

Software Configuration Management Course
Vrije Universiteit Amsterdam
- Practical Work -

Niels Veerman and René Krikhaar

Document version as of:
May 12, 2006

Contents

1	Introduction	2
2	Warming-up	3
2.1	CVS concepts and commands	3
2.2	Setting up and using a repository	4
2.2.1	Setting up a repository (done once per team)	5
2.2.2	Using the repository (done by several members)	5
2.3	The sample application <code>hello</code>	7
3	Case: Run your software company	9
4	CVS and Eclipse	10
4.1	Accessing the CVS repository	10
4.2	Configure and run an application	11
4.3	Accessing and comparing revisions	13
A	Resources	17
A.1	CVS quick command reference	17
A.2	Online resources	17
B	Source code of the sample application	18
B.1	<code>hello.java</code>	18
B.2	<code>runtests</code>	19
B.3	<code>Makefile</code>	21
B.4	<code>.cvsignore</code>	21

1 Introduction

The practical work is intended to let you practice with Software Configuration Management concepts. Your team represents a company which has several customers. Customers are represented by the SCM lecturers (René and Niels), which will come up with feature requests during the practice. Different customers request different features, so you will have to develop, maintain, and keep track of several releases of the same product. For proper and consistent coordination and communication among the team members, you must create and maintain a Software Configuration Management plan (SCM plan) which describes your company's SCM policies and procedures.

To support your Software Configuration Management process, your company uses CVS: the Concurrent Versions System. CVS can be used as a commandline tool, but is also available as a plugin for Intergrated Development Environments (IDEs), including the Eclipse platform. In the practical work, we start with CVS on the Unix commandline, and after that we move on to Eclipse.

Overview of the practical work

- **Warming-up (3/4 hour)** In Section 2, we give a brief overview of the CVS concepts and commands, and we explain how you set up and use a CVS repository.
- **Company case (2 hours)** In Section 3, your company starts its production. First your team should create an initial SCM plan. When the initial plan has been finished, you can briefly discuss it with the lecturers, and put it under version control. Then, you are ready to start working for the first customer. The lecturers will supply you with feature requests during the practice.
- **CVS and Eclipse (1/2 hour)** After you have finished the Company case from Section 3, continue with Section 4 to see how you can use CVS with the Eclipse IDE.
- **Evaluation (1/2 hour)** In the last half hour, we will evaluate the practical work with all participants.

Goal The goal of the practical work is to practice with Software Configuration Management concepts by creating and using an SCM plan in a company setting.

Preparation Study Chapter 4 and Chapter 5 of the CVS manual: "Version Management with CVS" by Per Cederqvist et al. (see Resources in Appendix A).

Assessment At the end of the practice, your team submits the final version of your company's SCM plan. Intermediate versions of the plan must be stored in CVS.

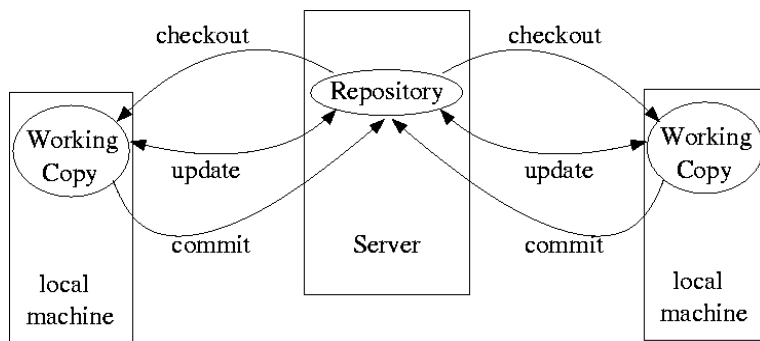


Figure 1: A CVS repository and the interaction with working copies.

2 Warming-up

2.1 CVS concepts and commands

CVS – Concurrent Versions System – is a tool to organize source-code in development. CVS is useful for large, distributed teams as well as for individual developers. Read the following concepts and commands carefully before starting the practical work. We also advise you to consult during the practice the CVS manual by Per Cederqvist for more details on specific CVS commands, in particular Chapter 4: 'Revisions' and Chapter 5: 'Branching and merging'.

- **Repository** The CVS 'repository' stores a complete copy of all the files and directories which are under version control. An illustration of the CVS repository concept is shown in Figure 1. You use CVS commands to get your own copy of the files into a 'working directory', and then work on that copy. When you have finished a set of changes, you check (or 'commit') them back into the repository. Furthermore, you can update your working copy with changes from the repository.
- **Module** A specific directory (or mini-tree of directories) in the repository, which can consist of files and other directories. A module can represent an individual component or even an entire product. To work on a module, it must first be checked out, which creates a local working copy.
- **Working copy** A personal copy of a module that a developer can edit independently. Can be retrieved by a check out and stored by a commit.
- **Log message** A message that can be entered when committing changes to the repository. This is very helpful to understand why a change was made.
- **Revision** CVS assigns revision numbers to files such as '1.1', '1.2', etc., which are *internal* versions. Each version of a file has a unique 'revision number'. It is also possible to end up with numbers containing more than one period, for example '1.3.2.2'.
- **Check out** Retrieve a working copy of a module from the repository that can be worked on or examined.
CVS command: `cvs checkout <module-name>`

- **Check in/commit** Store a modified working copy into the repository. CVS will ask for a log message.
CVS command: `cvs commit [file]`
- **Status** Check the status of your working copy, i.e. to see the revision number and to see whether it is up-to-date with the repository. Use `-v` to show tag information.
CVS command: `cvs status [-v] [file]`
- **Update** Update your working copy with changes from the repository. The update command can also be used to perform merges (see Merge).
CVS command: `cvs update [file]`
- **Tag** A tag is a symbolic name for a set of files with specific revisions. To create a tag based on the revisions in your working copy:
CVS command: `cvs tag <tag-name>`
For example, a set of files can be tagged as 'Release1'. This way, the particular revisions of the files of module 'OurModule' involved in 'Release1' can be retrieved with¹: `cvs checkout -r Release1 OurModule`
- **Branch** A branch allows to isolate changes on a separate line of development. Important: basing a branch on a certain working copy does not automatically switch that copy to be on the new branch, but the branch is created in the repository². To create a branch based on the revisions in your working copy:
CVS command: `cvs tag -b <branch-tag>`
- **Merge** Merge differences between revisions. For example, join a branch with the head of the main development line. Decide what you want to merge into what (e.g. merge a branch into the mainline or merge the mainline into a branch). To merge a branch into the head of the mainline, checkout the latest version of the mainline and perform an update which references the particular branch. After you solved any possible conflicts, perform a commit to create the merged revision, which becomes the new head of the mainline.
CVS command: `cvs update -j <branch-tag>`

2.2 Setting up and using a repository

You have been assigned to a team, and each team can use several computers. Teams have been numbered PW1..PW6 (Practical Work). This is important because each team creates exactly one repository in their team directory. Your team's repository directory is (at this moment it is empty):

```
/home/pracscm/PW<team-number>
```

In the Company case, your team will work on a small application, and different members will implement different functionality. We first explain in a few steps

¹When you check out a tagged revision, your working copy gets a 'sticky tag'. The sticky tag can be reset by using `cvs update -A`, which merges a tagged revision into the head of the main development line. See Chapter 4.9 of the CVS manual for more information on sticky tags

²You can use `cvs update -r <branch-tag>` to switch your working copy to the new branch. See Chapter 5.3 of the CVS manual for more details.

how your team can create and populate a repository. These actions are described in Section 2.2.1 and must be performed **once for each team**. Then, we explain in Section 2.2.2 how **each team member** can retrieve a personal working copy from the repository. Note that working copies must be retrieved into a member's home directory (~), so these should not be placed in the team directory.

2.2.1 Setting up a repository (done once per team)

Creating a repository One of the team members creates a repository for the team in the team's directory. To initiate a repository, run the following CVS command:

```
$ cvs -d /home/pracscm/PW<team-number>/cvsroot init
```

The above command creates the directory `/home/pracscm/PW<team-number>/cvsroot`, which is the root of the repository. A repository consists of administrative files, represented by the subdirectory `CVSROOT`, and user-defined modules.

Populating your repository We have already created a small sample application that should be put in the repository. The application resides in `/home/pracscm/exercise` and will be discussed in detail later. One of the team members can put these files in the repository as follows, using the `import` command (CVS will ask you to enter a log message):

```
$ cd /home/pracscm/exercise
$ cvs -d /home/pracscm/PW<team-number>/cvsroot import exercise vendor start
N exercise/.cvsignore
N exercise/Makefile
N exercise/hello.java
N exercise/runtests
```

```
No conflicts created by this import
```

This command creates the module 'exercise' in your repository and adds the files from the directory `/home/pracscm/exercise` to the module. The `vendor` and `start` arguments are tags that identify the initial revision. These initial tags can be ignored during the practice. To add a single file or directory to an existing module, use the `add` command followed by a `commit` command.

2.2.2 Using the repository (done by several members)

Telling CVS where the repository is Each member of the team that wants to access the repository (e.g. developer, tester) will have to tell CVS where the repository is located. There are two ways to do this: 1. specify the repository each time you run a CVS command using the `-d <directory>` option (this is shown above with the `init` and `import` command); 2. set the `$CVSROOT` environment variable to the location of your repository. We recommend that you set the environment variable (all subsequent commands assume that you have set the `$CVSROOT` variable):

```
$ export CVSROOT=/home/pracscm/PW<team-number>/cvsroot
```

Retrieving a working copy of the sample application Now each team member can retrieve a working copy of the module `exercise` from the team's repository into his/her home directory by issuing the `checkout` command:

```
$ cd ~
$ cvs checkout exercise
cvs checkout: Updating exercise
U exercise/.cvsignore
U exercise/Makefile
U exercise/hello.java
U exercise/runtests
```

This creates the directory `~/exercise` that contains a working copy of the module `exercise`. If you enter the directory, you see the files of the module and a subdirectory `'CVS'`. The `CVS` subdirectory contains the meta-data of the repository: `Root` contains the location of the repository; `Repository` contains the directory within the repository which the current directory corresponds to; `Entries` contains the names of the files and directories in the working copy.

To retrieve the status of the files in your working copy, use the `status` command (use `-v` to display tag information):

```
$ cd exercise
$ cvs status -v hello.java
=====
File: hello.java          Status: Up-to-date

Working revision:      1.1.1.1 Fri May 12 11:33:54 2006
Repository revision:  1.1.1.1 /home/pracscm/PW0/cvsroot/exercise/hello.java,v
Sticky Tag:           (none)
Sticky Date:          (none)
Sticky Options:       (none)

Existing Tags:
    start              (revision: 1.1.1.1)
    vendor             (branch: 1.1.1)
```

The output of the `status` command shows the status of the file, the revisions of the working copy and the repository, possible sticky tags, and the existing tags in the repository.

Building and testing the sample application Your working copy of the sample application consists of several files. We briefly explain the files here, in the next section we explain them in more detail:

- `hello.java`: the Java source file of the `hello` application. This application is the starting point for the Company case.
- `runtests`: a script for running tests of the `hello` application. Your team uses and modifies this script during the Company case to test new versions of your application.

- **Makefile**: the makefile for building and testing the application. It is not necessary to modify this file during the practice.
- **.cvsignore**: this file tells CVS what files to ignore when CVS commands are executed (e.g. generated files such as executables). It is not necessary to modify this file during the practice.

To build the sample application, type `make all` in the directory where it is located. To test the application, type `make check`.

```
$ make all
javac hello.java
$ make check
sh runtests
A test passed.
A test passed.
A test passed.
-----
ALL TESTS PASSED.
-----
```

2.3 The sample application `hello`

We explain the files of sample application that are now located in your team's repository. In Appendix B, you can find the contents of each file.

- `hello.java` is a Java program that prints a certain string depending on the supplied arguments. The program is the starting point for the Company case.
- `runtests` is a shell script for running small tests of the `hello` application. The script has a routine, `testCase`, which takes two arguments. The first argument is supplied to the `hello` application. The output of the application is then compared to the second argument. In case these two differ, the test has failed and a global variable `FAIL` is set to `true`. In the `runtests` script, the `testCase` routine is called with a number of test cases. After that, the `FAIL` variable is checked to see if one of the tests failed. Use and modify the script to test new versions of your application.
- **Makefile** is the makefile for building and testing the application. Each `make` rule consists of a target, dependents, and a body with commands. The basic operation is to update a target by ensuring that all of the files on which it depends exist and are up-to-date. For instance, the target `all` depends on `hello.class`, which depends on `hello.java`. This means that when you type `make all`, `all` checks if `hello.class` is up-to-date, which checks if it has a newer modification time than `hello.java`. If `hello.java` is newer, then it is (re-)compiled. The target `check` allows you to run the tests for the application using the `runtests` script. See the article in the SCM reader by Feldman for more information on `make`. It is not necessary to modify the **Makefile** file during the practice.

- `.cvsignore` tells CVS what files to ignore when CVS commands are executed. For instance, you may want to skip generated files such as executables when issuing a `tag` or `commit` command. In our case, `.class` files are skipped. It is not necessary to modify the `.cvsignore` file during the practice.

3 Case: Run your software company

You work in a company with several employees (your team members). Your company has a number of customers who are demanding on regular base new features and changes to your application. Different customers have different requirements and they may not be mixed up. After delivery of your software change requests may occur. Change Requests may be for all versions of your application, but also for a certain release. Develop a Software Configuration Management plan that covers these requirements. Describe policies how the developers in your company can work in parallel on the same or different releases. Describe in your plan the Development Streams of your application (in abstract terms) and define branching and tagging policies. The SCM plan will contain the planned Streams policy and the Actual Streams policy. They can be defined in a single diagram. The Streams diagram should include Branches, Release Tags, Feature descriptions and Merge activities. Follow the notation discussed during the lecture (see the slides from lesson 3).

As you run your business, new insights in the development process will cause evolution of your SCM plan. Store all intermediate versions of the SCM plan in your SCM system, CVS. Put the SCM plan versions in a separate CVS module (the plan should not be tagged with releases of the application). You can store an electronic version of the Streams diagram together with the SCM plan in the CVS module, but you are also allowed to make them on paper. Make sure that you keep the intermediate versions.

The practice includes a number of iterations. The iterations are not exactly defined in advance. You should manage your software product, using information as indicated above. During the practice, new events will pop-up (new feature, change request, problem report), which will be provided by the customers (the lecturers). Do not forget the regression test scripts, documentation like release notes and manual pages, but keep it simple.

Use the `hello` application that is located in your repository as a starting point for the development. You may want to tag the initial version as a baseline (e.g. Release 0).

Hints

- **Coordinate** Discuss how your team will work together, but you do not have to put this into the SCM plan. Each team can use several computers.
- **Keep it simple** The goal is to practice SCM concepts like branching strategies, so you do not need to put a project plan in the SCM plan. Decide yourself what releases, upgrade, updates and patches are.
- **Branching and merging** Determine branch and merge strategies and policies for you team, and how to apply them.
- **Testing** Before you commit a change, make sure that your application works and that it works with the current state of the repository. You should use and modify the provided test script to automatically test your application after you made changes and before you release an application.
- **Tagging** Choose a tag convention. Tag often. Tag before and after you merge. Tag when a release is finished.

4 CVS and Eclipse

After you have finished the Company case from Section 3, we want you to continue here to see how you can use CVS with the Eclipse IDE. The idea is to look at the versions in your repository that you have created during the Company case and review what you have done.

CVS is available as a plugin to the Eclipse platform. You can find Eclipse in the Windows startmenu, or start it from the Unix commandline. To start Eclipse, enter the following commands:

```
$ cd /home/pracscm/eclipse
$ ./eclipse
```

4.1 Accessing the CVS repository

Eclipse needs to know where your CVS repository is. Follow the menu sequence below to create a repository location.

```
File -> New -> Project... -> CVS -> Checkout projects from CVS ->
Create a new repository location
```

Now you should see a screen similar to the screen shown below:

The screenshot shows the 'Checkout from CVS' dialog box. It is titled 'Checkout from CVS' and has a close button in the top right corner. The main title is 'Enter Repository Location Information'. Below this, it says 'Define the location and protocol required to connect with an existing CVS repository.' There is a CVS logo in the top right corner of the dialog. The dialog is divided into several sections: 'Location' with 'Host' (galjas.cs.vu.nl) and 'Repository path' (/home/pracscm/PW0/cvsroot); 'Authentication' with 'User' (nveerman) and 'Password' (masked with asterisks); 'Connection' with 'Connection type' (extssh) and radio buttons for 'Use default port' (selected) and 'Use port:'. At the bottom, there is a checkbox for 'Save password' which is unchecked, and a warning icon with the text 'Saved passwords are stored on your computer in a file that is difficult, but not impossible, for an intruder to read.' Below the dialog are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

When you have filled in the proper values, click 'Next' to continue. Follow the next steps to checkout a module and open it with the 'New Project Wizard':

Use an existing module -> <select module exercise> ->
Check out as project configured using the New Project Wizard ->
Refresh Tags -> <select version or branch> -> Finish

When you are in the 'New Project Wizard', follow the next steps to set up the project:

Java Project -> <enter project name exercise> -> Finish

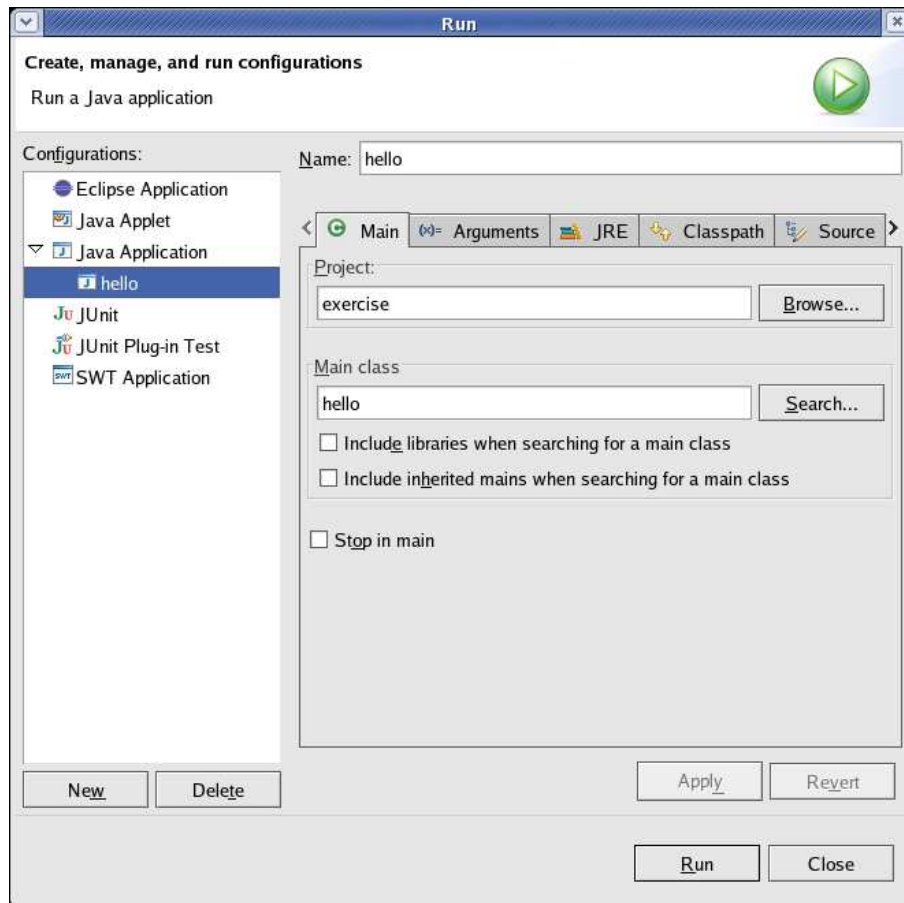
Now you can start working with the project.

4.2 Configure and run an application

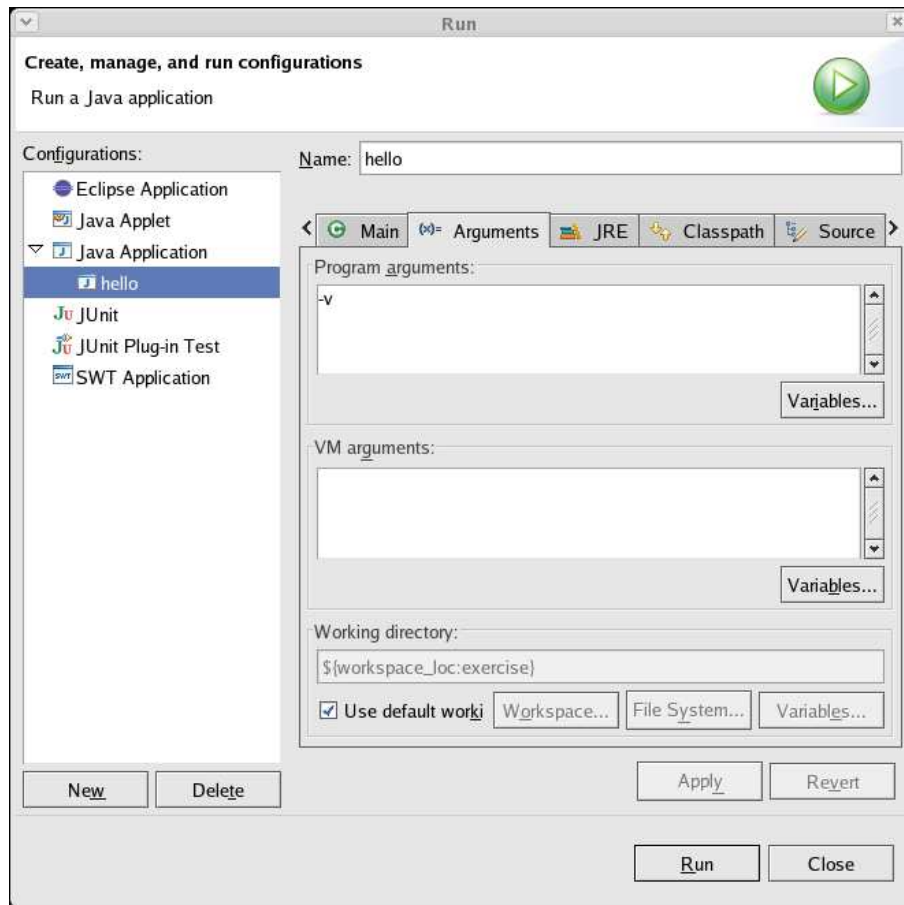
If you want to, you can run your application from the IDE. To configure and run an application, follow the next steps:

Right-click hello.java -> Run as -> Run...

Now you should see a screen like the one shown below:



Here, you can set up the configuration of your application. For instance, you can enter a `-v` program argument in the tab 'Arguments':



When you are done, click 'Run' to run the application using the current configuration. If you want to run the application again without entering the configuration menu, perform the steps below:

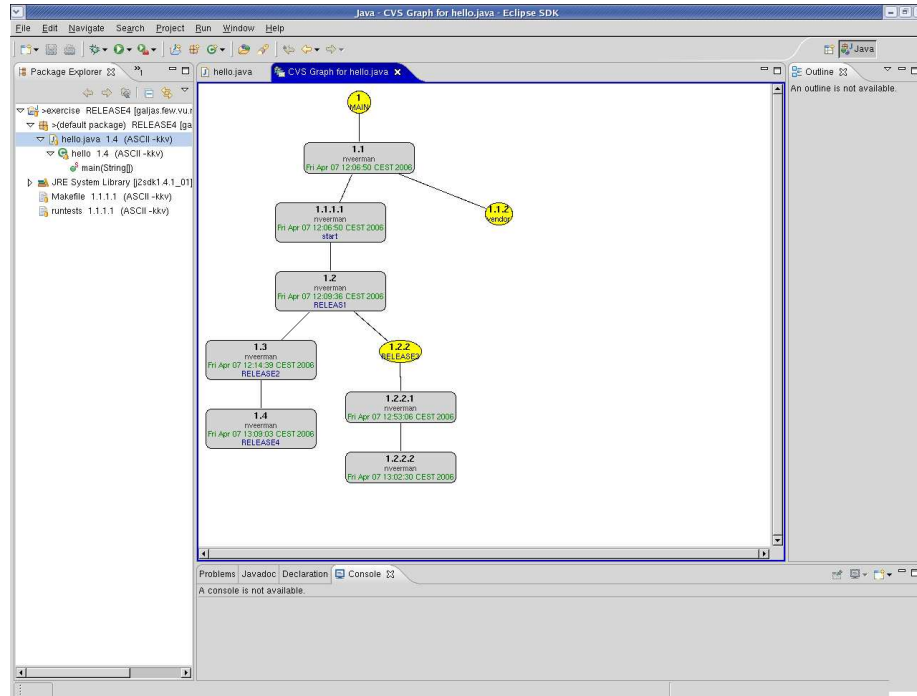
Right-click `hello.java` -> Run as -> Java Application

4.3 Accessing and comparing revisions

We have installed a plugin that can be used to visualise revisions. To have a graphical overview of the revisions in the repository, perform the following steps:

Right-click `hello.java` -> Team -> Show in Resource Graph

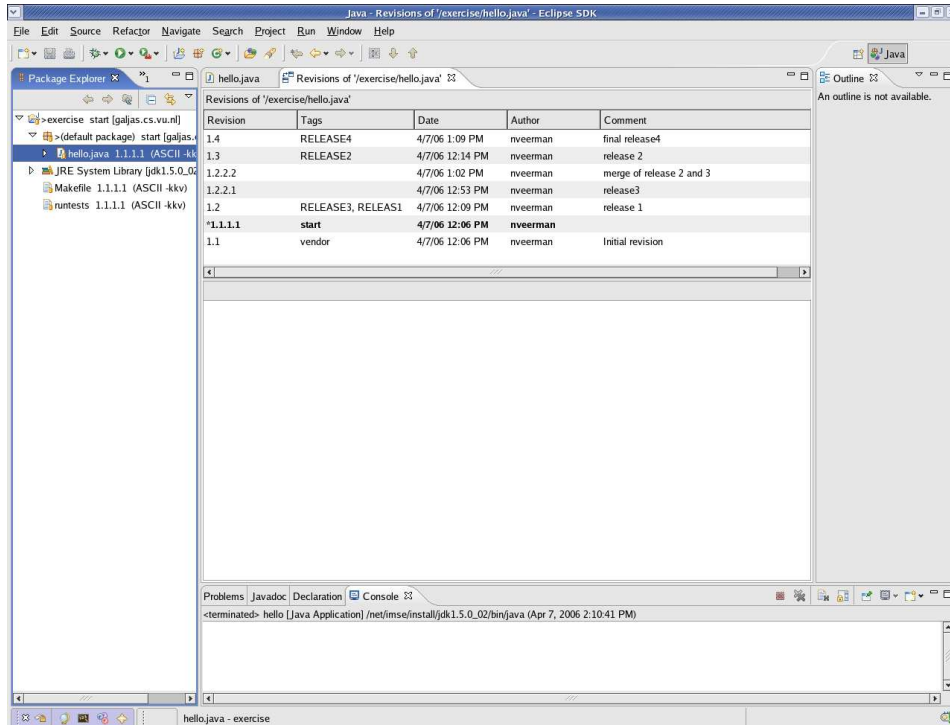
Unfortunately, the graph does not show merge actions, unless the source of a merge was labelled “Source_<label>” and the merged result “Merge_<label>”. By moving the mouse over the nodes in the graph, you can get more information about the revision and branches.



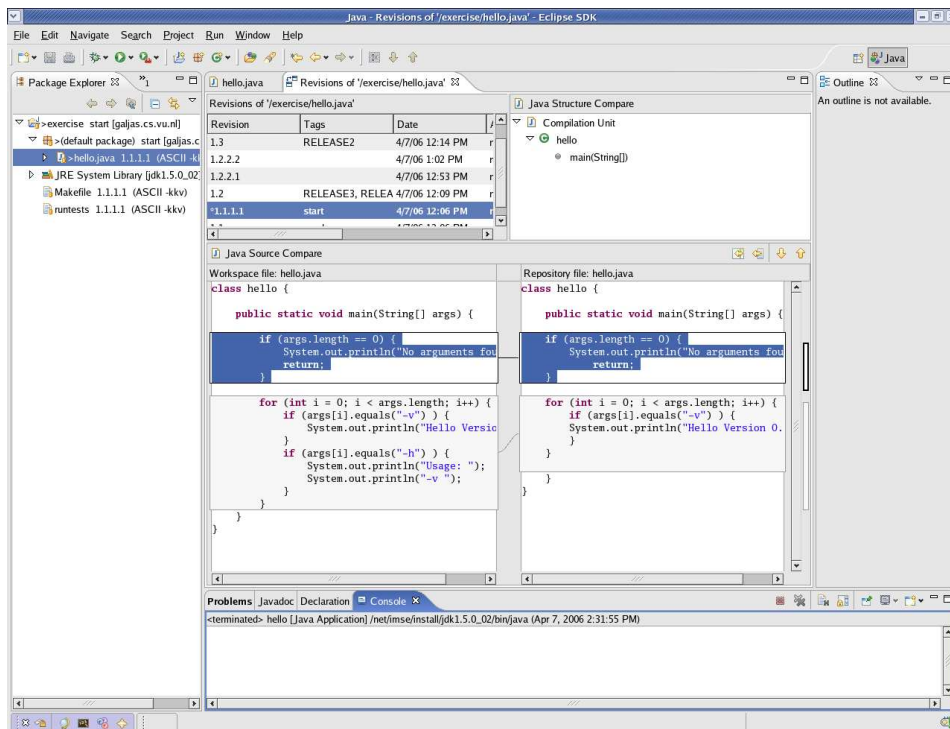
You can also compare the revision you have retrieved from the repository with another one. To do this, perform the next steps:

Right-click `hello.java` -> Compare With -> Revision...

Now you can select a revision from the table, which contains for each revision a revision number, tags, a date, the author and the log message. The table will look something like this:



After you select a revision, both revisions will appear in the project space. Similarities and differences are indicated by (shaded) boxes.



You can now look around and see what SCM policies have been applied during your team's software development!

That's all folks!

A Resources

A.1 CVS quick command reference

This list summarizes most of the available CVS commands. For a detailed description and possible options we refer to the online resources (see below).

add add new files or directories
admin control/administrate the repository
checkout get a new working copy from the repository
commit apply changes, additions, deletions
diff show differences between working copy & repository
export prepare copies for off-site shipment
history show report on commands executed
import incorporate a set of off-site updates
init initialize a new repository
log show stored log information
rdiff prepare a collection of diffs as patch file
release cancel a **checkout**, abandoning changes
remove remove files from the source tree
rtag specify a symbolic tag for a revision
status show current status of files
tag specify a symbolic tag from a working copy
update get the changes from the repository

A.2 Online resources

- “Version Management with CVS” by Per Cederqvist et al:

<http://ftp.gnu.org/non-gnu/cvs/source/stable/1.11.21/cederqvist-1.11.21.pdf>
<http://www.cs.vu.nl/~cats/scm-course/auxmaterial/cederqvist-1.11.21.pdf>

- The CVS website of Dick Grune, the original author of CVS:

<http://www.cs.vu.nl/~dick/ CVS.html>

- CVS Quick Reference Card

<http://tnerual.eriogerg.free.fr/cvsqrc.pdf>

- “Make: a program for maintaining computer programs” by S.I. Feldman

See the SCM reader

- Eclipse SDK Help

<http://help.eclipse.org/help31/index.jsp>

B Source code of the sample application

B.1 hello.java

```
class hello {  
  
    public static void main(String[] args) {  
  
        if (args.length == 0) {  
            System.out.println("No arguments found.");  
            return;  
        }  
  
        for (int i = 0; i < args.length; i++) {  
            if (args[i].equals("-v") ) {  
                System.out.println("Hello Version 0.1");  
            }  
        }  
    }  
}
```

B.2 runtests

```
#!/bin/sh

# Simple shell script to perform tests for the hello application.
#
# Each test case creates a file with the correct output and
# compares it with the actual output of the hello application.

# initialisation
JAVA=java
FAIL=false

# Routine for a test case.
# the first argument $1 is supplied to the hello application
# the second argument $2 is compared to the actual output
testCase() {
    TEST="$JAVA hello $1"
    ACTUAL_OUT='eval $TEST'
    EXPECTED_OUT="$2"

    if [ "$ACTUAL_OUT" != "$EXPECTED_OUT" ]
    then
        echo "A test failed!"
        echo "  Test:           $TEST"
        echo "  Expected output: $EXPECTED_OUT"
        echo "  Actual output:   $ACTUAL_OUT"
        FAIL=true
    else
        echo "A test passed."
    fi
    return 0
}

##### test cases #####

# a test case for no arguments
testCase "" "No arguments found."

# a test case for -v
testCase "-v" "Hello Version 0.1"

# another test case for -v
testCase "-X 1 -v -x" "Hello Version 0.1"

# add more test cases here
```

```
#####
```

```
# see if one of the tests failed
echo "-----"
if [ $FAIL = "false" ]
then
    echo "ALL TESTS PASSED."
else
    echo "TESTS FAILED."
fi
echo "-----"
```

```
exit 0
```

B.3 Makefile

```
JAVAC = javac

all: hello.class

hello.class: hello.java
    $(JAVAC) hello.java

check: hello.class runtests
    sh runtests

clean:
    rm -f hello.class
```

B.4 .cvsignore

```
*.class
```