Software Practice in the Ptolemy Project

John Reekie
Stephen Neuendorffer
Christopher Hylands
Edward A. Lee

Gigascale Semiconductor Research Center
Technical Report Series
GSRC-TR-1999-01

April 1999

This page is intentionally left not quite blank

Software Practice in the Ptolemy Project

John Reekie
Stephen Neuendorffer
Christopher Hylands
Edward A. Lee

University of California at Berkeley

In the Ptolemy project at UC Berkeley, we have been exploring ways of improving our software development process. Unlike many other efforts at improving software development, our efforts are focused on the particular needs of leading-edge academic research. High levels of creativity and flexibility are paramount in this environment, and the goals and qualifications of those producing the software are substantially different from those in commercial software production. This report describes our efforts to meet these needs and what we have learned.

1.0 Introduction

Increasingly, software is being seen as a publication medium for academic research. The Department of Electrical Engineering and Computer Science at UC Berkeley has a tradition of releasing software, and we believe that this tradition substantially magnifies the impact of the research behind the software. The Ptolemy project, which released its first version of the Ptolemy software environment in 1992, has grown through a series of supported releases incorporating the latest research results, an increasing user community, and an active newsgroup (`comp.soft-sys.ptolemy`). Other CAD projects from the University have experienced similar success.

This grounding of academic research in concrete and usable software is, we believe, becoming a necessity for researchers working in fields in which the primary product of the research is intellectual knowledge that is best embodied in software. In recent years, there has been a marked shift in the thinking of leading curriculum developers in Electrical Engineering and Computer Sciences, from a model based on physical models and low-level algorithms, to an emphasis on systems, networks, and communications. This shift is accompanied, particularly on the Electrical Engineering side, by a dramatic swing in the proportion of research output that is fundamentally software-based, and we

posit that many students in Electrical Engineering now spend the majority of their time writing software. Software is also increasingly rapidly in complexity, and validating research by embodying it in published software is increasingly important.

Although the importance of software publication is increasing rapidly, software processes designed for industrial use are becoming less relevant to this environment. The emphasis of processes such as the Capability Maturity Model (CMM) is on factors such as predictability and repeatability over a series of projects. In a research environment, we are more interested in flexibility and creativity, and projects that resemble earlier projects sufficiently to allow predictable and repeatable results are – by definition – generally not interesting in a research environment. For us, the questions are:

- What techniques can we use to improve our productivity and the quality of our "research" software?
- How much can we improve quality without increasing cost?
- How do we introduce and maintain new practices?
- How do we maintain creativity and excitement?

Since beginning work on Ptolemy II, the successor to the original Ptolemy – now called "Ptolemy Classic" – we have been working on improving our software practice. Our approach has been quite simple: look at industry best practice and see what we can incorporate or adapt to our environment in a suitable way. Our most common method for introducing a new technique is to hold a study group, in which members of the research group read and discuss selected materials describing the technique. We then try using the technique, initially in mock-up experiments or by gradually incorporating it into our development process. At each step along the way, we keep in mind the relevance of the technique to our needs and adapt (or discard) it accordingly.

The benefits of this approach have been substantial. Contrasted to Ptolemy Classic, the design and growth of Ptolemy II is less *ad-hoc* and more visible. Ptolemy Classic grew very large, and over the years became awkward to use as a research vehicle.[1] In Ptolemy II, every member of the research group has a thorough understanding of its architecture, as opposed to just one or two. Instead of finding bugs and problems by trying to use the infrastructure, we found them by reviewing and testing the design and the code. Having a more reliable infrastructure promotes good research because the researcher can focus on the problem at hand rather than fixing or extending core infrastructure.

The report is organized as follows:

- Techniques

  Section 2 describes the software design techniques that we have found effective. For each, we give a short summary of the technique, how we introduced the technique into our research group, and what we see as the benefits of the technique.

1. By this, we refer to its utility for experimenting with new models of computation and views of system modeling. The usefulness of Ptolemy Classic as a design tool in fields as diverse such as signal processing, architecture modeling and optronics, is not in question.

- The review process

  We have made formal reviews the mainstay of our development process. Reviews need a lot of “how-to” explanation, and so this section provides a detailed description of how we review. We document what has worked for us and what we need to improve.

- Guidelines

  The appendices contains copies of detailed information from our internal Web pages. We hope that this information will encourage other researchers to try applying a similar approach.

2.0 Techniques

The half-dozen or so individual techniques that have benefited us most are well-known techniques in software development. Nonetheless, they are not often applied in the production of academic software. Our experience has shown us that, in an academic research group with a commitment to publishing high-quality software, these techniques can be extremely effective, both individually and in concert.

In considering why these techniques are not more widely used when we have found them so effective, we believe the key issue is one of misunderstanding the scope and application of the techniques. Software methodologies are often perceived as being sold as "silver bullets" – techniques or methodologies that will solve all of your software problems. As pointed out by Weigers [14], adopting a methodology wholesale can be a big mistake; but there are nonetheless many valuable practices contained in these methodologies, and these can indeed be effective when adapted appropriately.

There is one point we would like to note before proceeding. Although these techniques are generally not particularly hard to learn, the real trick is making them stick – that is, in making their use habitual and part of the group's development culture. Simply knowing how to produce (say) a static structure diagram on demand is not useful if one does not consistently use this technique in the framework of the whole development process. To do so, one must find ways to encourage adoption of techniques. Often, this means causing changes in development habits. Ideally, people change because they see that a new technique helps them to work more effectively; but usually, it is necessary to use a combination of demonstrating by effectiveness, leading by example, and using a little authority where necessary.

2.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) [1][13] is a recent visual notation for modeling object-oriented software systems. UML is the product of three key figures in object-oriented analysis and design: James Rumbaugh, Grady Booch, and Ivar Jacobson. Although development of UML was sponsored by Rational Software Corporation, the language is an open standard, and has been adopted as an OMG (Object Management Group) standard. UML is well-positioned to succeed a plethora of different object modeling notations developed in the late 1980's and early 1990's.

2.1.1 Summary

UML consists of nine visual diagramming notations, each intended to represent a different aspect of a software system. The most commonly used notation is the *static structure diagram*, which is a development of OMT class diagrams [10]. A static structure diagram is a representation of a set of classes and the relations between them. The key features of a static structure diagram are shown in the UML fragment below:

- Classes

  A class is shown as a rectangle containing the class name, attributes, and operations. Depending on the level of detail represented by the diagram, attributes may represent abstract state of class instances (high-level), or actual instance variables (low

level). At a high level, the set of operations includes operations that are relevant to that class only; at a low level, operations that construct and traverse associations (see below) are also shown.

Several notations add detail to a class. Symbols such as "+" and "#" indicate public and protected attributes and methods, while italics indicate an abstract class or operation.

- Inheritance

  Inheritance is shown as a solid connecting line with a triangle at the superclass end. In this example, JCanvas inherits from JComponent, and GraphicsPane inherits from CanvasPane. CanvasPane is an abstract class, as indicated by the italic font used for its name.

  Not all classes are shown in one diagram – those that are not are assumed to be defined in a separate diagram. Note also that this diagram is conceptual; for example, constructors are not shown in detail.

- Interfaces

  An interface is marked by the stereotype ≪Interface≫. Implementation of an interface is shown by a dashed line with a triangle at the interface end. For example, CanvasPane implements CanvasComponent and EventAcceptor. An alternative notation is shown attached to CanvasLayer, which implements CanvasComponent.

- Associations

  Associations are the real substance of static structure diagrams, as they show detailed information about the relations between classes. The information shown by associations is *not* available from the source code itself, so their presence on static structure diagrams is an invaluable resource. Each association can have a *multiplicity* at each end, such as "1..n" or "0..1". (Unless shown otherwise, the default is one.) Here, for example, each CanvasPane can be contained by zero or one CanvasComponents, and zero or one JCanvasses. Each association can have a name (none do here), and each end of an association can be labeled with a role, indicating the role played by the class at that end of the association. Here, the association from CanvasComponent to itself has a "parent" role.

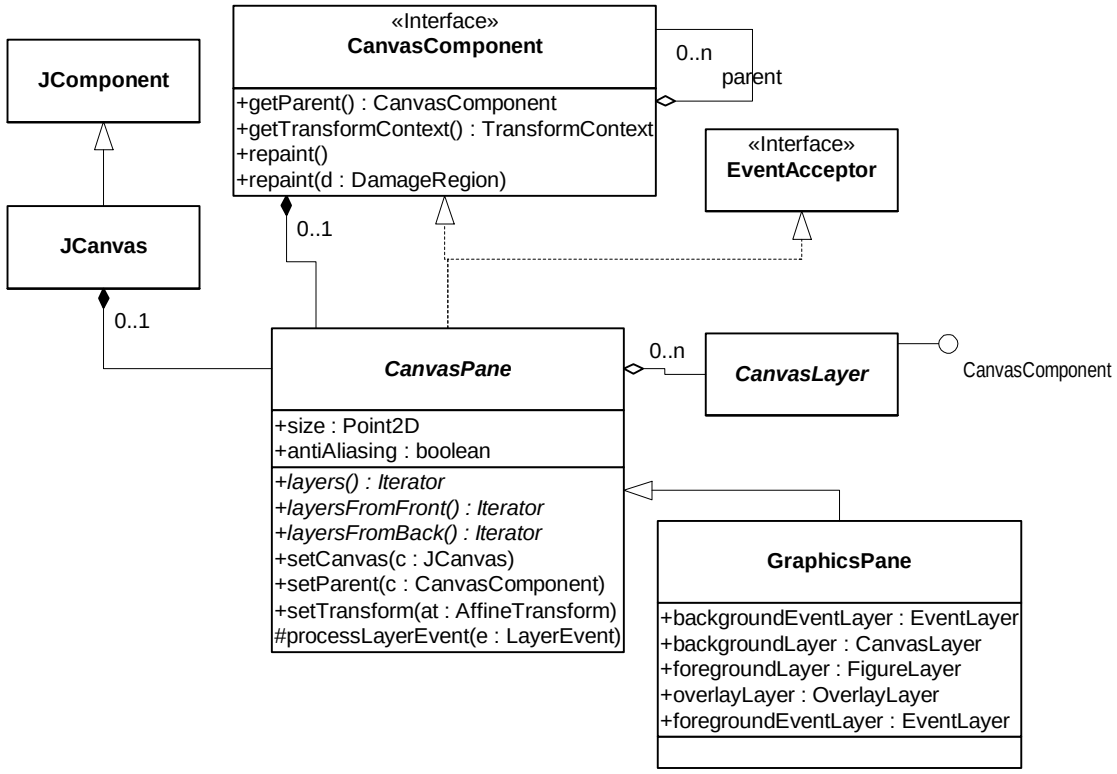

FIGURE 1. A sample UML static structure diagram

A second important notation in the UML is the sequence diagram. In contrast to static structure diagrams, sequence diagrams show individual objects instead of classes. Each diagram shows one or a few possible sequences of interaction between a set of objects. As shown in Figure 2, sequence diagrams order execution events along a set of time-lines. Patterns of communication, and object lifetimes and dependencies are easily seen.

In this example, a DataReceiver object constructs an Element, which it adds to a Model. It then draws the Element on a View, which performs some additional operations. Although conceptually simple, sequence diagrams are extremely effective for under-standing the flow of control between several objects.

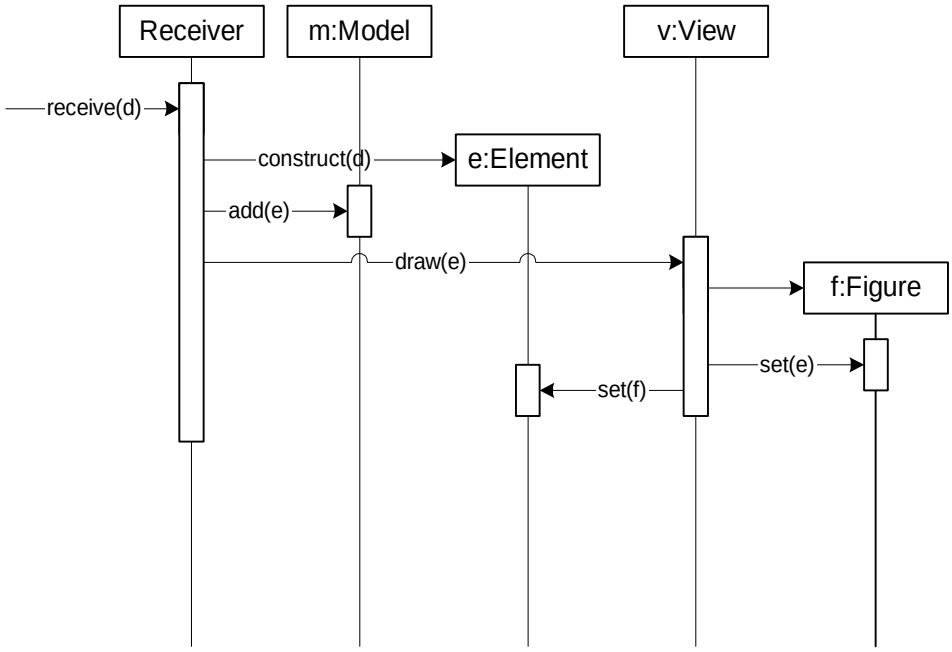

FIGURE 2. A sample sequence diagram

2.1.2 Introducing UML

We initially introduced OMT (Object Modeling Technique) into our research group in a series of study groups, and later changed to using UML. We originally used Rumbaugh *et al* [11] as the reference text, focussing mainly on the chapters on analysis and on the OMT class diagram notation, on which UML static structure diagrams are based. In later study groups, we added readings from other texts, in particular Rumbaugh's 1996 book [12] and Riel's *Object-Oriented Design Heuristics* [10]. Readings were chosen to emphasize particular aspects of static structure diagrams, with particular attention given to associations, which are both the most important concept and the hardest to grasp properly.

The process of learning how to draw OMT class diagrams was aided by a real design case study provided by one of the students (Mike Williamson). After creating an initial design, we worked on refining it. At the left of Figure 3 is one of the initial OMT designs, while the right shows a later, cleaner design. The process of actually trying to produce static structure diagrams with real designs is by far the most effective technique for learning the notation.

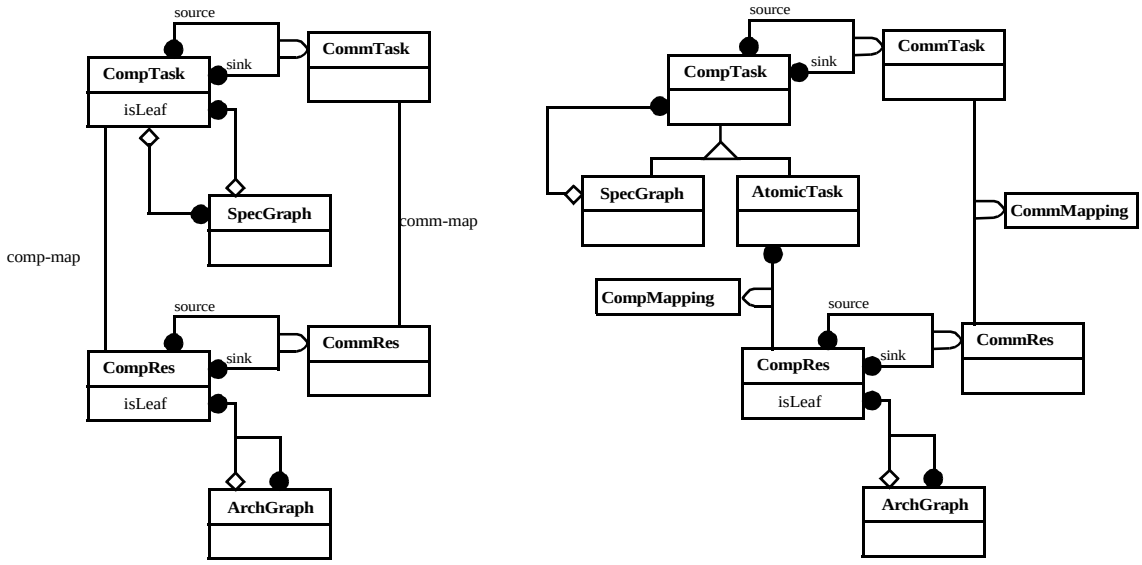

FIGURE 3. Examples of OMT class diagrams produced in a study group. At left, an initial design; at right, the design after some rework and refinement. Closed circles mean a multiplicity of zero or more, while the arcs connecting a class to an association show a class that is part of the association.

After introducing OMT, we took a look at the emerging UML standard, and decided that UML was probably a win. After a period of uncertainty, we chose Visio Professional as a drawing tool, which includes a fairly basic UML static structure editor. During this process, two other factors encouraged adoption of UML. First, we started performing design reviews using UML static structure diagrams. Second, we started work on a detailed design document for the Ptolemy II kernel. The document static structure diagrams extensively in the early chapters, thus setting a standard and precedent for later chapters.

2.1.3 Benefits

The use of UML static structure diagrams is an enormous benefit. All of the key parts of Ptolemy II are documented with static structure diagrams, providing a valuable design reference. (Some of the Ptolemy II developers have remarked that the most-thumbed pages of the kernel design manual are the pages with the static structure diagrams.) The static structure diagrams also play a key role in all design reviews.

A second benefit of drawing static structure diagrams is the aid they give to the developer. This benefit has not been fully realized, as many developers still generate UML diagrams after coding rather than before or during. This habit is changing slowly, though, as senior members of the group are starting to insist on UML diagrams as a requirement for and product of design discussions.

Finally, we are starting to find situations in which static structure diagrams are not enough – for example, design discussions that could be easily resolved with the aid of

sequence diagrams. Making sequence diagrams a habit, as static structure diagrams are now, is our next goal for our UML practice.

2.2 Design patterns

A design pattern captures a solution to a recurring problem in object-oriented design. Design patterns are lower-level than architectures: an architecture will typically exhibit several design patterns. The original reference on design patterns, and the one we used as a text, is the 1994 book by Gamma *et al* [5].

The figure below illustrates one of the design patterns from Gamma *et al*, the Strategy pattern. This pattern is a solution to the problem of dynamically altering the behavior of an object. By creating a separate abstract class that encapsulates a particular function, one can then substitute different sub-classes of the abstract class at run-time. There is no change to the interface presented by the Context object, but its behavior is different, as it delegates the operation to its Strategy object.

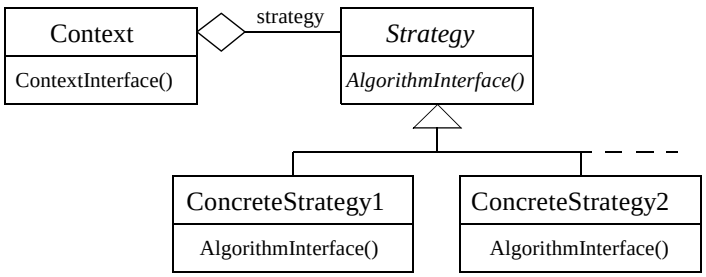

FIGURE 4. The **Strategy** design pattern

Patterns are presented in a stylized prose description. The description typically contains sections describing motivation, the pattern, known instances of the pattern, and so on. One of the best things about the way that patterns are presented is that issues in using and implementing the pattern are presented evenly: positive and negative aspects of each pattern and implementation choice are both dealt with frankly.

The patterns book by Gamma *et al* contains 23 design patterns, and there have probably been hundreds published since. There are two key benefits to studying design patterns. First, since a design pattern codifies design experience, familiarity with the basic patterns often enables one to arrive at a design more quickly, by reusing and combining known patterns. Less time is spent agonizing over alternatives, since the basic pattern and various implementation trade-offs are documented. Furthermore, inexperienced designers are more likely to "get it right" by following common patterns.

Second, design patterns provide a vocabulary with which to communicate fundamental aspects of a design. For example, a developer who describes a design as a *Composite* with a *Factory* here for window construction and a *Strategy* there for display painting, is

immediately conveying essential information about the design to someone that speaks the same vocabulary.

2.2.1 Introducing design patterns

Design patterns are easier to teach than to absorb! We started with two, two-hour study groups. Each participant chose one pattern, which he described and gave a concrete example of from our own design space. The patterns we covered from Gamma *et al* were *Composite*, *Decorator*, *Facade*, *Proxy*, *Observer*, *Strategy*, and *Template Method*. This is only a small selection, but we felt it enough to introduce the concept of patterns.

In design meetings following the patterns study groups, several of these patterns did appear in our designs, and we believe the designs were cleaner for it. In one case, the pattern appeared in our designs for a while but was eventually replaced by a more sophisticated pattern: *Observer* (aka publish-and-subscribe or model-view) was subsumed by the typed *Event-Listener* pattern implemented throughout Java AWT 1.1.

In retrospect (now that we write this report), we believe that it would have been worth holding some additional study groups to study additional patterns. Questions raised during design meetings indicated that a deep understanding of even simple patterns does not come immediately, and additional study would perhaps have yielded greater benefits.

2.3 Daily build and smoke test

One of the most important factors in releasing any kind of software is the ability to build and test the software on a regular basis. In Ptolemy II, the entire codebase is automatically rebuilt in several configurations every night. (Among other things, we need to build it with two versions of the Java compiler.) The output of the compilers and other tools is parsed to find errors in the build, and these errors are included in an email message sent to the whole group each morning.

Building every night helps identify configuration management issues earlier in the release cycle, which in turn makes the process of generating the release much less of a last minute rush. For example, the nightly build uncovered pathname length problems that were cropping up when the release was extracted with the Solaris `tar` program. In the past, similar bugs have only been detected in the alpha or beta releases. Building the release every night also makes it very easy for us to ship snapshot releases to off-site developers.

Regression tests are run together with the nightly build. Every package in Ptolemy II has a test directory, containing test scripts written in Tcl and using the Tcl Blend (see `http://www/scriptics.com`) package to create and exercise Java objects. We use Sun's JavaScope tool to generate code coverage statistics from the test runs, and generate annotated web pages that show the code coverage for each file. The web page that summarizes the code coverage on a per file basis is also annotated with its code rating (see Section 2.4). This page is discussed every week in group meeting so that everyone knows what the code coverage and rating statistics look like.

2.3.1 Introducing the daily build and smoke

In terms of mechanics, the daily build and smoke test was introduced simply by doing it: we wrote the scripts that perform the build and smoke test and generate the output email and Web pages, and turned it on. In terms of creating an understanding of issues such as the reason for the build and smoke test and the importance of not breaking the build, the process was less straight-forward.

To engender an understanding of the importance of the daily build, we held a study group where we read sections from McConnell's *Rapid Development* [8] and McCarthy's *Dynamics of Software Development* [6]. This was useful background material, but more important than this study group was an on-going effort to communicate the importance of a working build. Things we did to engender a spirit of shared ownership of the build were:

- When the build email shows broken compiles, figure out who checked in the code that broke and ask them to take a look at it.
- Buy a silly hat[1] and write "I BROKE THE BUILD" on it, and (humorously) threaten to make people wear it. Of course, we don't actually make anyone wear it, but its presence (and the fact that senior group members *do* wear it occasionally) emphasizes the importance of the issue in a light-hearted way.
- Complain a *lot* when we had to fix the build ourselves.

At the time, we already had an established infrastructure for writing test suites, so the smoke test tagged along with the daily build quite easily. Again, the physical mechanism for introducing regression tests is more straight-forward than the social mechanisms. One of the key problems is that most of us have simply not been in the habit of writing test suites as we develop code, yet this habit is one of the most valuable skills that a developer can have.

Apart from having a framework for writing test suites that is integrated into the makefile and build system, we have found that increasing the visibility of the test suites helps enormously. The code coverage tools generate coverage statistics in HTML, which we have modified to show the rating of each class. Particularly in the lead-up to a release, we run through these pages at our weekly lunch meeting, and highlight classes or packages that are short on tests or coverage and need attention. Of course, 100% coverage does not mean fully tested, but it is at least a clear goal.

2.3.2 Benefits of the daily build and smoke

The build and smoke test has had two key benefits. Although we have not instituted a check-in test – one that would require running a set of regression tests *before* checking in – the group members are now aware of the importance of keeping the build working. When the nightly build email shows that there is a problem, group members will generally fix the problem immediately, and then notify the rest of the group. This kind of concern is very confidence-inspiring!

1. For those who care about these things, we bought Cartman's hat, from "South Park." About twenty dollars at a novelty store.

Second, the regression and code coverage statistics have improved markedly in recent months. The lesson we have learned here is that simply making people aware of a) what the standard is and b) how their packages fare relative to that standard, encourages a more professional approach.

2.4 Code rating

One of the most important yet neglected factors in producing functioning research software is the ability to answer the question "Where are we now?" Our code rating system classifies code based on its progression towards an ideal quality standard. It acts as a framework for quality improvement by peer review and change control through improved visibility.

Code is rated on a per-class basis at four levels of confidence: red, yellow, green, and blue. The basic idea is that a package or class starts at red, and advances to blue. In our process, we have tied the rating to design and code reviews, but other processes could use different mechanisms to advance code through the rating levels. In more detail, the rating levels are:

- *Red*

  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*

  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*

  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*

  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.

Each class file contains a tag, `@AcceptedRating`, that marks its rating. The rating given by this tag appears in Web pages generated by the nightly build scripts, providing an at-a-glance indication of our overall level of confidence in the codebase.

Although we track the code rating of classes individually, it is very useful to indicate the aggregate code rating of packages as well. We generally view the code rating of a package as the lowest rating of any class contained within the package.

2.4.1 Introducing code rating

The idea of code rating was motivated by our realization that we were, at one point in one of our projects, spending a lot of time redesigning each other's code. To counteract

this, we invented the rating system. By visibly indicating our confidence in the quality of a class or file, we hoped to reduce the likelihood of unnecessary rework. Since then, we have formalized the rating system and integrated it with design and code reviews.

2.4.2 Benefits of code rating

Code rating has two key benefits:

1. It provides an instant summary of the perceived quality of a class or package.
2. It provides a vocabulary for summarizing the quality of a package. For example, we might say, "These packages need to be green before we release them," or "This set of packages has been red for a long time; we need to start moving some of them to yellow."

Given its simplicity, code rating combined with reviews is surprisingly effective, and is not something that we have seen published elsewhere.

2.5 Formal reviews

Formal reviews are a technique for improving software quality by peer review. By "formal," it is meant that the reviews are performed in a well-defined way, with clear goals and written results. The specific technique we use for reviews is generally known as an inspection, and was originally published by Fagan in 1976 [2]. A concise summary of inspections and other types of review is contained in chapter 24 of *Code Complete* [7]. A complete and detailed description of the review technique that we have evolved is contained in Section 2 of this report.

The basic premise of a peer review is that it will show defects that would not have been found solely by the author. For this process to be effective, reviews have a formalized structure that encourages defect detection, and discourages defensiveness, side-tracks, on-the-spot redesigns, and other barriers to effective completion of the review. Several points are key to a successful review. First, reviewers are expected to read the review material before-hand. Second, the emphasis of the review is on finding defects in the material at hand as a matter of fact and not as a reflection of the author's level of skill or ability. Third, reviewers are not permitted to discuss solutions to problems during the review itself, but must remain focussed on defect detection.

Each person in a review has a well-defined role. The key role is played by the *moderator*. The moderator's main task is to keep the review focused and moving. In particular, the moderator must cut short attempts to discuss solutions during the review itself. Other important tasks of the moderator are to organize the review and to start and finish it on time. In our reviews, the moderator also acts as *reader*, paraphrasing the design or code to focus the reviewers' attention on it.

The *author* is there solely to answer questions from the reviewers. Depending on the reviewers and the kind of review, the author may need to do more or less explanation. It is important that the author not attempt to defend his or her design or code during the review; rather, if the reviewers decide that they have found a defect, the author acknowledges the defect and moves on. The *scribe* has a critical role, which is to note the defects raised during the review. Finally, *reviewers* are there to find defects, both in

preparing for the review and during the review itself. (The moderator and scribe are also reviewers.)

Following the review, the scribe distributes the review notes (in our case, they are placed on a Web page). The author is then required to perform rework to address the defects raised in the review. Here, another key point: the author has responsibility to decide what to do about each defect. If he or she believes with good reason that a defect raised in the review is not in fact a defect, then it is sufficient merely to note the reasoning in the rework notes.

There are two key areas where we have not followed recommended practice. First, we have not implemented metrics that enable us to measure improvement in our quality. One reason for this is simply that we haven't felt that the cost of implementing metrics would pay off. Another reason is that we are a relatively small group of developers, so it is doubtful that we would be able to gather meaningful metrics anyway. Nonetheless, this is something that we should revisit.

The second area in which we have deviated is that we do not use checklists during the reviews. Although we started by adapting published checklists, we found that we ended up just not using them. We do, however, have preparation checklists for the author and moderator to ensure preparedness for the review. Again, this is an issue we should revisit to see what we might have missed.

2.5.1 Introducing reviews

We introduced reviews with a short series of study groups. In the first, we read Chapter 24 of McConnell's *Code Complete* [7], and a selection from the NASA *Formal Inspection Guidebook* [9]. Following that, we performed a mock code review of some of our code, using the checklists from the *Formal Inspection Guidebook* as a starting point. We repeated the process with a design review.

Having done these small test cases, we decided reviews would probably be beneficial, and started reviewing small parts of the embryonic Ptolemy II code. Somewhere along the way (we don't remember exactly – all of these techniques were introduced gradually into our development process), we incorporated design and code reviews into the code rating system.

Two final points are worth noting. First, we recall there being a lot of pressure to modify the format of the reviews. It is important not to let the reviews become too "comfortable" – keeping the reviews short and focussed is key. Second, reviews can be quite fun at first, with the role-playing enhancing the experience. It is important to keep a focus on the non-antagonistic nature of the review.

2.5.2 Benefits of reviews

Design and code reviews are immediately perceived as beneficial by most people that have tried it, because of the feedback they receive on their design or code. In the longer term, we have found that reviews conducted on a regular basis within a research group have a number of benefits:

- Improved quality. This is of course the primary goal of reviews, and the exposure of designs and code to criticism in a structured way almost always improves quality.
- Increased awareness and communication. By attending reviews of other people's work on a regular basis, group members become aware of others' work. They discuss and work on designs together more than in previous projects that did not have reviews, and there is less duplicated effort.
- Building a culture. Particularly in the early phases of a reviewed project, group members spend a certain amount of time querying style issues, coding standards, documentation requirements, and so on. The decisions made during these early reviews rapidly become part of the group's development culture.
- Dissemination of experience. Newer members of the research group rapidly become aware of expected standards by attending reviews. Reviews are an excellent means by which all group members, and in particular the less experienced programmers, gain design and programming knowledge.

3.0 The review process

This section is an expansion of Sections 2.4 and 2.5. It describes the review process that we have evolved for improving the quality and visibility of our code. We stress that our process is lightweight, and has very little overhead other than what is involved in performing the design and code reviews. It is essentially very simple: design and code reviews advance classes and packages through the code rating levels.

As much as possible, we try to build the attitudes and culture rather than rules and rigid structure. Nonetheless, we do have documents that look like tedious rules and check-lists – but we emphasize that these documents simply codify sequences of steps that we have found worked for us, and are intended to help make reviews as effective and productive as possible. These check-lists are reproduced in Appendix A. These documents are not perfect, and we are now (following the 1999 Ptolemy mini-conference) in the process of refining them again.

It is rather difficult to present the complete process, as so much of it is embodied in the way that group members interact and their attitude towards development. In the next few sections, we will give an overview of our process and then annotate and clarify the most important points. All of our current development is done in Java and so emphasizes packages (groups of classes), but it can probably be applied without much difficulty in other languages.

3.1 Overview of the process

Consider the following scenario.

Frank, an enthusiastic software developer (aren't we all?), writes a fabulous piece of foobar code. He tests it, installs it in the public repository, and then keeps working on other code that uses foobar. A few weeks later, Ernest starts using Frank's foobar code, and discovers that it has some design flaws that weren't apparent to Frank. Ernest, in the spirit of fixing problems when they're found, adds some code and moves some other code around to fix the flaws, and keeps on working.

A few days later, Frank is suddenly very unhappy. Firstly, his build broke because of a couple of small changes Ernest made to the types of a couple of methods. Easily fixed. Then he discovers that he is going to have to rework a lot of his code because Ernest redesigned part of the internal architecture of the foobar, which showed up in the order of method calling and the way event callbacks were handled. Eventually, the whole system is functioning again, but not without some acrimonious email debate, code rework, and perhaps some damage to the team dynamics.

How could this have been done better? Ernest could have tried to work around the flaw – but ultimately that's a negative impact on quality, and Ernest clearly felt that it was unacceptable. Or a change control "committee" could be established to evaluate the impact and cost/benefit of changes such as Ernest's. OK, but that's only half the story – the other half is that a peer review could have solved the problem before it occurred. Ernest (and others) could have voiced their concern about the design flaw at a time when fixing it would be relatively cheap. Either the flaw a) would have been fixed, or b)

it wouldn't have been fixed, but the reasons for not fixing it would be documented and – more importantly – accepted by other developers as a necessary decision in light of other constraints.

Our process is a simple framework for combining these two aspects: quality improvement by peer review and change control by improved visibility. It was designed as a light-weight technique for small teams – other than the cost of performing design and code reviews, it is very cheap. It is not intended to *prevent* change or creativity, since change is an essential part of the software development process. Ultimately, the framework is about *confidence in quality* and *commitment to stability*.

3.1.1 Rating levels

As described in Section 2.4, every package or class is assigned a *rating* indicating the amount of confidence we place in it. The rating levels are *red*, *yellow*, *green*, or *blue*. The basic idea is simple: a package or class starts at red, and advances up to blue by a series of reviews. At each step, the package or class author proposes an increase in its rating and finds a volunteer to act as review *moderator*. The moderator is responsible for organizing the review and advancing the rating after the issues raised in the review have been resolved.

Although every class has its own code rating, the classes within a package are usually very dependent on each other. Because of this dependency, we find it most useful to review all of the classes within a package at about the same time. This ensures that the code within a package has a consistent rating, and also reveals more of the interaction within the package. From now on, we will simply refer to the review process as being applied to entire packages.

The interpretations of the four rating levels are:

- *Red*

  The package has not been reviewed. The code may change without warning.
- *Yellow*

  The package has passed design review. The API is not likely to change significantly.
- *Green*

  The package has passed design review and code review. The implementation is not likely to change significantly, although it may not be optimal.
- *Blue*

  The package has passed final review. This means that we are confident that the code has reached a high level of quality, and we have made a final commitment to maintain backwards-compatibility.

3.1.2 The review procedure

The procedure suggested here is simple and easy to remember. For each level except red:

1. The author announces the package ready for review.
2. The moderator organizes and moderates the review.

3. The author responds to the issues raised in the review, redesigning or reworking as necessary.
4. The moderator signs off on the changes, and announces the new rating.

As a general rule, we believe the process will work better if the author initiates reviews, as the commitment to completing the review will be higher. However, new group members often need a little bit of encouragement to initiate reviews. Also, it is important that not too much time pass between the review and the rework, or the author will lose motivation and the review notes will start to lose meaning.

The most crucial part of the review is, of course, the reviewers. But how many reviewers should be present for an effective review? In our process, four is pretty much the minimum (author, scribe, moderator, and one additional reviewer). Any less would require the moderator and reviewer to split their attention between the topic and the mechanics of the review. (This is not to imply that the moderator and scribe are prevented from raising issues during the meeting – in fact, we rarely have reviews during which the author himself does not discover defects.) In general, about four to six people seems to be a good balance between having enough reviewers, and not wasting the reviewers' time – with more reviewers, there is a sense of diminishing returns, since are only a certain number of issues to be brought up. We have, however, run effective reviews with as many as ten people.

3.1.3 Requirements to enter review

Here are the requirements for entering a review, which we have found to work well:

- *Yellow*

  If a subsystem or package is being reviewed, then the review requires detailed design and API documentation, including UML diagrams and explanatory text. If one or a few classes with well-defined or obvious roles are being reviewed, then an API specification or `javadoc` output is generally sufficient.

- *Green*

  Green is a fairly large jump from yellow. A rating of green means that other developers are going to rely on the code, so entering a green review requires a test suite that is substantial enough to give the reviewers confidence that the code really does what it says it does.

- *Blue*

  Blue requires that the code has been stress-tested in use by other packages. This requirement improves the chances that a package really does fulfill its requirements, at least as far as its clients are concerned.

3.1.4 Managing change

Change is a fundamental aspect of software development. In *Code Complete*, McConnell suggests establishing a "change committee." Doing so is probably unnecessary for a small team, but the spirit of the idea is important: have someone else evaluate the change to see if it really is worth it. If it is, great – do it. At least you will know it's worth the pain!

Instead of a change committee, we will hold a mini-review. A mini-review is less formal than a full review, and takes much less time. Small changes to reviewed code can be covered at our weekly group meetings, while more extensive changes are handled at a separate review. If changes cannot be reviewed in a timely manner, then the package should be reverted to a lower code rating.

Here are some guidelines on how to approach making changes at each rating level. If the guidelines for change at each level cannot be met, then take the class back to a lower level. This is an action that is acceptable and in some cases probably inevitable. However, it is a very strong signal that the changes need to be evaluated very carefully, since the package has reached its current level by a careful review process. There is one overriding rule for all changes: *code checked into the public repository must not break the build* – changes that break the nightly build disrupt all work and are unacceptable.

- *Red*

  Changes are expected. If you know that other code already uses red code (if you have any), then you should at least notify whoever is responsible for that other code. Demos and test suites might be examples of code that depends on red code. Other than that, any changes to red specifications or code are acceptable.

- *Yellow*

  Minor changes to the interface of a yellow package are expected and acceptable. Major changes, such as reworking the collaboration of some of the classes in the package, are issues that should have been detected in the design review before the package went yellow, and thus require a re-review. In any case, major changes to a yellow package are to be considered with some care. Since the package has already been reviewed you need good reason to make very substantial changes.

  Change in the implementation code of a yellow package is expected as part of the development process.

- Green

  Changes to the interface of a green package are to be considered very carefully. A substantial amount of code may already depend on this interface, and adding new methods is better than changing old ones if it can be done without destroying the conceptual integrity of the package.

  Changes to the implementation of green packages are normal and expected, provided that the semantics behind the interface do not change. If they do, then take care to ensure that code that depends on this package will not break or is updated.

- *Blue*

  The interface to a blue package must not change in an incompatible way. Changes that extend the existing interface without changing its semantics are acceptable, but these changes should themselves be reviewed in a yellow-green-blue sequence. Implementation code can change, but this should be done only for bug fixes or for planned performance enhancements. *All* changes to blue code must be reviewed again to maintain confidence in the package.

  A blue package cannot be taken back to a lower level, as doing so would break the commitment to backward compatibility implied by blue.

  Remember that a blue package represents a substantial investment in intellectual effort, not to mention sweat.

There are other ways with dealing with change at more fundamental levels. Designing a system with change in mind will make it easier to accommodate large-scale changes with minimal disruption to existing packages. Consider implementing wrappers, facades, adapters, or (even) subclasses to deal with change instead of making major modifications to code that has been thoroughly reviewed and tested.

3.2 The process in perspective

There are three negative reactions we have experienced from people who first learn of our process. The first is that any process is too expensive or too rigid or too ineffective for a research environment. The second is that it's defective because we didn't implement, say, bug tracking or improvement metrics or name-your-favorite-technique-here. The third is that we have a waterfall process (red-yellow-green-blue sounds a bit like a waterfall model).

All of these are wrong.

Firstly, the goal of the whole process is to improve the practice of software development *in our environment*. In our case, that means academic research, a high level of innovation, and people who are not necessarily experienced software developers. If it doesn't produce results in that environment, then we won't use it. Although we have gone through some trial and error, and have further to go, the year or so since we started learning about design reviews and the other techniques summarized in Section 2.0, and putting them into practice has shown enormous benefits. The responses to the survey summarized in Section 3.4 show an overall very positive response to the process.

The second objection noted above is the other side of that same coin, and indicates a mind-set that we have been careful to avoid. We do not implement some practice or technique because it's "software engineering," we do it because we think it will make it faster and easier for us to produce better software that show-cases the results of our research. We hope that we can maintain this attitude, and be flexible enough to try any technique that looks promising, and discard any technique that – after trying with good intent and some perseverance – proves to be ineffective for us.

The third point is understandable, and arises from the apparent linear progression of the rating codes. But that is not the way they are supposed to work. Firstly, recall that the intention of the rating levels is one of *confidence and commitment*. If we say that certain classes are green, that means that we have high confidence that they are of a certain quality and stability, not that the implementation is cast in stone. Obviously, as we work on a piece of software, our confidence in its quality increases and our commitment to maintaining this quality increases, and how could it be otherwise?

Of course, change is inevitable in the life of a piece of software. Even in code that is green, thoroughly tested, and has been used daily for weeks, we discover bugs. Most of these bugs are conceptual errors: a new piece of research is using the package in some new way that shows a flaw or limitation in the original conception of the way those classes operate. So we fix it, and review the fix. We don't have any hard guidelines on how to do this, we just expect each of us to use our judgment, and to discuss the issues with others in the group. In extreme cases, we will take code that is green back to yel-

low, and start again from there. (We have also taken code that is yellow back to red.) This is not incompatible with the process, it is part of it.

Some other points are worth noting:

- We do not feel compelled to rate every piece of software we produce. In particular, code that was designed and written before we started the process is basically exempt, because we don't have time to retrospectively review it.
- We place more emphasis on reviewing kernel code than domain-specific actors (another way to put that: we place more emphasis on reviewing the application framework than the clients of the framework), because the kernel code is so much more critical than individual actors.
- In general, we try to consider packages as a whole as being at a given level. This is often not possible, as packages grow and change, but it is a good goal to aim for, especially for packages that a lot of other code depends on. For the same reason, it is better to prepare a whole package for review (in two or more review meetings) than to review it in bits and pieces.
- In general, code should not have a rating substantially higher than other code that it depends upon. For example, if package A uses package B and B is still red, then make B at least yellow (so its interface is somewhat stable) before making A green. A similar principle applies to inheritance hierarchies.

Finally, although we encourage others to try this process, we don't expect anyone to adopt it wholesale. Take the parts that work for you, add other techniques that you think will be effective in your environment, and keep refining it.

3.3 Additional notes

One of the crucial aspects of our efforts to introduce a software process and culture is to constantly strive to find ways to make the process more efficient and enjoyable. To highlight this aspect, here are some specific mechanisms that we have introduced, which work well in our environment. First, things that make reviews more efficient:

- Pre-allocate two time slots per week for reviews. These time slots are decided once at the beginning of each semester as time where most people in the group will be likely to make a review. When a review is announced, it always goes into one of these slots (except in the rare cases where we have more than two in one week). The reduction in frustration when trying to schedule reviews is dramatic!
- Create detailed “how-to” checklists for moderator and author. These lists are reproduced in Appendix A. The check-lists are intended to keep each review “on the rails,” and when they are used are quite effective. It is not clear how to encourage more consistent usage of the check-lists.
- Choose a scribe beforehand, and scribe into the computer. This ensures that the scribe is prepared in advance to scribe the review notes, and that the review notes are available on a web page immediately after the meeting.
- Require that the review material is stable. Early on, we had some reviews in which authors changed material under review the day before the review. As a result, the review meeting had two or more different versions of the material, which makes

effective reviewing impossible. We now require that, once the review is announced, the review material not change until after the review.

- Don't try to review too much. There is a temptation to "get through" a whole package or set of classes. Resist it. In our experience, reviews that run two hours or more are less effective at the end anyway, and it becomes much harder to get reviewers to commit to meetings that might run on endlessly. Now we generally schedule reviews for 60, 75, or 90 minutes, and the moderator is expected to stop the meeting on time, and schedule a follow-up meeting if necessary.

Here are some mechanisms that we have found make the reviews more useful:

- Use UML and `javadoc` for all design reviews. In general, the better the quality of the UML diagrams in a design review, the more time is spent focussing on the design itself rather than the trivialities of the presentation. We note, however, that focusing on the `javadoc` for design reviews is a potential trap. If an author has not kept the method comments in sync with the code, the design review becomes pointless because we are not reviewing the actual design.
- Create a Web page for each review. The Web page lists the reviewers, start and end time, and has pointers to all material to be reviewed. During the review, the scribe adds defects and issues raised at the review; during rework, the author annotates each defect or issue with a response. The Web page thus serves as a concrete artifact and record of the review.

3.4 Evaluating the process

Perhaps the most important ingredient in making any kind of software process successful is constant monitoring and improvement. Because there are many other demands placed on members of a research group – taking or teaching classes, basic research, writing papers, reports, and theses, and so on – any part of the process that feels unproductive or unnecessary will naturally tend to get ignored. It is thus critical to always aim to minimize overhead and maximize perceived benefits.

We have had two major points where we took a step back and looked at our process with the above in mind. For the first, we did a "review review" – that is, we subjected the review process itself to a review. Four members of the research group (other than the "author") attended the review, and raised over thirty potential defects and problems with the review process. (Because of the nature of the material being reviewed, we also allowed suggestions in that count.)

The second assessment was an informal survey conducted by John Reekie in preparation for writing this report. All of those who have been actively involved in the review process and who are still at UC Berkeley responded. Responses were considered to be anonymous, and so none of the responses published here include names. The full set of responses is reproduced in full in Appendix C, and summarized below.

3.4.1 Summary and analysis of survey responses

The survey consisted of fifteen questions intended to elicit opinions about the process. We tried to encourage frank responses. We were surprised by some of the responses, and there are clearly some aspects of our process that we need to keep working on!

1. *At how many design reviews have you been present as author?*

   Responses ranged from zero for the newer students through to five. Generally, each Java package takes one or two reviews.

2. *At how many code reviews have you been present as author?*

   Most people have not had their code formally reviewed yet. This partly indicates the greater importance we place on design reviews, but also the fact that we are still relatively early on the development curve. Edward Lee has led the way with code reviews, at a count of three.

3. *At how many design and code reviews have you been present as a reviewer?*

   Responses ranged from one to over 10 for students, and ten to fifteen for senior group members. Everybody in the Ptolemy group has attended at least five.

4. *What were the benefits of the reviews your designs went through?*

   Benefits listed included better design and more confidence, feedback about documentation and naming issues, development of a coherent style within the group, and affirmation of a sound design. In general, authors felt that the design reviews exposed design flaws and improved the quality and soundness of their designs.

5. *What were the benefits of the reviews your code went through?*

   Those who have had their code reviewed listed confidence, improved coding style, catching of subtle errors, and code cleaning.

6. *What were the costs/downside of the design and code reviews? (Time to prepare the reviews, time needed to rework your design into someone else's design, disruption of your regular workflow, discouragement caused by negative feedback, discouragement caused by the extra time involved, etc etc.)*

   This questions elicited a number of responses, but the most common was the time required to prepare for the reviews and to perform rework following the review. Most of those who raised this issue also remarked that the cost was paid off by the benefits of correcting designs before the package became more widely used. One respondent remarked that the benefits of the rework were not obvious to him, indicating that the purpose of the rework is not fully understood.

7. *List some good points about the* way *in which the reviews that you were present at were conducted.*

   Many respondents liked the formality of the reviews and the way that the assigned roles and the policy of not discussing solutions kept the review on track and focussed on the review material. One remarked that the reviews were good for not getting bogged down in details.

8. *List some bad points about the* way *in which the reviews that you were present at were conducted*.

   There were several issues raised in response to this question. The most prominent (that appeared in answers to other questions as well) was that some reviews got bogged down in trivialities, such as typos and small changes to the documentation. One respondent said " if an API is solid the reviewers should just acknowledge that it is solid."

   A second important issue raised is the quality of the notes taken by the scribe. This appeared in several places, and it is clear that we need to work on ensuring that the scribe is able to note raised issues precisely.

Other problems raised included: lack of *positive* feedback, authors not caring to respond effectively to issues raised in the review, and reviewers that do not prepare for the reviews or are unfamiliar with the overall architecture of the system.

9. *What differences did you note in the conduct of the reviews at which Edward was present? (Or me, or Christopher)*

This question was intended to explore a concern that we had early on that the presence of a professor (Edward Lee) at the reviews would count as "management." Quite the reverse seems to be true: most respondents felt that the presence of a senior group member at the review improved the review enormously. This indicates two things: i) that the right kind of participation in reviews by a professor and senior researchers can have an enormous benefit, and ii) that we have to be careful not to allow this to become an ingrained procedure, as inevitably it is not scalable. Since most of the members of the research group are in fact relatively early in their post-graduate careers, we hope that they will become stronger reviewers as they gain experience and confidence.

10. *Would you prefer more or fewer reviews (or the same number)? If so, why?*

This question was read as "reviewers" in some cases. In either case, it appears that our current number of both reviews and reviewers is about right. Two respondents noted the trade-off between the number of reviews and the time costs. Edward Lee felt that we should have more reviews, as it is a more effective use of his time than reworking poor designs or code himself!

11. *Do you think reviews should be conducted earlier or later in the development process? If yes to either, when and why?*

Some respondents suggested that reviews be done earlier so problems are found earlier. Others suggested that reviews be done later, so that the reviews spend less time working through obvious flaws and rough code. Several remarked that the point in the design process at which a review is conducted should just be left up to the author.

We are considering introducing an early design/architectural review that looks only at UML diagrams and not at API documentation.

12. *Do you think testing and the review process should be more integrated?*

Responses were divided into "How?" and "Yes!". Overall, it appears that most people are interested in reviewing test cases or in exercising the test cases as part of the reviews.

13. *How would you improve the review process? (This can include removing things as well as adding or changing things.)*

Responses here raised issues also raised in other questions (better notes from the scribe, better preparation from the reviewers). Other suggestions were that we should follow our review checklists more closely, and to have senior people in the group spend more time in reviews.

14. *How would you improve the way that software is developed in the Ptolemy group in general?*

There were two interesting suggestions. One was that each key package have two primary developers rather than one. Another was that we make better use of automated testing software.

15. *Do you have any other comments?*

Responses were extremely positive about the impact of the reviews on the quality of our code output. Two responses are “I really think that our review process has had a noticeable effect on the quality of code that has come out of the group recently. I very rarely look at Ptolemy II code that has been reviewed and see deficiencies in it, which seems to be a regular occurrence when looking at some Ptolemy Classic code” and “The review process is a great way to encourage/ensure high-quality software. I'm really impressed, surprised even, by the effectiveness of this approach.”

4.0 Concluding remarks

It is difficult to quantify the degree to which our software development has improved as we have adopted the techniques described in Sections 2 and 3. One measure of improvement is the quality of documentation: the Ptolemy II kernel manual is far clearer and more useful than earlier manuals for this level of code. A second is the lower rate at which developers encounter problems in our codebase. Although we have not yet officially released Ptolemy II, feedback from people who have used the developer releases has been extremely positive.

The intangible benefits are also substantial. We believe we now have more of a culture of professionalism, where group members are working together and helping each other identify bugs, propose fixes and build on each other's work. Members of the Ptolemy II team are much more aware of each other's work than the Ptolemy Classic team was.

Part of this change in our development culture has been a shift to finding and fixing problems much earlier in the release cycle; as a result, we are far more likely to identify code that needs more work in time to actually do the work. With earlier projects, we were surprised several times with problems in the implementation of certain features and packages, which we think would have been avoided if the code had been checked into the public repository, reviewed, and tested as we do now.

Apart from choosing the right techniques and adapting them appropriately, we have realized that one of the key ingredients in bringing about these changes has been visibility. Software is an intangible thing, so it is easy for developers to insert all kinds of cruft into the codebase and never quite get around to finishing it! Mechanisms such as the nightly build email, and the web pages with the build and test results and code coverage statistics, all contribute to an awareness and concern with the state of our product. Reviewing (informally) these pages on a regular basis highlights problem areas, and also rewards developers that have taken the trouble to have their code reviewed and to test it.

Finally, we note that introducing these techniques requires commitment and involvement by senior personnel. Because the real goal is a change in culture, having the research group leader deeply involved in the process, both leading by example and (where appropriate) demanding that the standard be maintained, has been a key ingredient in making these changes.

Acknowledgements

The work described here is also the work of all of the people who have participated in the process. In particular, we thank

- John Davis
- Ron Galicia
- Mudit Goel
- Heloise Hse
- Jie Liu
- Xiaojun Liu
- Lukito Muliadi
- Michael Shilman
- Neil Smyth
- Michael Williamson
- Yuhong Xiong

Most of these people are part of the Ptolemy project, directed by Professor Edward A. Lee and sponsored by the Electronics Technology Office of the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO Program, and the following companies: the Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications (now merged with Raytheon), Motorola, NEC, and Philips.

Michael Shilman and Heloise Hse are part of the JavaTime project, directed by Professor Richard A. Newton and sponsored by the Defense Advanced Research Projects Agencies (DARPA), Hewlett Packard and Hughes Research Laboratories.

References and Further Reading

[1] Grady Booch, James Rumbaugh, and Ivar Jacobsen, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[2] Michael E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* 15, nr 3, pp 744 - 751, 1976.

The original paper describing formal inspections. Historically interesting, but Chapter 24 of *Code Complete* is more useful now.

[3] Daniel P. Freedman and Gerald M. Weinberg, Handbook of Walkthroughs and, Inspections, and Technical Reviews, Dorset House, 1990.

Essentially structured as a long FAQ, this rambling set of course handouts is a good reference but is not essential.

[4] Martin Fowler and Kendall Scott, *UML Distilled*, Addison-Wesley, Inc., 1997.

A concise and frank introduction to UML. Recommended if you haven't already been exposed to UML or if you don't have time to study it in detail.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, October 1994.

The original object-oriented patterns book. Highly recommended.

[6] Jim McCarthy, Dynamics of Software Development, Microsoft Press, 1995.

A collection of "rules" and anecdotes for software development. Favorites adopted as development mantras include "Don't break the build" and "Get to a known state and stay there." Highly recommended by Reekie, who first saw software development as a social act while reading it; however, a lot of people don't like this book.

[7] Steve McConnell, *Code Complete*, Microsoft Press, 1993.

A wide-ranging treatment of practical software issues, covering topics from coding style to formal inspections to personal character. Many of McConnell's arguments are backed by published studies, and each chapter includes a wealth of references. Very highly recommended.

[8] Steve McConnell, *Rapid Development*, Microsoft Press, 1996.

Although not immediately as appealing as his first book, this book is invaluable for both its frank coverage of topics such as specification and life-cycles, and for its capsule descriptions of "best-practice" techniques. Highly recommended.

[9] NASA Software Assurance Technology Center, *Software Formal Inspection Guidebook*, http://satc.gsfc.nasa.gov/fi/ , 1996.

A good description of a formal inspection process. Includes a collection of check-lists that would provide a good starting point for developing your own check-lists.

[10] Arthur J. Riel, *Object-Oriented Design Heuristics*, Addison Wesley, 1996.

[11] James Rumbaugh, Michael Blaha, et al. *Object-Oriented Modeling and design*, Prentice Hall, 1991.

The original book on OMT. Chapters 3 and 4 are one of the best and most concise introductions you will find to static structure modeling, and translates easily into UML. Other than that, a bit dated now.

[12] James Rumbaugh, *OMT Insights*, SIGS Books, 1996.

A collection of reprints of Rumbaugh's columns from the Journal of Object-Oriented Programming, some of which are very insightful. Recommended.

[13] James Rumbaugh, Ivar Jacobsen, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[14] Karl Weigers, *Creating a Software Engineering Culture*, Dorset House Publishing, 1996.

Common sense guide to creating a healthy development culture. We wish we had discovered it earlier.

A Specific guidelines in the Ptolemy Project

(From our Web pages.)

This document lists guidelines and checklists for the review process in the Ptolemy project. It is based on practical experience with making the process as smooth and effective as possible. The checklists seem long, but we highly recommend that you go through these to make your reviews as smooth and productive as possible.

A.1 The review process

This is the suggested process for conducting a review in the Ptolemy project. All reviews follow this same basic format, which deviates from the documented formal inspection process where we have felt it necessary to streamline the process in our environment.

A.1.1 Organizing a review

The author is primarily responsible for getting the review process started. When an author has code that is ready for review, he or she verifies this by running through the *Development checklist* for the type of review. (See lists below).

The author then finds someone willing to be the moderator and they agree on a suitable time. The first-choice times as of February 1999 are **Friday 1:30 – 2:30** and **Monday 3:00 – 4:00 pm**. The author then runs through the *Review preparation* checklist. When completed, the moderator announces the review and requests reviewers.

Here are the steps:

1. Author steps through *Development* checklist
2. Author finds moderator and they schedule a date
3. Author steps through *Review preparation* checklist
4. Moderator announces review and calls for reviewers

A.1.2 Conducting a review

The review requires four roles:

- The **moderator** runs the meeting. In our process, the moderator also acts as **reader**.
- The **author** answers questions about the design or code.
- The **scribe** notes down issues and defects raised at the meeting.
- The **reviewers** are everyone else. (The moderator and scribe are also reviewers.)

Points to note:

- The scribe is to be determined before the meeting, should the scribe want to bring a laptop to use.
- Scribing directly into a computer is preferred, since the review notes will be available much sooner.

- The scribe must be careful to note the raised issues and defects. Poor scribing makes rework much more difficult and substantially reduces the effectiveness of the review.
- The moderator should schedule a second meeting if one meeting proves insufficient.

A.1.3 Completing a review

The review is not completed until the author has completed rework. To be sure, the author should run through the *Rework* checklist for the level being reviewed.

Once the author completes rework, he or she informs the moderator, who verifies that the Web page has been updated with responses to the review. The author then updates the rating tags in the appropriate files, and announce the updated page. Note that we do not require that the moderator actually verify that rework has been completed properly, only that the Web page has been updated with responses to each issue raised in the review meeting. An author is of course always free to sit down with the moderator or someone else and check some part of the rework.

A.2 Mini-reviews

Sometimes a mini-review is required. A mini-review is typically a good idea if the author has made a significant change to the design during rework, but not significant enough to warrant a full re-review. A mini-review is a very good idea if someone modifies code that has already been reviewed.

We have not codified a process for a mini-review. Use your judgment.

A.3 Moderator checklist

The role of the moderator is crucial for successful reviews. If you are moderator for a review, print out this list and refer to it throughout the review process.

1. With the author, schedule a time for the review.
2. Announce the review and recruit reviewers.
3. Remind the author that review materials must be ready and announced at least three days prior to the review. Remind the author that review materials must be stable for this time.
4. Verify that the proposed material can be covered in the time scheduled for the meeting (75 - 90 minutes).
5. Ensure that enough reviewers of the right type will be present. If not, re-announce the review and/or walk around and ask for reviewers.
6. Make sure that there is a scribe nominated for the review.
7. Send a reminder to the `eallocal`[1] mailing list the day before the review.
8. At the start of the review meeting, state:

   The purpose and scope of the meeting.

   The time at which the meeting will finish. For example, “This is a design review of the Expression package. We are starting at 1 o’clock and will finish at 2:15.”

1. `eallocal` contains all of the developers in our group.

9. During the meeting, paraphrase the review material to focus the reviewers' attention on the right things.
10. During the meeting, keep the reviewers focussed on the review material. Do not allow discussion not directly related to the stated purpose of the meeting.
11. Finish the meeting on time. Schedule another meeting if the review materials were not completed.
12. At the conclusion of the meeting, thank the participants.
13. At the conclusion of the meeting, remind the scribe to update the review Web page with noted issues within 24 hours.
14. When the author informs the moderator of completion of rework, verify that the Web page has been updated with the author's responses.

A.4 Author checklist – yellow

Yellow is a commitment to an initial architecture and design. Yellow requires a description of the design, UML static structure diagrams, and `javadoc`-produced API documentation.

A.4.1 Development

1. The package has a design document.
2. The design document includes a complete static structure diagram.
3. `make docs` runs with no errors or warnings on classes scheduled for review.
4. All methods produce documentation in the `javadoc` output.
5. Review the output of `ptspell *.java`.[1]

A.4.2 Review preparation

1. Probable review date and time agreed with moderator.
2. Web page created for review and under version control.
3. Web page contains HTTP links to:

   Design document including UML

   `javadoc` output
4. Moderator notified of readiness.

A.4.3 Rework

1. Each item on review Web page annotated with a response contained in `<blockquote> ...</blockquote>`.
2. Notify the moderator that the rework has been completed.

1. `ptspell` is a spell-checker that understands, for example, `thisVariableName`. It also has other commonly-occurring words, such as our names, in its dictionary. This makes spell-checking code quite do-able.

3. After the moderator approves, update the `@AcceptedRating` tag[1] of all relevant files and announce the completed rework to the `ptdesign` mailing list.

A.5 Author checklist – green

Green is a commitment to an implementation of the design. Green requires solid and tested code.

A.5.1 Development

1. Run `make clean all JFLAGS=-deprecation` and fix any warnings. If it is not possible to fix the warnings, consider adding the `@deprecated` tag.
2. Review the output of `ptspell *.java`.
3. `make docs` does not produce any errors or warnings.
4. `util/testsuite/chkjava *.java` does not produce any warnings.[2]
5. Run `jindent *.java` to reformat code to Ptolemy group style standards.

A.5.2 Review preparation

1. Probable review date and time agreed with moderator
2. Web page created for review and under version control.
3. Web page lists the classes under review and contains an HTTP link to the design document (for reference, not for review).
4. Moderator notified of readiness.

A.5.3 Rework

1. Each item on review Web page annotated with a response contained in `<blockquote> ... </blockquote>`.
2. Notify the moderator that the rework has been completed.
3. After the moderator approves, update the `@AcceptedRating` tag of all relevant files and announce the completed rework to the `ptdesign` mailing list.

A.6 Author checklist – blue

Blue is a firm commitment to backwards compatibility. Blue requires a complete design, implementation, testing, and documentation. A blue review looks for defects in completed documentation, presentation, and packaging. Currently, we have not had any blue reviews, so we haven't developed a check-list.

1. The `@AcceptedRating` tag contains the code rating level. We are about to make `@Rating` a synonym.
2. `chkjava` is a script that processes source code files looking for problems, such as bad revision control keywords or `javadoc` tags, methods with missing comments, and so on.

B Notes on Study Groups

(From our Web pages.)

We have been running a study group for about eighteen months now, with topics initially chosen from various fields of software development, and then expanding into other areas of interest. Participation in the study group is voluntary. This presents some challenges for the moderator in encouraging interest and participation.

In my view, the prime measure of a "successful" study group is that each participant feels that they got something out of it. Here are some thoughts on what can make this happen.

B.1 Choosing a topic: relevance, concreteness, achievability

Topics need to be something that all participants could conceivably use, if not right now, then sometime. In the same vein, concrete and "hands-on" techniques are more likely to be relevant than abstract or hypothetical ones.

To get the most out of a meeting, participants need to do some preparation – this always makes progress faster and the discussion more lively. To encourage this, the reading material needs to be short enough to be read and understood fairly rapidly. Ten to fifteen double pages, although small, nonetheless seems to be a good size for a study group of this kind.

Participants are much more likely to join and enjoy the study group meetings if what they get out of it is large relative to what they put into it.

B.2 Moderating the meeting: focus, openness, and enthusiasm

The purpose of the study group is to study something new. To aid this, the moderator should encourage a "we are doing it by the book" approach, solely for the purpose of enabling understanding of new ideas without premature rejection.

If a participant disagrees strongly with ideas presented in the reading material, the moderator needs to steer the discussion away from subjects that cannot be verified directly from the reading material. A participant with a strong dissenting voice should be encouraged to find suitable reading material to present at a later study group.

The moderator does not need to be an expert in the topic – in fact, I think meetings work better if the moderator is also learning the material for the first time (although perhaps having done more background research than the others), as this avoids any possibility of the meeting turning into a lecture.

Meetings that focus on the topic and material at hand seem to move faster and be more satisfying than those that wander off into other areas.

Finally, a moderator who is excited about the topic has a much better chance of inspiring the other participants into, um, participating. I have found this surprisingly difficult to maintain. The moderator also needs to encourage an open and non-threatening environment, and to be excited about the input of the participants.

B.3 Participation

The best way to encourage participation in the meeting is to build it into the meeting's structure. Here are some techniques that have worked so far:

Take turns in presenting. Each person is assigned a small section of the reading material, and presents it at the meeting. We used this technique to cover object-oriented design patterns, where we covered seven patterns over the course of two meetings.

Passing the marker. People have the whiteboard marker passed to them and are in control of what goes on the whiteboard. This basically forces each person in turn to take an active role in the discussion. This works particularly well if the topic is about or includes visual notations – drawing the notation on the white-board gives participants a useful “first step” that overcomes the natural resistance to learning new notation.

Role-playing. Each person assumes a well-defined role and plays it out in the meeting. This worked well for the software inspection meetings, in which people were assigned roles such as moderator, scribe, reviewer, and so on.

Round-robin. This tends to be a last-ditch attempt to get people involved. If the topic has a list of some kind, taking turns around the table to assess, explain, or just read each element of the list is still better than the moderator doing all the talking!

Another technique is to have a *case-study*. Case-studies are a little difficult because a) it is hard to find case-studies that are relevant yet small enough to be covered in a single meeting, and b) the moderator may have trouble both getting someone to provide a case-study and for the others to read it prior to the meeting. Nonetheless, when it can be done, a case-study provides a useful incentive and relevance to the meeting.

A key point to remember in all meetings is that the whole purpose is to learn something new. It is important not to be distracted by a case study or the broader context in which the meetings take place (a research project) into turning the study group meeting into a design or technical meeting. The purpose of the study group meeting is to learn – any by-products that happen to be useful, such as design insight into the current project, are great, but must be seen as a happy accident.

C Complete Survey Responses

In preparation for writing this report, we prepared an informal survey to solicit direct feedback from those involved in the development process. We received responses from most of those who have taken part, with the exception of two who have now left UCB. All responses are reproduced below. To distinguish the nature of responses, we have marked those from professor and staff with a dagger. The responses are unedited except for spelling and minor grammatical errors.

1. *At how many design reviews have you been present as author?*
2. *At how many code reviews have you been present as author?*
3. *At how many design and code reviews have you been present as a reviewer?*

The responses to these three questions are summarized in the following figure:

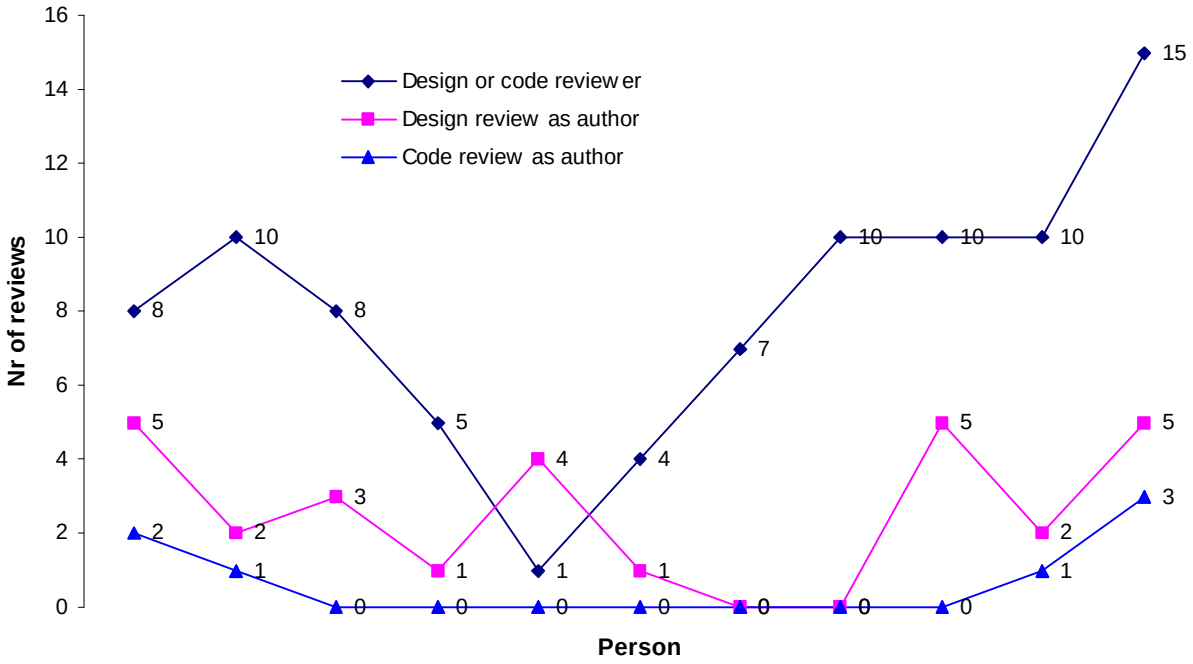

FIGURE 5. Number of reviews attended, as reviewer, author at design review, and author at code review

4. *What were the benefits of the reviews your designs went through?*

   Better design and more confidence.

   I got a lot of extremely good feedback about the documentation and naming issues. This was extremely helpful not only to make the use of the class more clear, but also helped me rethink the way I designed it.

   The reviews revealed quite a few flaws ranging from class structure to method naming. This was very helpful.

One big benefit is a coherent style. This is very important in a large software project, such as ptolemy, especially for new group members. I also view design reviews as an affirmation that your architecture is sound, which gives a good sense of closure on a software package and provides good incentive to move onto other things. Lastly, reviews encourage other people in the group to reuse code that has already been written, through the familiarity that was gained by being a reviewer and the commitment to stability. (I find myself leery to use other peoples working code, even if it is not changing, simply because the commitment to stability is not there.)

a) Forcing function to get documentation in order and really focus on cleaning up interfaces; b) good comments and critiques which drastically improved design.

Inputs from the reviewers are generally very useful and give me ideas of designing certain things differently.

N/A.

N/A.

Better understanding of OO modeling techniques (thus leading to a better design), clarification of the issues central to the problem and revealing of flaws. Also having the reviews forced the design to achieve coherence at that point (instead of just plodding along). Finally, and perhaps most importantly, they also gave me, as an author, time to explain my design, something that may not otherwise have happened except through the Masters report.

† Forced us to clean up design flaws.

† Exposed quite a few design flaws (some of which were not really correctable, but most of which were).

5. *What were the benefits of the reviews your code went through?*

Even better design and more confidence.

My coding style changed (hopefully for the better :-)). The code that I write now is more compact and the names (including the variable names) are more informative.

[No response]

N/A.

N/A.

N/A.

N/A.

N/A.

† We cleaned the code, started writing tests.

† Caught lots of minor errors, and quite a few insidious errors that would not have shown up in more thorough testing but would have bitten us later.

6. *What were the costs/downside of the design and code reviews? (Time to prepare the reviews, time needed to rework your design into someone else's design, disruption of your regular workflow, discouragement caused by negative feedback, discouragement caused by the extra time involved, etc etc.)*

Sometimes, since there is a review arranged say in the next week, I have to stop the current development and wait for the result from the review.

The main downside was the time it took me to convert my design to some other design when the usefulness of the new design wasn't that obvious. Also quite often the design changes would just be the name of the methods, which can be quite difficult to update if you have a lot of methods using them. Regarding negative feedback, etc., I don't think that it is a hindrance at all. It's just that I would prefer if the people who review one package even help review the sub-packages. Otherwise, some of the changes suggested at the sub package levels are conflicting with the packages that have already turned yellow. (Eg. actor.process and pn.kernel). I think it is a lot to do with personal tastes.

The time needed to rework is not trivial. It may take a few weeks for me to find time to do the rework and write the response. But it is worth it.

When doing reviews, it seems that a lot of time is spent preparing material for the review, which often must be rewritten as a result of the review (such as documentation, web pages, etc.) However, I feel this time is very small compared to the amount of time that would be wasted attempting to use packages that were poorly designed in the first place, and the time that it takes to redesign after a release has occurred.

Takes a long time to prepare for the review, and often times the design is changed enough that you might have to undo a lot of that work. In contrast, if you just code the thing, and then use the code some, it's often easy to tell some of the short-comings and you can change your design without having to rewrite comments, etc.

I think the time to prepare the reviews is sort of the downside. There's some stress too.

I think these costs have sufficient payback from the reviews. We can do better in conducting reviews of packages used more extensively earlier, since changes in these packages will cause rework on more designs.

Design/code reviews take time both in preparation and attendance. Often the reviews seemed to ignore issues such as algorithm used and focused more on issues such as comments.

Preparing for review does take a good bit of time, and they also reveal design flaws one would prefer to brush under the carpet...

† It took some time to finish things.

† Time is expensive. But I think these are well worth it.

7. *List some good points about the* way *in which the reviews that you were present at were conducted.*

Don't get into so much details like looking for a particular data structure.

Just discussing the design with a person not familiar with the package brought about many obvious flaws to the design, that wont be quite apparent to the author. The good points were that at some of the reviews the reviewers had actually read the code and had some good suggestions to offer!

I think the format of the review is very good, the job division between the moderator, the scribe, and the reviewer is good.

I think the most important thing about a review is to keep it moving, and not to get bogged down in small points (which can be very tempting). Reviewers must remember that they are there to identify defects, not to fix them.

Very well-moderated and kept on track. I like the formality of the reviews.

I like that we try to keep the review under 1 hour and 30 minutes, so it doesn't drag on. Also if somebody goes off track, the moderator would step in so that review is focused on the right thing. These have been done effectively in the reviews that I've been to.

Lively discussion and exchange of ideas.

John Reekie does a good job of moving things along.

Strong moderator. Also the author sometimes explained aspects of a design that were difficult to grasp.

† a) Getting someone else to look over the code is invaluable. b) The code reviews have prompted me to write test suites.

† The policy of not discussing solutions, when the moderator enforces it, is essential to keeping the process from getting bogged down.

8. *List some bad points about the* way *in which the reviews that you were present at were conducted.*

We need powerful person as moderators.

Quite often, the reviewers weren't familiar with the code or had put in not effort to familiarize themselves with the designs and the suggestions they had to offer were limited to some typos and gratuitous changes to the code or documentation. Also at least in one case the author and scribe was the same person which slowed down the review a lot. The moderator would in some cases let the discussion digress completely.

Occasionally, the scribe does not have enough time to fully understand the point raised, so the wording of some of the points is not very clear.

Reviewers that have read the code, but don't understand the architecture. It is OK to ask questions of the author before the review about "the big picture" a) After the review in which I was reviewer, the author didn't really address most of the points that were raised. b) I think the attention to detail increases with the quality of the code. So that by the time an API is bullet-proof, reviewers are nit-picking over the most trivial little details. I think reviewers shouldn't be obliged to come up with 50 points in every review, and if an API is solid they should just acknowledge that it is solid and it'll be a better use of everybody's time.

I can't think of anything right now.

Often the approach of "say something positive first and then criticize" is not applied. Authors will [know] that their work is appreciated if they are told so.

Sometimes a review would spend too long on one issue.

† We are always rushed for time. Some people are not prepared.

† Sometimes the reviewers and moderator focus on trivialities... Time is wasted.

9. What *differences did you note in the conduct of the reviews at which Edward was present? (Or me, or Christopher)*

They are the powerful persons. Also, they has more designing/coding experience than anyone else in the group, so the feedback is more constructive.

With Edward present, the reviews were extremely effective. Though mainly the feedback came from only Edward. (I think quite often other reviewers - not including you or Christopher - were intimidated and refused to come up with useful sug-

gestions.) Sometimes even the moderator would not try to be assertive. With you and to a certain extent Christopher, the review qualities were certainly enhanced. Without you or Edward, the reviews would normally be a ritual with the author reading the code, the reviewers nodding their head, point out some typos and in a hurry to get back to their cubicles. The reviews in these cases completely lost their relevance.

I think Edward (or you, or Christopher) does not ignore any problems, even seemingly minor ones. So the reviewers pay more attention, which is very good.

Reviews where a strong personality modereadered (A new word!) tended to stay on track better. It is important to take charge as a modereader.

N/A - you were present at all the reviews I was at. Christopher/Edward wasn't present at any of the reviews. The reviews that [other professors] were at were a lot more tense than the ones we did solo.

Don't know, I've only been to your reviews.

Edward: more capable of relating problems under review to “big picture” issues. John: gives very useful comments on software eng. and coding issues, also makes the discussion more lively. Christopher: no chance to observe.

Edward's criticisms come across as harsh sometimes. John Reekie is a good time manager. Christopher generally makes meetings more enjoyable.

I was never at a review where one of the three staff were not present. I guess for my own design reviews I always tried to have one of the three present, partly due to the relative levels of software experience.

† a) Edward arrives late and leaves early, but this is to be expected, he is busy. b) I think that reviews that have two of the three of us are better than reviews where I'm the only one of the three. c) When Edward is around, we do more detailed discussion about the design. If he is not present, then the review sort of turns into a typo finding exercise - big design problems are not really discovered.

† The best reviews had at least one senior person present.

10. *Would you prefer more or fewer reviews (or the same number)? If so, why?*
[Authors’ note: this question was read as “reviewers” in some cases.]

I am not sure. More reviews mean less materials for each one so the reviewers can have a better understanding about the design. And we don't have to rush in the review. But, on the other hand, more reviews means long waiting time. For example, the three CT reviews made me waited for nearly a whole winter break...

About 4 reviewers is just fine. But I think, no review should be held without you or Edward, or at least Christopher.

I think the same number is OK.

A small number of active reviewers is best. Author, Scribe, reader/moderator, and 1 or 2 reviewers. (The scribe should be kept scribing, not reviewing.

Probably more reviews, but mostly that's my fault for not polishing things up. Also see #11 below.

I don't know what's the standard. I think too many reviews of one package may not be necessary. I guess it depends on how big the package is. But too many reviews can be time consuming.

This can be determined by how extensively the stuff to be reviewed will be used: more extensively, more reviewers. For a very specialized package, 3-4 is good. For things like kernel, maybe we should have 5-6.

Same.

Same, it worked well.

† I think that we have roughly the right number. I like having fixed times for the reviews. Having the reviews all over the week makes it less likely that I'll be prepared.

† More. The leverage on my time is very high, I feel. When I'm reviewing other people's code, I think it's more efficient than just rewriting the code, which was my old technique. When reviewing my code, I feel it improves the code considerably, and also helps disseminate both good practice and an understanding of existing code.

11. *Do you think reviews should be conducted earlier or later in the development process? If yes to either, when and why?*

Early and it would be nice to review more than once, from a designer point of view. But from reviewer's point of view, may be later.

I think this should be left to the author. But if there are enough people (or for that matter even one other person) using it, then the review should be quite early.

I think the timing of the review is about right.

I like the time when we do them now, but I also try to get feedback on my design decisions well prior to a formal review. I think that our timing works because we generally have informal design reviews as well (meetings with Edward, talking amongst ourselves, whiteboard discussions, etc.)

I feel it's up to the person being reviewed. My personal bias is towards *later* in the development process. I think I like to have "my best possible" design reviewed, so that every criticism "counts" and I've worked out a lot of the bugs by myself. If I do a design review before things are polished, then it seems like a lot of the comments are things I thought of anyway. And since in general it's not the reviewer's job to discuss alternatives during the review, this is of less help to me.

I think design reviews should be conducted once before implementation, once after some implementation.

Earlier, maybe after the designer has completed his first pass of design, coding, and testing.

The current timing is fine. This issue is really up to the author though.

One of the key things in getting a review to happen is someone, i.e. Edward, John or Cxh, suggesting that a review is due/needed. It seems somewhat haphazard, and there could be benefits in having the design review earlier in the cycle.

† a) If anything, they should be conducted earlier. Lots of people are writing code that it likely to be thrown away. b) The review process breaks down when we are in a crunch for a demo.

† Later is more useful. Reviewing very rough code is extremely difficult.

12. *Do you think testing and the review process should be more integrated?*

It would be nice to look at the test suite at some point of the review, say after design review and before code review.

Perhaps. I think what we need is the code review coupled with testing of the code. The reviewers should get a chance of looking at the test suites and what it does before promoting it to yellow. Leaving the test suites to the author is just not good enough. I think the reviewers need to see whether the test suites actually check the methods or are present just to save the author from some criticism.

I'm not sure how testing and the review can be more integrated.

I think we should start reviewing our tests, after code coverage has been demonstrated, to ensure that good test cases are run.

I think testing should be by committee. More specifically, I think a group of people should design a set of test cases on paper, and then the author (or possibly somebody else) should implement these test cases and make sure they pass.

Hmm... I'm not sure what you mean by integrating them.

Can we cover the test suite in code reviews?

Level green should require a certain level of code coverage by tests.

Nice idea...

† Yes, a thousand times yes. I think we need much better test case coverage *before* code review. Also, people need to run the spell checker and formatter etc.

† I think testing and coding should be more tightly integrated. I still don't understand how people can write code without tests.

13. *How would you improve the review process? (This can include removing things as well as adding or changing things.)*

I don’t know how you can force this, but I would certainly prefer to penalize the reviewers if they haven't given even the least thought to the code being reviewed. Most of the time the reviewers had never even seen the code/documentation till the review and obviously they did not have any good suggestions to offer.

I would like more detailed notes because the notes that most scribes take are very brief and hard to follow outside of the context of the review (i.e. a day later when you are responding to the comments).

I don't have suggestions here, 'cause I've only been to a few and they've been okay for me.

Authors should come with a list of concerns that they have about their code. I.e., “I'm torn between making this algorithm recursive or...” Reviewers can then respond to these issues. It might even be useful for these points to be sent via mail prior to the review.

No suggestions.

† Make people follow the checklists. Perhaps if the checklist was not done, we would cancel the review for that day.

† Find more time for JohnR, Christopher, and me to spend on it.

14. *How would you improve the way that software is developed in the Ptolemy group in general?*

I would suggest that more than one person take charge of a domain. There is a major developer. And a second person who can be fairly familiar about the semantics and the design of the domain. There are usually many subtle things that can not be

resolved (or even explained in the review process). Having a regular people to discuss with during the design is better than after a design.

More rigorous cross-testing?

Don't know.

Use testing software. Perhaps JTest by Parasoft.

Again, no suggestions as I thought the process was pretty good and really helped me better understand software engineering.

† Good question.

† Find more time for me to spend on it.

15. *Do you have any other comments?*

Thanks John!

I think reviews should be more tightly integrated with the testing of the code and the reviewers should be more involved in the reviews rather than being there just to add up to the number.

I really think that our review process has had a noticeable effect on the quality of code that has come out of the group recently. I very rarely look at Ptolemy II code that has been reviewed and see deficiencies in it, which seems to be a regular occurrence when looking at some Ptolemy Zero code. Furthermore, the insistence on reviews has encouraged me (and others, I believe) to be better programmers individually, as well.

The review process is a great way to encourage/ensure high-quality software. I'm really impressed, surprised even, by the effectiveness of this approach.

Good job John!

† You've done a great job with the reviews and other enhancements (UML etc.) to the software development process.

† We are producing the best code ever to come out of Berkeley. I don't mean that as hype. I really believe it.