the svntest repository. If you don't have write permission on the source.eecs.berkeley.edu repositories, then use your local repository.

```
bash-3.2$ svn co svn+ssh://source.eecs.berkeley.edu/chess/svntest
A    svntest/README.txt
Checked out revision 7.
bash-3.2$ cd svntest

bash-3.2$ echo '$Id$' > testfile.txt

bash-3.2$ svn add testfile.txt
A         testfile.txt
bash-3.2$ svn commit -m "A test for svn keywords: testfile.txt " testfile.txt
Adding         testfile.txt
Transmitting file data .
Committed revision 8.
bash-3.2$ cat testfile.txt
@version $Id: testfile.txt 1.1 2010-03-31 18:18:22Z cxh $
bash-3.2$ svn proplist testfile.txt
Properties on 'testfile.txt':
  svn:keywords
  svn:eol-style
```

Note that testfile.txt had `$Id$` properly substituted. If testfile.txt had only `$Id$` and not something like
`$Id: codingStyle.tex,v 1.19 2010/09/30 22:26:37 cxh Exp $` then keywords were not being substituted and that ~/.subversion/config had a problem.

7.1 Checking Keyword Substitution

To check keyword substitution on a file:

```
bash-3.2$ svn proplist README.txt
Properties on 'README.txt':
  svn:keywords
  svn:eol-style
bash-3.2$ svn propget svn:keywords README.txt
Author Date Id Revision
bash-3.2$ svn propget svn:eol-style README.txt
native
```

See $PTIII/doc/coding/releasemgt.htm for information about how to use $PTII/adm/bin/svnpropcheck to check many files.

7.2 Fixing Keyword Substitution

To set the keywords in a file called MyClass.java:

```
svn propset svn:keywords "Author Date Id Revision" MyClass.java
svn propset svn:eol-style native MyClass.java
svn commit -m "Fixed svn keywords" MyClass.java
```

7.3 Setting svn:ignore

Directories that contain .class files should have svn:ignore set. It is also helpful if svn:ignore is set to ignore the jar file that is created by make install. For example, in the package ptolemy.foo.bar, a makefile called bar.jar will be created by make install. One way to set multiple values in svn:ignore is to create a file /tmp/i and add what is to be ignored:

```
svn propget svn:ignore . > /tmp/i
```

If the directory has svn:ignore set, then /tmp/i will contain the files to be ignored. If the directory does not have svn:ignore set, then /tmp/i will be ignored. Edit /tmp/i and add files to be ignored:

```
*.class
bar.jar
```

Then run

```
svn propset svn:ignore -F /tmp/i .
svn commit -N -m "Added *.class and bar.jar to svn:ignore" .
```

We use the -N option to commit just the directory.

8 Checklist for new files

Below is a checklist for common issues with new Ptolemy II files.

8.1 Infrastructure

1. Is the java file listed in the makefile? (section 6.1)
2. Are the subversion properties svn:keywords and svn:eol-style set? (section 7)

8.2 File Structure

1. Copyright - Does the file have the copyright? (section 2.1)
2. Is the copyright year correct? New files should have the just the current year (section 2.1)

8.3 Class comment

1. Is the first sentence of the class comment a cogent and complete summary? (section 3)
2. Are these tags present? (section 3.2):

```
@author
@version
@since
@Pt.ProposedRating
@Pt.AcceptedRating
```

3. Are the constructors, methods and variables separated by the appropriate comment lines? (section 2)

8.4 Constructor, method, field and inner class Javadoc documentation.

1. Within each section, is each Javadoc comment alphabetized? (section 2)
2. Is the first sentence a cogent and complete summary in the imperative case? (section 3.5)
3. Are all the parameters of each method clearly documented? (section 3.7)
4. Are the descriptions of each exception useful?
5. Did you run doccheck and review the results? See http://chess.eecs.berkeley.edu/ptexternal/nightly/ (section 3.1)

8.5 Overall

1. Did you run spell check on the program and fix errors? (section 1)
2. Did you format the file using Eclipse? (section 1)
3. Did you fix the imports using Eclipse? (section 2.2)
4. Did you add the new file to the makefile?

9 Checklist for creating a new demonstration

Ptolemy II ships with many demonstrations that have a consistent look and feel and a high level of quality. Below is a checklist for creating a new demonstration.

1. Does the name of the demonstration match the directory? Demonstrations should be in directories like demo/Foo/Foo.xml so that the code generators can easily find them. Model names should definitely be well-formed Java identifiers, so Foo-Bar.xml is not correct, use FooBar.xml instead.
2. Does the model name begin with a capital letter? Most demonstrations start with a capital letter because if we generate code for them, then the corresponding class should start with a capital letter.
3. Is there a makefile in the directory that contains the demonstration and does the upper level makefile in the demos directory include the jar file produced in the directory? If your model is demo/Foo/Foo.xml, then demo/makefile should include Foo/Foo.jar in PTCLASSALLJARS.
4. Does the demonstration have a title?
5. Does the demonstration have an annotation that describes what the models does and why it is of interest?
6. Does the demonstration have any limitation or requirements for third-party software or hardware clearly marked in red. Usually these limitations use a smaller font.
7. Does the demonstration make sense? Some models require third party software or hardware and might not be the best demonstration.
8. Is there a gray author annotation in the bottom corner? The color values should be 0.4,0.4,0.4,1.0, which appears as grey. One trick is to copy the author annotation from another demonstration.
9. Do all the models use paths that are relative and not specific to your machine? In general, it is best if fileOrURL parameters start with $CLASSPATH so that they can be found no matter how Ptolemy is invoked.

10. Has the demonstration been added to $PTII/doc/coding/completeDemos.htm?

11. Is there a separate test that will exercise a model similar to the demo? Usually these tests go in directories like domains/sdf/test/auto because the domains/*/test/auto models are run after all of Ptolemy has been built.

10 Checklist for creating a new directory

To create a directory that contains Java file follow the steps below:

1. Copy a makefile from a similar directory. (section 6)

2. Verify that the makefile works by running the command below

```
make sources
make
make install
```

3. Check the style of the makefile: first line, author, copyright date etc. (section 6)

4. Add the package to doc/makefile so that Javadoc is created.

5. Create package.html and README.txt by running adm/bin/mkpackagehtml. For example, to create these files in the package ptolemy.foo.bar:

```
$PTII/adm/bin/mkpackagehtml ptolemy.foo.bar
cd $PTII/ptolemy/foo/bar
svn add README.txt package.html
```

6. Don't forget to edit package.html and add descriptive text that describes the package.

7. Set the svn:ignore properties (section 7.3)

References

[1] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software practice in the ptolemy. Technical Report GSRC-TR-1999-01, Gigascale Silicon Research Center, April 1999.