



Universiteit Utrecht

[Faculty of Science  
Information and Computing Sciences]

# Software Configuration Management

Jurriaan Hage

e-mail: [jur@cs.uu.nl](mailto:jur@cs.uu.nl)

homepage: <http://www.cs.uu.nl/people/jur/>

Department of Information and Computing Sciences, Universiteit Utrecht

December 15, 2010

# Overview

## Software configuration management

Scenarios

Software configuration management areas

## Versioning

Subversion tutorial

Branching and merging

Other versioning tools

Darcs



# 1. Software configuration management



What it's not about:

- ▶ How to configure your printer driver for duplex printing in Windows XP.
- ▶ How to configure Doom 3 to use anti-aliasing.



- ▶ Webster: *configuration*, noun, from Latin verb *configurare*, “relative arrangement of parts or elements”.
- ▶ General CM: “the discipline of controlling the evolution of complex systems” (Tichy).
- ▶ Necessary for complex systems since:
  - ▶ Design evolves.
  - ▶ *System families* (or *product lines*).



- ▶ SCM is CM applied to software systems.
- ▶ Required since software products are composable, structured and evolving.
- ▶ Difference to general CM:
  - ▶ Software evolves much faster.
  - ▶ Potentially more automatable.



# 1.1 Scenarios



# Scenario: multiple developers, multiple workspaces

## §1.1

- ▶ Alice and Bob are working on some software product.
- ▶ They stay synchronised by emailing each other their changes.
- ▶ Problems:
  - ▶ Manual labour. Which files change?
  - ▶ What if they forget a file? Loss of synchronisation.
  - ▶ What if they both changed the same file?



# Scenario: multiple developers, single workspace §1.1

- ▶ Alice and Bob now use a network drive and work on the *same* files simultaneously.
- ▶ No longer any sync issues, but...
- ▶ Problems:
  - ▶ What if they are editing the same file at the same time?
  - ▶ Alice is making big changes to support component X. Bob is working on application Y that uses X. But while Alice is changing X, Bob is stuck because his stuff doesn't compile.



- ▶ Alice and Bob have released version 2.0 to the customers and start working on version 3.0.
- ▶ A bug is discovered in 2.0 which necessitates a version 2.1.
- ▶ So now there is development on two *branches*: the 2.x stable branch, and 3.x development branch.
- ▶ Problem:
  - ▶ They fix the bug relative to 2.0 and release 2.1...
  - ▶ But forget to port the fix to the ongoing 3.0 development.
  - ▶ Thus, a *regression* occurs.



Like the previous one, but:

- ▶ The source code for 2.0 has disappeared.



# Scenario: supporting old releases

§1.1

Like the previous one, but:

- ▶ The source code for 2.0 has disappeared.

Or:

- ▶ The compiler has disappeared.



Like the previous one, but:

- ▶ The source code for 2.0 has disappeared.

Or:

- ▶ The compiler has disappeared.

Or:

- ▶ The people who know how to build the system have disappeared.



Like the previous one, but:

- ▶ The source code for 2.0 has disappeared.

Or:

- ▶ The compiler has disappeared.

Or:

- ▶ The people who know how to build the system have disappeared.

Or:

- ▶ The hardware necessary to build the system has disappeared.



- ▶ Attempt to keep Skylab in space for a longer period (it was slowly falling back to earth due to atmospheric drag).
- ▶ This required uploading new control software.
- ▶ *“When IBM began to make preparations to modify the software, it discovered that there was almost nothing with which to work. The carefully constructed tools used in the original software effort were dispersed beyond recall, and, worse yet, the last of the source code for the flight programs had been deleted just weeks beforehand. This meant that changes to the software would have to be hand coded in hexadecimal, as the assembler could not be used—a risky venture in terms of ensuring accuracy. Eventually it became necessary to repunch the 2,516 cards of a listing of the most recent flight program, and IBM hired a subcontractor for the purpose.”*



# Real example: Skylab reactivation mission (cont'd)

## §1.1

- ▶ *“What happened after the manned Skylab program demonstrated the need for foresight and proper attention to storage of mission-critical materials until any possibility of their use had gone away. [...] The destruction of the flight tapes and source code for the software by unknown parties was inexcusable. A single high-density disk pack could have held all relevant material.” (NASA, Computers in Spaceflight—The NASA Experience)*



- ▶ To prevent all the sync overhead, every team member works on just one component of the system.
- ▶ At the end (just before the deadline), they put everything together (*big-bang integration*).
- ▶ Nothing works—expected and actual interfaces are subtly different from the design documents, many bugs surface only now in the complete system, and so on.



- ▶ For very small systems it may be doable to build the system by hand.

```
$ javac *.java
```

- ▶ Doesn't work anymore when we have
  - ▶ many build steps,
  - ▶ different languages,
  - ▶ many source files,
  - ▶ lots of build time configuration options.

```
$ gcc -g -DDEBUG=1 -c foo.c  
$ yacc parser.y  
$ gcc -g -DDEBUG=1 -c parser.c  
$ gcc -o app foo.o parser.o
```



- ▶ So we could just make a build script.
- ▶ What if we have a system consisting of 10,000s of source files and tens of millions of lines of code?  $\Rightarrow$  slow.
- ▶ And developers should rebuild/test the system after every change!
- ▶ Result: developers will skip testing.



- ▶ Solution: just rebuild the things that “changed”.
- ▶ If `foo.c` changes, rebuild only `foo.o` and `app`.
- ▶ But if `bar.h` changes, and `foo.c` includes that file...
- ▶ And what if the compiler changes?
- ▶ *Dependencies* are very tricky.
- ▶ If done manually: easy to forget things  $\Rightarrow$  binaries not consistent with sources.
- ▶ If done automatically: is the dependency information correct?



- ▶ The product is done and you want to release/ship it.
- ▶ Should be installed automatically.
- ▶ But at installation time, it turns out that you forgot to add a required file to the distribution.



- ▶ The user installed your application, but the installation breaks some other application because you overwrote some common DLL file in `C:/Windows/System32` with an incompatible version.



- ▶ The user tries to run the application, but gets an error message about `foo.dll` missing.
- ▶ Problem: your application required third-party `foo.dll`, but you forgot to ensure that it's installed.
- ▶ Testing didn't reveal this since `foo.dll` happened to be present on your machine.





- ▶ The user wants to get rid of the application. How to do that?
- ▶ Manually delete files: dangerous.
- ▶ Automatically delete files: need complete manifest of files belonging to the application.
- ▶ What about shared files?



## 1.2 Software configuration management areas



- ▶ Support the evolution of source code ⇒ *version management* (a.k.a. source revision control).
- ▶ Control building of derivative artifacts ⇒ *build management*.
- ▶ Manage transmission and installation of software ⇒ *software deployment / package management*.
- ▶ Support (continuous integration) testing and releasing ⇒ *continuous build systems* (a.k.a. build farms).

Related:

- ▶ *Issue tracking systems*.



## Goals:

- ▶ Provide safe storage for source code (a *repository*).
- ▶ Store history:
  - ▶ Source code at all (or many) points in time.
  - ▶ Reason for the change, and who made the change.
- ▶ Allow parallel lines of development.
- ▶ Allow flow of changes between developers and branches.
  - ▶ *Merging of changes.*
  - ▶ *Resolve conflicts.*
- ▶ Provide identification.
  - ▶ Give names to releases/branches.



## Goals:

- ▶ Build *derivates* (e.g., binaries) automatically from sources.
- ▶ Efficiency: prevent unnecessary rebuilds.
- ▶ Correctness: do rebuild if necessary.
- ▶ Support variability
  - ▶ Multiple platforms, debug on/off,...



## Goals:

- ▶ Get software from the producer site to the consumer site.
- ▶ Manage installations (install, upgrade, uninstall).
- ▶ Correctness: ensure that software works the same at producer and consumer sites.
- ▶ Deal with dependencies / interferences.



Integration of version management and automated builds (and maybe deployment).

Goals:

- ▶ Verify after every source change that the system builds / passes automatic tests.
- ▶ Portability testing: build on multiple platforms.
- ▶ Report build errors.
- ▶ Link back to version management ( $\Rightarrow$  “blame” facility).
- ▶ Maybe build and/or deploy packages automatically.



- ▶ Should be integrated with version management system.
- ▶ E.g., so we can query: “is bug X fixed in branch Y?”



## 2. Versioning



## Goals:

- ▶ Provide safe storage for source code (a *repository*).
- ▶ Store history:
  - ▶ Source code at all (or many) points in time.
  - ▶ Reason for the change, and who made the change.
- ▶ Allow parallel lines of development.
- ▶ Allow flow of changes between developers and branches.
  - ▶ *Merging of changes.*
  - ▶ *Resolve conflicts.*
- ▶ Provide identification.
  - ▶ Give names to releases/branches.



## 2.1 Subversion tutorial



- ▶ Central *repository* (usually stored on a separate server) holds all source code in a *versioned* file system.
- ▶ I.e., not just the “current” file system tree, but also previous versions (called *revisions*).
- ▶ Developers each have their own *working copy* of a revision.
- ▶ A working copy is created by doing a *checkout* of a certain path in the repository.
- ▶ Working copies are private, so changes don't affect others.
- ▶ When you are happy about your changes, you copy them to the server by doing a *commit* (creating a new revision).
- ▶ Others get your changes by doing an *update*.



- ▶ Repositories (and paths inside them) are identified using URLs.
- ▶ E.g., `http://server/repo/projectX/docs` or `file:///home/alice/myrepo/projectY/javasrcs/componentA`.
- ▶ A URL has a *repository part* and a *path part*. E.g., `http://server/repo` and `/projectX/docs`.



- ▶ Alice does a checkout:

```
alice$ svn checkout http://server/repo/interpreter
A interpreter/Print.java
A interpreter/Abstract.java
A interpreter/buildme.sh
...
Checked out revision 23.
```

```
alice$ cd interpreter
alice$ ls
Abstract.java  buildme.sh  Libs.jar....
```



- ▶ Alice edits a file (`Main.java`).
- ▶ The `svn status` command shows what files have changed.

```
alice$ svn status
M      Main.java
```



- ▶ The `svn diff` command shows the changes.

```
alice$ svn diff
--- Main.java      (revision 1)
+++ Main.java      (working copy)
@@ -12,6 +12,8 @@

    public static void main (String args[])
    {
+       System.out.println("Hello World");
+
        if (yyparse()) return 1;

        evalExpr(rootExpr);
```



- ▶ The `svn commit` command copies the changes to the repository. It will ask for a *log message* which should (briefly) describe the changes.

```
alice$ svn commit
(enters a message, like
  "added 'hello world' message")
Sending          Main.java
Transmitting file data .
Committed revision 24.
```



- ▶ Bob has his own working copy (still at revision 23).
- ▶ He can get Alice's changes by doing an `svn update`.

```
bob$ svn update
U  main.java
Updated to revision 24.
```



- ▶ But suppose that Bob also changed `Main.java` in his working copy.
- ▶ If the changes don't overlap, Subversion will automatically resolve this situation by *merging* the changes.



- ▶ For example, Bob has:

```
bob$ svn diff
--- Main.java      (revision 1)
+++ Main.java      (working copy)
@@ -17,6 +17,8 @@
     evalExpr(rootExpr);

     printExpr(rootExpr);
+
+   System.out.println("Goodbye!");

     return 0;
}
```



- ▶ Then Subversion will merge the changes:

```
bob$ svn update
G Main.java
Updated to revision 24.
bob$ cat Main.java
public static void main (String args[])
{
    System.out.println("Hello World");

    if (yyvsparse()) return 1;
    ...
    printExpr(rootExpr);

    System.out.println("Goodbye!");
    ...
}
```



- ▶ But this is not always possible. Suppose:

```
bob$ svn diff
--- Main.java      (revision 1)
+++ Main.java      (working copy)
@@ -12,6 +12,8 @@

public static void main (String args[])
{
+   System.out.println("Hello Utrecht!");
+
   if (yyparse()) return 1;

   evalExpr(rootExpr);
```



- ▶ Then the update will signal a conflict that requires *manual resolution*:

```
bob$ svn update
C Main.java
Updated to revision 24.
```



- ▶ The conflicting lines will be marked in Main.java:

```
bob$ cat Main.java
...
<<<<<<< .mine
    System.out.println("Hello Utrecht!");

=====
    System.out.println("Hello World");

>>>>>>> .r2
    if (yyvsparse()) return 1;
```



- ▶ Subversion will block attempts to commit until the conflict is resolved.

```
bob$ svn commit
svn: A conflict in the working copy obstructs
the current operation
svn: Commit failed (details follow):
svn: Aborting commit: '.../bob/interpreter/Main.java
remains in conflict.
```



- ▶ Subversion will block attempts to commit until the conflict is resolved.

```
bob$ svn commit
svn: A conflict in the working copy obstructs
the current operation
svn: Commit failed (details follow):
svn: Aborting commit: '.../bob/interpreter/Main.java
remains in conflict.
```

- ▶ Bob should edit Main.java to resolve the conflict (how?) and tell Subversion about it.

```
(edit Main.java)
bob$ svn resolved Main.java
Resolved conflicted state of Main.java
```



- ▶ Files created in the working copy will be ignored by Subversion unless you explicitly *add* them.

```
(Bob creates Optimiser.java)
```

```
bob$ svn status
```

```
?      Optimiser.java
```

```
bob$ svn add optimiser.java
```

```
A      Optimiser.java
```

```
bob$ svn commit -m "added an optimiser"
```

```
Adding      Optimiser.java
```

```
Transmitting file data .
```

```
Committed revision 26.
```



- ▶ Files can be removed using `svn remove`.
- ▶ File will also be deleted locally!

```
bob$ svn remove parser.cup
D          parser.cup
```

```
bob$ svn commit
(types in log message)
Deleting          parser.cup
Committed revision 27.
```



- ▶ The repository never forgets anything. You can always get access to any committed file or directory.
- ▶ E.g., you can checkout a *specific* revision:

```
alice$ svn checkout -r 26 \  
    http://server/repo/interpreter  
...  
Checked out revision 26.
```



- ▶ Bob has discovered that deleting `parser.cup` wasn't a good idea.
- ▶ How to undo the change?
- ▶ This was the change from revision 26 to revision 27.
- ▶ So Bob *merges* the *difference* (or *delta*) between revision 27 and revision 26 (a reverse delta). He applies this delta to his working copy.

```
bob$ svn merge -r 27:26 .
```

```
A parser.cup
```

```
bob$ svn commit
```

```
...
```



- ▶ You make a branch (a new line of development started from existing code) by making a *copy*.
- ▶ This is a *versioned copy*: Subversion remembers the historical link between the target and the source.



- ▶ Subversion does not require any specific repository layout, but a common layout is like this:

```
/projectX/trunk/  
/projectX/branches/  
/projectX/branches/1.x-stable/  
/projectX/branches/bob-experimental/  
/projectX/tags/  
/projectX/tags/1.0/  
/projectX/tags/1.1/  
/projectX/tags/2.0-beta/  
/projectX/tags/2.0/  
/projectX/tags/2.0.1/
```



- ▶ The *trunk* is the main development branch.
- ▶ The *branches* directory contains non-mainline development (such as bug-fix-only development on the 1.x series, or Bob's experimental changes). These are copies of the trunk or another branch at some point.
- ▶ The *tags* directory contains *releases*; these never change ( $\Rightarrow$  identification).



- ▶ Alice creates a branch for some experimental stuff.

```
alice$ svn copy http://server/repo/interpreter/trunk \  
    http://server/repo/interpreter/branches/alice-test \  
Committed revision 28.
```

(in her working copy of trunk)

```
alice$ svn switch \  
    http://server/repo/interpreter/branches/alice-test
```



- ▶ Many commits later, Alice has completed the work on her test branch. The experimental stuff is now stable, so it should be merged back into the trunk.

```
alice$ svn switch \  
    http://server/repo/interpreter/trunk  
  
alice$ svn merge \  
    http://server/repo/interpreter/trunk@28 \  
    http://server/repo/interpreter/branches/alice-test \  
    .
```

- ▶ This applies the difference between the *branch point* (the trunk at revision 28) and the current test branch to Alice's working copy of the trunk.
- ▶ There probably will be merge conflicts.



- ▶ What if Alice continues to work on experimental?
- ▶ Solution: merge only the stuff changed since the previous merge.
- ▶ Otherwise you may get duplicate merges.

```
alice$ svn merge \  
http://server/repo/interpreter/branches/alice-experimental@34  
http://server/repo/interpreter/branches/alice-experimental \  
.
```



- ▶ `svn mkdir` creates a directory.
- ▶ `svn move` (a.k.a. `svn rename`) moves or renames a file or directory.
- ▶ `svn help` and variants like `svn help merge`
- ▶ `svn log` shows list of log messages.
- ▶ Most commands work both on working copies as well as URLs.



## 2.2 Branching and merging



Support different lines of development. E.g.,

- ▶ The “main” branch: development for the next major release.
- ▶ Support branches: bug fixes for previous releases.
- ▶ Multiple “new” development branches: next major release, next minor release.
- ▶ Experimental branches for things too dangerous for the main branch.

What branches to use is a policy decision.



Advantages of branching:

- ▶ Isolation between lines of development.

Disadvantages:

- ▶ Danger of late integration.
- ▶ Merging can be hard.



One extreme:

- ▶ Never branch (except maybe for maintenance branches that will never be merged).

Other extreme:

- ▶ Create branches for each developer / each issue.



- ▶ Continuous integration is generally better (faster feedback on incompatible changes); no big-bang integration.
- ▶ But do branch for large changes that might break the trunk for a long time.



- ▶ Be careful changing layout, method/function ordering, etc.
- ▶ It makes merging much harder.
- ▶ Doesn't apply if there is no need for merging (e.g., just a single developer working on the source file at the time).



- ▶ Commit often.
- ▶ Don't put multiple logical changes in a single commit — makes it harder to undo changes.
- ▶ Don't commit when it breaks the branch (even more) [unless it's a private branch].



- ▶ Merging loses history.
- ▶ Central repository: you cannot commit if you don't have commit rights. It's not (easily) possible to branch (and merge) a repository.
- ▶ Revisions and branching/tagging through copying are not orthogonal concepts. Why not make every commit operation a copy (with delta) of the previous trunk? E.g.,

```
/trunk/0
```

```
/trunk/1
```

```
/trunk/2
```

```
...
```



## 2.3 Other versioning tools



- ▶ What are the “first class” objects of the version management system?
  - ▶ Revisions (CVS, Subversion, ...)
  - ▶ Deltas between revisions (*changeset-based systems*: Darcs, BitKeeper, ...)
- ▶ How to deal with simultaneous edits?
  - ▶ Lock, modify, unlock
  - ▶ Copy, modify, merge (e.g., Subversion)
- ▶ How to specify configurations of source files?
  - ▶ E.g., “I want version 2.3.6 of the project with the FOO version for version 3.0 but without the BAR bug fix added in version 2.3.1”
- ▶ How to represent branches?
  - ▶ Extra dimension (CVS)
  - ▶ “First class” copies (Subversion)
  - ▶ Copy the repository (Darcs)



- ▶ What to version?
  - ▶ Files
  - ▶ Directories
  - ▶ Semantic entities (e.g., classes)
- ▶ You typically don't version derivatives
  - ▶ but there are exceptions.
- ▶ Merge support
  - ▶ Repeated merge problem
  - ▶ Syntactic vs. semantic merging
- ▶ Distributed operation
  - ▶ Open source development
- ▶ Integration with build management
- ▶ Tool for moving between different kinds of versioning:  
Tailor



- ▶ BitKeeper (<http://www.bitkeeper.com/>)
  - ▶ True distributed system.
  - ▶ Each user has his/her own repository (cloned from a “parent” repository).
  - ▶ Branching = cloning.
  - ▶ Changes can be moved between parents/children, but also between any repositories in the same tree.
  - ▶ Deltas (“changesets”) are first-class entities.
  - ▶ Proprietary.
- ▶ GNU arch (<http://www.gnu.org/software/gnu-arch/>)
  - ▶ Similar to Bitkeeper, but open source



- ▶ Mercurial (since 2005): distributed SCM system
  - ▶ scalable, well-documented and mature
  - ▶ delta-mechanism, but smart (compression etc.)
  - ▶ history-aware merge: the system remembers that a merge took place
  - ▶ not very good at “cherry-picking”
- ▶ Vesta: focus on repeatability of builds.
  - ▶ Any past build can be redone in the future
  - ▶ Builds also depend on compiler used, flags,...
  - ▶ Scalable
- ▶ ICE: feature selection leads to a specific build.
  - ▶ Specified as `#ifdefs` in code
- ▶ Darcs: a distributed system
- ▶ Git: a rather popular distributed system



## Compared to Subversion:

- ▶ Directories are not versioned (renaming/moving files is impossible).
- ▶ Branches are identified along a separate dimension—less intuitive.
- ▶ Commits/checkouts are not atomic.
  - ▶ Conflicting files are skipped, while others are still committed
- ▶ Versioning is per file, no global repository revision numbers
- ▶ Network operation is poorly implemented.
- ▶ Branching is  $\Theta(n)$  time, not  $O(1)$  time.



## 2.4 Darcs



- ▶ Changeset: set of changes to be applied to a source tree
  - ▶ Edit file
  - ▶ Add file
  - ▶ Remove file
  - ▶ Rename/move file
- ▶ In Subversion changesets are implicit: they are the difference (delta) between repository revisions.
- ▶ `svn diff -r(N - 1):N`  
gives the changeset corresponding to revision  $N$ .
- ▶ In changeset-based systems *changesets*, *not revisions* are the main objects of the system.



- ▶ Ideally a changeset corresponds to a “logical” change:
  - ▶ Addition of a new feature
  - ▶ A bug fix
  - ▶ A refactoring
- ▶ We can then (ideally!) easily select the features/bug fixes etc. that we like.
- ▶ E.g., given changesets  $\{A, B, C, D, E\}$ : give me the source code resulting from changesets  $\{A, B, D\}$  (i.e., I don't want  $C$  and  $E$ ) *Cherry Picking*.
- ▶ I.e., the working copy is  $A + B + D$ . A source tree is the sum of a set of changesets.



- ▶ Changesets are useful in loosely organised projects (such as some open source projects, e.g., the Linux kernel).
- ▶ Not all developers need to have write permission to a central repository (as opposed to CVS and Subversion).
- ▶ Rather developers can copy (branch) the central repository, work on the copy, and send back changesets.
- ▶ Supports many different policies: e.g., one central repository, hierarchy of repositories, or ad-hoc organisation.
- ▶ Also makes cooperation between branches easier: developers can selectively apply changesets from other branches.



- ▶ Changeset-based version management system
- ▶ Free software
- ▶ <http://abridgegame.org/darcs/>
- ▶ Example: used for development of GHC.
- ▶ In fact, darcs was written in Haskell.



- ▶ Working copies and repositories are the same.
- ▶ Alice creates an empty repository/working copy:

```
$ mkdir /repos/alice
$ cd /repos/alice
$ darcs initialize
```

- ▶ Then she creates some files and *records* the changes into a changeset:

```
$ darcs add foo.c bar.c
$ ls
_darcs foo.c bar.c
$ darcs record
...
```

What is the patch name? Initial revision



Bob imports (“pulls”) patches from Alice to his own repository:

```
$ mkdir /repos/bob
$ cd /repos/bob
$ darcs initialize
$ darcs pull /repos/alice
Pulling from "/repos/alice"...
Finished pulling and applying.
$ ls
_darcs foo.c bar.c
```

*Branching = cloning!*



Bob makes some changes, then

```
$ darcs record
hunk ./foo.c 219
-   assert(MAX_BLOCK_SIZE >= 16);
+   assert(MAX_BLOCK_SIZE >= 32);
Shall I record this patch? (1/?) [ynWsfqadjkc]: y
What is the patch name? Increased block size
```

There are many ways to get changes to Alice:

- ▶ Alice can *pull* the changes from Bob's repository.
  - ▶ Selectively adds changesets not already in her repository.
- ▶ Bob can *push* the changes to Alice's repository (if he has write permission).
- ▶ Bob can send them by email.



Bob sends the changeset by email:

```
$ darcs send  
Mon Sep 19 22:45:32 CEST 2005 bob@example.org  
  * Increased block size  
Shall I send this patch? (4/4) [ynWvpxqadjk]: y  
What is the target email address? alice@example.org  
Successfully sent patch bundle to: alice@example.org.
```



Alice receives the changeset and applies it after review:

```
$ darcs apply --interactive bobs-email.txt
```

```
Mon Sep 19 22:45:32 CEST 2005 bob@example.org
```

```
* Increased block size
```

```
Shall I apply this patch? (1/1) [ynWvpxqadjk], or ? for help: v
```

```
[Increased block size
```

```
bob@example.org**20050919204532] {
```

```
hunk ./foo.c 219
```

```
- assert(MAX_BLOCK_SIZE >= 16);
```

```
+ assert(MAX_BLOCK_SIZE >= 32);
```

```
}
```

```
Mon Sep 19 22:45:32 CEST 2005 bob@example.org
```

```
* Increased block size
```

```
Shall I apply this patch? (1/1) [ynWvpxqadjk], or ? for help: y
```

```
Finished applying...
```



Not having a central place that holds the artifacts of a project can cause problems:

- ▶ Development work might be lost: what if a developer throws away his local repository?
- ▶ An SCM tool should support *awareness*: it should help developers to know what other developers are doing. A distributed systems makes the following harder:
  - ▶ Commit notifications
  - ▶ Locking
  - ▶ Ability to see unmerged work on other branches
  - ▶ Continuous integration (i.e., automatic building and testing in a build farm)
  - ▶ Integration with issue-tracking system



Doing a pull can take a very long time: it downloads all patches, rather than the current files. Bad scalability. And,

```
[eelco@hagbard:/tmp]$ \  
  darcs get http://www.abridgegame.org/repos/darcs/  
Welcome to the darcs darcs repository!
```

```
This is the stable release branch.  
*****  
Copying patch 769 of 3519...
```



- ▶ Eelco Dolstra made most of the slides.
- ▶ Paper: Walter Tichy, *Software Configuration Management Overview*.
- ▶ Book: Stephen P. Berczuk, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*
- ▶ The Subversion handbook:  
<http://svnbook.red-bean.com/>.

