Getting Started Guide
for All TAU Projects

Jennifer Dempsey, Andrey Kvochko, Kyriacos Pavlou, Tapasya Patki,
Richard T. Snodgrass and Benjamin Dicken

August 5, 2014

TD-2

A TAU TECHNICAL DOCUMENT
Title: Getting Started Guide for All Tau Projects

Author(s): Jennifer Dempsey, Andrey Kvochko, Kyriacos Pavlou, Tapasya Patki, Richard T. Snodgrass and Benjamin Dicken

Publication History: January, 2013
May, 2012
February 16, 2012
October 2, 2011

TAU Participants

Faculty
Richard T. Snodgrass (Director)
James R. Cordy (Queen’s University)
Sabah A. Currim
Clayton T. Morrison

TAU Coordinator History
Jennifer Dempsey (June 2014–present)
Young-Kyoon Suh (January 2014–March 2014)
Andrey Kvochko (March 2012–December 2014)
Kyriacos Pavlou (August 2011–March 2012)

Chief Programmers
Jennifer Dempsey (τPSM, LoCuS)
Brian Godshall (SPOCK, τDOM, τXSHEMA, AMELIE)
Young-Kyoon Suh (AZDBLab)
Stephen Thomas (IBM Ottawa, τBENCH)

For additional information, see The TAU Homepage:
URL: http://cs.arizona.edu/projects/tau

TAU is an umbrella project comprised of a number of inter-related and complementary projects, all with the goal of providing to users, through sophisticated user languages and APIs, facilities to manage time-oriented data. The name of the project was inspired by three associations. The acronym is “Temporal Access for Users” which concisely describes the project. The project name is the English spelling of the Greek letter “τ”, which is commonly used in scientific formalisms to denote time. And the Greek letter resembles an umbrella, emphasizing the sub-projects comprising this overall project. Our goal is to realize the theoretical advances over the last decade in temporal databases in practical, efficient, correct, well-engineered and documented interfaces that can be used by the application developer or database user. A second, consistent goal is to leverage each system with the other systems, to provide comprehensive support. Hence, the temporal data types and granularities made available in τZAMAN can be utilized in XSHEMA directly and also in τXSHEMA, the latter to describe time-varying XML data, stored efficiently by τBDB, supported in Persistent Stored Modules in τPSM, queried and returned as a result by τXQUERY, evaluated for correctness and performance in τBENCH, and utilized in τDOM when manipulating time-varying XML documents.
# Contents

1 Introduction ............................... 1
   1.1 Projects .................................. 1
   1.2 Technologies .............................. 2

2 Terminology .................................. 3
   2.1 Projects ................................... 3
   2.2 Systems .................................... 3
   2.3 Features .................................. 3
   2.4 Plugins .................................... 4
   2.5 Releases ................................... 4
   2.6 Documentation Taxonomy ................... 5
   2.7 Repositories ............................... 5

3 The Tau Project Directory .............. 7

4 SVN Repositories for Papers .......... 9
   4.1 The Structure of the Paper Repository .. 9
   4.2 Checking Out an Existing Paper Repository 9

5 Interaction of Technology Components 11

6 Setting Up Tools ............................ 13
   6.1 Setting Up SVN .............................. 13
   6.2 Setting Up Eclipse ......................... 13
   6.3 Setting Up VisualParadigm ................. 14
   6.4 Using Pygments and Minted ............... 14

7 Guidelines for the Structure of the Working Directory 15
   7.1 The Structure of the Working Directory of a Tau Project ....... 15
   7.2 Executables ................................ 17
   7.3 Documents ................................ 17
   7.4 UML Diagrams .............................. 17

8 Setting Up the Working Directory ...... 19
   8.1 Fetching the Working Directory .......... 19
   8.2 Importing the Working Directory into Eclipse ..................... 19
   8.3 Importing the LocalDocs/VisualParadigm Directory into VisualParadigm 20

9 Using SmartSVN .............................. 21
   9.1 Obtaining SmartSVN ....................... 21
   9.2 Checking out a Part of an Tau SVN Project ....................... 21
   9.3 Opening an Existing Working Copy of Tau Documentation ........ 22
Chapter 1

Introduction

First of all, welcome to this exciting project! The purpose of this document is to equip you with the major technologies that we use, more specifically, CVS, SVN, Eclipse and Mantis. This is the general guide to all the projects included under the Tau umbrella project. For getting up to speed with your specific project, please refer to the project-specific Getting Started Guide.

1.1 Projects

There are currently eleven projects within Tau. A few, such as AMELIE, AZDBLab, LoCuS, and SPOCK are not strictly in the Tau sphere, as they do not concern temporal databases, but are included here because they benefit from the structure that Tau provides. \( \tau \text{Bench} \) and \( \tau \text{XQuery} \) reside in an SVN repository at Queen’s University, and so do not have technical documents.

AMELIE: a cyber workspace that provides computational support for causal modeling throughout the process of an empirical investigation

AZDBLab: a system to execute experiments, collect data, and analyze that date

Dragoon: adds auditing to \( \tau \text{BDB} \) and couples it with facilities to validate a database to detect tampering and tools to perform forensic analysis of such tampering once it has been detected

LoCuS: a collection of web-based ergalic labs to explore the science of computing

SPOCK (Student Prediction of Classes): an experimental domain for AMELIE.

\( \tau \text{BDB} \): our version of BerkeleyDB with transaction-time support

\( \tau \text{Bench} \): a temporal extension of the XBench benchmark, consisting of a suite of temporal and non-temporal benchmarks, all derived from XBench, as well as a suite of tools for generating and validating each benchmark

\( \tau \text{DOM} \): an extension to standard DOM API to support manipulating temporal XML documents

\( \tau \text{PSM} \): an extension to SQL/PSM (Persistent Stored Modules) to allow such modules to be invoked from temporal queries

\( \tau \text{XPointer} \): a temporal extension to XML Pointer

\( \tau \text{XQuery} \): an extension to XQuery to enable temporal support

\( \tau \text{XSHEMA} \): a system for constructing and validating temporal XML documents

\( \tau \text{Zaman} \): a native Java system for formatting and manipulating times and dates in multiple calendars and languages
1.2 Technologies

We employ established technologies for ease of development, consistency and usability. These technologies include:

- Ajax
- Ant
- bash
- doxygen
- Eclipse
- Eclipse Memory Analyzer (MAT)
- Hibernate
- JAutodoc
- javadoc
- Java Preferences
- LaTeX
- log4j
- lint
- Mantis
- make
- minted
- SVN
- SmartSVN
- Valgrind
- VisualParadigm
- Visual VM

We use VisualParadigm for all UML diagrams. Eclipse Memory Analyzer (http://www.eclipse.org/mat/) is a Java memory leak tool. It is installed on lectura. VisualVM is a visual tool integrating several command line JDK tools and lightweight profiling capabilities (http://visualvm.java.net/download.html). Java Preferences (http://docs.oracle.com/javase/8/docs/technotes/guides/preferences/index.html) allow for user settings to be easily saved and retrieved for a project. The QUERYEXECUTOR, used in τPSM, is one example of a program that makes use of Java Preferences to save user login information for a database, as well as, the location of directories containing database queries.

All commands given in this document are for the bash shell; therefore we recommend that users work with the bash shell.
Chapter 2

Terminology

Tau is an umbrella project: a project that contains other projects. Given the complexity of many of these projects, it helps to have some common terminology and shared practices.

2.1 Projects

A Tau project is an effort by a collection of faculty, graduate students, undergraduate students, and external colleagues focused on a general problem with a goal of producing a system to explore solutions to that problem. Each project has a chief programmer who maintains oversight: managing the SVN repository, planning releases, and monitoring the state of the project. One example of a Tau project is AZDBLab.

2.2 Systems

Each project is associated with a software system, varying widely in size and configuration. τDOM and τZaman are Java APIs: a set of classes that provide additional functionality to Java programs. τPSM and τXQuery are source-to-source translators, converting a temporal language into a conventional language. τBDB is a full-fledged temporal DBMS. τXSschema is a collection of tools for managing temporal XML documents. Dragoon, AZDBLab, and AMELIE are systems (the first a DBMS, the rest experimental management systems) that each include several auxiliary programs. LoCuS is a Java program that can present a lab, along with included Java-based apparatuses. τBench is in some ways the most complex, comprising a half-dozen interrelated benchmarks, each with data and a workload and auxiliary tools.

All systems, with few exceptions (like Dragoon), have a helloworld example. Helloworld is a minimal example that shows off all functionality provided by the system. For instance, a helloworld example for τDOM would be a set of temporal XML files and Java source code that uses the τDOM API to read the files. LoCuS has a helloworld lab, as well as a helloworld apparatus plugin.

2.3 Features

A feature is a system that can be integrated with an existing Tau project to expand its functionality. A feature is general enough not to be associated with any particular system. A feature is integrated with a Tau project at the time of development and is compiled as a part of that system. An example of this is the Watcher feature, which is integrated with the AZDBLab and LoCuS systems. Another example is QueryExecutor, which is a critically-useful tool for both the AZDBLab and τPSM projects.

A Feature Owner is a person designated to developing and supporting the feature, as well as, in case of multiple developers working on the feature, coordinating the team. Feature Owners are responsible for documenting the features. Each feature must have at least one technical document.
2.4 Plugins

Plugins are compiled classes for which interfaces are defined in a Tau project. Plugins can be dynamically loaded and used by the main software system during run time. For example, LoCuS apparati are plugins to the main system. An AZDBLab experiment subject is another example of a plugin.

2.5 Releases

There are a series of releases of a project. An internal release is visible only to the project personnel. An external release is given to others and/or put on the web. The eventual goal is for every project to be released externally. That said, a project release must be of higher quality (have adequate documentation, be reasonably complete in its functional capabilities, have examples) to be considered for external release.

A project exists initially only in its repository. Often the system can be run directly from this repository, say by coupling Eclipse to it. Or project personnel can check it out from the repository and run it in their local system.

A release is more formal. A release has a major and a minor number, e.g., 6.2. While the differentiation is project-specific, a major number means significant new functionality and/or incompatibility with the previous release, say because the schema has changed. A release is stored within a directory of /cs/projects/tau/releases, named with the date it was created and the version number. Once such a directory is created, it is permanent. So we will have a collection of subdirectories, one for each release of the system.

Releasing a system requires first performing the release creation step, which creates a .tar file containing a README file that explains the steps needed to build the system from the files in the tar file. Once a system is built, it can be installed (where the executables are moved to one place (e.g., /usr/local/bin) and the include files moved to another place (e.g., /usr/local/include)). Installed executables for a particular release are used within the Tau project (e.g., τBDB) and are stored within a directory (named for the release and the installation date) in /cs/projects/tau/installations. Once the executables are in place the system can be run.

For example, in the case of τBDB the script used to create the tar file of a release is called deploy.sh. This script is run inside the scripts directory of a user’s τBDB working directory. The README file contained in the τBDB release references the INSTALL file (also found in the release) which details the build and installation instructions. Files (including the binaries) are placed during installation according to the pathname specified when executing the .configure –prefix=pathname command. The variable pathname denotes the destination of the binaries. To install τBDB then we execute make followed by make install. The binaries can then be moved to the installations folder unless of course the pathname specified was the installations folder.

The structure of /cs/projects/tau/ is described in more detail in Chapter 3 of this document.

For releases, remove unnecessary statements in the code and scale back logging. Using the JAutodoc Eclipse plugin or a shell script, the release number should be inserted in the header of source files.

It is important to emphasize that the directory structure of the project’s repository might be different from the directory structure of the tar file (such as the former including some documentation used by us but not distributed), and the directory structure of the tar file might be different from the directory structure used in building the system, and indeed, the final location of the system. In the other extreme, the release creation step just extracts a subset of a specified version of the repository and constructs a tar file from that, the build step just untars and runs make, the install step doesn’t do anything, and the system is run right from that directory.

A source release (the common kind) includes all source (that is, everything in the src directory, with the build step compiling the source. τBDB is a good example. In fact, this system is a patched version of the BerkeleyDB system from Oracle; the user must first download BerkeleyDB, with the build step merging that source with some changed and additional files in the τBDB release. DRAGOON is more interesting, as it patches τBDB with an audit module, and then adds some auxiliary programs (a notarizer and a validator) along with some helpful GUI tools.

A executable release does not include the source; so the build step itself compiles the source into an executable. LoCuS is a good example: building produces a jar file that can be downloaded and directly
executed by the Java-based JNLP system, present on most machines.

### 2.6 Documentation Taxonomy

Documentation is vital to a substantial project. Through documentation the project maintains its institutional history, enables new people to come up to speed on the project, and helps users of the system understand what the system does.

Documentation can be classified along a number of orthogonal dimensions.

**Publishable:** A document may be publishable (a paper that may eventually be submitted or has been submitted to a conference or a journal) or nonpublishable (not planned to be submitted anywhere).

**Formality:** Documentation may be formal (a numbered Tau Technical Document, listed in [http://www.cs.arizona.edu/projects/tau/technicaldocuments/](http://www.cs.arizona.edu/projects/tau/technicaldocuments/)) or informal (a file found in a project’s repository).

**Usage:** A document may be user documentation (intended to be read by the user of the system) or internal documentation (intended to be read by developers and maintainers of the system, and thus describes aspects of the system of less interest to users).

**Distribution:** A document might be released (included in a release) or local (intended only for use by the project developers).

**Venue:** A publishable paper may be targeted or appear in a journal, conference, workshop, poster, or demo.

A publishable document must have a venue while a nonpublishable document must have formality, usage, and distribution.

Since these are orthogonal, each documentation file is publishable or not, has a formality, has a usage, and has a distribution. So a .txt file describing a particular design decision might be “nonpublishable/informal/internal/local,” whereas the README for a system might be “nonpublishable/informal/internal/released,” and this document is “nonpublishable/formal/internal/local.” A paper describing τPSM might be “nonpublishable/informal/internal/released.” A journal paper on Dragoon will be “publishable/journal.”

While a project may be released (built and then installed), a publishable paper is submitted. When a paper is submitted, a tag should be created in the repository for that paper of the current files that make up that submission.

Two technical documents are especially important: the “Getting Started” and “Chief Programmer” manuals, which are provided by each project. A student new to a Tau project should first read Getting Started Guide for All Tau Projects (this document) and then the Getting Started manual for their individual project. A new chief programmer for a project (who will generally have been working on that project for awhile) should first read the Chief Programmer’s Guide for All Tau Projects for an overview, and then the Chief Programmer manual for their individual project.

Required documents for each project are: Chief Programmer’s Guide, Getting Started Guide, Schema document, UML diagrams (both sequence and class diagrams), and Plugin Development Guide (for projects that utilize plugin functionality).

Tau technical documents are located in an SVN repository. Location and structure of this repository is described in detail in Section 9.1 of this document.

### 2.7 Repositories

There is an SVN repository for each project, managed by that project’s chief programmer. These repositories are located under /cs/svn/tau/. When a repository, or an SVN project within an SVN repository, or a directory within an SVN project, or even an individual file within a directory within
an DVN project within an SVN repository, is checked out into a user’s directory, that portion that was
checked out is termed a *working copy*.

SVN can get confusing about derived files. All `.pdf`, `.bst`, `.cls`, and `.sty` files should be versioned. The following kinds of derived files should remain *unversioned* in SVN.

- latex : `.aux` `.bbl` `.blg` `.dvi` `.log` `.toc` `.ps`
- Java : `.class`
- C : `.o` `a.out`
- other : `.bak` #file# `file~`

However, derived files from drawing programs should be versioned, because it is a pain to recreate them each time. Derived files from drawing programs that should be versioned in SVN:

- gnuplot or xfig : `.ps` `.eps` `.pdf`
- omnigraffle : `.pdf`
- GIMP : `.jpg` `.png`

In addition to the repositories just discussed, there is also a separate SVN repository for each publishable paper (`/cs/svn/tau/papers/`).

Nonpublishable documentation resides in the project’s SVN repository. Technical documents related to TAU generally (such as the TAU Chief Programmer’s Manual) resides in the repository for the project called `tau`.

Source code for experiments related to the results in a publishable paper resides in a subdirectory of that paper’s SVN repository. There should be a README file there that states which release of any system used by that experiment (e.g., for the paper on stamping the τBDB, the README would mention the version of τBDB used in those experiments, i.e., the release directory). Most publishable papers will use at least one system.

You can find additional information on SVN repositories for publishable papers in Chapter 4.
Chapter 3

The TAU Project Directory

The TAU project directory is /cs/projects/tau and contains all the material related to the TAU projects. Figure 3.1 shows the structure of /cs/projects/tau.

![Diagram of directory structure]

Figure 3.1: The structure of the tau directory.

We now review the subdirectories.

fossilized This folder contains several inactive folders like BerkeleyDB, from_cvs. It also contains the folder people which has the working directories of all students who previously worked on the TAU projects.

installations This folder contains installed executables for a particular project release used within the TAU project (e.g., τBDB). The executables are stored per project within a directory named for the release and the installation date, e.g., /cs/projects/tau/installations/azdblab/v6_01_20121009. Installations are created by the chief programmer.

live This folder contains the working directories of all the students currently involved in the TAU projects, for example /cs/projects/tau/live/rts. All files related to working within a TAU project should be in a subdirectory of live. Directories should have group set to tau and have g+rwx permissions.
media This folder contains media such as videos that undergraduate students prepared in order to showcase their involvement with TAU projects during their REU work.

projectInfo This folder contains useful information about the TAU projects, like the list of machines maintained, the information about the Chief Programmers, overview of each TAU project etc. It also contains information about the duties of the DBA for all TAU projects.

Machines.txt Information on each of the TAU dedicated servers.
restartingmachines Information on how to restart machines remotely.
Summary Other useful information about TAU directories.
AZDBLab/ Information for the AZDBLAB chief programmer.
DBA/ Information for the TAU database administrator.

README This folder contains information about all the folders under /cs/projects/tau.

releases This folder contains the releases of each of the TAU projects. Each release named with the date it was created and the version number, e.g., /cs/projects/tau/releases/azdblab/azdblab_release_5.3.tar.gz. Once such a directory is created by the chief programmer, it is permanent. Releases are usually .tar.gz files.

third_party_releases This folder contains releases of software/systems developed outside of the TAU research group. The .tar.gz release files can be found in individual subfolders with the release number and release date in their name.

third_party_installations The actual installation of the third party releases is found here, thus mirroring the structure of installations and releases.
Chapter 4

SVN Repositories for Papers

A separate SVN repository (paper repository) must be created for each paper under the TAU umbrella project. This includes published papers, submitted papers, demos, workshops, technical reports, and posters (c.f. Section 2.6). To create an SVN repository, please, refer to the Chief Programmer’s Guide for All TAU Projects. All TAU paper repositories are located in /cs/svn/tau/papers/.

4.1 The Structure of the Paper Repository

The name of the SVN repository must be the name of the paper in lower case; each word should be separated by an underscore. The last part of the name should indicate the type of this paper (journal, conference, workshop, poster, etc), e.g., forensic_analysis_of_database_tampering_journal. Figure 4.1 shows the structure of a paper repository. It is recommended that project names for the papers should be descriptive, unique, and as short as possible.

The process of writing a new paper starts with creating an SVN repository and importing a project into trunk (forensic_analysis in the figure). Changes and revisions of the paper are made under the trunk. The trunk always contains the most current version of the paper. Whenever the paper ready for submission and the venue has been decided (VLDBJ in Figure 4.1), a new tag is created, as described in Section 5.5 of the Chief Programmer’s Guide for All TAU Projects (TD-1). The first time a paper is submitted to a particular venue the paper is tagged as venuename_initial, e.g., VLDBJ_initial in Figure 4.1. When the paper needs to be revised as per the referee comments the process of writing continues under the trunk. When the revision is ready to be submitted to the same venue the paper project is tagged as venuename_revisionnumber, e.g., VLDBJ_revision1 in Figure 4.1. The number the number of the revision to be submitted. If the paper is rejected and is intended to be sent to a different journal (TODS in Figure 4.1), a new tag indicating an initial submission is created for that journal, e.g., TODS_initial. The process is repeated until the paper is accepted at which point the final version of the paper is tagged as venuename_final, e.g., TODS_final in Figure 4.1. Note that this mean every submission of a paper to either a conference or a journal should have its own tag.

Branches are allowed and should be used when multiple authors are working on the same paper simultaneously. However, for most papers, only one author works on a paper at a time (“has the modify token”) and in such cases a branch is not needed. The important point is that under all circumstances the most current version of the paper should reside in the trunk.

4.2 Checking Out an Existing Paper Repository

Please refer to Sections 9.4 and 9.6 for instructions on how to check out a paper repository using SmartSVN.
Figure 4.1: The structure of a paper SVN repository.
Chapter 5

Interaction of Technology Components

The working directory is set up by executing the `fetchProjectName.sh` script. The script checks out the appropriate working copies from the TAU project repository and places them in the working directory.

**Important:** do not checkout the project by means other than the fetch script!

Any documentation in the working directory is managed through SmartSVN. The Java source code in the working directory is imported into the Eclipse workspace by reference. Any project changes committed to the SVN repository are reflected in Mantis through the execution of the `resolve_bug.php` script which resolves the appropriate Mantis issue.

![Diagram of interaction between Eclipse, Mantis, and SVN](image)

Figure 5.1: The interaction between Eclipse, Mantis, and SVN

The description given below should be considered as simple guidelines and not as requirements. Please consult the project-specific guides on how the working directory is structured for each project. Some projects are managed entirely through SmartSVN.

We’ll use the DRAGOON project as a running example within this Chapter. Suppose a user would like to set up his working directory under `/cs/projects/tau/live/username`. To do so the user must run the `fetchDragoon.sh` script under the `username/` folder. The script itself can be found within the DRAGOON SVN repository and can be exported using the following command:

```bash
svn export svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/dragoon/trunk/dragoon/scripts/fetchDragoon.sh
```
After running the script the DRAGOON working directory (/cs/projects/tau/live/username/dragoon) will have the following subfolders: LocalDocs, docs, and src. The user checks out, and manages all the documents in LocalDocs and docs using SmartSVN (see Chapter 9). The Java source code located in src is imported into the user’s Eclipse Workspace by reference, i.e., no physical copy of the code is made. (NB: The C code in the src directory is not managed through Eclipse.) Eclipse has the Subclipse plugin installed thus allowing direct SVN management of the Java source code. The source can be updated and committed through Eclipse. When the user commits any changes to the source in response to a reported Mantis issue, the resolve_bug.php script is run automatically to resolve the issue in Mantis.
Chapter 6

Setting Up Tools

In this chapter we describe how to set up Eclipse to work with SVN by employing the Subclipse plugin. We also describe how to set up VisualParadigm and how to use Pygments.

6.1 Setting Up SVN

We use SVN client version 1.6 for all Tau projects. To view your SVN client version, execute the following.

```
svn --version
```

If you don’t have SVN client installed, or your client version is below 1.6, you can download client packages for multiple platforms from the official website: [http://subversion.apache.org/packages.html](http://subversion.apache.org/packages.html).

Note, SVN client 1.6 should already be installed on all department machines. If this is not the case you will need to contact lab ([lab@cs.arizona.edu](mailto:lab@cs.arizona.edu)).

6.2 Setting Up Eclipse

We assume that the most recent version of Eclipse is installed on your machine. Eclipse can be set up to work with SVN via the Subclipse plugin.

To install and set up Subclipse, do the following steps. These steps are for Mac OS X.

1. Make sure you have SVN client installed as described in the previous section.
2. In Eclipse, go to Help → Install New Software.
3. In Work with field, type in [http://subclipse.tigris.org/update_1.6.x](http://subclipse.tigris.org/update_1.6.x) and press Enter.
4. Check all the boxes and click Next. In the next window click Next.
5. Select I accept ... and click Finish.
6. If asked whether you want to continue, click OK. When installation finishes, click Restart now.

For Ubuntu the steps are slightly different. Steps 1–3 are the same.

4. Click Add... . A new window will appear titled Add Repository . Name the repository Subclipse. The Location: field will already be filled with the url you provided in step 3. Click OK .
5. Check all the boxes of the names which appear by clicking on the Select All button and then click Next. In the next window click Next.
6. Click the radio button **I accept the terms of the license agreements** and click **Finish**.

7. If asked whether you want to continue, click **OK**. When installation finishes, click **Restart now**.

### 6.3 Setting Up VisualParadigm

The Computer Science department provides one floating license for Visual Paradigm For UML 10.0 software. This means that anyone on the CS department network can use the license, with a restriction that only one person can be using Visual Paradigm at any one time.

Please note that Visual Paradigm floating license can only be used from the CS department network. Steps below describe how to set up Visual Paradigm on a Department–issued Linux machine.

1. From a department Linux machine, go to [http://www.visual-paradigm.com/download/vpuml.jsp](http://www.visual-paradigm.com/download/vpuml.jsp), select **Linux (No Install)** and click **Try Visual Paradigm for UML FREE NOW**.

2. Unpack Visual Paradigm to a desired folder.

3. Navigate to the folder chosen in step 2, go to **Visual_Paradigm_for_UML_10.0/bin**.

4. Double-click on **Visual_Paradigm_for_UML_10.0** and click **Run**.

5. Select **Perpetual License**.

6. In the next window, select **+ Floating License**. In the center **Host** text field, enter `sodb7.cs.arizona.edu`. Click **Test Server**. Once the server is successfully tested, click **Apply**.

7. Click **Start**. The floating license is now set up.

### 6.4 Using Pygments and Minted

Pygments is a Python module for highlighting the syntax of code fragments. Pygments is required by minted (see Section 15.3 for details) to highlight code snippets in Tau LaTeX documents. For instructions on how to install Pygments, please follow this link: [http://pygments.org/docs/installation/](http://pygments.org/docs/installation/).

Note that Pygments has already been installed on lectura, so it is best to compile any Tau Technical Documents using Minted there.
Chapter 7

Guidelines for the Structure of the Working Directory

This chapter describes the required structure of a TAU project under a user’s working directory. It also describes the structure of the TAU projects’ releases and installations folders.

7.1 The Structure of the Working Directory of a TAU Project

Figure 7.1 shows the required directory structure that all TAU projects under a user’s working directory must have. In the next chapter we discuss how to set up the structure of the working directory by using the `fetchProjectName.sh` script. The working copy should not be checked out of SVN manually.

We explain the contents of each subfolder in the working directory structure by visiting each node in a breadth-first-manner. Note that all subfolder names in level 2 are rendered in CamelCase, e.g., **Deployable**, while all subfolders in level 3 and beyond are rendered in lower case, e.g., **apparatus**.

![Diagram of the required structure of a TAU project in the working directory.](image)

Figure 7.1: The required structure of a TAU project in the working directory.

**Level 1**

**SVN-project** This folder is named for the project name of a TAU project. For example it could be **AZDBLab**, **locus**, or **tBDB**. Each such **SVN-project** folder holds the different components of the
project. Under the **SVN-project** folder there are three subfolders. Each subfolder name should be rendered in CamelCase and contain no Greek letters.

**Level 2**

**Deployable** This subfolder contains all the files included in a release that is to be released.

**LocalDocs** This contains UA-specific documentation only. All documents are designated as nonpublishable and local. Additionally, each document can be formal or informal and be part of either user documentation or internal documentation. This directory contains all meeting minutes kept during project meetings.

**Scripts** The scripts in this subfolder are used for the release step, e.g., scripts for creating the `.tar` file. The scripts can also be used for local builds (e.g., a script that would fetch) during the build phase, and required third-party software such as `beecrypt` or drivers.

**Level 3**

All the subfolders below are under the **Deployable** folder.

**docs** This subfolder contains documentation that is released. All documents are designated as nonpublishable and released. Additionally, each document can be formal or informal and be part of either user documentation or internal documentation.

**lib** This subfolder contains statically-linked libraries used by the source code. If a library was obtained from somewhere else, the READE file of that folder should give the source URL and describe the relevant permissions.

**src** This subfolder contains the source code.

**resources** This subfolder contains files used when the project is built. Such files may include, images, data, `xml` files, and schemas.

**Level 4**

All the subfolders below are under the **src** folder.

**apparatus** The subfolder contains files necessary for the LoCuS project.

**scripts** These are scripts included in the release, e.g., Makefiles or `ant` scripts (`build.xml`). The contents of this folder differ from those of the Scripts folder described at level 2. To differentiate between the two folders we capitalize the first letter of the Scripts folder at level 2, while the name of the scripts folder at level 4 is rendered in all small letters. The Scripts folder contains scripts that are used locally (for development, setting up workspaces, creating releases etc.) and are never released. The scripts under the scripts folder are part of the release mainly used for building the project.

**Java-projects** These are Java source files which are part of the system. For example, Java-projects could be GUIs, or AZDBLab Hive.

**Java-packages** These are required Java packages which are imported in other Java classes.

**C++-directories** These are directories containing necessary C++ code.

**C-directories** This subfolder represents C directories. Such directories might be the daemons folder in Dragoon (which contains the notarizer, validator, and forensic analysis algorithms) or the hashing patch for TauBDB.
Level 5

include This subfolder contains C header files.

7.2 Executables

Figure 7.2 shows the required structure of a TAU binaries folder.

 beers /cs/projects/tau installations projectName executableName
 releases projectName projectName.tar

Figure 7.2: The required structure of a TAU project.

There are two directories under /cs/projects/tau named releases and installations. Under each directory there are folders for each project (e.g., azdabl). Under project-specific directory there should be named releases (e.g., 20111123_v5_19) and under that - tar files (or VM images as required by some projects like DRAGOON). Also keep in mind that when we create a release though SVN we just create a tag of the existing files in SVN. Do not tag the tar file. For an internal release place the tar file under a separate directory, namely, /cs/projects/tau/releases/projectname. Under tt installations will be a similar structure, but with executable files. (Recall that when we deploy a project we build the tar file and then install to get executables so that the project can be run locally.)

7.3 Documents

If a document is released then:

(a) If it is a .txt file then it should be placed in under Deployable/docs.

(b) If the file is generated from a source, e.g., from a .fig to .pdf but the source is not released, then the pdf file is placed under Deployable/docs, while the source is placed under the LocalDocs directory for that document.

(c) If the source is to distributed, then under Deployable/docs there is a directory for that document containing both the source (e.g., the .fig file) and the printable (e.g., the .pdf file).

If a document is not released then it is placed under the LocalDocs directory of the project it belongs to.

7.4 UML Diagrams

UML provides a convenient way of representing the structure of large projects, and as such, we have created UML diagrams for most of the TAU projects. As with other documentation in our projects, we have decided on a specific structure of the diagrams, uniform across all the projects. The structure describes the location of the UML diagrams (which are pdf files as well as the VisualParadigm source files) in the SVN repository, in the folder tree for UML diagrams directory, as well as where the diagram source is located.

Some of the TAU projects are too complex to be displayed in one class or sequence diagram. To simplify the diagrams, for each such project there should be three levels of UML diagrams.

1. Package Diagram shows the structure of the packages in the project and dependencies between packages.

2. Class Diagram (name only) shows generalized associations between classes. For large projects, multiple class diagrams, each with a subset of the classes, will be necessary.

3. Class Diagram (complete) shows all attributes, all methods, associations, and generalizations.
It should be noted that while a complete class diagram shows all public attributes and public methods of classes, it does not necessarily need to show all classes. The main goal is to create an understandable and informative representation of the project’s architecture. Thus, which classes to show, and which to leave out of the diagram, is left for the author of the diagram to decide.

Below is the structure of the diagram folder as it rests in the SVN repository.

![Diagram Structure](image)

**Figure 7.3: Structure of the UML folder in the SVN repository.**

There are two files and a number of folders directly under the UML folder. One file is the package (that is, Level 1) diagram of the entire project, and another is a Readme.txt file, which describes what functionality the UML diagram depicts. Each subfolder is the name of the package, obtained by eliding `edu.arizona.cs.projectName` from the beginning of the full name of the package. For example, LoCuS has a package named `edu.arizona.cs.locus.core`. The path to the UML diagrams of the package will be: `.../UML/core`. Note that `package-name` might be composite, e.g. `edu.arizona.cs.locus.gui3d.view` (in this case the subdirectory name would be `gui3d.view`). `package-name`, as well as `subpackage-name` a level below, are the names of the subpackage, without mentioning the names of the packages containing them.

The UML diagrams within these directories are either `ClassDiagram_Level2.pdf` or `ClassDiagram_Level3.pdf`. If there are multiple diagrams at a particular level within a particular folder, their name will be extended by the notion they are describing, e.g., `ClassDiagram_Level2XML.pdf` for those classes dealing with XML processing. An optional `Readme.txt` file provides additional information on these diagrams.

Sequence diagrams are organized with Scenarios, each with a descriptive name, e.g., `Initialization`. Each sequence diagram with a scenario is named by its left-most method, e.g., `main.pdf`, `Locus2d.pdf`. The scenario is located in the folder of the package containing the declaration of the method, e.g., `main.pdf` describes the `main()` method in the code of `Locus2D.java` in the `core` package.

TAU projects use Visual Paradigm as a standard software tool for creating UML diagrams. It is of importance to include the Visual Paradigm project file together with the diagrams, since the project file serves as a source for the diagrams. For details on how to setup Visual Paradigm, please refer to Section 6.3. To find out how to import the project file into Visual Paradigm, see Section 8.3. In SVN, the Visual Paradigm project should go under `SVN-project/LocalDocs/VisualParadigm`.
Chapter 8

Setting Up the Working Directory

As mentioned in Chapter 3, all files related to working within a Tau project should be in a subdirectory of /cs/projects/tau/live/username.

8.1 Fetching the Working Directory

Inside the project SVN repository under trunk/projectname/Scripts there is a script named fetchProjectName.sh. (Note that the script could have the project name as part of its filename, e.g., fetchDragoon.sh.) Export this script by executing:

svn export svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/projectname/Scripts/fetchProjectName.sh

Execute the script in the directory where you would like to set up your SVN working copies for future development. This directory will contain the SVN-project. So for example, to create a directory MyProject as the directory-of-choice execute the following

cd directory-of-choice
./fetchProjectName.sh

the script will create a directory named after the Tau project under the directory of your choice. It will the checkout the working copies from SVN and put them under the newly-created directory. For Dragoon, a subdirectory named dragoon (the SVN-project) will be created, extracted from the Dragoon repository, and containing a Deployable subdirectory.

8.2 Importing the Working Directory into Eclipse

To import a working directory into Eclipse do the following.

1. Locate the working directory that was created by the fetchProjectName.sh script, for example, locus. This will also serve as your Eclipse workspace directory.

2. Open Eclipse. Go to the File menu, click Switch Workspace, select Other, and then select the directory mentioned in step 1. Click OK.

3. Go to Window → Preferences. Under the Mac OS X environment the step is slightly different: go to Eclipse → Preferences.

4. Under Team, select SVN. Make sure that SVN Interface option is set to SVNKit (Pure Java) and click OK.

5. Go to File → Import.
6. Under **General** select **Existing Projects into Workspace** and click **Next**.

7. Make sure that the **Select root directory** radio button is selected and click **Browse**.

8. Select the locus folder created in step 1.

9. Click **Finish** and wait for the projects to be imported into the workspace.

10. Proceed to the next section.

### 8.3 Importing the LocalDocs/VisualParadigm Directory into VisualParadigm

Visual Paradigm stores its diagrams in project files with .vpp extension. Such project file contains all diagrams created for a Tau project. Importing an existing project into your Visual Paradigm workspace is therefore straightforward.

1. Launch Visual Paradigm,

2. Click **File → Open Project...**, and

3. Select the project under LocalDocs/VisualParadigm and click **Open**.
Chapter 9

Using SmartSVN

For all TAU projects we use SmartSVN to manage documents. These include all documentation and papers under project-name/Deployable/docs, project-name/LocalDocs and /cs/svn/papers/ (see Chapter 4). Please do not use Eclipse for managing these SVN projects. We give directions on how to check out specific folders from a TAU SVN project (i.e., LocalDocs and docs) and on how to check out entire papers from /cs/svn/papers/.

9.1 Obtaining SmartSVN

SmartSVN is a graphical client for Subversion (SVN), an Open Source version control system. SmartSVN runs on all major operating systems. It can be downloaded from:

http://www.syntevo.com/smartsvn/download.html?all=true

Select the appropriate .tar.gz file according to your operating system. Unzip and untag the file by double clicking on it. In the resulting folder locate the executable file. For Mac OS X and for the current version of SmartSVN the file is SmartSVN 6.6. (Note that it is best to move the resulting folder under your Applications folder.) For Ubuntu the executable is smartsvn-6_6_11/bin/smartsvn.sh. Double click on the executable to start the client.

A window will ask you what you would like to do. You will be presented with four options. (This is true for both in the Mac and Ubuntu versions of SmartSVN.)

- Check out project from repository
- Open existing working copy
- Import project into repository
- Open existing project(s)

9.2 Checking out a Part of an TAU SVN Project

If you would like to check out part of a project from an SVN repository then

- Choose the first option Check out project from repository and click OK.
- Then specify the project you would like to check out. Select the radio button Quick Checkout (less configuration).
- Specify the URL of the project you want to check out
  svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/project-name/directory-name/
- Specify the **Local Directory** where the checked-out working copy should be placed. You can use the provided **Choose...** or **Browse** button to specify the absolute pathname.

- Click **Continue**.

- In the SSH Connection window enter your login name and password and click **Login**. Both login name and password are your lectura credentials.

Once you have checked out the working copy then under the **Transaction** window you will see a single entry: “No cache information could be found (Build cache now)”. Click on “Build cache now” to get a list of the commits and associated messages.

### 9.3 Opening an Existing Working Copy of TAU Documentation

If you have already checked out a project and you want resume with its management through SmartSVN click the second option from above. (An example of this situation would be when you have an existing working copy which you have modified and now you would like to commit the changes to the repository trunk.)

- In the **Open Working Copy** window specify the **Versioned Directory** field by providing the absolute pathname of the directory where the working copy is located. You can use the **Choose...** button to help you locate the working copy. (Recall that the versioned directory is the one which has the **.svn** file in it.)

- Click **Continue**.

- Select **Don’t manage as project**.

- Click **Finish**.

- In the SSH Connection window enter your login name and password and click **Login**. Both login name and password are your lectura credentials.

### 9.4 Checking out an Entire Paper from SVN

In this section we give instructions on how to check out the entire repository; recall that each paper is a repository. Again you should choose the first option **Check out project from repository**. Then specify the entire **repository** (instead of just a project under that repository) you would like to check out.

- Select the radio button **Quick Checkout (less configuration)**.

- Specify the **URL** of the project you want to check out

  `svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/papers/paper-name/`

- Specify the **Local Directory** where the checked-out repository should be placed. You can use the provided **Choose...** or **Browse** button to specify the absolute pathname. Note that SmartSVN will place the **trunk** directory (and the **branches** and **tags** directories if they exist) under the directory you choose. If the directory does not exist, SmartSVN will create it for you, after asking permission.

- Click **Continue** or **Next**.
Now a new pop-up window will warn you if you want to check out the entire repository. The window message says [Do you really want to check out the whole repository?] . Select [Check Out] .

Select [Finish] .

In the SSH Connection window enter your login name and password and click [Login] . Both login name and password are your lectura credentials.

Once you have checked out the repository then under the [Transaction] window you will see a single entry: “No cache information could be found (Build cache now)”. Click on “Build cache now” to get a list of the commits and associated messages.

### 9.5 Checking Out a New Branch

The steps to check out a new branch are given below.

- Change directory to the parent directory where the new branch will be placed.
- Execute the command

  ```
  svn co svn+ssh://username@lec.cs.arizona.edu/cs/snv/tau/papers/papername/branches/branchname
  ```

A directory of name `branchname` will be created.

### 9.6 Opening an Existing Working Copy of a Paper

In the new window you will be prompted to choose how to manage the project you want to open. Here we can just open existing project.

- Select the fourth option [Open existing project(s)] .
- Underneath the fourth option SmartSVN will provide a list of the existing project(s). Select the project you would like to open.
- Click [OK] .
- In the SSH Connection window enter your login name and password and click [Login] . Both login name and password are your lectura credentials.
Chapter 10

Command Line SVN Techniques

This chapter briefly explains general techniques of using SVN command-line tools to create and checkout a project, commit the changes and merge the changes from a branch into the trunk. However, in general we suggest using the fetch script discussed in Section 8.1 and SmartSVN discussed in Chapter 9 over these commands.

10.1 Checkout

If you wish to checkout project1 from /cs/svn/tau/locus/project1, execute the following:

```
svn co svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/locus/project1
```

(this will create a working copy of project1 in your current directory)

Note that username is your username in lectura. You will also be prompted for your password.

After the initial checkout you can use a caret ^ symbol to denote the repository address, for example:

```
svn co ^/project2
```

will checkout project2 from lec.cs.arizona.edu/cs/svn/tau/locus/.

10.2 Checkout from a Particular Tag or Branch

If you are working with a branch, you would want to checkout from that particular branch. In SVN, branches are implemented as folders under the /branches directory.

Execute the following commands to checkout from a branch:

```
cd checkout-destination
svn checkout svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/locus/branches/branch-name
```

10.3 Keeping a Branch Up to Date

As the development of the project goes on, oftentimes changes are made simultaneously to /trunk and to a separate branch. It is agreed that the latest stable version of the project should be kept in /trunk. SVN provides an easy way to keep a feature branch in sync with the main line of development. Doing this procedure regularly will facilitate integration of a branch back into trunk as well as allow to find and resolve conflicts at an early stage. To update your branch with the latest version of trunk, do the following:

1. Go to the parent folder of the project you’ve been working on, for example:

```
cd /projects/helloworld
```
2. Make sure you have no uncommitted changes by running
   `svn status`

3. Execute the following command:
   ```
   svn merge ~/trunk/helloworld
   ```
   where `helloworld` is the name of your SVN project.

   After this command finishes you might need to resolve some conflicts or modify your code to work with the new revision. Build and test your branch. After ensuring that everything works properly, commit changes to the repository.

10.4 Reintegrating a Branch Into trunk

Assume the scenario were an individual developer has finished his work, e.g., added a new feature to their working copy of the code. If the developer is working on their own SVN branch then the branch changes need to be merged back to the trunk (so the rest of the team can use the new feature). To do this the process is simple.

1. First, bring your branch in sync with the trunk as discussed in Section 10.3
   ```
   svn merge ~/trunk/helloworld
   ```

2. Commit your branch code:
   ```
   svn commit -m "Final merge of trunk changes to my-branch."
   ```

3. Use `svn merge` with the `--reintegrate` option to replicate your branch changes back into the trunk. You'll need a working copy of `~/trunk`. You can do this by either doing an `svn checkout` or using an existing `trunk` working copy somewhere on your disk. This `trunk` working copy cannot have any local edits and must be up-to-date. While these are typically best practices for merging, they are required when using the `--reintegrate` option. Once you have a clean working copy of the trunk, you are ready to merge your branch back into it.

4. Navigate to the folder where the working copy of the `trunk` exists, e.g., `~/home/user/my-trunk`.

5. Using `svn status` make sure there are no uncommitted local changes to the trunk.

6. Issue `svn update` to make sure the `trunk` working copy is up to date.

7. Perform the merge:
   ```
   svn merge --reintegrate ~/calc/branches/my-branch
   ```

8. Commit the merged `trunk` working copy to the repository.
   ```
   svn commit -m "Merge my-branch back into trunk!"
   ```

9. Your branch has now been remerged back into the main line of development.

Please note that once a `--reintegrate` merge is done from branch to trunk, the branch is no longer usable for further work. It is not able to correctly absorb new trunk changes, nor can it be properly reintegrated to trunk again. For this reason, if one wants want to keep working on their merged branch, it is recommended that the branch be destroyed and then re-created from the trunk.
10.5 Committing the Code

To commit your changes to the repository, follow these steps:

1. Navigate to your project’s folder, for example:
   
   `cd ./projects/helloworld`

2. Execute the following command:
   
   `svn ci -m "Comment on this code"`
Chapter 11

Eclipse SVN Techniques

As an alternative to the command line SVN can be used in conjunction with Eclipse. These techniques may be easier than switching between Eclipse and the command line when making modifications to code. Note, these techniques should only be applied to project components that are already being used in Eclipse. Do not use these techniques for other repositories such as those for papers or Technical Documents. Instead use command line techniques or SmartSVN for these projects as described in the previous chapters. For more detailed information on Eclipse SVN techniques see the Eclipse Subversive documentation which can be found at http://www.eclipse.org/subversive/documentation/.

11.1 Checking Out Projects into Eclipse from SVN

To check out projects into Eclipse from SVN, do the following.

1. Create an empty directory, for example, locus. This will be your workspace directory.
2. Open Eclipse. Go to the File menu, click Switch Workspace, and select the directory your created in step 1.
3. Go to Window→Preferences. Under the Mac OS X environment the step is slightly different: go to Eclipse→Preferences.
4. Under Team, select SVN. Make sure that SVN Interface option is set to SVNKit (Pure Java) and click OK.
5. Go to File→New. Click Project..., select Checkout projects from SVN under SVN. Click Next.
6. Select Check out into the workspace as projects, click Next.
7. Select Create new repository location, click Next.
8. Under URL, type in

   svn+ssh://username@lec.cs.arizona.edu/cs/svn/tau/locus

   where username is your department login. Locus repository is used as a reference. Refer to your project-specific documentation to set the correct repository location for your project, branch or tag.
9. Fill in your password and click OK.
10. When asked if you trust the unsigned certificate, click Yes.
11. Review your name and click OK.

12. Under /branches click a branch and select the projects you wish to checkout. See the specific getting started guides for more information on which projects you should checkout.

13. Click Finish and wait for checkout to finish.

14. Right click on the project, e.g., locus, click Run As → Run Configurations. Refer to your project-specific document to set the correct Run Configuration.

11.2 Keeping your Branch Up to Date and Committing Code

In order to keep your branch up to date use the options found in the team menu that can be found by right clicking on the name of your project. Note, certain revisions and actions to your branch will require you to comment before they are performed in a separate window that will appear.
Chapter 12

Using Mantis

The Mantis Bug Tracker system is used for tracking bugs and new features.

12.1 Users

Rick is the sole administrator, doctoral students are designated as managers, and undergraduate and MS students are designated as developers. An administrator can do anything a manager can do, and a manager can do anything a developer can do.

12.2 Mantis Issue Lifecycle

An issue in the Mantis goes through the following lifecycle.

**New** - This is the landing status. Anyone can create an issue.

**Assigned** - The issue has been assigned to one of the team members. The assignment is generally made by Rick, though sometimes the chief programmer will assign an issue. Rick and chief programmers can switch an assignment or unassign an issue.

**Feedback** - When a team member thinks the issue has been resolved, she should add a note explaining how she resolved the issue and how she convinced herself that the issue has been resolved. At that point, she should change the status to feedback and reassign to the chief programmer, as described in Section 12.4.

At that point the chief programmer for the project will decide whether to switch the status to resolved, with a couple of word summary, or switch the status back to assigned, with a note that indicates what need to be done. In either case, the issue should be assigned back to the original team member.

**Resolved** - The issue has been implemented or the bug has been fixed, with the changes committed to the repository (see below). Rick will periodically scan the resolved issues as a sanity check. The team member who completed the issue will remain assigned, thus getting the credit.

Anyone can add a note to any issue. Mantis is our primary repository for what has been done, when, by whom, and why. Team members should not use email for this purpose, as email doesn’t allow the structure that Mantis provides. All technical information regarding a Mantis issue should be communicated via Mantis notes rather than email. (It is perfectly fine to say in an email message, “See issue xyz.”)
12.3 Priority

An issue has a priority.

**Low** - This feature will appear in a future release. A team member should address this issue only if no higher priority issue exists.

**Normal** - This feature or bug fix is planned for the next release.

**High** - This feature or bug fix should be resolved before the next release.

**Urgent** - This feature or bug fix should be resolved quickly, within the next few days.

**Immediate** - This issue should be resolved within 24–48 hours, as it is on the critical path for an important project objective.

12.4 Completing a Mantis Issue

We wish to ensure small granularity of commits to the SVN repository—ideally, a commit per Mantis issue. That said, sometimes a team member will work on a set of related issues, and will commit the files for this entire set, when all of these issues are resolved together.

It is also reasonable for a programmer to work on a few disjoint issues in parallel. When that is done, though, each issue (or set of related issues) should be resolved independently, with an SVN commit for each done separately.

Before committing, for each file in which you made a “substantial” change, add your name to the header of the file.

To simplify the process of integration of new code into the project, we use a script between SVN and Mantis. Upon each commit to the SVN repository, if the text of the comment contains a substring in a format `\[Ii]ssue.*#\[0-9]*\]`, the script will locate the issue referred to in the text of the comment in Mantis and leave a comment there with the list of files changed in the current commit and the developer’s comment, along with some additional information, like the time of the commit, revision number or developer’s name. The script will also locate the current user in Mantis with the role of a manager for the project the issue belongs to, reassign the issue to the manager, and change the status of the issue to “feedback.”

Below is an example of the comment generated by the script from an SVN commit that included in its comment issue #22.

```
------------------------------------------------------------------------
Developer: akvochko, revision 1617 at Sun, 19 Mar 2012 19:50:59 -0700
Issue 0000022 resolved: Added a draft of PlotTool apparatus spec as TD-18.
Files changed:
 AM .../trunk/locus/LocalDocs/ApparatusSpecifications/PlotTool/plotter.pdf
 AM .../trunk/locus/LocalDocs/ApparatusSpecifications/PlotTool/locus.sty
 A .../trunk/locus/LocalDocs/ApparatusSpecifications/PlotTool/locusPlotToolSpec17.tex
 AM .../trunk/locus/LocalDocs/ApparatusSpecifications/PlotTool/locus-logo.png
 AM .../trunk/locus/LocalDocs/ApparatusSpecifications/PlotTool/locusPlotToolSpec17.pdf
 A .../trunk/locus/LocalDocs/ApparatusSpecifications/PlotTool
Details:
------------------------------------------------------------------------
```
Chapter 13

Using Doxygen

Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D. We recommend doxygen for creating source-code documentations, particularly for projects developed in C/C++.

Doxygen has the following three main features.

1. Doxygen can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in \LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

3. You can also use doxygen for creating normal documentation.

Doxygen can be invoked and configured via command-line options. A more visually-oriented approach is eclox, which is a Eclipse plugin that builds doxygen functionalities into Eclipse. This plugin can be installed from http://home.gna.org/eclox/.

To use eclox (inside Eclipse), a developer first needs to select a source code directory. By right-clicking the selected directory, a pop-up menu will show. The developer then selects the New→Other... option. Under the Other option listed under the Wizards, the developer clicks on Doxyfile and chooses Next. By clicking on Finish a doxygen-configuration file will be created under the currently selected source-code directory.

To actually generate the source-code documents, the developer again right-clicks the directory selected in the above step. (Note that this directory must contain the .Doxyfile file.) The developer then selects the Build Documentation option and the documentation-generation task will run automatically. The produced documents will be placed (by default) into a docs directory under the currently-selected directory.
Chapter 14

Reporting Errors and Program Status

For all Tau projects it is essential that we have the ability to access errors that have occurred. This capacity allows for bugs that are found in each project to be reported and eventually resolved. As an example, consider AZDBLab if an executor fails during an experiment we would like to be able to figure out the underlying cause to prevent the failure from reoccurring. Additionally, it is often useful, and sometimes necessary to be able to track information about the current state of an application. For example, LoCuS requires that we maintain updated records of each student’s answers for a lab.

In order to create and maintain these types of records Tau projects make use of either the log4j library and/or Watcher. Both of these methods will be further explained in the subsequent sections.

14.1 Logging Guidelines

For projects developed in Java, we use the log4j library for logging. Logging is necessary for two purposes. First, instead of printing information out onto the screen, for instance, with the `System.out.println()` method, all information should be stored in log files. This will ensure that the execution of the released program is clean, with little to no output to the console. Second, the logs preserve comprehensive history of the execution of a program, with which the debugging and problem tracking is made feasible and easier.

It is recommended to employ a convention of three levels of logging verbosity. The specification of the logging level should be provided as a program invocation argument, for example, `-log 1`. The Java program should include a `public static` variable to keep track of the logging level. It is recommended to utilize one log file per program execution so that it is easy to map the log file to a particular execution session. The name of the log file should contain the program name as well as the time of the execution, such as `azdblab_03212012_15:48:06.log`.

**Level 0** is the default in which no logging messages will ever be written out. Specifically, the logging level variable is defaulted to 0 if no logging-level argument is provided to the program at execution. This level results in no log files being created.

**Level 1** creates log files. The log should include only the errors and exceptions captured during the execution of a program.

**Level 2** contains level 1. Additional program execution details, such as the current cardinality of query execution present in the AZDBLab project, should be logged. This level of logging should not be enabled in the production system, i.e., in installations of the project, because logging extensively can compromise program execution performance as well as taking significant disk space.
14.2 **Watcher**

**Watcher** is a Tau feature for the purpose of monitoring a project and reporting important data from a program. For example, it is used in LoCuS to monitor a student’s progress throughout each lab, record that student’s answer to each questions, and to log any errors that occur in a database.

Watcher can be easily adapted to place information in a log file as described in the previous section, in a database, or in an email if immediate action is required. **Watcher** is thoroughly documented in the “**Watcher Feature Getting Started Guide**”. This guide details the major components of the feature, recommended steps for installing **Watcher** into other projects, as well as, two examples of how **Watcher** has been specifically configured for two separate projects (LoCuS and AZDBLAB).
Chapter 15

Technical Documents

As students often work for a TAU project for as little as one semester, it is important that documentation be accurate, complete, and comprehensible.

If you the reader is just starting with a TAU project, you are in a perfect position to evaluate and contribute to this and other technical documents. Should you find anything hard to understand, or find it technically inaccurate, or find it incomplete, please (a) find out the ground truth and then (b) work with the main author of the document to get this corrected, so that the next person will not have such problems. Each document and reader of that document has benefited from such correction and improvement from prior students. Only through constant involvement by new students will our documentation suite continue to improve.

Pdf versions of technical documents for all TAU projects can be found at http://www.cs.arizona.edu/projects/tau/technicaldocuments/.

15.1 Naming a TD

Naming of the technical document should follow the scheme: projectNamedocumentNameTD#.pdf, where projectName is the name of the project in lower case, documentName is the name of the document rendered in CamelCase and TD# is the number of the technical document. For example: tauGettingStarted1.pdf is valid name for a technical document.

In order to obtain a TD number, send a request to the TAU Chief Programmer that includes the full title of the document.

15.2 The Project Style File

For each TAU project a separate style file is used. If a project has multiple technical documents (each with its own folder) then the (single) style file should be placed at the top level of all the folders so that it can be shared among all technical documents. The style file guarantees that each chapter begins on the recto (“front” side) of a leaf.

The style file includes:

• the rendering of a title page,
• a list of participants and general description of the project,
• a publication history of the document,
• a table of contents.

NB: If the table of contents fits in a single page then add the command \mbox{} immediately before the last \clearpage command in the style file to force the insertion of a blank page.
For the Getting Started Guide for All TAU Projects and the Chief Programmer’s Guide for All TAU Projects the list of participants comprises the following (in the order given).

- The faculty currently (at the time this version is prepared) involved in TAU. The names should appear in alphabetical order with the exception of the director whose name appears first. The faculty member who serves as director should be designated as such by having the word “Director” appear in parentheses after their name. All faculty members external to the University of Arizona should have their names followed by their current affiliation in parentheses.

- A list of everyone who has served as a TAU coordinator in chronological order (latest first) with their tenure period appearing in parentheses after their names.

- A list of the current chief programmers for each TAU project appearing in alphabetical order. Each name should be followed by a comma-separated list (in parentheses) of the projects for which the person serves as a chief programmer.

In all other project-specific technical documents the list of participants comprises the following (in the order given).

- The faculty currently involved in the specific TAU project the technical document belongs to. The same rules apply as the ones described for the Getting Started and Chief Programmer’s Guides for All TAU Projects.

- A list of everyone who has served as a chief programmer in chronological order (latest first) with their tenure period appearing in parentheses after their names.

- A list (in alphabetical order) of current graduate students—other than the chief programmer.

- A list of current undergraduate students in alphabetical order.

In general, the position of director and the affiliation of all external members should be included in parentheses following the names of faculty and students whether current or past. There are several new commands defined in the TAU style file.

- \publication_history fills the Publication history field on the second page of the document.

- \tdnumber fills the TD field on the title page.

- \titlebreak should be used as a linebreak command in the document’s title.

- Commands \Tau, \tdom, \tbdb, \tbench, \azdblab, \dragon, \txschema, \txquery, \tzaman, \tpsm, \locus and \amelie render respective projects’ names.

At the beginning of the document, each TD must have the following code:

```
1 \author{} % author of the TD
2 \publicationhistory{} % publication history of the TD, separated by newline symbol
3 \tdnumber{} % technical document number
4 \logo{tau.eps} % project logo
5 \maketitle % make title page
```
15.3 Listings

TAU technical documents use minted package to render listings. (If your local environment does not have minted, you will need to install it. Follow the minted.pdf instructions included in the distribution. lectura has minted already installed.) The format of listings is described in the project’s style file. For example, the following code renders the listing in C++ language.

```cpp
#include <iostream>
using namespace std;

int main() {
    for (int j = 0; j < 10; j++)
        cout << "Hello, World!" << endl;
}
```

The result is the following.

```cpp
#include <iostream>
using namespace std;

int main() {
    for (int j = 0; j < 10; j++)
        cout << "Hello, World!" << endl; // prints out "Hello, World!"
}
```

The TAU style file defines the following language environments: ccode, cppcode, javacode, latexcode, and bashcode. To find out which environments are available in your project’s style file, please refer to the project-specific Getting Started Guide.

When running pdflatex, remember to use the -shell-escape flag.

15.4 Guidelines for Formatting the Text in Technical Documents

The following are various guidelines that should be observed in preparing the TAU technical documents.

Running prose should be rendered in roman. Quoted text in prose, e.g., “abc”, should be expressed in \textit in LaTeX as ``abc''.

Variables should always be rendered in italicized Courier font using \texttt, e.g., \texttt{username}. Multi-word variables should be hyphenated or rendered in CamelCase where appropriate, e.g., \texttt{directory-of-choice}.

Code examples and fragments should be rendered using the minted package using fixed-width Courier font. Please see the minimum functionality example given above.

Inline code fragments found in running prose should be rendered in Courier font using \texttt, e.g., the line \texttt{for (int j = 0; j < 10; j++)} is responsible for loop iteration.
**Inline typed text** should also be rendered using `{\tt ...}`. This applies to filenames, pathnames (e.g., `/cs/projects/tau/working`), inputs, and parameters. If the inline text ends the sentence, precede with a space. Likewise, if the inline text ends a clause, precede the comma with a space. Any typed text which includes quoted text should be enclosed in " ", e.g., to render "quote" one must use the \texttt{\textquotesingle \textquotesinglequote \textquotesingle} code. Also any typed text that requires single quotes should be enclosed in ' ', e.g., to render 'quote' one must use the LATEX code `{\tt \textquotesingle quote\textquotesingle}`.

**Shell commands** (e.g., `bash`), packages (e.g., `minted`), machine names (e.g., `lectura`), and technologies (e.g., `ant`) should be rendered in lowercase Courier font using `{\tt ...}`. All shell commands given in the text of a technical document should be for the `bash` shell.

**Shell commands on a separate line** should be separated from the main text on either side by a vertical space of \texttt{2mm}. The commands themselves should be rendered in footnote size text. For example, \texttt{\vspace{2mm}\footnotesize export TBDB\_DIR=/home/johndoe/tbdb\vspace{2mm}} produces the following.\vspace{2mm}

```
export TBDB_DIR=/home/johndoe/tbdb
```

The "\vspace{2mm}" on either side of the LATEX code ensures that the command appears separate from the rest of the text. The newline (\texttt{\\}) is used instead of a new paragraph so that the command does not appear indented (the command \texttt{\footnotesize} does not produce the desired result). The command is rendered in \texttt{\footnotesize} so that it is visually consistent with the code fragments rendered under the \texttt{listing} environment.

**Project names** should be inserted using macros (e.g., `{\Tau}` found in the provided LATEX style file, rendered using small caps (\texttt{\sc ...}), e.g., `\Tau`.

**Tab names** and menu selections in GUIs and IDEs should be rendered in Courier font with a box around them and right arrows to denote a sequence of clicks, e.g., `Run As $\rightarrow$ Run Configurations`. This can be achieved by using the \texttt{\fbox{\tt ...}} command (e.g., \texttt{\fbox{Run As}$\rightarrow$\fbox{Run Configurations}}).

**Chapter names** as well as section names should be in quotes when referred to in running prose. For example: The “SVN Repository of Papers” chapter is too long. Titles of technical documents, on the other hand, should not be in quotes. For example: The Getting Started Guide for All Tau Projects is the first technical document all users must read.

**Captions** end with a period, even if they are not a full sentence.

**URLs** should be underlined and rendered using `{\underline{\url{}}}`, e.g., `www.google.com`.

**Technical terms** being defined should be given in italics at definition using `{\em ...}`, e.g., release creation.

### 15.5 Creating Figures for the Technical Documents

The recommended software for creating figures and diagrams are `xfig` and `gnuplot`. If neither has the required features to create a particular image then please use Omnigraffle. Make sure that figures are exported as a `pdf` document.

If you would like to manipulate `jpeg` images, e.g., screenshots, please use GIMP.

### 15.6 Recommended LATEX Typesetting and Packages

All technical documents should be typeset using `pdftex`. This method produces a `pdf` file directly without producing an intermediate `dvi` file. Images in both `pdf` and `jpeg` format are supported and can be included in your `tex` source.
We recommend using the \texttt{minted} package to render code fragments inside the technical documents. See Section 15.3 of the current document for an example.
Chapter 16

Development

16.1 Coding Conventions

Use the standard Sun Coding Conventions for Java. These can be found at
http://www.oracle.com/technetwork/java/codeconv-138413.html

For C, the coding conventions can be found at
http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

For Javadoc conventions please refer to
http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

16.2 Header

TAU projects use specifically-formatted headers in the source files. The header should look like the following.

```c
/*
 * Copyright (c) 2012, Arizona Board of Regents
 * See LICENSE
 * See README
 * Component: daemons
 * File: deployable/src/daemons/notarizer.c
 * Release 1.0, SVN $Revision: 143 $
 * Authors:
 * Kyri Pavlou (http://www.cs.arizona.edu/people/kpavlou)
 */
```

The first line is the copyright holder. For all TAU projects the copyright holder is Arizona Board of Regents. This line should not be modified.

The second line points to the license under which the project is distributed. This information is contained in the LICENSE file at the top level of project releases directory structure.

The third line points to README file for the project. The README file contains information about the documentation generating tools, change list for every minor and major release, and any other information that should be released with the software, but does not fit into the format of a technical document.

The next part specifies the project the file belongs to. This should include the URL for the project. The next line is the component of the project the file belongs to. This is followed by the file name. The SVN revision number is automatically maintained by SVN. So when this file is edited and committed
to SVN the revision number will be automatically incremented. The revision number is updated by the release script as discussed under release management in the Chief Programmer’s Guide for All TAU Projects.

The final part is the list of authors along with each author’s personal webpage URL in parentheses (if the developer doesn’t have a personal webpage, use their email address, by permission). There are no specific rules when a developer should be added to the author list. The only requirement is that the number of changes should be “significant.” Simple changes or bug fixes do not count as significant changes. The order of authors is chronological, with earlier names on the list contributing earlier in time, and with the first name being of the developer who created the file.

After the header, in the appropriate position (immediately after the header in Doxygen, and right above the class declaration for Javadoc), is the description of the file.

16.3 External Program Interactions

Below is a list of any programs that lie outside of TAU that currently interact with at least one TAU project.

- JDBC-ODBC interaction (e.g., SPOCK)
- Interactions with network drivers. (e.g., SPOCK)
- Interactions with MS Office components, such as Access (e.g., SPOCK)
- Interactions with an operating system (e.g., native 32-bit or 64 bit code within a jar file.)
- External databases (e.g., storing lab notebooks external to the cs network as in AMELIE or MOSAIC)
- Connection Strings which are generally hard-coded (SPOCK). Alternatively they are used as part of a plugin (AZDBLAB).
- A plugin for one system that invokes another system (e.g., the AMELIE plugin that is used in SPOCK and HAWK).
Chapter 17

Security Guidelines

When developing server-side software that will network with other client-side software (e.g., smartphone application, website, etc.) outside of the CS departmental domain, extra security measures need to be taken. This chapter includes some guidelines.

The first rule of security when receiving communication for the outside is “never trust the end user.” This may sound odd, but it is quite necessary, as you never know what person is on the other end of the network connection to your server. If the end user has malicious intent, then we need to take precautions to prevent any security loopholes. Sometimes, even users that do not have malicious intent will send malicious content without even meaning to.

Data (strings, integers, etc.) that are sent to any Tau or CS server should never be used directly in any function call (if it will be used to execute something) or database query. If there is a need to call a function based off of data sent from an end user, always first check the data to ensure that it is of the expected type and format. If any data submitted by an end user are going to be used directly in a database query, always cleanse the data first (to avoid SQL injection attacks). In Java, this is as simple as using PreparedStatement for all SQL query submissions. There are methods for cleansing SQL queries in most other languages as well.

One of the most critical aspects of web security is properly dealing with usernames and passwords. Whenever a password is to be stored somewhere server-side (in a database, XML file, or other resource file), the password should always be encrypted. Preferably, only one-way encryption algorithms should be used for passwords. This way, once a password is set it is very difficult to retrieve it maliciously. Usernames need not be encrypted, since usernames are generally publicly viewable information about users.

There are a vast number of security loopholes and vulnerabilities that need to be considered when working with mobile and web applications. For more information, look into the following security topics:

- Password Storage
- Cryptographic functions
- SQL injection
- Cross-Site Coding
- Cross-Site Scripting (XSS)
- Public-key cryptography and SSL
- Input validation and cleansing
Appendices
All active TAU projects have moved to SVN. These appendices apply only to legacy projects.

A Deprecated: Command Line CVS Techniques

This chapter briefly explains general techniques for using CVS command-line tools.

IMPORTANT When working with the `cvs` command from the command line, always make sure that the `CVSROOT` variable is set correctly:

```
export CVSROOT=:ext:username@lectura.cs.arizona.edu:/cs/projects/cvsroot/tau
```

Note that `username` is your username in `lectura`. If this is the only cvs repository you use in `lectura`, you could put the above command in your `.bash_profile` file (in your home directory) so that you don’t have to define `CVSROOT` every time.

After the `CVSROOT` variable is defined (check by executing the command `echo $CVSROOT`), every `cvs` command you execute will be executed relative to the directory defined by `CVSROOT`. For example,

```
cvs update
```

A.1 Checkout

If you wish to checkout `project1` from `/cs/projects/cvsroot/tau/locus/project1`, execute the following:

```
cvs co locus/project1
```

(this will checkout `project1` in your current directory)

A.2 Checkout from a Particular Tag or Branch

If you are working with a branch, you would want to checkout from that particular branch. In CVS, branches are implemented with tag-names.

Execute the following command to checkout from a tag or branch:

```
cd dir-where-you-want-to-checkout

cvs checkout -r tag-name locus/project-name
```

For example,

```
cvs checkout -r tag version-4 locus/helloworld
```

B Deprecated: Setting up Eclipse for CVS

Using Eclipse is the easiest way to work on a Java project. Eclipse has built in support for CVS and is a good IDE for doing Java development. To set up Eclipse, do the following steps.

1. Create an empty directory, for example, `locus`. This will be your workspace directory.

2. Open Eclipse. Go to the `File` menu, click `Switch Workspace`, and select the directory your created in step 1.

3. Go to `File`→`New`. Click `Project`, select `Project from CVS` under `CVS`. Click `Next`.

4. Set `Host` to `cvs.cs.arizona.edu`. Set `Repository path` to `/cs/cvs/tau`. Set `User` to your UA CS username. Set `Password` to your password. Set `Connection Type` to `extssh`. Click `Next`.
5. Select **Use an existing module...**.

6. Select all the project folders/subfolders that you wish to check out and click **Next**. Use **Ctrl** key to select them.

7. Select **Check out into the workspace as projects**, click **Next**.

8. Select **Use default workspace location**, click **Next**.

9. Click on the **Branches** option, then click the refresh tags button.

10. Type in the name of your branch and choose that branch, for example, you may want to access the *locus-branch-qsim* branch for the *locus* project.

11. Click **Finish** and wait for checkout to finish.

12. Right click on the project *locus*, click **Run As** → **Run Configurations**. Refer to your project-specific document to set the correct Run Configuration.