Software Development

The Softdev SIG is for the discussion of software development issues that pertain to any and all GSRC research groups. This group is open to GSRC members, the primary means of communication is the mailing list.

GSRC Software Development Notes

Bookshelf/Softdev Meeting

Where:
  BWRC
When:
  9-12, with lunch 12-1, followed by the datamodeling meeting
What:

  0900 - 0950    Andrew and Igor present Bookshelf in general + specific (proposed) release standards
  0950 - 1015    Christopher presents the proposed GSRC development framework
  1015 - 1030    Followup discussion as necessary on (Bookshelf perspective) acceptable specific development environments
  1030 - 1045    Break
  1045 - noon    Discussion

Conference Call info

I've set the conference call for 8:30 -12:30 on Thursday

Phone number:     1 888 422 7105
Participant code: 297 393
Host code: 383 983
I'm not sure how many people will be calling in. I set it up so that we are guaranteed 5 lines (1 host line at BWRC + 4 call ins) If we need more, we will get them on a space available basis, or I could change the call.

  • PowerPoint presentation: GSRC Software Guidelines

GSRC Software Development Guide

Overview
Process
Languages
Platforms
Version Control
Style Guides
Build System
The importance of good software practices in academia is not to be underestimated. Software is used by more and more EE research projects to explore research areas. Published software should be looked upon as a form publication similar to publishing a paper.

When students leave the university, they often continue developing software in industry. We should be training students on platforms and products similar to what they will use, and we should train students in modern software engineering techniques where we can. We should favor common off the shelf (COTS) tools where ever we can, and avoid custom, baroque, hard to maintain tools.

The process of releasing research software has many benefits

- Software is actually finished with a specific set of features.
- Documentation and tests are written.
- A stable snapshot is created for users who do not want to ride the tiger of development.
- A release gives a team a concrete deadline, and is a good way to handle the graduation of students.
- Sponsors can see concrete results in the form of software downloads and documentation.

The release effort needs to be intimately associated with the development effort. It can be very difficult to pull together a package that was not developed with a release in mind. Such software is usually poorly documented, has few tests and is very disorganized.

With the advent of 'Internet Time' releases, where companies such as Netscape had two separate development teams working on a leap frog system of releases, the line between release and development has blurred.

If nightly builds are used, then in effect, a release is created every day. The nightly build release can be valuable for quick developers releases to interested parties. If nightly builds are done, then changes that break the build are quickly identified, and developers can quickly fix them.

If GSRC software groups follow a common set of standards, then it will be easier to cooperate and use each other's work. If there are no GSRC-wide standards, then each group will have to spend time determining their own standards for software development. It would be even worse for groups to have no standards, and that are just hacking away as heroic programmers running from fire to fire.

**Acknowledgements**

This document came about from work with John Reekie and Edward A. Lee, and a very good discussion.
Development Process

Below are some notes about systems that can help the development process. Software developers are encouraged to explore these areas further.

Study Groups

John Reekie proposed that the Ptolemy group have weekly study groups, and many of the study groups were about software engineering issues.

The format of a study group is that at the beginning of the semester, individuals proposed topics and we decided as a group what topics we would cover, and who would facilitate the study group. The facilitator then chose the reading material and distributed it to individuals. Everyone was encouraged to read all the material, but each section of the material was assigned to an individual who would make an attempt to really understand the section.

This participatory aspect of the study group is, in my mind, the single most important aspect of the study groups since it promoted involvement by each individual.

We covered a number of different software engineering issues, including UML, design reviews and CMM.

We strongly encourage individual research teams to set up study groups. We chose to do ours from 4-5:15 on Fridays, and then servered refreshments afterwards.

UML

UML diagrams are a good way to communicate and discuss the design of a system. UML diagrams give a development group a common language, where a picture is worth a thousand words.

When UML diagrams are combined with design patterns, then it becomes much easier to identify recurring themes, and for developers to quickly communicate ideas about class structure and how classes interact.

The "Software Practice in the Ptolemy Project," paper has more information about UML.

Design and Code Reviews

The Ptolemy Project system of Design and Code Reviews is documented in "Software Practice in the Ptolemy Project."

Each source file in the Ptolemy II System has a rating

Red

Unreviewed code, interface (method names etc.) subject to change
A design or code review involves the following people:

The Moderator
  who keeps things moving
The Author
  who answers questions about the code
The Reviewers
  who raise issues about the code
The Scribe
  who takes notes about issues raised

The Ptolemy group usually scheduled two reviews a week at predetermined times.

To have a review, the author finds a moderator who in turn finds reviewers and a scribe. The moderator put up a web page with the review material, which usually included the javadoc output, and the source code. Design reviews usually had UML diagrams.

Design and code reviews take an hour to an hour and half.

The point of a design and code review is raise issues about the code, and not to provide solutions.

After the review, the scribe posts the notes on a web page, and the author goes through each issue and either fixes the problem, or explains that the issue is not really a problem.

The literature is full of statistics about the efficacy of detecting bugs by doing desk checks of the code. In the Ptolemy project, we found that many fundamental design flaws were detected early on by having a design review. This early detection helped us fix bugs earlier in the release cycle.

**Capability Maturity Model (CMM-SW) for Software**

The Capability Maturity Model (CMM-SW) for Software, developed by Carnegie Mellon's Software Engineering Institute, is described at [http://www.sei.cmu.edu/cmm/cmm.html](http://www.sei.cmu.edu/cmm/cmm.html)

The CMM-SW consists of 5 maturity levels:

1. Initial - chaotic and ad 'hack' methodology, success depend on heroics
2. Repeatable - project management skill are used to analyze and determine the schedule and cost. The success of previous projects can be repeated with newer, similar projects
3. Defined - the software process is documented, and all projects use an approved version of the process
4. Managed - results are carefully instrumented and controlled
5. Optimizing - feedback from analytical data is used to tune the software process.
Obviously, it is probably not appropriate in an academic environment to require CMM-SW level 5 for all software activities. However, being aware of the different levels can pay off in terms of higher quality product, and fewer last minute all nighters.

## Languages

Java is the preferred language for GSRC Software, though C++ is acceptable.

### Java

Software written in Java should run under the most recent released version of the Sun JDK.

**Rationale:**

- Java software is somewhat more portable between platforms than C/C++ software, though 'write once, test everywhere' applies.
- Java includes useful features that are not found in a portable fashion in C++: garbage collection, packages, array bounds checking
- Java includes off the shelf packages for things like database access, user interfaces, graphics, sound, CORBA
- We encourage Java developers to create high quality applet demonstrations of their work. Applets provide an easy way to let web visitors browse software packages without downloading and compiling packages. Ideally, Java applets should run in a vanilla JDK1.1 compliant browser. However, Java applets may also use the JDK1.2 plugin

### C++

The preferred C++ compiler is the most recent version of the GNU g++ compiler. Java is the preferred language, C++ should only be used when

1. The performance of Java can be demonstrated with benchmarks as not being sufficient when compared with C++ benchmarks.
2. There is already a large body of legacy C++ code. C++ authors are encouraged to look at the [Java Native Interface (JNI)](http://www.cygnus.com/misc/g++FAQ_toc.html) as a way of combining C++ and Java
3. To make up for deficiencies in C++, authors should consider using assertions, memory access checkers such a Purify and providing high coverage test suites.
4. Joe Buck's G++ FAQ is a great resource: [http://www.cygnus.com/misc/g++FAQ_toc.html](http://www.cygnus.com/misc/g++FAQ_toc.html)

### binutils

The issue of whether GSRC C++ software requires binutils is unresolved at this time.

## Sharing libraries between GNU and Sun C++ compilers

Igor Markov points out that it is nearly impossible to share code between GNU and Sun C++ compilers if
templates are involved. Sharing libraries between GNU and Sun C compilers can be tricky enough.

**Scripting languages**

**Perl**

Perl certainly has the most vocal community of all the scripting languages. Well documented and easily read Perl scripts are permitted. Hyperoptimized Perl scripts should be avoided.

**Bourne Shell**

Bourne shell is the preferred language for small scripts, since it is likely to be on most platforms, and can easily be use inside makefiles

**Tcl**

The Ptolemy project has shown good results using Tcl as a glue language for writing regression tests. The use of Tcl for regression tests is not mandated, but it is encouraged. Tcl should probably not be used for other scripting tasks, try using Bourne shell instead.

**C Shell**

C Shell scripts should be avoided, /bin/csh is not everywhere. Cygwin comes with bash, which is basically a superset of csh. Bourne shell can do basically everything that C Shell can do, though sometimes Bourne shell is not as clean.

If you really want C Shell, and Bourne shell will not work for you, then consider using Perl.2

For more details about csh issues, see [http://gonzo.tamu.edu/csh.whynot.html](http://gonzo.tamu.edu/csh.whynot.html) which basically says, use Bourne shell or Perl.

**Platforms**

GSRC software should run on the following platforms in roughly the following order of priority. Obviously, each group will have different priorities, the priorities below are suggestions.

Note that the version numbers below will change over time, they correct in August, 1999.

1. NT4.0sp5 with
   - Cygwin b20.1 - The Cygwin toolkit is a set of programs that provide Unix like functionality under Windows. The Cygwin toolkit includes many GNU tools, such as make, bash and a compiler
   - JDK1.2.2 - The Java compiler
   - gcc-2.95 - The GNU C++ compiler which is available for Cygwin as a separate download. Note that as of this writing, gcc-2.95.1 was not yet available for Cygwin.

   Rationale: Most industry users have PC desktop environments, we should work towards supporting
these users.
2. Solaris 2.7 with JDK1.2.2 and gcc-2.95.1
   Rationale: Solaris 7 is a stable, well designed multi-user multi-tasking environment used extensively in Academia.
3. Red Hat Linux 6.0 with Blackdown JDK1.2 with gcc-2.95.1
   Rationale: Linux has large market share and an active community.
4. Solaris 2.7 with Sun CC 5.0 (list $1895 for a node locked license) Academic pricing may be lower
   Rationale: C++ programs are more robust when compiled on multiple compilers.
5. NT4.0sp5 with JDK1.2.2 and MSVC++ 6.0 Microsoft Visual Studio 6.0 lists for $1079, Academic price, $259
   Ideally, this target platform should not require Cygwin to build or run.
   Rationale: Industry prefers to use Non-GNU software.
   Being able to ship binaries that do not require other software is a good feature.
   C++ programs are more robust when compiled on multiple compilers.

NT, Red Hat Linux, and Solaris with the JDK1.2 or the GNU C/C++ compilers are the primary development platforms. Solaris and NT4.0 with their respective commercial compilers are secondary development platforms.

## Version Control

The importance of using version control (VC) to help manage a software development process cannot be underemphasized.

A little background:

SCCS and RCS basically operate at the same level. SCCS is older, and there was no open source version available until recently, so it was not available on all platforms. RCS has been around a long time, and the source code has been freely available. Emacs has a vc facility that allows the user to manipulate both RCS and SCCS files transparently.

CVS is a system that works on top of RCS. Originally, CVS called RCS programs, now the RCS functionality is folded into CVS. CVS allows developers to operate on files in a concurrent fashion, whereas RCS and SCCS are really set up for locking files so that others cannot modify them. If two developers modify the same file, CVS attempts to merge the changes, and if the changes cannot be merged, it notes the conflict.

Here's what we get with CVS and version control in general.

1. CVS provides platform independence. I can check out the Ptolemy II system under NT with Cygwin or under Solaris, and the CRNLs or NLs in text files are handled properly
2. The directory structure of the package is automatically generated when I do 'cvs checkout ptII'. This means I can build a tree in no time, try out something wild, and then abandon the tree or check in my changes as I choose.
3. CVS allows for concurrent modification of files. Yes, problems can result, but this is a serious win. Professor Lee can hack away on his laptop while traveling, and then easily check in all his changes when he dials in. This is a big win.
4. Fixing bugs is much easier. I can narrow the time the bug was introduced down by checking out
trees from different dates and then running a test. I can then look at the log messages for the files that were changed when the bug was introduced and get an idea what was causing the problem. Or, I can tell the author that their changes broke something, and that they need to have a look. Version control is a serious win for having accountability.

5. The GSRC machine gigasource.eecs.berkeley.edu provides CVS support so that projects with developers in more than one university can easily set up and share a development tree.

6. CVS can handle binary files, so I can check in jar files (collections of java .class files), and PDF files. Basically, the entire release that we ship is checked in to CVS. This makes it very easy to create interim developer's releases.

7. By using CVS branches, I can easily create a release tree. I freeze the features of the release tree, and then fix bugs in the master tree and the release tree. The changes I made are visible in the log file, as is the location where I made the branch.

8. The CVS module facility seems like it could be of use in software systems that include subsystems.

9. There are web based interfaces to CVS. Lots of projects are doing distributed development with CVS.

10. CVS and version control systems in general are an excellent form of backup. When I work on a website, I put it under CVS, so then if I destroy a file, I can easily get a recent copy without bothering the backup operator. I work on a PC, but my files are in CVS repositories on reliable and backed up Unix boxes.

Style Guides

With regard to indentation standards, I strongly feel that publically available code should be looked upon as a published document.

- There should be no spelling errors.
- The comments should be syntactically correct.
- The comments should produce readable documentation when tools like javadoc are run.
- There should be a common documented style for each of the languages. Ideally, the style between languages should be as similar as possible.
- The style should be checked by automatic tools.
- Style guidelines are a fact of life in industry, they are almost always set by someone else.

Choosing a common style can be very controversial. The most important thing is that the members of a development team all agree on the same style.

Java Coding Style

The Ptolemy II Coding style is documented in $PTII/doc/coding/style.htm

To set up Emacs to follow the Ptolemy II coding style, download ptjavastyle.el and follow the instructions at the top

The Sun Java Coding Conventions are documented at http://java.sun.com/docs/codeconv/

Java tools

Emacs
GNU Emacs includes cc-mode which has a reasonable Java format

jindent
jindent is a shell script that invokes uses Emacs to indent java files. jindent is part of Ptolemy II, but is available as a separate download via

http://ptolemy.eecs.berkeley.edu/java/jindent.htm

chkjava
chkjava is a shell script that checks for common problems in Java source files. chkjava is part of Ptolemy II, it can be viewed at $PTII/util@testsuite/chkjava

ptspell
ptspell is a short shell script that uses a local dictionary with the Unix spell command to find misspellings. ptspell is part of Ptolemy II, but it can be used on any document, not just Java. ptspell can be found at $PTII/util@testsuite/ptspell

C++ Coding Style

The GNU tool gindent supports three common styles:

- The GNU style
- The Kernighan & Ritchie style
- The original Berkeley Style

We are open to proposals for a standard GSRC C++ style. Ideally the style would be one of the above styles with few if any modifications.

Ideally, C++ code would use an embedded documentation tool that is similar to javadoc. Doc++, at http://www.zib.de/Visual/software/doc++/index.html might be of use.

Filenaming conventions

Below are some subtle issues involved in naming files

- NT has a rather strange way of handling case in file names. It appears that file names are case insensitive, but case is preserved. Thus, under NT, you may have problems if a directory contains a file or subdirectory named Foo and a file or subdirectory named foo. Note that NT also has problems with DOS special names like com1, lpt1, or aux
- Under NT filenames with spaces are a fact of life. Scripts should be written and tested with filenames with spaces.
- GNU make looks for makefile before looking for Makefile. If a software package standardizes on Makefile, and someone accidentally introduces a file named makefile, then the accidentally introduced file is read and the proper Makefile is ignored. Thus, it is better to standardize on makefile. Also the NT filenaming issues can also cause problems

Build environment

A build environment is the system of files used to recompile source files and generate jar files, libraries and stand alone applications.
A build environment should have the following features

- Ease of setup - New users should be able to configure and build with minimal instruction.
  
  
  
  
  
  
  ./configure; make; make install should work.

- Use freely available, common off the shelf (COTS) tools such as GNU autoconf.

- Platform independence - Using Cygwin under NT allows us to use the same makefiles under NT and Unix.

Autoconf

GNU autoconf is a tool that can generate a portable /bin/sh script usually called configure. The configure script can find executables and libraries, and test for known bugs in compilers and such and then substitute variables inside files.

Most modern Unix source trees come with a configure script, and most people that regularly install Unix software are very familiar with how configure works.

GNU autoconf reads in a configure.in file and generates a configure shell script. The configure script is fairly portable, at install time, the user need not have GNU autoconf available.

At install time, the user can type ./configure --help to see what sort of command line arguments are available. For example, in Ptolemy II, the user can choose which Java Development Kit (JDK) to use by calling configure --with-java=/opt/jdk1.2 If the user does not supply a --with-java argument, then configure will find the first JDK in the user's path.

After the configure script determines the value of various variables it then reads in files that end in .in (makefile.in), substitutes variable names that are begin and end with @ (@JAVAC@), and then generate the resulting file.

For example, if makefile.in contains

JAVAC=@JAVAC@

and configure determines that the Java compiler is at /opt/jdk1.2/bin/javac, then makefile will contain

JAVAC=/opt/jdk1.2/bin/javac

Case study: How Ptolemy II is built

Ptolemy II consists of about 115 thousand lines of code in about 500 Java files spread over 133 source directories. Since Java .class files are platform independent, different platforms can share the same .class files.

The Ptolemy II build system uses GNU autoconf to generate a Bourne shell script called $PTII/configure.

To configure the Ptolemy II build environment, the user runs $PTII/configure, which does things
checks the version of the Java compiler and looks for optional tools.

configure reads in files such as $PTII/mk/ptII.mk.in and generates files such as
$PTII/mk/ptII.mk which has the location of the Java compiler defined as a makefile variable

The makefiles in the other directories include $PTII/mk/ptII.mk via a relative path. The makefiles also
include $PTII/mk/ptcommon.mk, which defines common rules that use the makefile variables defined in
ptII.mk and the current makefile

Advantages

• Uses configure to build
• Generates multiple jar files so that applets can only download what they need

Disadvantages

• makefile structure is somewhat complex
• Jar file structure is confusing. However, there is one large jar file that contains everything.

Case study: How the released version of Diva is built

http://ptolemy.eecs.berkeley.edu/diva/download/install.html says

1. Download the gzipped tar file or the zip file. Change to a directory that you want to place
Diva in, and either

    gzcat diva-0.2.tar.gz | tar -xf -

or

    unzip -a diva-0.2.zip

This will create a directory diva in the current directory. On Windows, you can also use a
utility such as WinZip.

2. Change to diva/mk and copy one of the files vars.OSTYPE.mk to vars.include.mk,
where OSTYPE is the value of the environment variable with that name. Currently, there
are three: one for Solaris (solaris), one for Cygwin bash on Windows NT (cygwin32),
and one for an anonymous port of tcsh (WindowsNT). (Feel free to send more in.)

3. Edit the new vars.include.mk so that the JAVAHOME variable points to the root of your
JDK1.2 installation.

4. Copy the file local.default.mk to local.mk. If you have any of the packages listed in that
file (which are not distributed with the JDK or with Diva), edit the file appropriately.

5. In the diva directory, type make packages.

6. Go to the demo page to run the demos.

Advantages

• Easy to set up
• makefile structure is easy to understand
- Creates on large jar file

**Disadvantages**

- Released version does not have ./configure, so each installer has to grok the instructions
- Requires copying files around by hand
- Produces large monolithic jar file, which must be downloaded over the net for each diva applet

**Case study: How the development version of Diva is built**

The development version of Diva uses configure to create a diva/vars.mk which is included by each of the makefiles.

**Case study: How Ptolemy Classic is built**

Ptolemy Classic consists of roughly 420 thousand lines of code in 2600 C/C++ files. C++ .o files are not shareable between platforms and sometimes not shareable between compilers so the Ptolemy Classic build system uses separate directories for the bin lib and obj trees on each platform.

The user sets the PTARCH environment variable to the name of the platform (i.e. for Solaris 2.5.1 PTARCH is set to sol2.5).

The user runs $PTOLEMY/MAKEARCH, which is a Bourne shell script that creates platform dependent directories like $PTOLEMY/bin.$PTARCH and $PTOLEMY/obj.$PTARCH. obj.$PTARCH is consists of directories that mirror the Ptolemy source tree.

The user then runs make, which goes into obj.$PTARCH and starts compiling the C++ source files, which are accessed via the makefile VPATH facility. Each makefile includes $PTOLEMY/mk/config-$PTARCH.mk, which consists of platform or compiler dependent makefile variable settings. Each makefile also includes $PTOLEMY/mk/common.mk, which includes common makefile rules.

**Advantages**

- Supports multiple platforms and compilers from one tree

**Disadvantages**

- The MAKEARCH script requires constant maintenance.
- The config-$PTARCH.mk file must be configured by hand for each compiler. Using a GNU autoconf configure script would make configuration easier.

**Build speeds**

If your nightly build is taking longer and longer, you might try splitting the nightly build up onto multiple machines. This is especially useful if you are compiling from scratch with different compilers or with different compiler settings.
Note that if your software is taking a long time to compile, then users will be facing the same long compile time when they install it, so you may want to look into what is causing the long compile time.

We found that at one point, the Java `-depend` flag was causing the compile to take 79 minutes. Removing the `-depend` flag caused the complete rebuild to take 19 minutes.

Another trick to speeding up a compile is to add a `fast` rule that runs the compiler on all the files in a directory at once. The following rule runs `javac *.java`

```bash
fast:
    CLASSPATH="$(CLASSPATH)$(AUXCLASSPATH)" $(JAVAC) $(JFLAGS) *.java
```

The advantage of this is that the Java compiler is invoked only once per directory, instead of once per file. At one point, the Ptolemy II compile was 11 minutes, 11 seconds, the addition of the fast rule allowed the compile to finish in 4:53.

Using `-j` option with GNU make will cause make to run more than one job at a time. `make -j 4` will run 4 jobs at a time. Note that there are limitations about commands that read stdin. `make -j -l 2.5` will not let make start more than one job if the load is more than 2.5. See the GNU make documentation for details.

## Release versions

Selecting version numbers for a release can be tricky.

Ptolemy classic released software as version 0.x, where the 0 was to remind the downloader that the software is experimental. Other packages may or may not want to follow the 0.x numbering scheme.

A version number has two parts, a major version number and a minor version number. Major version numbers should be incremented when the software changes in non-backward compatible ways. Code that uses the 1.1 version of a package should continue to work with the 1.2 version. However, when the 2.0 version comes out, then the 1.1 based code may require modification.

Releases usually go through an alpha1->alpha2->beta1->beta2->release->patch1->patch2 etc. cycle. The preferred version numbers are: 1.1a1, 1.1a2, 1.1b1, 1.1b2, 1.1, 1.1p1, 1.1p2.

- If you are having only one alpha or beta, then you can use 1.1a or 1.1b
- An alternative is to spell out alpha, beta and patch: 1.1alpha1, 1.1alpha2, 1.1beta1, 1.1beta2, 1.1, 1.1patch1, 1.1patch2.
- An alternative to using the p1 or patch1 is to release 1.1.1, 1.1.2 etc.
- If you use a different numbering scheme, it should be obvious which version is the most recent version.

## Documentation

Well written documentation can make all the difference in whether a software package is used by anyone other than its developers.
Well written documentation also helps sponsors and potential sponsors evaluate a package. Documentation that is integrated into the release via html pages can also help.

Software documentation can be split into several categories

**Installation**

The Installation documentation should usually have two distinct sections:

- one section for experienced installers, who just want the facts in a concise manner.
- one section for less experienced users. A troubleshooting section can help here.

The installation documentation should include a list of the requirements of the software, and should be visible from the download page so that users can see what they need to do after downloading the software.

If the installation procedure is very similar to other software, then there are likely to be far fewer problems. Ideally `./configure; make; make install` should be all that is necessary.

**User Level**

User level documentation consists of information about what the package does, and how to run the demonstrations.

User documentation is sometimes written using MS Word or Framemaker, and shipped as a PDF file, or a collection of HTML files.

**Programmer Level**

Programmer documentation is for software developers who want to extend the software system, or include the system in their own package.

The code itself should be considered a publication, each file should have the following:

- A copyright notice
- A primary author
- A list of contributors
- A version control tag such as

  ```
  ```

  so that each file can be associated with a branch in the tree
- The source files should have a common style
- The files should be spell checked

Programmer documentation usually includes class and method documentation. Using an embedded documentation system can be a big win in this area.
- Java code should use the javadoc system to generate readable html documentation.
- C++ code should probably use a similar embedded system

**Online Help**

Providing well thought out online help, error handling and error messages can greatly improve the user's experience.

The Java exception handling mechanism should be used properly. Software packages should define exceptions and use them.

**Documentation formats**

Documentation is usually in several formats:

**Text**
- ASCII Text files are useful for short files. CVS takes care of the newline issues, but shipping tar and zip files can be a little tricky.

**HTML**
- HTML files are the preferred format, especially for Java packages that include applets.

**PDF**
- PDF is the preferred platform neutral format for longer documents such as user guides.

**PostScript**
- PDF is preferred over PostScript. Adobe Distiller can be used to convert PostScript to PDF. The GSRC has a copy of Distiller.

**Microsoft Word and Framemaker files**
- These formats are vendor specific and are not usually shipped with a release. It is far better to ship PDF files, or convert the document into HTML. Word and Frame files can be checked into the CVS repository with the -kb option.

**Testing**

Regression tests are tests that compare new results with a known results and report any differences. Regression tests are a good first step towards creating software that will survive multiple developers over time.

**Coverage**

Regression tests are of little use without a measure of coverage. Note that 100% code coverage does not mean the code is completely tested, but high level of code coverage is a start.

Java developers can use JavaScope from Sun, which formerly was freely available to schools, but now lists at roughly $800/seat. The educational pricing is unknown at this time. JavaScope can generate HTML reports.

C++ developers can use tools like PureCov from Rational.
**Tcl**

The Ptolemy group has had good results using Tcl as a glue language for test suites.

Using scripting to write tests for Java or C/C++ is quick and easy.

Writing tests is much more of an incremental process than writing system code - a scripted language makes sense.

Being able to easily modify tests, and then run them from an interpreter makes test case development faster.

**Java and Tcl**

Java classes can be accessed via Jacl and Tcl Blend. Jacl is a 100% Java implementation of a subset of Tcl, which is contained in one 675k jar file. Tcl Blend is an extension that can be loaded at runtime into a standard C based Tcl interpreter. The Ptolemy project is currently using Jacl for the test bed.

The Tcl command to instantiate a Java object: si

```
set a [java::new classname]
```

This returns a handle, like `java0x4`. We can then call Java methods on the handle:

```
$a toString
```

C++ authors can use Swig ([http://www.swig.org](http://www.swig.org)) to generate wrappers for C++ classes.

**Tcl Testing Framework**

The Tcl testing framework was first implemented by Mary Ann May-Pumphrey of Sun Microsystems.

The Tcl testing framework is fairly straight forward, test files consist of blocks of code like

```
test testname { testcomment} {
    # Do the test
    # The return value from the last line is compared with the
    # known result
} {known result}
```

A primitive example would be:

```
test Add-1.1 {test the tcl expr command} {
    expr { 2 + 2 }
} {4}
```

If for some reason 2+2 did not equal 4, then when the test was run, a message would be printed. A simple test of the java.lang.String constructor might look like:

```
test SimpleTest-1.1 {Test Foo} {
```
set a [java::new {String String} "A string"]
$toString
} {A string

An actual test from Ptolemy II looks like:

test NamedObj-2.1 {Create a NamedObj, set the name, change it} {
    set n [java::new ptolemy.kernel.util.NamedObj]
    set result1 [$n getName]
    $n setName "A Named Obj"
    set result2 [$n getName]
    $n setName "A different Name"
    set result3 [$n getName]
    $n setName {}
    set result4 [$n getName]
    list $result1 $result2 $result3 $result4
} {{} {A Named Obj} {A different Name} {}}

Floating Point

Tests that return floating point results can present problems, especially when moving the tests to another platform. Often, the floating point representation is slightly different, so one platform might return 2.00000001, and another might return 1.9999999.

Unfortunately, Unix diff will indicate that such a change is an error, because the characters are different.

A better approach is to use an epsilon method to compare two numbers and return whether the results are close enough or not.

Nightly Builds

Nightly builds are a good thing, they provide immediate feedback to developers if they broke the build. Having a nightly build means that the software stable and is likely to build for new users.

Software that is not built regularly is difficult to release. McCarthy says "If you build it, it will ship".

Nightly builds and Testing

An essential part of the nightly build is to actually run the software, ideally with a test suite.

Test results from nightly builds provide developers with feedback on how their test suite is progressing.

Nightly builds are very important when there is deadline pressure, running the nightly build several times a day can result in improved stability.

Nightly Build Strategies

Nightly builds really should be separate from your development tree, so the nightly build should not interfere with work by developers. If a dummy user account is used to run the builds, then it is much easier to configure the build environment properly and not have it break from minor environment
changes.

If the nightly build involves bootstrapping a compiler, it is best if the path of the dummy user account includes the basic directories that the operating system originally included, and whatever other directories are necessary.

The cron facility can be used to start up a script that runs the nightly build itself. The cron entry below runs the ptIInightly.cron script that sends email to the ealtech alias.

```
30 3 * * * csh -c "/users/ptII/adm/bin/ptIInightly.cron ealtech"
```

The cron entry should be as simple as possible. If the nightly build is run as a script, it is much easier to run the script by hand when debugging.

Ideally, the nightly build will check out the current source tree and build using those sources.

**Build Culture**

Nightly builds promote cooperation between developers, since there is a fair amount of social pressure not to break the build.

Usually, it is better when developers work in fairly small steps, and check in their working code on daily regular basis in such a way so that their changes do not break the build.

Obviously, this is not always possible, but we have found that code that is kept out of the tree for a long period of time tends to cause problems when it is finally checked in.

Nightly builds don't solve this problem, but the build can help identify code that is not present in the build.

**The future**

Eventually, nightly builds could be run on gigasource. Groups could receive nightly email with build log information and with code coverage statistics.

**Tiers**

Obviously, not all software will meet all standards. The enforcement of standards is a politically sensitive issue, and ultimately is the responsibility of the Principal Investigator (PI) of each software project. For academic projects, software engineering techniques really only work if the PI agrees that the standard or tool is valuable, and allocates time towards using it.

We can use a set of software standards to rate individual projects, which gives collaborators an idea of the stability of a package.

For each of the standards, we could have four tiers, Red, Yellow, Green and Blue, where Blue is the highest level of compliance.

Software with low levels of compliance is not necessary bad software, it might be software under
development by only a few authors, or it might be a project that is just starting out.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Red</th>
<th>Yellow</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td>UML diagrams never used</td>
<td>UML diagrams or similar aids sometimes used</td>
<td>UML diagrams available for all source code</td>
<td>UML diagrams shipped with source code</td>
</tr>
<tr>
<td>Reviews</td>
<td>No Reviews</td>
<td>Design and code reviews sometimes used</td>
<td>Design and code reviews regularly performed and available to developers online</td>
<td>Software not released until design and code reviewed</td>
</tr>
<tr>
<td>Platforms</td>
<td>Runs under one platform</td>
<td>Runs under two platforms</td>
<td>Runs under both Unix and Windows NT</td>
<td>Runs under all platforms</td>
</tr>
<tr>
<td>Version Control</td>
<td>Release Tree is not in version control</td>
<td>Release Tree is under version control of some sort</td>
<td>CVS</td>
<td>CVS Notification via email</td>
</tr>
<tr>
<td>Style Guide</td>
<td>No style guide</td>
<td>Documented style guide. No style checker tools available</td>
<td>Documented style guide. Style checker tools available</td>
<td>Documented build system. ./configure; make; make install works</td>
</tr>
<tr>
<td>Build System</td>
<td>No Build System</td>
<td>Custom, one off build system using custom scripts</td>
<td>Build system uses configure and makefiles</td>
<td>Complete installation, user and programmer documentation available, uses source based documentation system such as javadoc</td>
</tr>
<tr>
<td>Documentation</td>
<td>No documentation</td>
<td>Installation docs available, some user and programmer documentation available</td>
<td>Complete installation, user and programmer documentation available</td>
<td>Tests run every night with code coverage reports and 100% method test coverage</td>
</tr>
<tr>
<td>Testing</td>
<td>No test suite</td>
<td>Regression test suite available</td>
<td>Tests run every night with code coverage reports</td>
<td>Test release created and built, test suite run, code coverage reports generated</td>
</tr>
<tr>
<td>Nightly Builds</td>
<td>No Nightly Build</td>
<td>Nightly Build</td>
<td>Nightly Build with test release built</td>
<td></td>
</tr>
</tbody>
</table>

Ratings for various software projects

Probably the least controversial way to rate software is to have the authors rate themselves and see where they stand. Below is a set of ratings for various software projects, where the packages are listed by start date, with the oldest package listed on the left.
What can the GSRC do for me?

Today, the GSRC can offer the following facilities to software developers

The Softdevel SIG

The softdevel special interest group has been set up. The softdevel SIG is open to everyone in the GSRC, we hope that the mailing list will be used on an as needed basis to discuss and debate GSRC software development issues.

Mailing Lists

Each GSRC special interest group can have a mailing list associated with it. The mailing lists are archived and include a search facility.

CVS

A CVS repository has been set up on gigasource.eecs.berkeley.edu.

Software projects that have developers at more than one institution often have problems setting up accounts at each institution for each developer. Such projects may find it easier to give each developer a GSRC account and use this repository.

There is also a CVS SSH installer available for use under Windows NT.

Training

In April, 1999, John Reekie presented a GSRC Software Day. We hope to present other courses on areas such as CVS.
Advice

We are more than willing to look over your software development process and make suggestions and answer questions.

Future

In the future, we hope to offer

Nightly Builds

Eventually, we would like to be able to offer nightly builds, where the CVS repository is checked out in a clean location and ./configure; make; make install; make test is run and the results mailed back to the developers.

Nightly builds that are run on gigasource could take advantage of commercial software such as code coverage tools.

Installers

We are experimenting with the InstallShield Installers for C/C++ and Java. The C/C++ version runs only under Windows, but the Java installer runs under Windows and some Unixes.

Resources

Papers

  http://ptolemy.eecs.berkeley.edu/publications/papers/99/softwareprac

Courses

- Notes from GSRC Software Day, April 1999  
  http://gigascale.eecs.berkeley.edu/pubs/courses/software99/

Web pages

- GSRC Softdevel CVS FAQ: http://gigascale.eecs.berkeley.edu/softdevel/faq/cvs