Version Control Systems

ESA 2016/2017

Adam Belloum

a.s.z.belloum@uva.nl

Material Prepared by Eelco Schatborn

- Today
  - Introduction to Version Control Systems
  - **Centralized** Version Control Systems
    - **RCS**
    - CVS
    - SVN
    - …
  - **Decentralized** Version Control Systems
    - Git, (written in C) used by the Linux Kernel and Ruby on Rails.
    - Mercurial, (written in Python) used by Mozilla and OpenJDK
    - Bazaar (written in Python) used by Ubuntu developer
    - Darcs, (written in Haskell).

Version Control systems

A version control system provides **support** for the **development** of **software** in particular.

- documenting an annotated history of a project is a **complex matter** (changes, bugs)
- multiple editors in use, keep developers from overwriting each others work
- **backtracking** to a previous version
- **branching**

Version Control Systems

Most important Unix Version Control programs

- **SCCS** (Source Code Control System) Bell Labs, **1980**. Proprietary.

  Introduction of most important concepts. Not used much anymore.

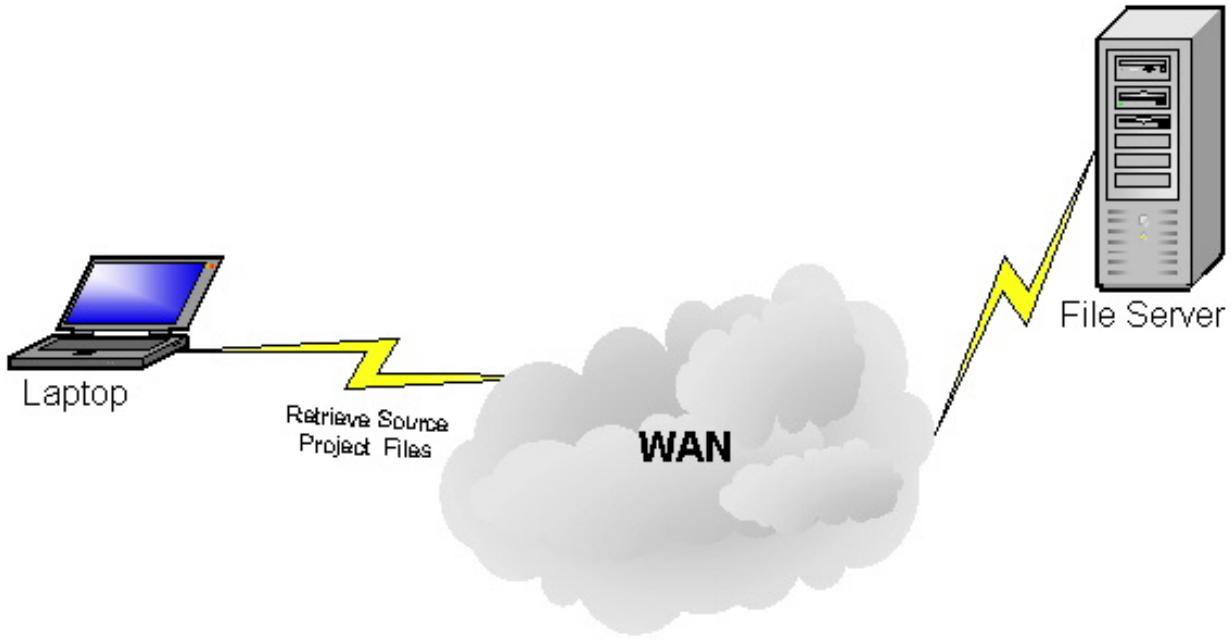
- **RCS** (Revision Control System) Purdue University, **early 80's**. Open source. Basis of CVS.

- **CVS** (Concurrent Version System) **Early 90's**. Maintained by FSF (Free Software Foundation)

- **SVN** (Subversion) **Early 2000**. CollabNet

- Today
  - Introduction to Revision Version Control Systems
  - Centralized Version Control Systems
    - RCS
    - CVS
    - SVN
  - Decentralized Version Control Systems
    - GIT

Centralized Version Control Systems



It's like having your source files on a File server and coordinate with your colleagues on the version number you're working on.

By Durai Raj "CVS" Concurrent Versioning System

How Does centralized VCS Work?

- centralized VCS uses a client / server architecture
- A sys admin would normally install the server
- The Server Maintains THE **repository**
- Developers use the **client**
- The client allows for **checking in/out**, and Updating, etc..

Versioning Systems

Versioning Systems allow:

- **multiple users** to modify the same code
- To store **ONE master copy** of the source code
- To **automate** the **update** between versions
- **Access** to **any previous** state of the source code

VCS do's and don'ts ……

- Facilitates **bug detection** when software is modified
- **Economy** in disk space while saving versions
- **Prevents code over-writing** in a team project

- **Not a build** system
- **Not a substitute** for communication between developers or for management
- It will **not create any magic** for you.

Check-In, and Check-Out

- The **version file** for a document contains extra **administrative information**
- before you can go and look at or change a file, a **working version** has to be created; this is called **`check-out`**.
- If you want to make changes, **PUT a lock** on the file at check-out (Very OLD WAY, RCS)
- submitting your changes to a file to the system is called a **`check-in`**

Check-In, Check-Out: commands

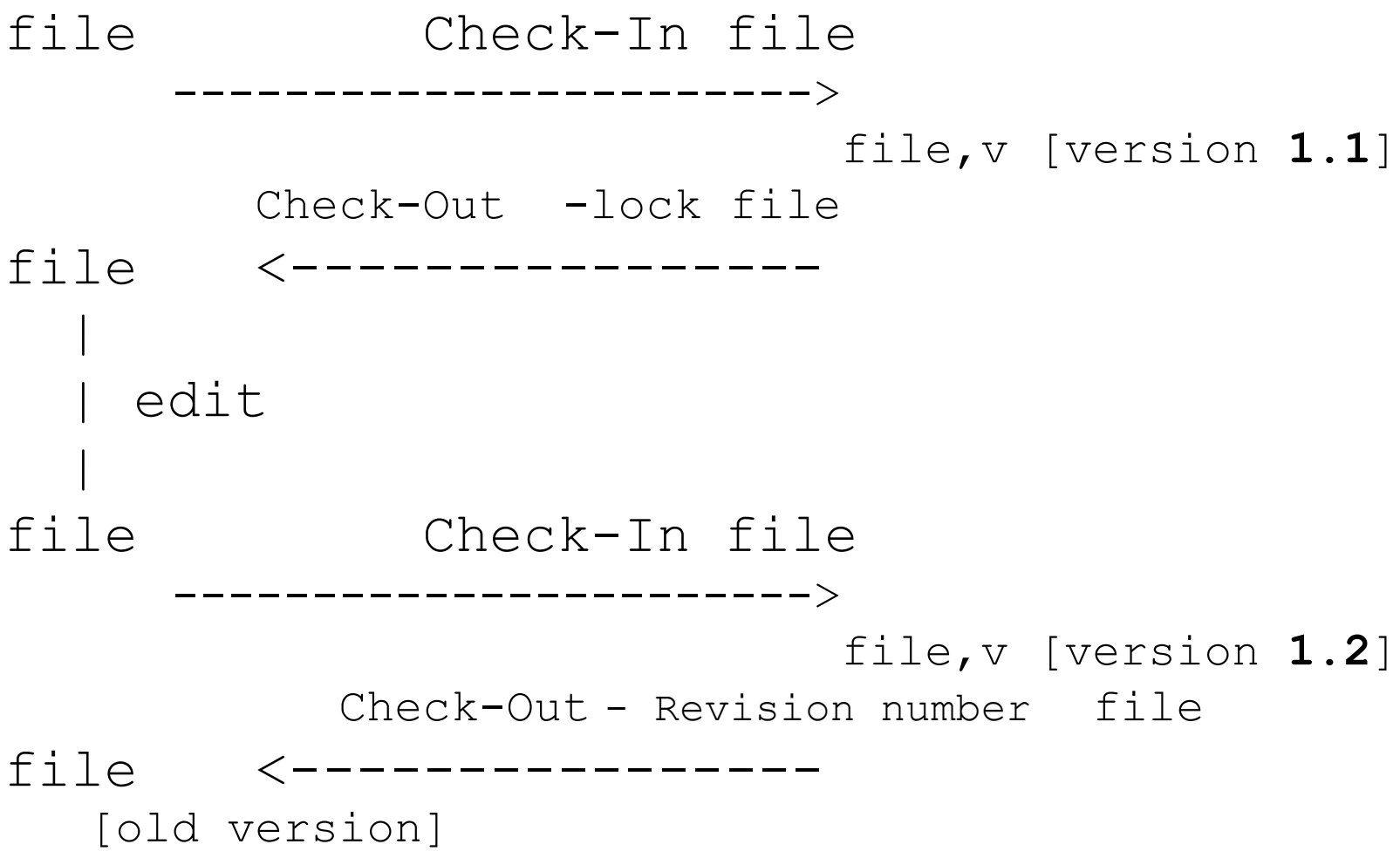
`ci` Check-in: put the **work file** into the **version file**;
→ create the version file if it does not exist. The system requests a description of the changes

`co` Check-out: create a **new work file** from the version file (read-only)

`co -l` Check-out with lock; write permissions

`co -rN` Check-out file with revision number N

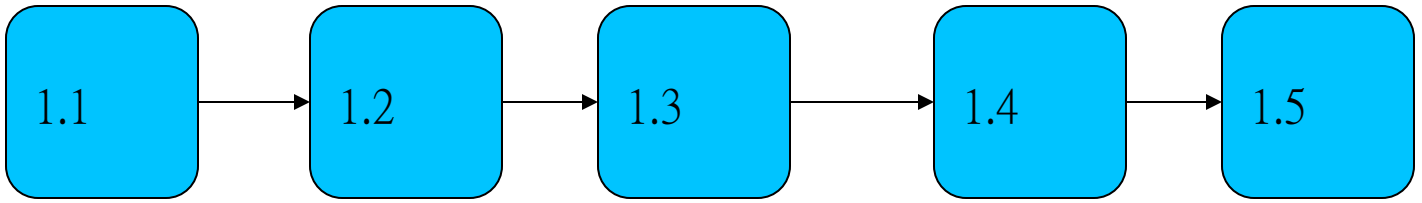
Check-In, Check-Out: Example



Revision numbering

- Numbering starts with **1.1**
- With every check-in the **second number** is increased by **1**
- The **first number** can be increased by using the -r option;

  for instance, using **`check-in revision 2`** will set the revision number to **2.1**



When to commit

- Commit to **mark** a **working state** that you might want to **return to later**.
- Commit **related files** in a single operation. Use a common log message for all the files.

- (useful but not very wise if the code is not fully tested !!!)
  - Commit to **backup** your sources
  - Commit from an office desktop to be able to access the files from home much faster than through filesystem sharing.

Collaboration (old time)

- If a user A locked a file using **`check-out -lock,`** he or she gets write permissions on the file
- If user B the tries to lock the file in the same way, a warning is given that the file is locked
- B can decide to go forward anyway, in which case A will get a warning (by email)

RCS Information

- Information in a revision can be added to **the working file**
- The information itself is part of the version file
- To copy that information to the working file during check-out one uses **keywords** for example use:
  - **Id** to insert information on the author, date, version
  - envelop the keyword with **$** . . . **$**
- Other keywords are **Author**, **Log**, **Date**, **Locker**, . . .

Branching

- say you are working with version 2.5, but want to start an **alternative** development from version 1.8
- using **`checkout -l -r1.8`** a working file can be created from version 1.8

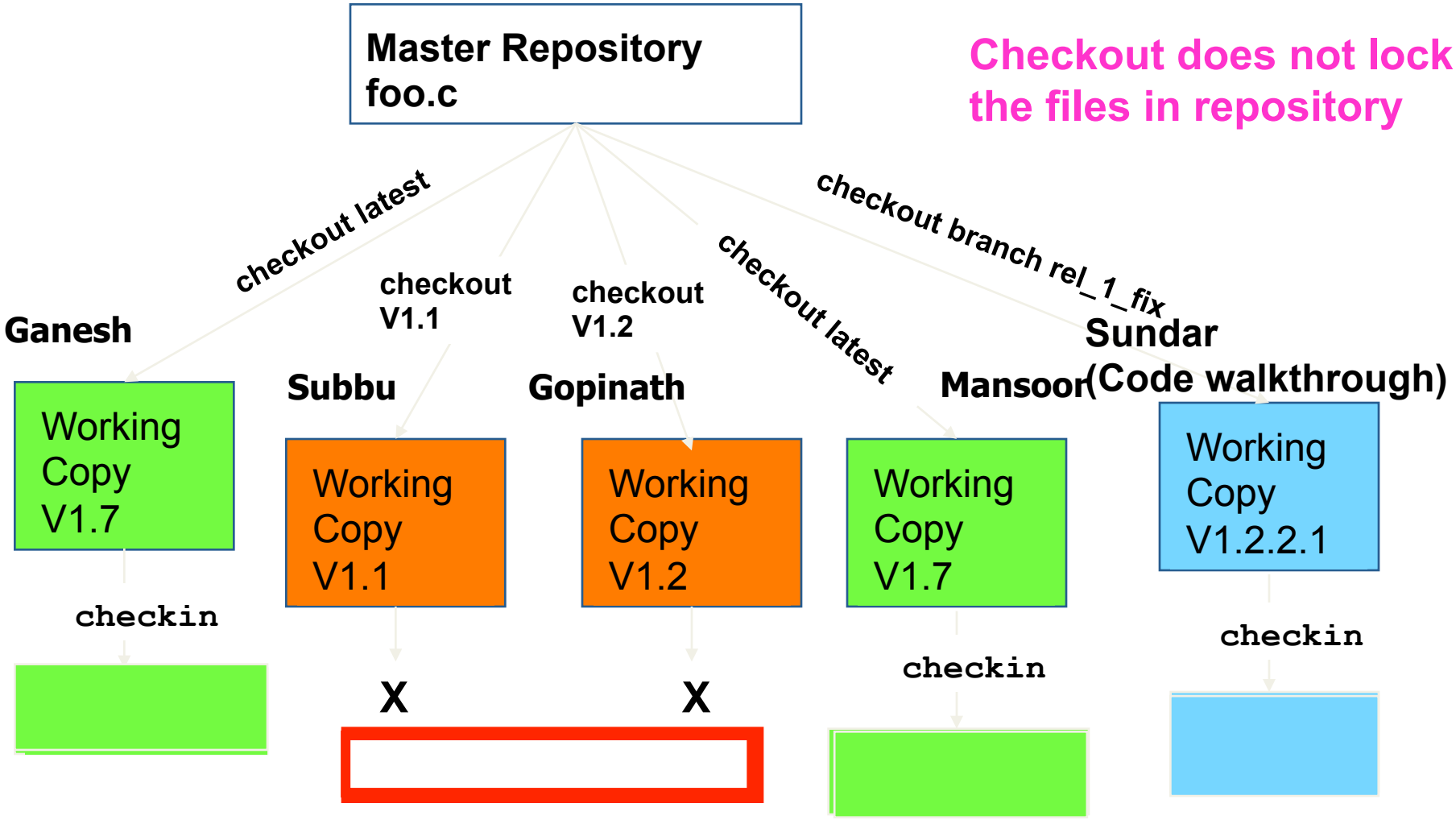
Note: should you check-in that file you would get version 2.6!

  - use **`ci -r1.8.1`** to start a new branch from version 1.8 (first new version is 1.8.1.1)

Branch if you need ...

- to **create** sustaining (patch) releases
- to have multiple development lines from a single repository
- to do **experimental** development to **merge** later or forget about it
- to keep temporary state of development without affecting builds

Concurrent checkout



Merge

- Branches can be merged. Say a file is a work file for revision 1.8.1.3, then

```
rcsmerge -r1.8 -r2.5 file
```

- will give a merge of the file with version 2.5. A check-in will now create version 2.6.

- A merge operation looks at the **differences** between **two versions** an places as much as possible into the new file. If there is a **conflict** it will be shown between <<<<<< and >>>>>>

- Today
  - Introduction to Revision Version Control Systems
  - Centralized Version Control Systems
    - RCS
    - CVS
    - SVN
  - Decentralized Version Control Systems
    - GIT

CVS: Concurrent Version System

- **initially** a front-end to **RCS**
- **newest** implementations are no longer dependent on RCS
- they do use the **RCS format** for the version file
- **more functionality** than RCS, **more complex** interface,
  - RCS can be used for small projects with few Editors
  - CVS can handle large projects with many designers in different places (worldwide) **no exclusive locks**

THE Repository

- CVS keeps the **version files** for a project in a **CVS repository**
  - a directory on a machine reachable by all participants (CVS server)
- people can work locally or remotely through ssh. In both cases the environment variable **CVSROOT** must point to the repository
- The repository can be split up into **modules** -- a group of files and/or directories
- to work on module foo a working directory has to be created Using

  cvs checkout foo
- CVS then creates a subdirectory foo containing the work files of the foo module

Example checkout

```
$ cvs checkout foo
cvs checkout: Updating foo
U foo/.cvsignore
U foo/file1
U foo/file2
U foo/file3
$ ls foo
CVS file1 file2 file3
```

Commit and Update

- After making changes a check-in can be done file using `cvs commit`
- if more than one person is working on the module it is standard practice to do an **update** first using

  `cvs update`

  – an update will check if others have committed anything since your last check-out
  – if so, your last work file will be merged with the latest version

Example update

```
$ cvs update
cvs update: Updating .
U file1
U file2
RCS file: <CVSROOT-path>/foo/file3,v
retrieving revision 1.4
retrieving revision 1.5
Merging differences between 1.4 and 1.5
  into file3
M file3
$
```

Commit

- if your files have been merged after an update you should check if everything **STILL** works

- Then do another update

```
$ cvs update
cvs update: Updating .
M file3
```

  - CVS tells us file3 is the only file you have adapted, no others have committed the file since your last update

- using cvs commit file3 you can check-in the file

Log message

- After the cvs commit command CVS will open an editor (**`$CVSEDITOR`**) so that a description of the changes can be given

- Very important advise: DONOT PUT DUMMY TEXT, explain exactly what you did

The CVSROOT module

- To initialize the repository run:

  `cvs init` in the root directory `CVSROOT`

- After the command the CVSROOT directory contains the special module CVSROOT

- That module contains configuration information on the repository

Starting a new module

- When a **new project** is started a new module must be created in the repository using cvs import

1. go to the directory you wish to import
2. use:

```
cvs import <module name> <vendor tag> <release tag>
```

3. CVS will then ask you to type in information on the project in an editor

Import Example

```
$ mkdir example
$ mv file* example/
$ cd example
$ cvs import example example_project ver_0_0
```

A module example is now made; the example directory
in the repository contains the version files file1,v and file2,v

Adding and removing files

- **adding** a file to a module
  1. put the file in the work directory
  2. `cvs add <file>`

- **removing** a file from a module
  1. first remove the file from your working directory
  2. `cvs remove <file>`

- Note Directories cannot be removed

  All changes are made at the next commit command

Viewing changes

- `cvs log <file>`

  command gives an overview of the log data of a file. Who made what changes and when.

- `cvs diff -c -r 1.6 -r 1.7 <file>`

  will give a list of differences between version 1.6 and 1.7

Tagging

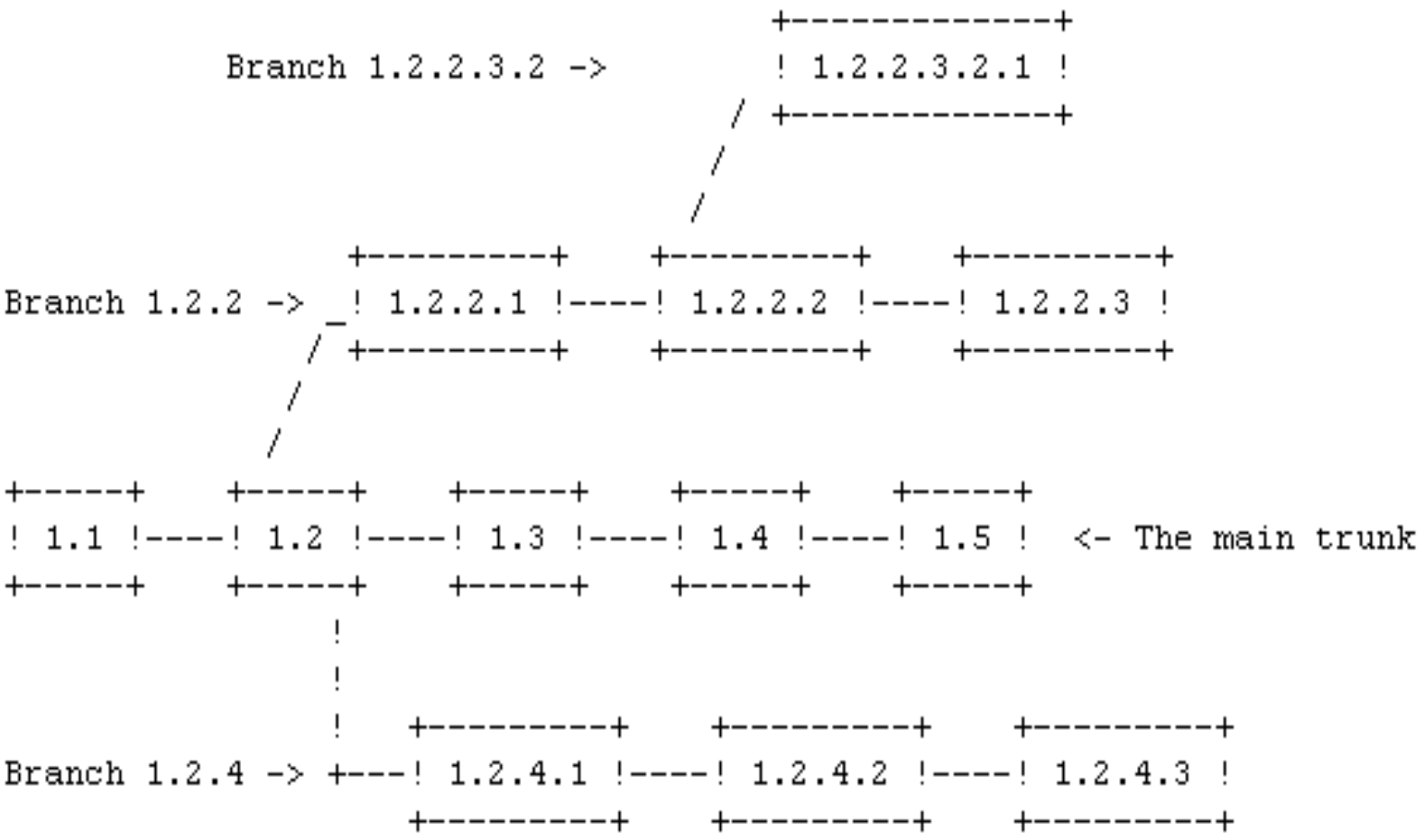
- a project MyProject can be given a tag **MyTag** to which you can refer later on
- **revision number** can be referred to, but each file can have a different version number
- use the command cvs tag **MyTag** in the working directory of the module
- a tagged version can be retrieved using

```
cvs checkout -rMyTag MyProject
```

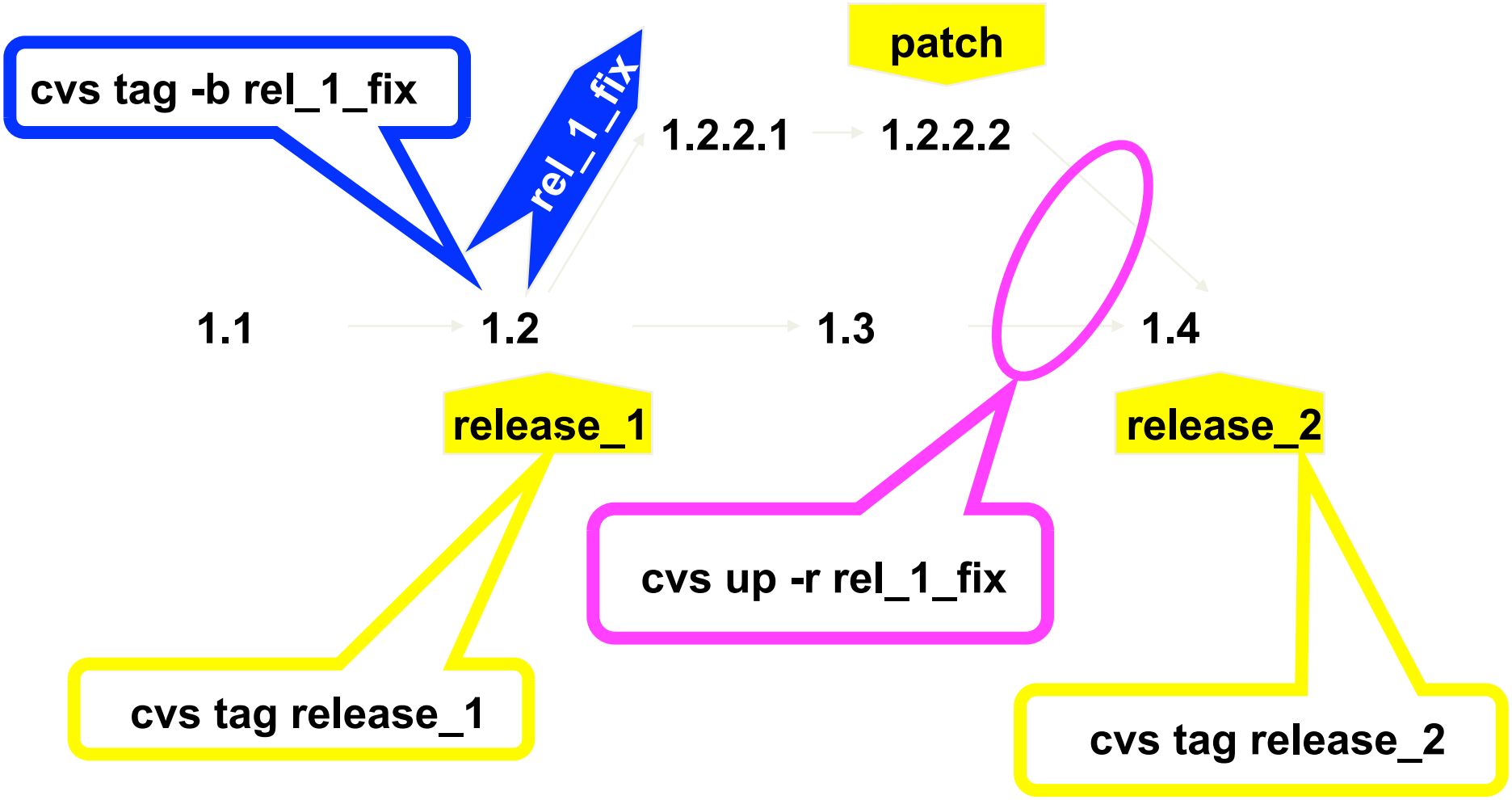
- there is always one central branch, the `**trunk**'
- the tag name for the latest revision of the trunk is always **HEAD**

Branching

- CVS work with branches in terms of tags
- a branch tag refers to the branch
- use the command `cvs tag -b <branchName>` in the work directory of the module to be branched
- use the command `cvs checkout -r <branchName>` to get a working copy of a branch, if you commit now the new revision will be added to the branch


Branch 1.2.2.3.2 ->
! 1.2.2.3.2.1 !
Branch 1.2.2 ->
! 1.2.2.1 !
! 1.2.2.2 !
! 1.2.2.3 !
! 1.1 !
! 1.2 !
! 1.3 !
! 1.4 !
! 1.5 !
<- The main trunk
Branch 1.2.4 ->
! 1.2.4.1 !
! 1.2.4.2 !
! 1.2.4.3 !

Working on branches



Merging

- to merge a **branch** with **HEAD** use
  `cvs update -j branchName`
- this gives a new version of the **trunk**
- a warning
  `rcsmerge: warning: conflicts during merge`

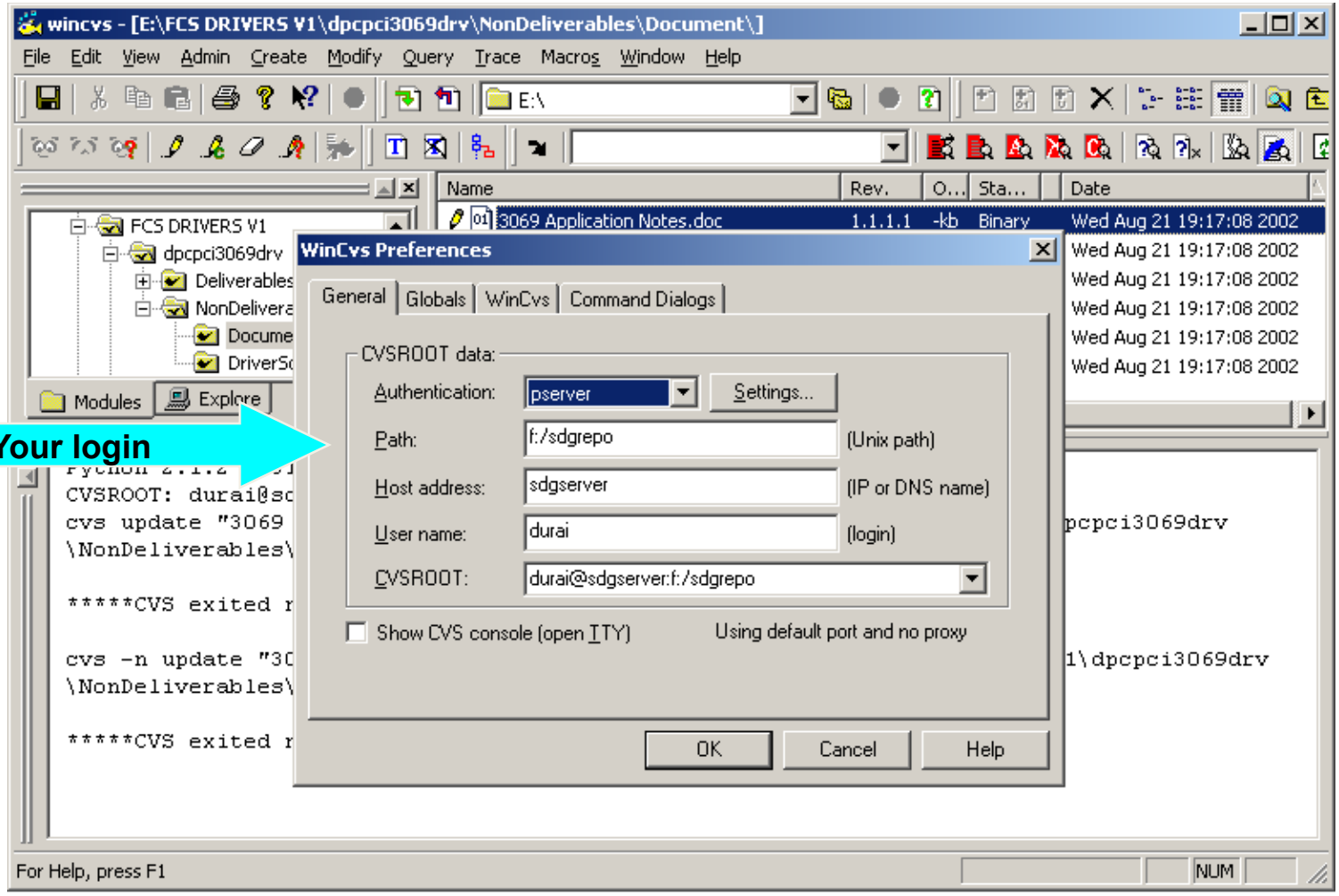
  if there are **conflicts** with a certain file

  if that happens a copy of the HEAD version is kept as .file1.5 (file is the name and 1.5 the revision number)

  in the le itself CVS will mark the lines where the conflicts are (between <<<<<< and >>>>>>)

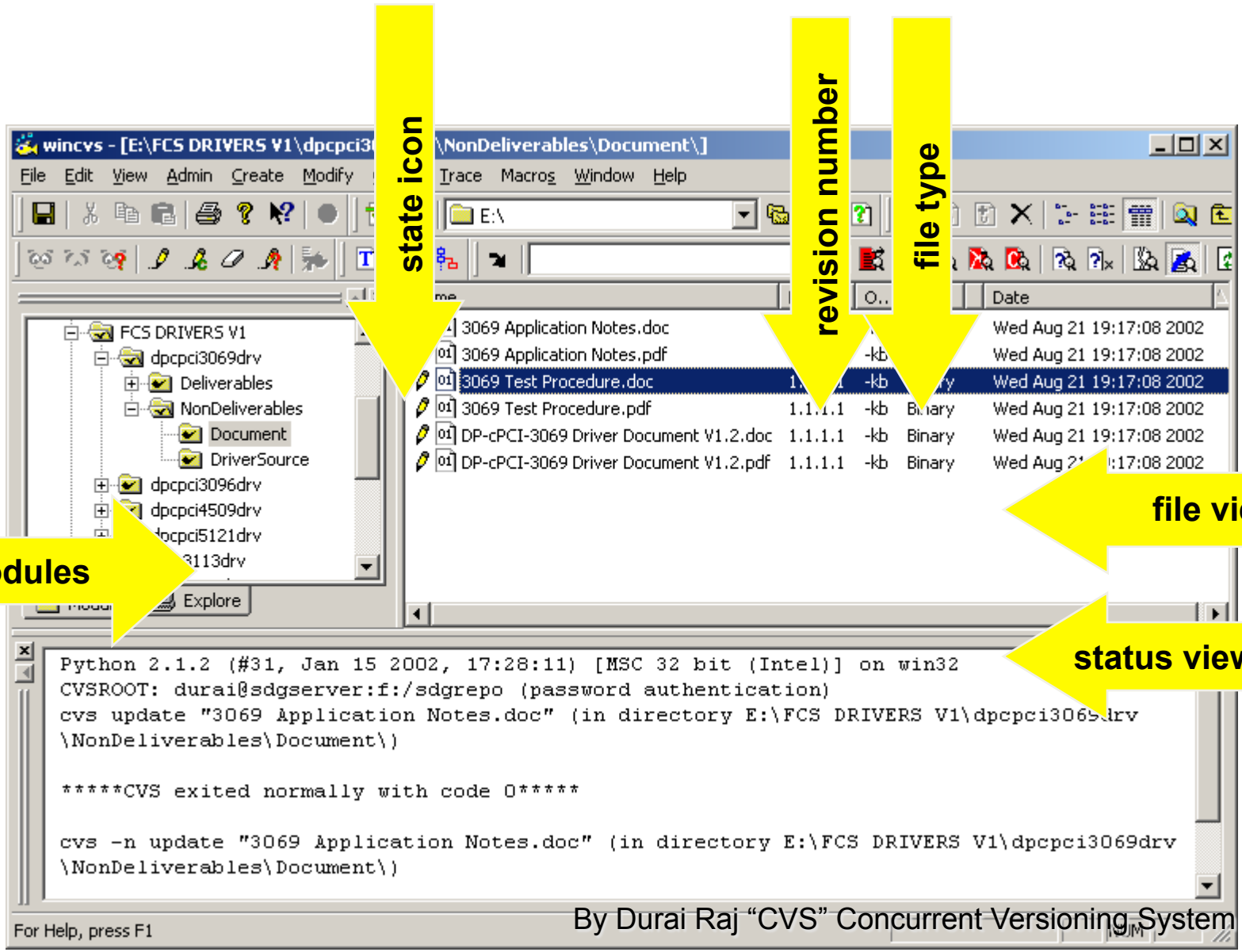
WinCVS: Configuration

http://cvsgui.sourceforge.net/index.html

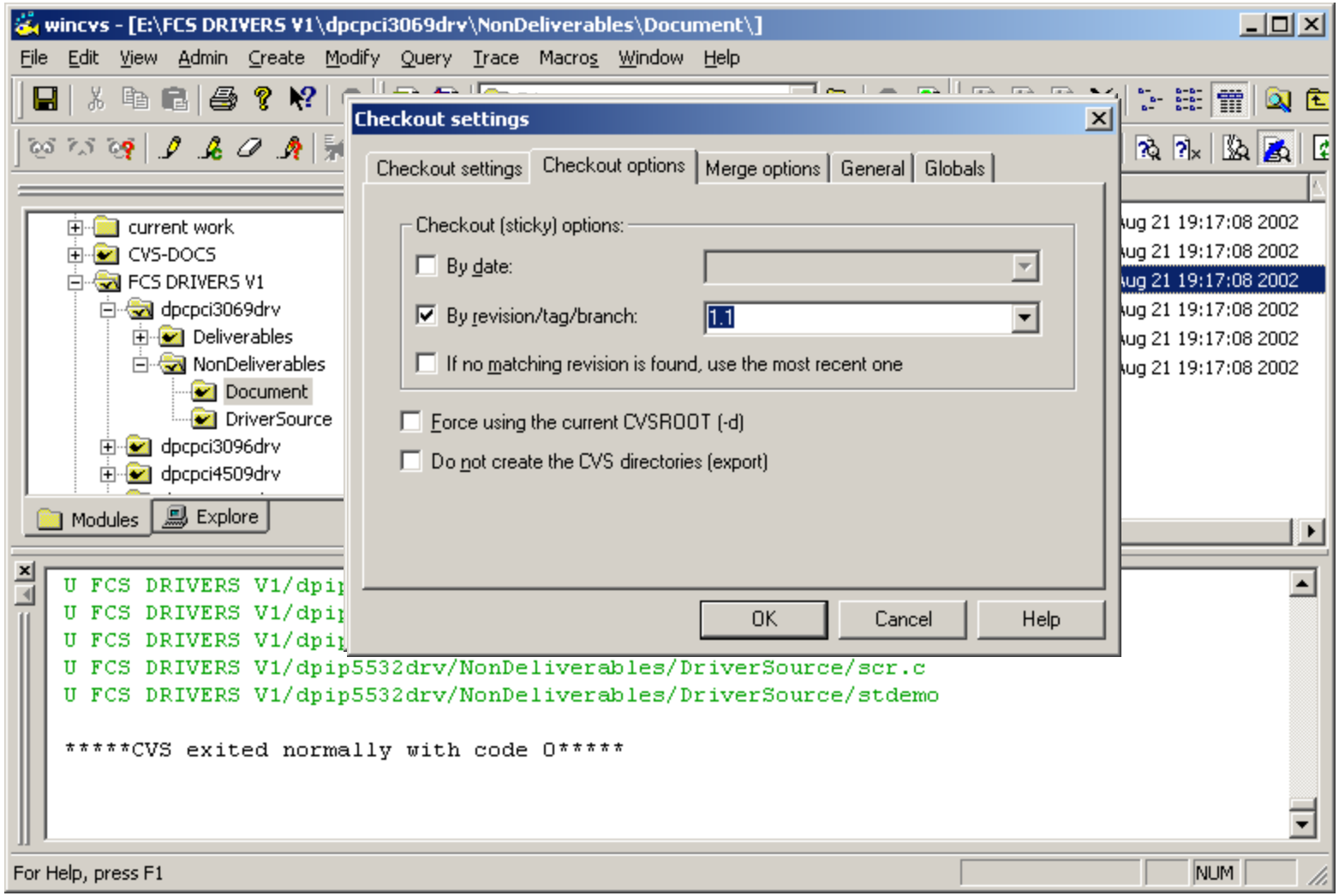


By Durai Raj “CVS” Concurrent Versioning System

WinCvs: Main screen

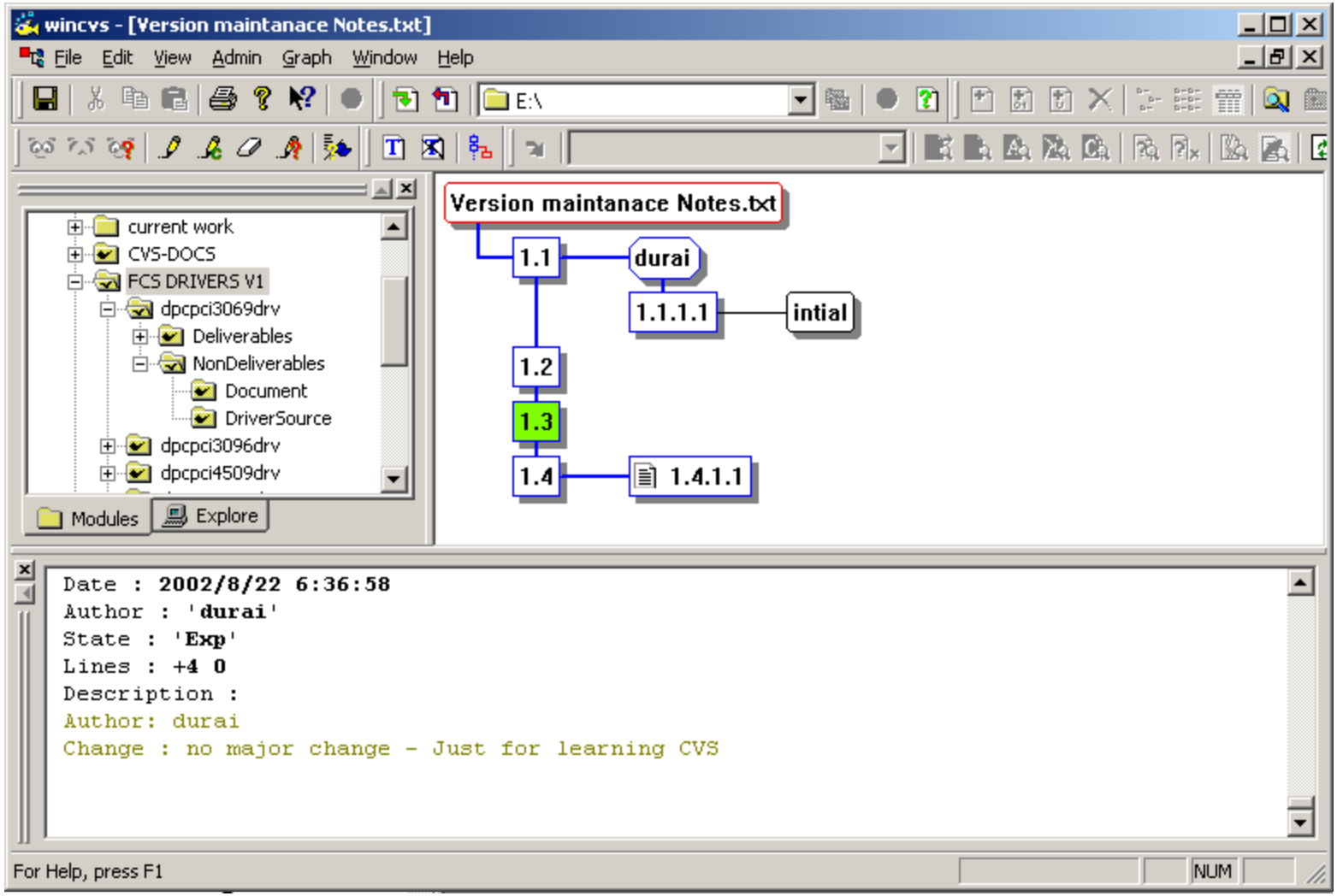


Checking out the sources



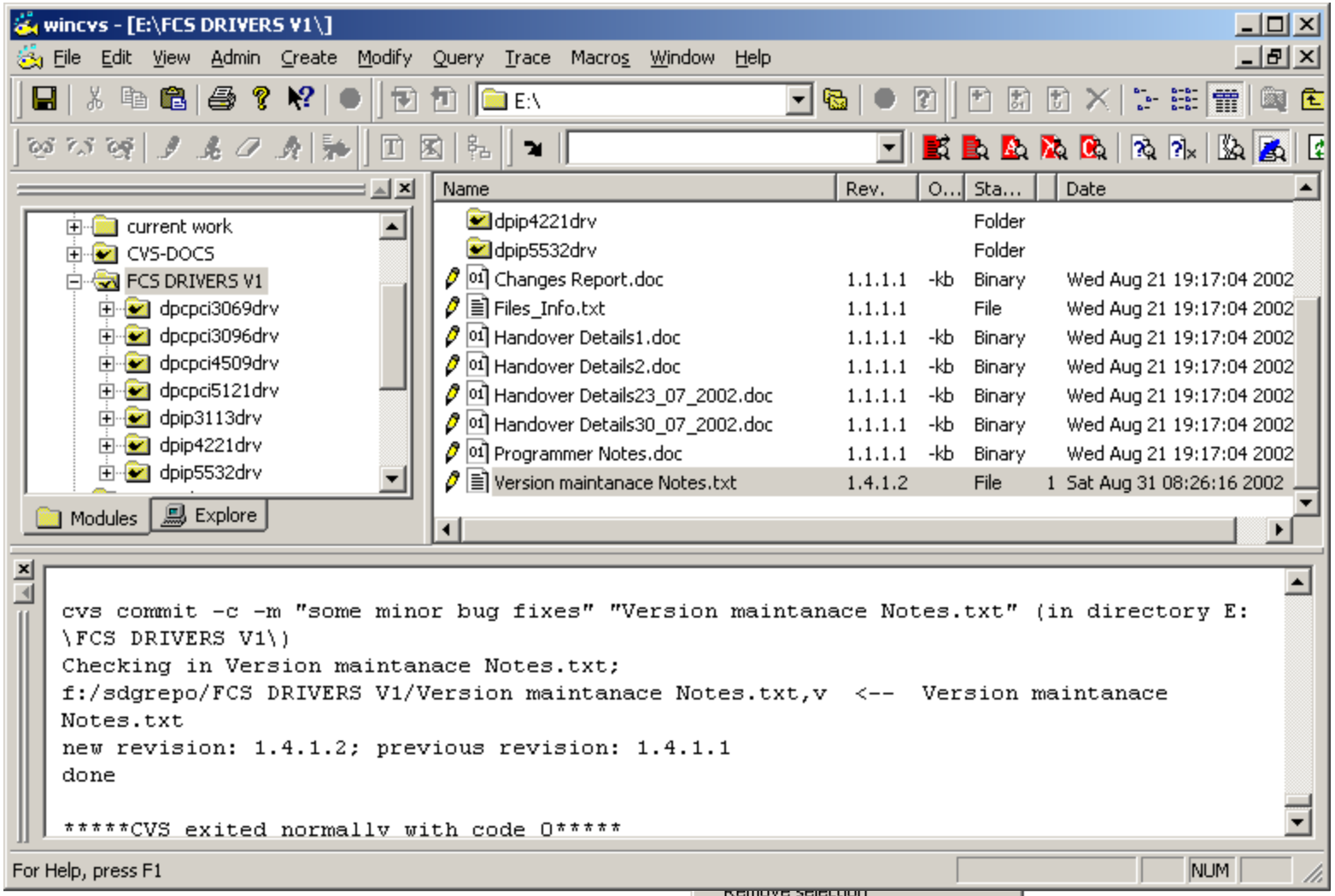
By Durai Raj “CVS” Concurrent Versioning System

Source history & diff



By Durai Raj “CVS” Concurrent Versioning System

Commit, Update & Tag



- Today
  - Introduction to Version Control Systems
  - Centralized Version Control Systems
    - RCS
    - CVS
    - SVN
  - Decentralized Version Control Systems
    - GIT

Subversion (SVN)

- Project summary A compelling replacement for CVS
- Offers the functionality of CVS using more or less the same interface
- Better implementation

Differences with CVS

- **Atomic commit** When you commit a group of files, they should either be successfully committed, or not at all. CVS allows for partial commits
- **Renaming** in CVS renaming can only be done by creating a new file, adding it to the repository and removing the old one. That way all version information on a file is lost. SVN provides move.
- **Versioning directories** directories can also be moved and renamed in SVN
- **Binary diffs** The diff mechanism of CVS is geared towards text files, which makes it very **inefficient** when working with **binary files**. SVN improves on that

Differences with CVS

- Revision numbering in CVS files are numbered individually.
- In SVN the whole module is numbered
  - **CVS :** revision 2.6 of <file> is always different from revision 2.7
  - **SVN:** it does not have to be a difference between revision 30 and revision 31 for a given file

Subversion Commands

co/checkout

ci/commit

up/update

add, remove/rm

cp/copy

mv/move

Log

diff

- Today
  - Introduction to Version Control Systems
  - Centralized Version Control Systems
    - RCS
    - CVS
    - SVN
  - Decentralized Version Control Systems
    - GIT

GIT

- a distributed revision control system.
- most features from Subversion
  - identical interface
  - file handling (binary, etc)
- more branching oriented
- everyone **keeps Its own root**, you **can** merge with other People
- manual access control (**YOU decide** who to merge with)
- see http://git-scm.com/

Snapshots, Not Differences

- most other systems store information as a list of **file-based changes**

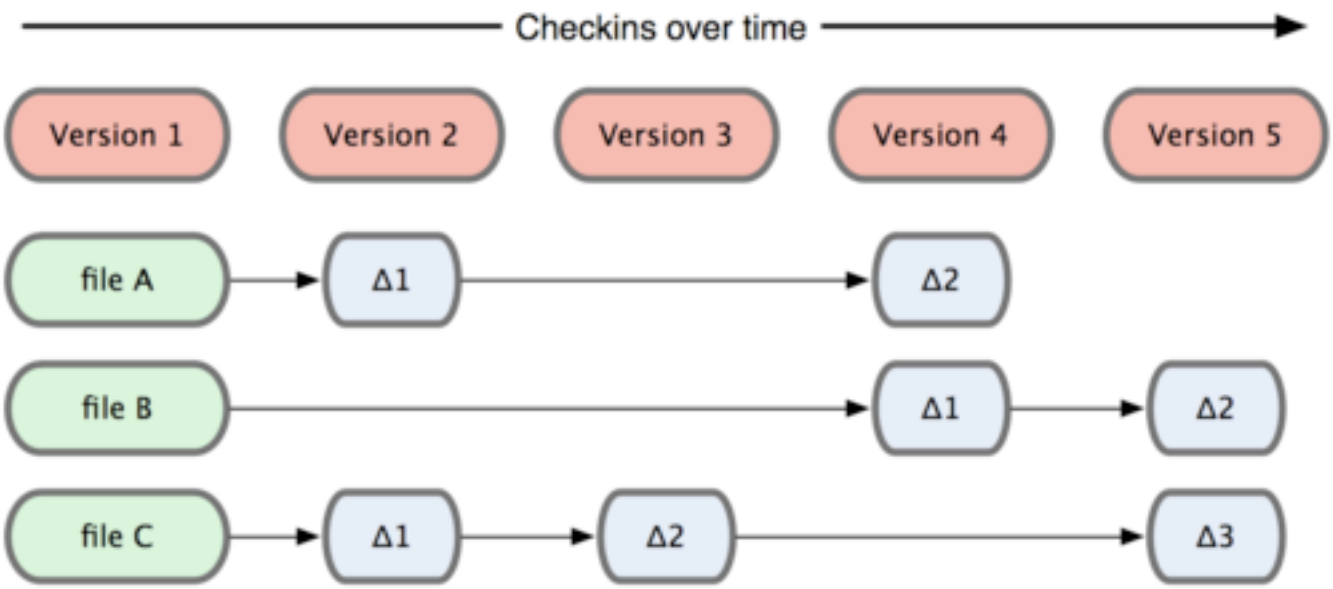


Figure 1-4. Other systems tend to store data as changes to a base version of each file.

http://progit.org/book/ch1-3.html

Snapshots, Not Differences

- a set of **snapshots** of a mini filesystem

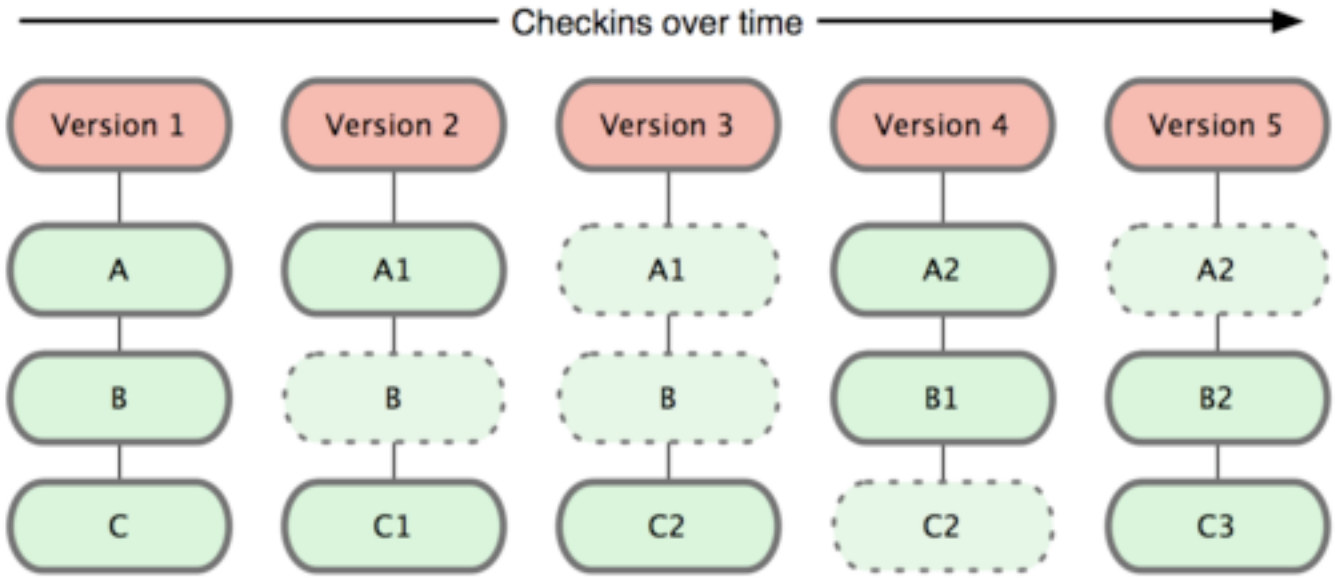


Figure 1-5. Git stores data as snapshots of the project over time.

http://progit.org/book/ch1-3.html

Nearly Every Operation Is Local

- Most operations in Git **only need local files** and resources to operate
  - generally no information is needed from another computer on your network.

- Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

http://progit.org/book/ch1-3.html

Git Has Integrity

- Everything in Git is check-summed before it is stored and is then referred to by that checksum.
  - This means it's impossible to change the contents of any file or directory without Git knowing about it.
- Functionality built into Git at the lowest levels and is integral to its philosophy.
  - **You can't lose information** in transit or get file corruption without Git being able to detect it.
- Mechanism that Git uses for this check summing is called a SHA-1 hash.
  - This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

http://progit.org/book/ch1-3.html

Git Generally Only Adds Data

- When you do actions in Git, nearly all of them only **ADD data** to the Git database.
  - difficult to get the system to do anything that is not undoable or to make it erase data in any way.

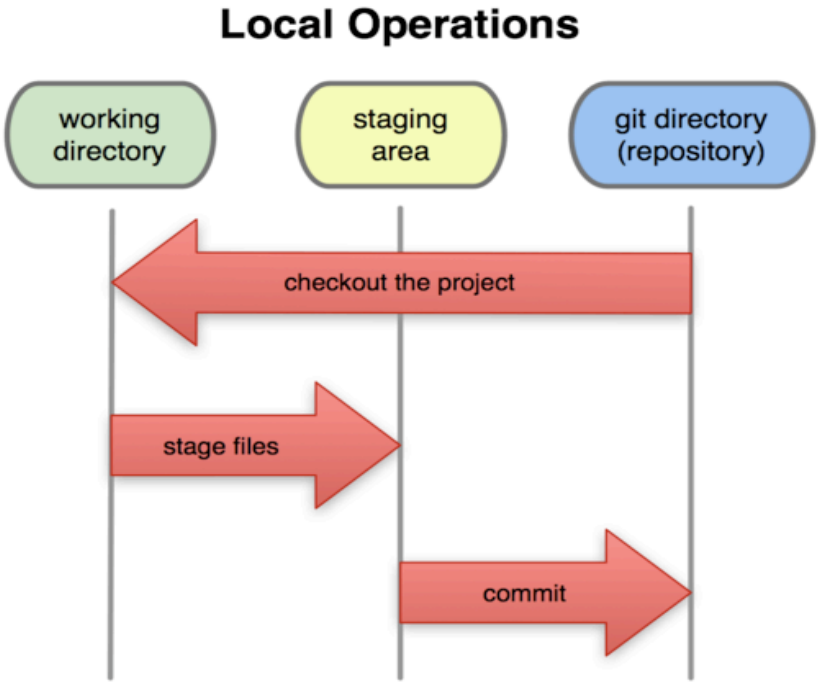
- As in any VCS, you can lose or mess up changes you haven't committed yet;
  - but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

http://progit.org/book/ch1-3.html

GIT Three States

- Git has three main states that your files can reside in: committed, modified, and staged

- **Committed** file is stored in the local database.
- **Modified** file is modified but have not committed it to the database yet.
- **Staged** file is marked a modified file in its current version to go into the next commit snapshot.



http://progit.org/book/ch1-3.html

http://gitolite.com/uses-of-index.html

Skipping the Staging Area

- Although it can be useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow.

- You can skip the staging area, Git provides a simple shortcut.

  -a option to the **git commit** command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the **git add**

http://progit.org/book/ch2-2.html

Initializing a Repository in an Existing Directory

- start to track an existing project in Git, go to the project's directory and type:

```
$ git init
```

- start version-controlling existing files (as opposed to an empty directory), do an initial commit on these files:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

http://progit.org/book/ch2-2.html

Cloning an Existing Repository

- get a copy of an existing Git repository

```
$ git clone git://github.com/schacon/grit.git [myGrit]
```

- important distinction: Git receives a copy of nearly all data that the server has. **Every version of every file for the history of the project is pulled down**

http://progit.org/book/ch2-2.html

Cloning an Existing Repository

- Checking the Status of Files `$git status`
- Tracking New Files `$ git add file`
- Staging Modified Files `$ git add file`
- Committing Your Changes `$ git commit`
- Removing Files `$ rm file; git rm file`
- Moving Files `$ git mv file1 file2`
- Ignoring Files create a `.gitignore file`

http://progit.org/book/ch2-2.html