

Project manual

Steffen H. Prochnow

File: project-manual.tex
Revision: 1.20
Revisiondate: May 31, 2007
Revisiontime: 13:04:38

This document describes the management plan for the KIEL project. It tries to put all the ideas collected by the organizers and developers of the KIEL project into one document by which the KIEL project can be developed and maintained.

Contents

1	Introduction	2
2	General guidelines and remarks	3
2.1	Code Organization	3
2.1.1	Top-level Directory	3
2.1.2	Module directories	5
2.1.3	Eclipse Setup	7
2.2	Information resources	8
2.3	Exchanging information	9
3	Coding conventions	9
3.1	General remarks	9
3.2	Code documentation	10
4	Testing	10
4.1	Module Testing	10
4.2	<i>Jlint</i> Testing	12
4.3	Code Review	12
4.3.1	Why do we do code reviews	13
4.3.2	How to do reviews	13
4.3.3	Checklist	14
4.3.4	References doing code reviews	14
4.3.5	System Testing	14

4.4	Documenting Bugs	14
5	Working with code repository	15
5.1	Getting the Code	15
5.1.1	Revision Control Background	15
5.1.2	CVS over SSH Background	16
5.1.3	Getting Your Own Working Copy	16
5.2	Working with the code	18
5.3	Sample CVS Session using command line	18
5.4	CVS Merging	19
6	Writing student papers	20
6.1	Language	20
6.2	Document Composition	20
6.3	Useful things for L ^A T _E X	21
A	User Administration	22
A.1	A Developer joins the Project	22
A.1.1	Project Access	22
A.1.2	Code	22
A.1.3	Documentation	22
A.2	A Developer leaves the Project	22
A.2.1	Code	22
A.2.2	Documentation	22
A.2.3	Project Access	22
B	Revision log	23

1 Introduction

The project manual contains a nearer description of rules of collaboration in developing software for **K**iel **I**ntegrated **E**nvironment for **L**ayout (KIEL). These rules are not arbitrary, but have been proven over numerous developments to help with the following:

- Improving programming efficiency.
- Allowing the re-use of software functions.
- Improving software documentation.
- Making it possible to modify old programs without spending weeks relearning old code.
- Reducing errors by improving the readability of algorithms.
- Improve structure of project artefacts (software, documentation)

Johann Gjertz	pt00jgj@student.bth.se
Tobias Kloss	tkl
Lars Kühn	lku
Florian Lüpke	flu
André Ohlhoff	aoh
Jan Täubrich	jat
Mirko Wischer	miwi

Table 1: Participating persons of the project

- Facilitate dissemination of knowledge within project
- Aid synergy—learning from experience (and errors) of others
- Reduce risk of software-faults by using proven procedure
- Help newcomers to the project to become productive quickly
- Make software usable even after their authors have left the team
- Give team members “project experience”

Programmers are expensive and the goal of any software design should be the reduction in time needed to complete the task. Careful analysis and design of the project are crucial for minimizing the expensive debugging time. By generating structured, welldefined code and algorithms, we can considerably reduce software errors and omissions.

Table 1 lists the participating persons of the KIEL project with their e-mail or account identification. All short identifications should be completed to an e-mail adding the string “@informatik.uni-kiel.de”.

2 General guidelines and remarks

2.1 Code Organization

It is a good idea to look through the code as you read this section. All modules of the project are organized as subdirectories below the KIEL directory (see Figure 1). In the directories you will mostly see two different types of files (excluding the build-scripts). The first have a ‘.java’ suffix and the second a ‘.class’ suffix. The ‘.java’ files are the source files: the *Java* code. These are the files you change and also the only files that are checked into the source code repository. The ‘.class’ files are the results of the compilation process (i. e., when you type ‘make’ in that directory); **they should not be added to the code repository**. A ‘.class’ file is to *Java* what a ‘.o’ file is to *C++*.

2.1.1 Top-level Directory

Go to the top-level of your local development area. Assuming your environment is properly set up, this can be done with:

```

kiel
|-- CVS
|-- documents
|  '-- CVS
|-- examples
|  |-- CVS
|  |-- esterelstudio
|  |-- kiel
|  '-- statemate
|-- kiel
|  |-- CVS
|  |-- browser
|  |-- build
|  |-- configMngr
|  |-- dataStructure
|  |-- doc
|  |-- editor
|  |-- esterel2estudio
|  |-- fileInterface
|  |-- graphicalInformations
|  |-- layouter
|  |-- simulator
|  |-- src
|  '-- util
'-- tools
   |-- CVS
   |-- apache-ant-1.6.1
   |-- apache-ant-1.6.2
   |-- checkstyle-3.4
   '-- graphviz

```

29 directories

Figure 1: Tree-listing of code directories of the KIEL project

```
kiel
|-- CVS
|-- documents
|-- examples
|-- kiel
'-- tools
```

5 directories

Figure 2: Tree-listing of the top-level directory

```
> cd WORKAREA_ROOT
```

Looking in this directory (e. g., ‘ls’) you will see the following subdirectories (see Figure 2):

CVS: This directory contains files used by the ‘cvs’ revision control system. You should not need to do anything in here. Also note that every subdirectory in your development area will have a ‘CVS’ subdirectory: this is how CVS keeps track of the state of your local development environment.

documents: Project documentation. This includes the User’s Guide and other miscellaneous files that are needed. This is also the place to put internal development-related documents that are to be shared among all developers (e. g., functional specifications, design docs, coding conventions, etc.)

examples: Samples that might be delivered as part of the project software.

kiel: The place where all source code will live.

tools: General tools for building and checking the project.

2.1.2 Module directories

A directory listing for the module directories is depicted in Figure 3.

CVS: This directory contains files used by the ‘cvs’ revision control system (see section 2.1.1).

browser: Environment for showing statechart simulation results (contact person: miwi)

build: Compiled java class files of the runnable KIEL project

configMgr: Creates configurations, fundamental for simulator (contact person: aoh)

dataStructure: Java source files of the topological structure of statecharts

doc: Javadoc HTML files of the KIEL project

editor: Environment for editing statechart (contact person: flu)

```

kiel
|-- CVS
|-- browser
|-- build
|-- configMgr
|-- dataStructure
|-- doc
|-- editor
|-- esterel2estudio
|-- fileInterface
|-- graphicalInformations
|-- layouter
|-- simulator
|-- src
'-- util

14 directories

```

Figure 3: Tree-listing of all module directories

esterel2estudio: Transformation of Esterel language to Esterel Studio charts (contact person: lku)

fileInterface: Dialogue window with read/write functionality for several Model data (contact person: lku)

graphicalInformations: Data structure containing graphical Data

layouter: Contains of several layout engines for pretty drawing statecharts (contact person: tk1)

simulator: Environment for simulating behavior of statechart (contact person: aoh)

src: Java source files of the runnable KIEL project.

util: utilities generated by a programmer of any module to provide often used program snippets

All the short account identifications are explained in section 1. Every module directory contains of the following subdirectory structure (Figure 4 shows exemplarily the directory structure of module browser):

CVS: This directory contains files used by the ‘cvs’ revision control system (see section 2.1.1).

build: contains files created by the build scripts

doc: API of the module

lib: common library files and other libraries the code depends on

src: the source code of the module

```
browser
|-- CVS
|-- build
|-- doc
|-- lib
'-- src
```

5 directories

Figure 4: Exemplary tree-listing of module browser

2.1.3 Eclipse Setup

The following steps need to be undertaken once and exactly in the specified order to obtain a working copy of the entire KIEL project for editing, compiling, debugging, and running in Eclipse 3.1.

1. Get a fresh copy of the repository, i.e., type `cvs co kiel` in your workspace directory
2. Type `cd kiel; mv default.classpath .classpath; mv default.project .project`
3. Type `cd kiel; ./build.sh cleanall; ./build.sh KIELbuild`
4. Start Eclipse
5. Import an existing project into workspace.
6. Select your workspace directory as root directory.
7. Make sure that the project `kiel` is selected.
8. Create a new run configuration, New Java Application.
9. Set `kiel.KielFrame` as "Main Class".
10. Set the working directory on page "Arguments" to `${workspace_loc:kiel/kiel}`.
11. Enter the following VM arguments
 - a) `-Djava.library.path=${workspace_loc:kiel/kiel/checking/lib}`
`:${workspace_loc:kiel/kiel/layoutter/lib}`
 - b) `-Dlayoutter.path.other=${workspace_loc:kiel/kiel/layoutter/lib}`
 - c) `-Xmx512M`
12. Apply the settings.
13. On page "Environment", enter the following variable

- a) Variable: LD_LIBRARY_PATH,
Value: `${workspace_loc:kiel/kiel/checking/lib}`
`:/home/java/j2sdk1.4.2_02/jre/lib/i386/client`
`:/home/java/j2sdk1.4.2_02/jre/lib/i386`
`:/home/java/j2sdk1.4.2_02/jre/./lib/i386`

14. Apply the settings and hit "Run".

2.2 Information resources

The project manual does not contain of all relevant informations, which are important for developers of KIEL. Further briefings will be distributed in course of working or published at the project home-page (<http://www.informatik.uni-kiel.de/~rt-kiel/>). At the project homepage literature lists will be published as well as hints and documentation for tools and techniques.

- Programming language: *Java*, J2SE 1.4.2
- Language for code documentation and object naming: english
- Infrastructure:
 - Software modelling: ArgoUML, Eclipse, Rational Rose
 - Version management: CVS
 - Development: JBuilder
 - Checker: Checkstyle, Jlint
- coding conventions: *Java* coding conventions as plugin for several IDEs
- Copyright Guidelines—Facts in Brief
 - Copyright is a form of protection provided to authors of “original works in authorship,” including literary, musical, artistic and other intellectual works.
 - Copyright is secured automatically upon creation—even for works not bearing a copyright notice.
 - Copyright registration is not required but establishes a public record of the copyright claim and is necessary prior to filing an infringement suit.
 - Standard forms of copyright notice:
 - Copyright 2004 Jane Doe
 - © 2004 Jane Doe
 - Copyright © 2004 Jane Doe

ArgoUML:	http://www.argouml.de/
Checkstyle:	http://checkstyle.sourceforge.net/
U.S. Copyright Office:	http://www.copyright.gov ,
Fair Use Information:	http://www.copyright.gov/fls/fl1102.html
CVS:	https://www.cvshome.org/ http://www.gnu.org/software/cvs/
Eclipse:	http://www.eclipse.org/
Java:	http://java.sun.com/j2se/index.jsp
JBuilder:	http://www.borland.com/jbuilder/
Jlint:	http://artho.com/jlint/
Rational Rose:	http://www-306.ibm.com/software/rational/

2.3 Exchanging information

3 Coding conventions

3.1 General remarks

The main task that the group must accomplish for this assignment is to make the code conform to the project coding conventions. Since only the owner is allowed to modify their code, this will require everyone to chip in to get this done. The end result of this assignment should be all the modified code checked into the CVS repository, conforming to the code conventions and still working (i.e., able to be compiled and run without errors).

Before embarking on making code changes, everyone will first need to read these conventions: <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>. I doubt that you will get it completely right the first time, but I am expecting you to give it a good try and get most of the major points correct. As you get more experience with them, and as others point out where there are problems with your use of the conventions, the code will begin to creep closer and closer to compliance.

Since full conformity will only happen gradually, and is not necessarily required early on, I have decided to explicitly enumerate the minimal things that your code must conform to. The list of things you must do are:

- Write English code (e.g. names for classes) and code documentation
- Add headers to all .java files
- Add original copyright within the header (see below)
- Properly indent all the code (emacs will help)
- Ensure only one variable declaration per line (no comma separated lists)
- Ensure no lines are longer than 80 characters
- Ensure opening and closing curly brackets on their own line

- Explicitly enumerate, sort and group all ‘import’ statements.
- Separate the public, protected and private vars/methods into separate sections

Note that this is just a listing of the tasks that must be done. The detailed requirements for each of each are found in the coding conventions.

Java Code Conventions: <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

3.2 Code documentation

Important: Your code documentation has to satisfy the criteria of *Java Coding Conventions*, p. 5 ff. When you are altering the source code to make it conform to the Java Coding Conventions, you should also do the following: The documentation comments just prior to the ‘class’ definition of each file should contain an ‘@author’ line. Here you should preserve the original author names and also add new line documenting the owner and co-owner by adding author1 and author2 (see Figure 5). Here the owner is “Tobias Kloss” and the co-owner is “Florian Lüpke”.

Java Code Conventions: <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

4 Testing

There are two sections to program testing. The first is testing of each module in the program. This is similar to conventional debugging. The second is testing of the complete program or instrument. This tests not only how well the modules behave together but it also tests how well the software fits the application.

4.1 Module Testing

It is absolutely essential that after the software developer codes each module, that he/she debugs each module. Don’t wait until you code all the modules before testing them—test each module as you write them. If you wait, an oversight in one module might have a ripple effect throughout all the other coded modules requiring a lot of effort compared to coding and testing each module separately. If you follow the encapsulation techniques mentioned earlier, the testing should be easy and straightforward.

The software developer must test each module before incorporating it into the program. Sometimes it is not possible for you to test a module stand-alone. In this case, you must identify a technique that unequivocally tests the module. You may also test a module using a “test” program to verify its functionality, or you can check it with a software debugger for proper operation. Whichever route you take, you must document the testing technique and the applicable test results. If you generate a separate test program, you must save this as part of the documentation.

```

// $Author: spr $
package kiel.fileInterface;

/**
 * <p>Description: A generic File Interface including GUI Elements for the
 * KIEL Projekt. It has a Plugin mechanism for including FileFormats.</p>
 * <p><b>Details on this special class:</b> This the Interface all
 * Fileformat Parser/Generator must implement.
 * </p>
 * <p>
 * References: <a href="http://www.informatik.uni-kiel.de/~rt-kiel/kiel/
 * documents/papers/examples-wristwatch/documentation/documentation.pdf">
 * Examples</a>
 * </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Uni Kiel</p>
 *
 * @author <a href="mailto:miwi@informatik.uni-kiel.de">Mirko Wischer</a>
 * <br> <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss</a>
 * <br> <a href="mailto:flu@informatik.uni-kiel.de">Florian Luepke</a>
 * @version $Revision: 1.2 $ last modified $Date: 2005/03/15 12:46:15 $
 *
 * $Log: ex-header-cvs.txt,v $
 * Revision 1.2 2005/03/15 12:46:15 spr
 * changed cvs specification
 *
 * Revision 1.1 2004/10/14 08:07:03 spr
 * *** empty log message ***
 *
 * Revision 1.3 2004/10/06 14:02:56 spr
 * *** empty log message ***
 *
 * Revision 1.1 2004/07/12 08:25:32 miwi
 * Adding new Plugin for serialized Kiel
 *
 */

```

Figure 5: Example header in source code file

Independent of the language translator, you must evaluate the compiler during initial system development if it has not been used before. For the first 500 lines of source code produced by the programmer, the programmer will evaluate the output of the compiler for correct translations. In addition, the programmer will obtain from the compiler's (and library) manufacturer a list of all known bugs within the compiler and will, if at all possible, have his/her name on a list to receive all future bug reports. All bugs that the programmer finds and that are not on the bug report will be listed on an internal bug report, including ways to work around the problem. The programmer will send this internal bug report list to the compiler's manufacturer in a timely manner. If the manufacturer releases a new version of the compiler, you must check this new version for errors and compatibility with the older version. Occasionally, manufacturers release new versions that have more errors than what is being replaced.

As you test and make each module work, enter the testing date and test method for each module in the Module List. If while testing a module you find a previously undetected error, fix the module, modify the testing procedure, and then retest the module. Once you test a module, don't change it. If you do change it, you need to retest it.

4.2 *Jlint* Testing

Jlint is a utility program that identifies questionable sections in source code (at least in *Java*). It does this by looking at all the source code, and identifying common programming mistakes, oversights, and questionable constructs. It will not necessarily find program "bugs". Typical examples of what *Jlint* finds includes uninitialized static variables, unused variables, and unintentional endless loops. An analogy for *Jlint* is using a spelling checker—it can find spelling mistakes but it cannot find if you are using a wrong word, like than instead of then. You can easily tailor the *Jlint* program to a specific coding style so that it does not log unrealistic problems.

The *Jlint* procedure is an iterative process. It is not worth running *Jlint* and then not changing the source code to reflect what it found. On the other hand, just because *Jlint* says that something is questionable, this does not make it necessary to change the source code. The programmer knows the code better than *Jlint* and therefore the programmer's intuition must always win out.

Jlint is run not only on a module, but you also run it on the entire program. Unlike a compiler, *Jlint* can find problems and inconsistencies across modules.

Jlint: <http://artho.com/jlint/>

4.3 Code Review

After *Jlint* evaluates the program source code, the next step is a Code Review by someone other than the programmer. A different programmer performs the Code Review, who looks at the source code in more detail than *Jlint* can. The reviewer is looking, for example, for bad constructs, poor structure, ambiguity, commenting, and even poor algorithm implementation.

4.3.1 Why do we do code reviews

Code reviews will result in higher quality and a better conformance to standards which will result in a better maintainability of the software. The reviewers will gain better knowledge about what the others in the project are doing, and also learn from others mistakes and solutions.

4.3.2 How to do reviews

A review should be done with one reviewer together with the developer. The code should already be in a tested and stable state when the review takes place. Doing the review too early can be a waste of time. Before the human review is made you should use the checkstyle tool to make sure that the code follows the standard coding conventions. The scope of the review should be limited. One of the reviewer's key responsibilities is selecting the appropriate amount of content to review. You should focus on reviewing selections from the code that are most critical. To make the review as effective as possible the reviewer should use a checklist (section 4.3.3) when doing the review. For every review there should be some information reported:

- the reviewer,
- the developer,
- lines of code reviewed and
- errors found, (types and severity of the error).

The reviewer will log all comments, suggestions, and problems either on a standard form or in a database. The reviewer will not amend or change the source code in any way. It is also not necessary to print out the entire program. The reviewer can do the review using a PC screen. At the end of the code review, the reviewer will give all the forms to the programmer.

During the course of the code review, the reviewer may wish to test, check, or verify the operation of a particular function or module. The reasons for this might be because of code ambiguity, algorithmic questions, or just as a spot check. It is not necessary for the reviewer to explain why he/she tested a section of code. The reviewer will log any testing methods that he/she does and the results of the test.

Sometimes the reviewer will identify a section for testing but cannot devise a proper test. The reviewer will also log this information. Logs should be done to the documents you find in the repository directory `kiel->documents->code-review`. After logging don't forget to commit your file to the repository.

After the reviewer finishes the code review and any testing, he/she will give the results to the programmer. The programmer will then either fix the source code or not. Whatever the programmer decides to do must be logged onto each of the forms.

After the programmer finishes with the forms and changing the source code, the programmer gives the forms back to the reviewer. The reviewer will review the changes and

make sure that the programmer has satisfactorily resolved or explained the questionable areas. The reviewer will again log onto the forms any additional comments concerning the source code. In other words, this is an iterative process where the iterations are finished when the source code satisfies the reviewer. Hopefully this should not take more than three iterations.

4.3.3 Checklist

This is the checklist that should be used when doing a review.

4.3.4 References doing code reviews

- <http://www.linuxjournal.com/article/4388> ([2])

Statements:

“... There are two basic types of reviews: walk-throughs and formal inspections. In a walk-through, the author takes one or more colleagues on a tour of the document under review. In my experience, about 80 % of the errors found in a walk-through are actually found by the author in the process of explaining the document. Eli Weber, a former colleague, said ‘If I could talk to the wall as if it were a person, I wouldn’t need someone else in order to do a code review.’...”

“... it is better to have a modest inspection methodology that actually gets implemented versus a grandiose scheme for which there is never quite enough time to get implemented at all...”

- http://mentalmodels.mitre.org/cog_eng/reference_documents/asurveyofsoftwareinspectionchecklists.pdf ([1])

4.3.5 System Testing

Besides individually testing each module, you need to test the entire program to make not only sure that all the modules function together properly but that the software and instrument meet the needs of the end-user. Before performing system testing, you must create a *system test document*. You must then test the system according to this document to prove that the system works. When the system passes its testing, only then can you deliver it to the end-user. Remember during the system testing that things other than software problems can make a system appear to fail its tests. Sure, software a lot of times has bugs but keep an open mind and realize that other parameters, especially in a jury-rigged or simulated environment, might not truly represent the intended application.

4.4 Documenting Bugs

Developers participating the project KIEL use the Bugzilla system to report the occurrence of Bugs. Bugzilla is a bug-tracking system and is used to submit and review defects that have been found in software products. Bugzilla is not an avenue for technical

assistance or support, but simply a bug tracking system. If you submit a defect, please provide detailed information in your submission after you have queried Bugzilla to ensure the defect has not been reported yet. Defects will go directly to the developer responsible for the component you filed the defect against. Developers have many responsibilities and will get to your defect in due time. It's absolute essential to follow the Bug Writing Guidelines given by Bugzilla.

What is Bugzilla?

<http://rtsys.informatik.uni-kiel.de/bugzilla/docs/html/whatis.html>

Why Should We Use Bugzilla?

<http://rtsys.informatik.uni-kiel.de/bugzilla/docs/html/why.html>

Bug Writing Guidelines

<http://rtsys.informatik.uni-kiel.de/bugzilla/bugwritinghelp.html>

Bugzilla Usage FAQ

<http://www.bugzilla.org/docs/2.16/html/faq.html#faq-use>

5 Working with code repository

Everyone needs to change some code and everyone needs to be able to put their changes back into the CVS code repository. This will require using CVS to “commit” your changes and to use CVS to “update” your local copy with changes from other developers on the project.

5.1 Getting the Code

5.1.1 Revision Control Background

The basic premise for tracking changes and versions of your source code and project files is that there is a central code repository that has the “master” copy, including all the revision history of what was changed, who changed it and when it was changed.

If everyone on the team had to share a single copy of the code, there is too much coupling of the process and too many opportunities for each team member to interfere with the other. A much better way is for everyone to have their own personal copy of the code as this decouples their work and let's them tailor things to their needs and habits if they so desire.

Of course, with n developers there potentially n different versions of the code at any given point, but we really just need a single copy and need it all eventually to be consistent and coherent. This is where the source code repository comes in.

The repository is the central master version. The individual developers then have the responsibility to synchronize their local copy of the code with this central repository. The individual members are then free to alter there local copy in any way that is convenient. However, they will have to keep track of which changes are for their convenience and which one will actually be needed by everyone. In general, this is a hard problem, but luckily tools exist to help the developers keep things in synch.

The tool used is generally referred to as a revision control system. There are many of them out there, most of them commercial and costing hundreds or thousands of dollars.

However, they all operate on the same basic premise and most of the more expensive one are also some of the worst ones from the developers perspective.

For this class, we will be using an open source system called CVS. It is simple enough that it does not get in the developers way, but sophisticated enough to do everything you need a revision control system to do.

5.1.2 CVS over SSH Background

Before you can start doing any development on the project, you need to get your own personal working copy of the code. The source repository lives on a GNU/Linux. For now, don't worry too much about who or how the source repository was set up initially. Just be happy I have done it for you, so you can just assume it exists.

If everyone on your team was developing code in the same physical building, or on the same physical, secured network, then your connection to the bug database and the source repository would be simplified as the security aspect would not be as big a concern. Since in our case this machine is world-accessible, we have to be very careful. Just like the bug database required some extra machinery to protect it from worldwide abuse, your access to the source code repository has an extra twist to it.

In particular, you need to use a secure connection to interact with the repository: this takes the form of using SSH. To do this requires what looks like computer wizardry. For now, I will not explain it, just do as it says below, but it really isn't that complicated.

Note: When you are working on the code development, generally you do not need any network connectivity. All your work is done on your local copy. It is only when you want to add your changes to the "master" repository or fetch someone else's changes that were put into the repository that you will need a connection.

5.1.3 Getting Your Own Working Copy

Your first step is for you to decide where in your personal file space you want to put the code as you work on it. It can go anywhere, but if you don't have a preference I would suggest the directory named (this is just a directory name, not a command or something you need to type):

```
~/proj/workarea
```

Everything to follow assumes you will be using this location. It is not necessary to use this location, but if you choose another, you will need to adjust the discussion below for your directory.

First thing to do is create the directory where the code will go (this is a command to be typed in a shell):

```
> mkdir -p ~/proj/workarea
```

With this directory created, make it the current directory with this command:

```
> cd ~/proj/workarea
```

Next up is to set some environment variables. These are like variables in a programming language, but are used as you run commands. Note that below we'll show you how to set these so that they are permanent so you get them for "free" each time you log in rather than having to manually type them over and over again.

If you work with Linux you have to add the following to your `.bashrc`:

```
if [-z "$CVSEEDITOR"]
then
  if [ -n "$EDITOR" ]
  then
    CVSEEDITOR=$EDITOR
  else
    CVSEEDITOR=emacs
  fi
fi

[ -n "$CVS_RSH" ] || CVS_RSH=ssh

[ -n "$CVSROOT" ] ||
CVSROOT=:ext:cvs@launchpad:/home/cvs/CVSROOT

export CVSEEDITOR CVSROOT CVS_RSH
```

This actually sets the following environment variables:

- `CVSEEDITOR`: Specifies the program to use to edit log messages for commits.
- `CVS_RSH`: Specifies an external program for connecting to the server when using the `:ext:` access method.
- `CVSROOT`: This specifies the path to the repository.

To get in the way of typing your password connecting the CVS server you should create a public key for your account using the command `ssh-keygen -t dsa`. After them send your `.ssh/id_dsa.pub` to the project manager, to authenticate your account to the CVS account.

If you get a password prompt, but it does not accept your password, verify that your password is correct and that you are typing it exactly as shown (it is case-sensitive). If you still have problems, let me know. If you never get to the password prompt, then either you do not have a network connection to that machine, ssh is not installed, cvs is not installed, or you typed something wrong (or any combination of the above). Check to eliminate all these possible problems.

If all the work is done, now can check out your local copy of the code repository. Type

```
> cvs co kiel
```

and the whole repository establishes on your local system.

5.2 Working with the code

Important Convention: No code should be checked into the CVS repository if it does not compile. Far better than this, it is better that you can not just compile your code, but that you have also tested your code reasonably enough to know your changes didn't break anything. It is a mortal development sin to check in code which does not compile. Eventually, you will be running “nightly builds” of your project. Here, each evening an automated process will check out all the code from CVS, compile it and run some tests. This is a sanity check to make sure the development team's efforts of the day did not cause major disruptions to the code-base. When you check something in that does not compile, the nightly build will break and everyone will get email the next morning showing not only the failure, but the reason for the failure. This is known as “breaking the build” and usually entails some punishment, if only in the form of disdain from your co-workers.

CVS is a full-featured revision control program. The full documentation for command line using can be found at the CVS Manual Page <https://www.cvshome.org/docs/manual/>. You should at least read through the introductions and parts of the manual pages that talk about the repository. For the class assignments a few CVS commands will give you the bulk of what you'll need. You should read the CVS manual pages about these specific commands:

- `cvs commit [filename(s)]`—for storing your local changes in the repository;
- `cvs update [filename(s)]`—for getting other people's recently committed changes from the repository;
- `cvs diff [filename(s)]`—for a line-by-line comparison of your local copy to what is stored in the CVS repository;
- `cvs status [filename(s)]`—for a high-level view comparison of your local copy against what is stored in the CVS repository;
- `cvs add [filename(s)]`—adding new files/directories into the CVS structure;
- `cvs remove [filename(s)]`—removing files/directories from the CVS structure.

You should note that if you execute these commands without giving any filenames, then it will perform the action on the entire subdirectory as well as any of its subdirectories. **Note:** Be careful with adding directories. Once checked in, they can't be deleted by the users.

CVS Manual Page: <https://www.cvshome.org/docs/manual/>

5.3 Sample CVS Session using command line

Let's suppose I am responsible for the Board.java, Rules.java and Move.java source files. I spend a half-hour working on my local copy of Board.java making it conform to the

coding conventions. When I think I am done, I first recompile the code and make sure it still runs. If I find I've broken the compile, I go and fix the problems. Once it compiles, I should test it to see whether or not I broke the program logic. If it is broken, I go and fix it, otherwise I am ready to check it in.

Once I am satisfied that my reformatting allows the code to still compile and work, I decide to check these changes in before moving onto the other code. This is both a safety to make sure I do not accidentally wipe out these changes and also to make sure the rest of the development team can access this new version.

To commit my changes I simply type:

```
> cvs commit Board.java
```

Now CVS will fire up the editor (see the section on editors). It will be predefined with some comments indicating which files are being committed and such. You should enter your comments before the predefined comments. When you are finished with the message, save the file and exit the editor. When the editor shuts down, CVS resumes it's work and goes about the process of committing the code.

Now, before I start working on the other code, I want to check to see if anyone else on the project has checked in any of their changes. I can check this with:

```
> cvs status *.java
```

This will give me a status message for all '.java' files in that directory. It tells me if my local versions are up-to-date, modified or if they need a checkout. IF I need a checkout, then that means someone has committed a change to the repository. Let us suppose the HumanPlayer.java file said I needed an update. I could get the latest CVS version with the command:

```
> cvs update HumanPlayer.java
```

CVS fetches the new versions and update my local copy.

5.4 CVS Merging

So what happens if I make changes to my local copy and then find someone has modified and committed their own changes to the same file? In theory, on your project, this should not happen, since only one person is responsible for modifying any particular piece of code. However, in reality this happens and someone needs to deal with it.

The philosophy of working with CVS says that the first person to commit a change to the repository is the "winner". Anyone with other changes to the same code that was not yet committed is the "loser" and becomes responsible for reconciling their changes with the winner's changes.

CVS actually provides a mechanism where it tries to "merge" the two versions. In this case, the "loser" (i. e., the person that did not commit their code) will do an update to fetch the "winner's" version. During the update, CVS will attempt to merge the two pieces of code. This merging process is actually fairly robust, though not infallible. In

effect, if the winner's changes and the loser's changes concern different parts of the code, CVS will be able to merge them just fine. This doesn't guarantee that the changes are not in "logical conflict", but it does mean they are probably not in "syntactic conflict".

In the cases where CVS is able to successfully merge the the winner's and loser's code, it is still the responsibility of the loser to ensure that the code still works before checking this merged versions back in. Notice that when CVS does a merge, it is not making chnages to the code in the repository. It is only giving the loser a merged version. The loser will have to separately commit this merged version after verifying that nothing has been broken.

There are times where "syntactic conflicts" cannot be resolved by CVS. When this happens, during the 'update' messages, you will see CVS warning you that there were "merge conflicts". You need to pay particular attention to these, because in this case CVS actually inserts text into the file. The text that is inserted is meant to highlight the conflicting pieces of code. It is guaranteed to make your code un-compileable so you will have no choice but to edit this file and resolve the conflict yourself.

6 Writing student papers

6.1 Language

It is up to you to compose your documentation in English or in German. If you decide to write the documentation in another than your first language a qualified corrector should read it.

6.2 Document Composition

Your document should be composed of the following parts:

- Title
- *Eidesstattliche Erklärung*
- abstract
- preface
- table of contents
- table of tabulars
- table of figures
- possibly other tables
- your own chapters
- appendix, consisting of

- a manual for the developed software
- a general survey of code implementation,
- the class and other UML diagrams and the
- the full and documented source code
- a bibliography
- possibly a glossary
- index.

Former information about writing student papers gives the Website <http://www.informatik.uni-kiel.de/inf/von-Hanxleden/Diplom/hinweise.html>

Functional specifications: <http://mojofat.com/tutorial/>

6.3 Useful things for \LaTeX

The Not So Short Introduction to $\LaTeX 2_{\epsilon}$	http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf
$\LaTeX 2_{\epsilon}$-Kurzbeschreibung	http://www.dante.de/tex-archive/info/lshort/german/l2kurz.pdf
$\LaTeX 2_{\epsilon}$ FAQ (in English)	ftp://cam.ctan.org/tex-archive/help/uk-tex-faq/newfaq.pdf
$\LaTeX 2_{\epsilon}$ FAQ (in German)	http://www.dante.de/faq/de-tex-faq/de-tex-faq.pdf
listings, fancyvrb	typeset programming code within \LaTeX

A User Administration

A.1 A Developer joins the Project

A.1.1 Project Access

- Hand out the project manual and the review form.
- Add the new user's public key to the `ssh` key ring of user `rt-kiel`. Therefore, edit the file `authorized_keys` in the `CVSROOT` directory.
- After a adequate period of vocational adjustment register the bachelor project/-master thesis/diploma thesis.

A.1.2 Code

- Add the new software module directory to the KIEL root directory.

A.1.3 Documentation

- Add the new user's documentation directory to the KIEL directory `papers`. The user's documentation directory should be identical to the user's e-mail short name.

A.2 A Developer leaves the Project

A.2.1 Code

- The code should conform to sections 2 and 3 of the KIEL project manual.
- The code should contain understandable and compilable *javadoc* documentation.
- The code should pass the *Checkstyle* run.
- The code should pass the *JLint* run.

A.2.2 Documentation

- All the sources of the final documentation paper should be organized in the working repository.
- The \LaTeX source should be compilable by the project manager.
- All the read electronic literature should be organized in a subdirectory `literature` of the main documentation directory.

A.2.3 Project Access

- Remove the user's public key from the `ssh` key ring of user `rt-kiel`. Therefore, edit the file `authorized_keys` in the `CVSROOT` directory.

B Revision log

Revision 1.20 2007/05/31 13:04:38 spr
misc

Revision 1.19 2006/10/23 17:36:23 spr
added user's administration

Revision 1.18 2006/06/14 15:26:34 spr
added user administration structure

Revision 1.17 2006/04/10 14:51:31 gsc
renamed robustness to checking

Revision 1.16 2006/03/27 08:24:18 gsc
removed .classpath and .project from repository,
updated eclipse setup instructions in project manual

Revision 1.15 2006/03/21 12:42:12 gsc
changed eclipse setup to accomodate jcvcl lib

Revision 1.14 2006/01/30 09:59:27 kbe
added VM arguments to eclipse setup

Revision 1.13 2006/01/30 09:42:15 kbe
modified the eclipse setup

Revision 1.12 2006/01/27 15:48:20 rt-kiel
added eclipse setup intructions

Revision 1.11 2005/03/11 14:48:18 spr
added section Writing student papers

Revision 1.10 2005/03/10 14:36:50 spr
added cvs log

Revision 1.9 2005/02/21 16:07:01 spr
added code review references

Revision 1.1 2004/10/13 15:31:34 spr
project-management moved to own subdir

References

- [1] Bill Brykczynski. A survey of software inspection checklists. *SIGSOFT Softw. Eng. Notes*, 24(1):82, 1999. <http://doi.acm.org/10.1145/308769.308798>.
- [2] Larry Colen. Code Reviews. *Linux Journal*, 2001(81es):11, 2001.