
9. Implementation and Change: Coordination, Versioning and Tracking

9.1 Introduction

9.2 Manual Means

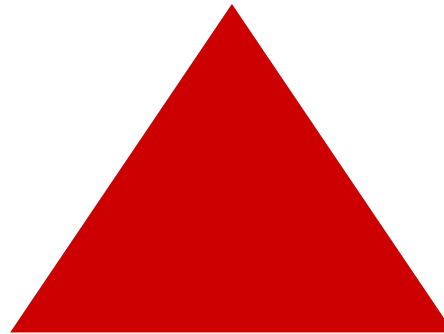
9.3 Repository Systems

Implementation and Change

9.1 Introduction

SOFTWARE DEVELOPMENT. THE KEY Triangle

Large and Long-lived
Application Systems

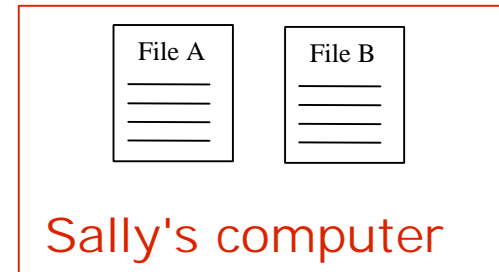
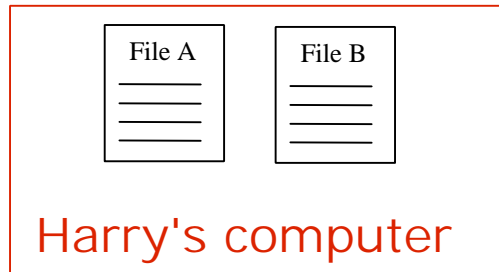


Cooperating
Development Experts

Well-engineered
Software Structures

Strong Points of OO Analysis,
Design and Implementation

The Situation



- o Scenarios:
 1. Harry changes File A.
 2. Sally changes File B.
 3. Both Harry and Sally change File A at the same time.
- o Different problems arise:
 - n How do you propagate the changes to the others?
 - n How do you merge changes to the same file?
 - n How do you keep track of different revisions of a file?

Propagating Change

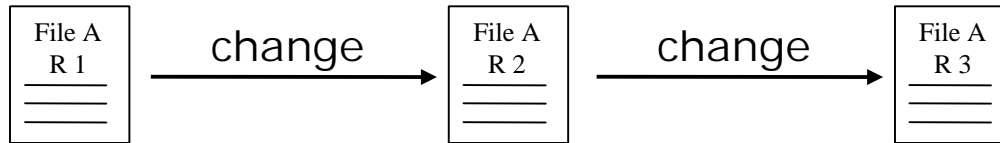
- o In teams, multiple people change artefacts at the same time.
 - n They need to inform each other of the changes.
 - n They have to be careful to only make non-conflicting changes.
- o Several issues arise:
 - n Concurrency: How do you merge two changes to the same file?
 - n Consistency: Will the sum of all changes result in a consistent system?
 - n Durability: How do you make sure, no change is lost?

Merging Changes

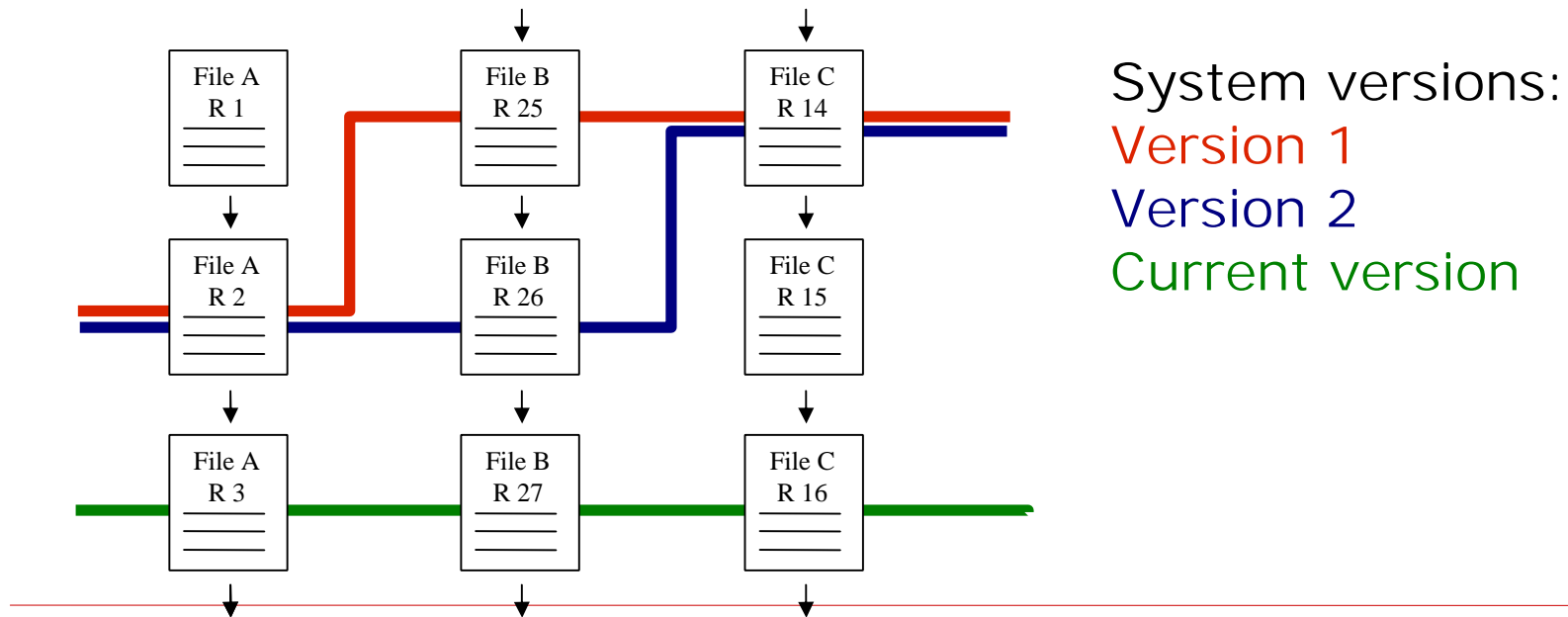
- o Changes applied by different people have to be integrated.
 - n Even if the changes are to different files, you still have to make sure, the result works (i.e., at least compiles).
 - n If the changes are to the same file, you have to create a new version of the file that combines all the changes.
- o Note: To make sure that the merge results in a consistent system, you have to understand the changes.

Versions and Revisions

- Revisions of a file:

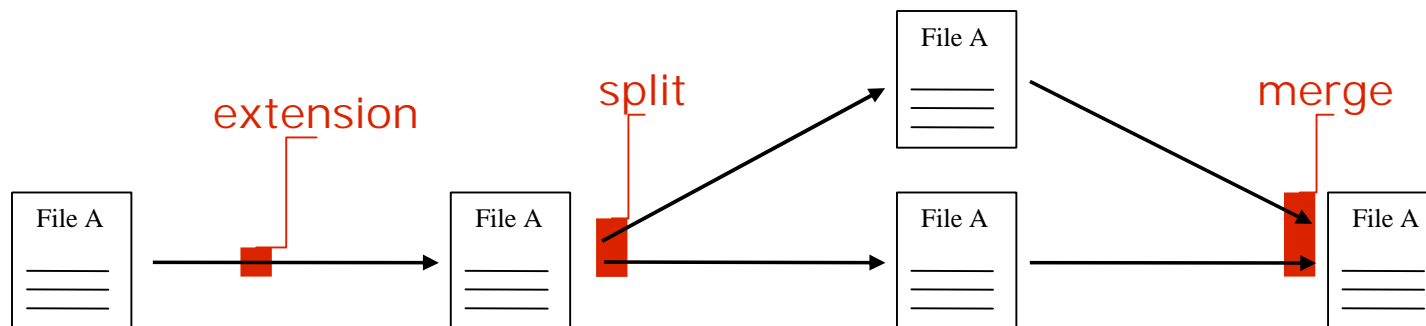


- Versions of the system:



Revision Graphs

- o Show revisions of a file
 - n Node: Revision
 - n Edge: ... stems from ...
- o Cases:
 - n extension
 - n split
 - n merge



Version and Change Tracking Tools

9.2 Manual Means

The manual way: Diff & Patch

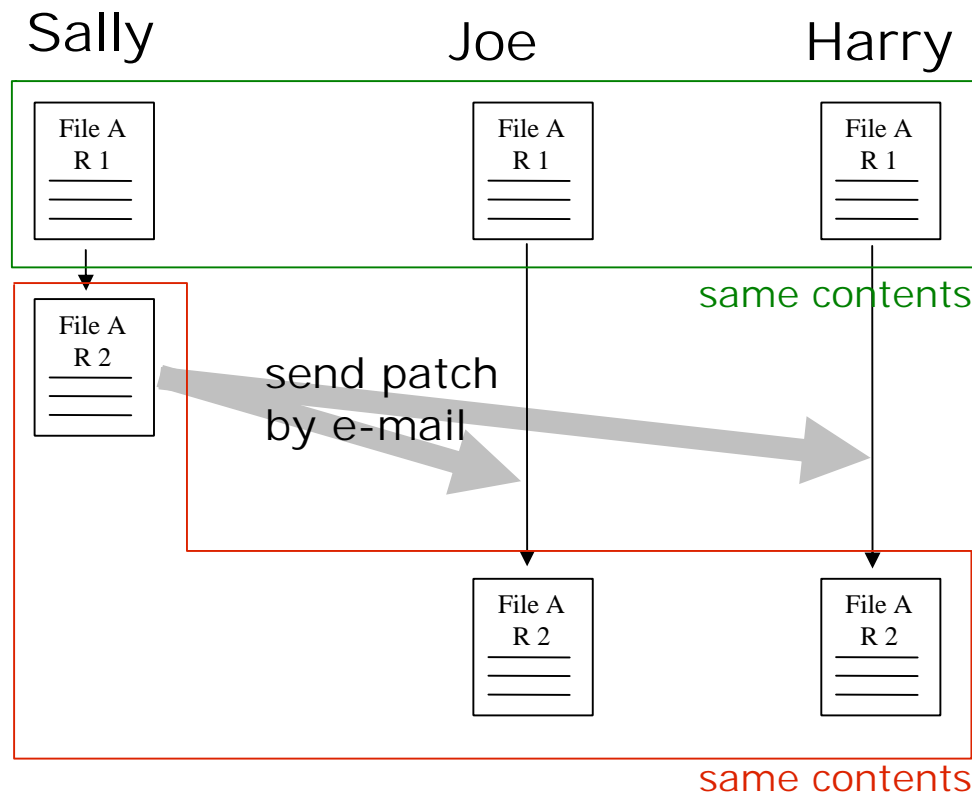
- To propagate a change manually you can do the following steps:
 1. You create a diff-file (UNIX has a tool called `diff` that does it for you). This file is also called a patch.
 2. You e-mail this file to everybody concerned.
 3. Your colleagues use the `patch` utility to merge your changes into their code.

There is no master version of the code, everybody keeps its own version. It is therefore very easy to loose a change or to have the versions out of synch.

Don't do this! Generally it does not work!!

Diff & Patch Workflow

- Sally changes File A and sends the differences as a patch file to Joe and Harry by e-mail.



Disadvantages of Diff & Patch

- o Many manual steps
 - n Error-prone.
 - n Same work has to be done many times.
 - n Easy to defer and forget (“I’ll apply that change tomorrow”).
- o No version of the code that everybody agrees on.
 - n What do you ship?
 - n How do you make sure the system still works?
- o You have no guarantee that everybody actually applies your changes.
 - n It is so easy to loose updates!
(Sometimes such losses are unnoticed for weeks.)
- o No management of versions and revisions.

Where to use Diff & Patch

- o The diff & patch approach can be of some use:
 - n to convey changes of the source code to people who are not members of your team.
 - o This is a one-way road, you do not usually receive changes from outside. Thus low on conflicts.
 - o The external people don't have to look at all the code to find out which parts changed.
 - n to visualize changes (given in the diff file) in order to discuss them (e.g. on a mailing-list).
 - o The real changes (i.e., the results of the discussion) are usually propagated by different means.
- o In neither of these approaches diff & patch is used for actual development!

Version and Change Tracking Tools

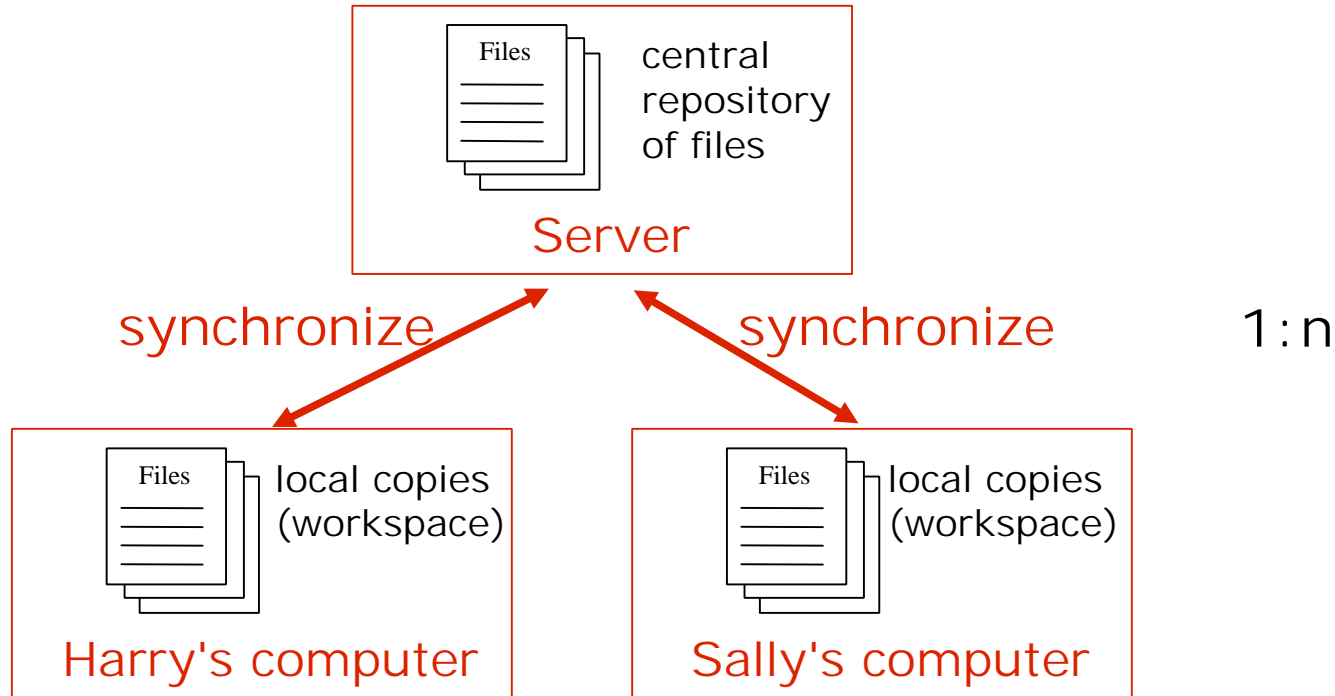
9.3 Repository Systems

Repository Systems (1)

- o Repository systems try to address the issues of diff & patch consistently.
- o Important properties of repository systems:
 - n Serve as central code repository.
 - o Everybody synchronizes with the repository. The propagation situation is no longer n:m but 1:n.
 - n Provide facilities to merge changes.
 - n Do version and revision tracking.

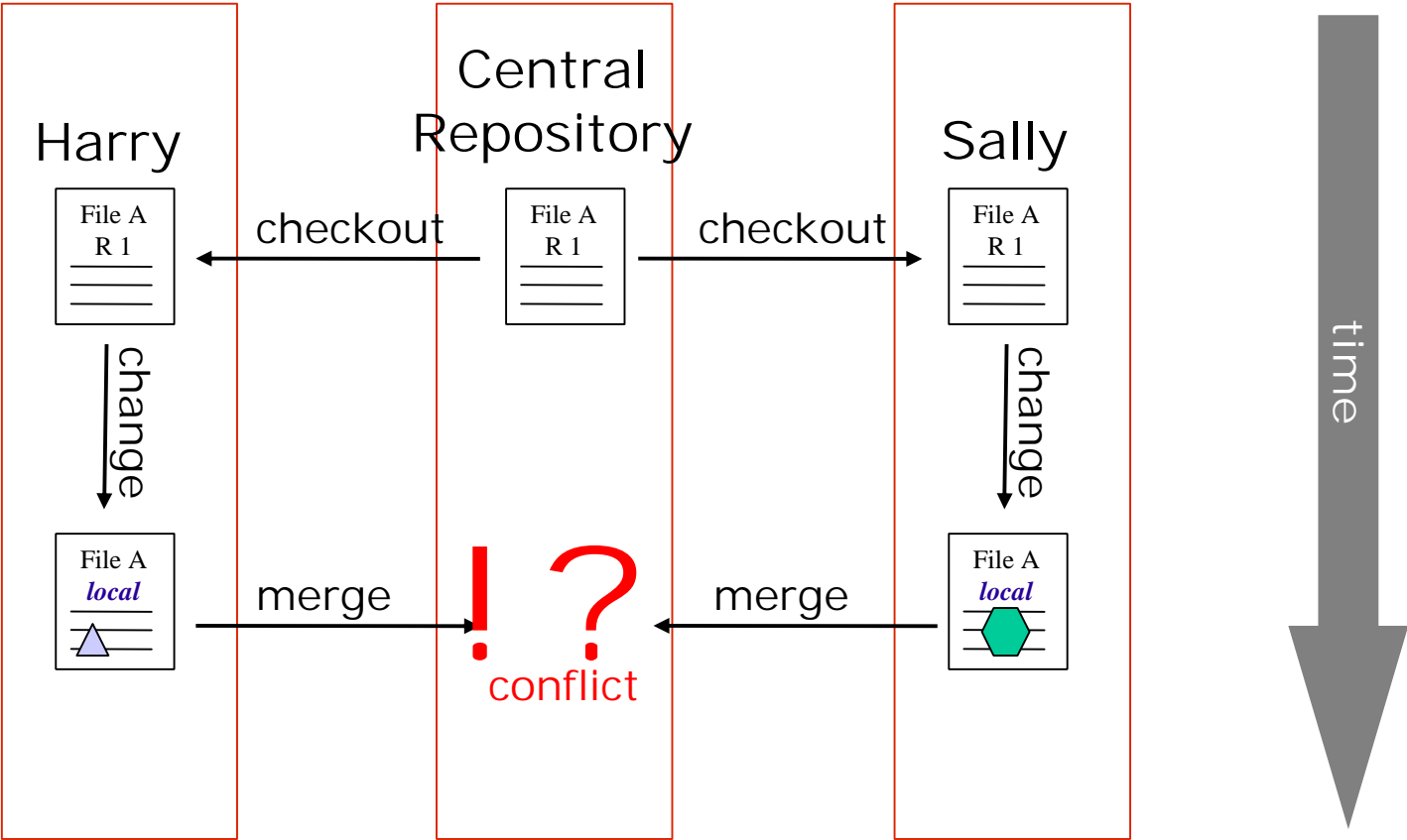
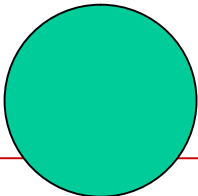
There are various systems that implement these properties in slightly different ways.
(Their terminology is not consistent.)

Repository Systems (2)



- Harry and Sally only synchronize with the server.
- They never share files with each other directly.
- Both work on local copies and upload changes to the server.

Conflict (1)



Conflicts (2)

- o How do you solve the problem of clashes when concurrent changes occur?
 - n Prevent clashes altogether.
 - n Try to resolve as many conflicts as possible.
- o Observations:
 - n When developers produce a conflict, they are working on the same “thing” (package, component, module, object, ... entity). This should not have happened in the first place.
 - n When developers work on different “things”, this usually does not create real conflicts, even if they work on the same file.
- o Repository systems need to have some sort of solution for conflicts.

Conflict Resolution Paradigms

- o Locking (“pessimistic approach”)
 - n When developers need to edit a file, they have to lock it first. This will ensure that only one of them is editing it.
 - n After editing (and putting the changes back into the repository) they have to unlock the file.
- o Merging (“optimistic approach”)
 - n When developers need to edit a file, they go ahead and just do it.
 - n After editing a file, developers have to upload the changes into the repository.
 - n If conflicts occur, they have to resolve them manually.

Locking

- Have to remember to lock before you edit (there usually is tools support for this).
- Have to unlock after editing, this is easy to forget. (Especially if you decide not to make the change.)
- Often the lock is unnecessary (no conflict would have occurred).
- Issue: Locking granularity (see: database transactions)

Locking is safe, but often too restrictive. Update conflicts cannot occur, but serialization of concurrent development can seriously delay and annoy developers and cause new problems.

Merging

- o Assume, the worst case (manual resolution) will not occur.
- o Pseudo-conflicts, i.e., changes to disjoint places of the same file, can be resolved semi-automatically.
 - n Developer usually has to approve the semi-automatic resolution.
 - n What file formats does your tools support?
- o Real conflicts can require quite a bit of work to resolve them.

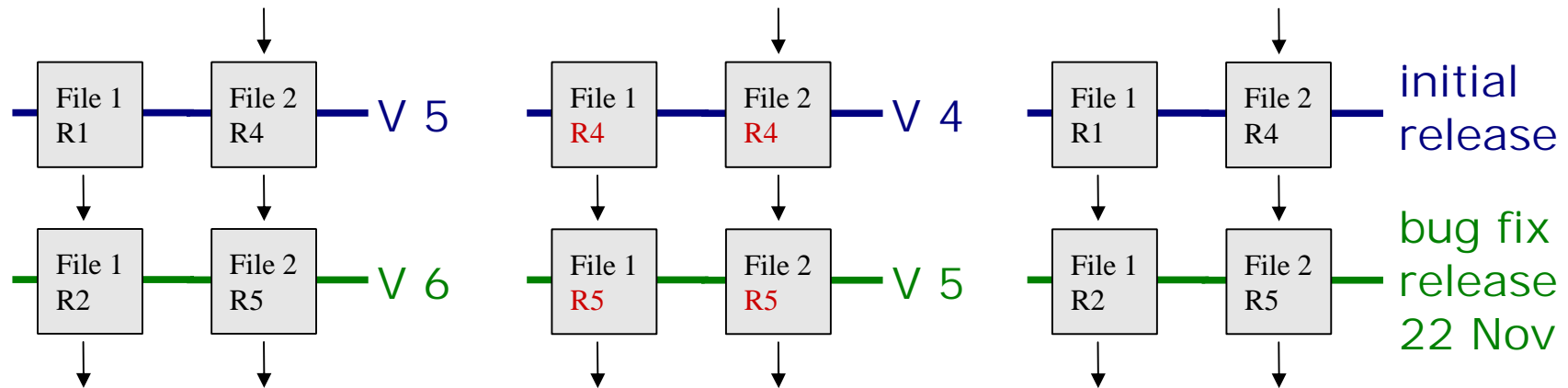
If your repository system uses merging, try to keep the change cycles short. This will make real conflicts less likely.

Revision History

- o Repository systems keep track of revisions of a file.
- o This allows you to:
 - n track changes
 - n go back to old revisions (or even versions of your system)
- o The repository also knows about different versions of the whole system
 - n these are sets of revisions of files
 - n are usually named/numbered
- o Application: Be able to debug system version 3.2 even though the current version is already 10.8
 - n important for support and bug fixing tasks

On the Naming of Versions

- o Different approaches:
 1. a version contains many (different) revision numbers
 2. all revisions of a version have the same revision number
 3. give each important version a unique textual name



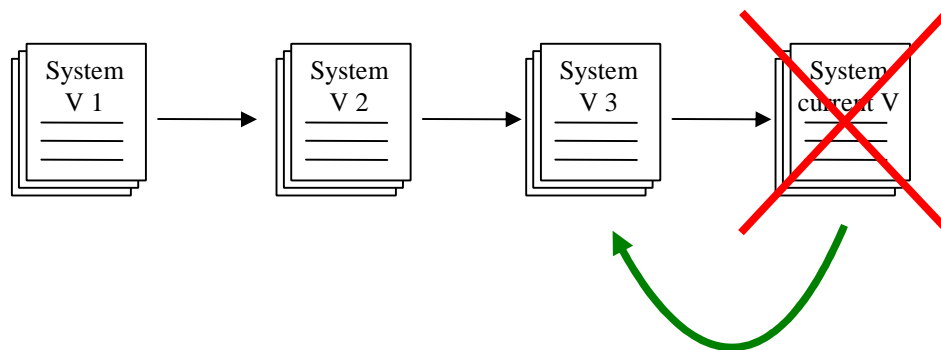
1. revision-# tells you nothing about the version

2. revision-# is equal to the version-#

3. versions have meaningful names

Rollback

- o Not all changes turn out to be good.
- o History gives you the opportunity to undo changes by going back to previous versions.
 - n you can try what-if scenarios
 - n only works if you upload your changes often
 - n you should name correctly functioning versions to make sure you can later safely roll back to them

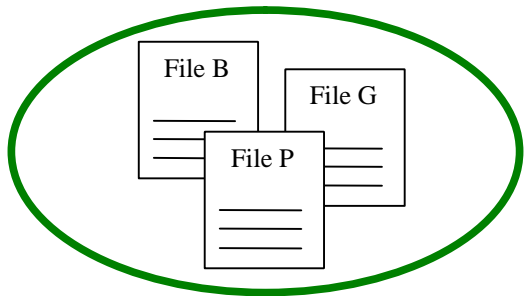


Semantics of Changes

- o History of your system is a list of changes.
 - n Either of changes to each file individually
 - n but more understandable if grouped by implementation task.
- o Recording changes:
 - n most repository system ask for a comment every time you upload changes
 - n can later browse these comments
 - n limited use if comments are not stored in a central place but on each file
(e.g., CVS, Concurrent Versions System)
- o Solution: Change-sets, Change-list
 - n group file changes according to tasks
 - n changes can be tracked in context

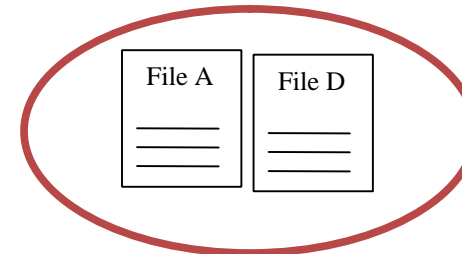
Change-Sets

- Group related changes into change-sets.
- Every change you make belongs to a change-set.
- You can even work on multiple change-sets at the same time.
- Example:
 1. Open new change-set
 2. Change some files
 3. Upload the change-set.



change-set 1:
fix bug #184

change-set 2:
implement /F01

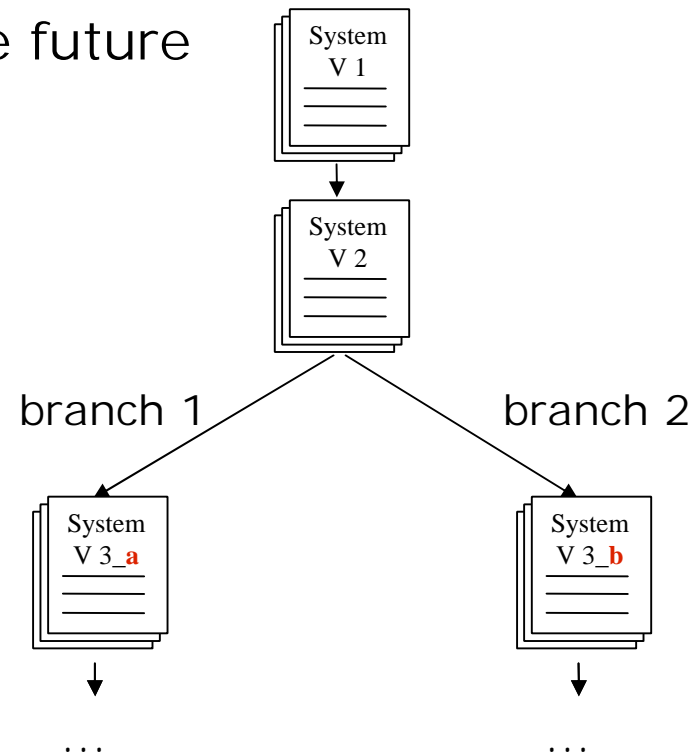


Access Control

- o (Code) repositories contain important assets
 - n you need to control who has read/write access rights
 - n unauthorized access can lead to various problems:
 - o stealing of code
 - o smuggling in of malicious code
 - o disclosure of secret information
- o Access control within the development team?
 - n depends on the development process you run
 - n e.g., eXtreme Programming does not use it

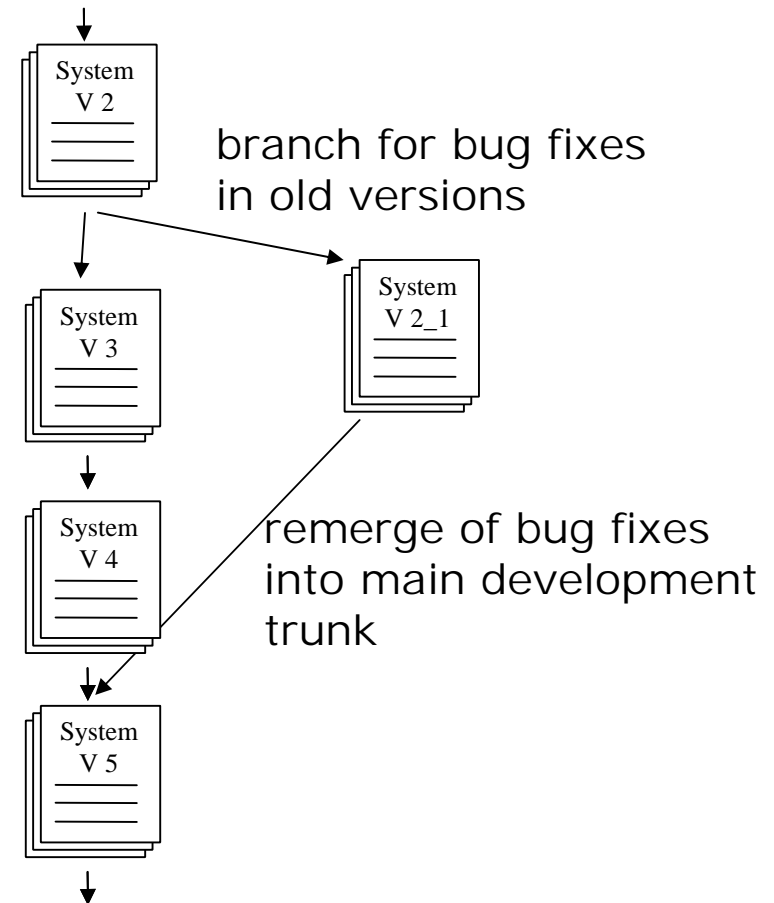
Branching (1)

- o Creates two version of the same repository
 - n based on same code
 - n independent in the future



Branching (2)

- o Fix bugs in old versions while still developing the current system
 - n Avoids fixing the same bug twice (in each version)
 - n Independent teams can work on each branch



General Workflow of a Developer

(1)

1. Get an initial version from the repository.
2. Write code (create changes).
3. Upload the changes into the repository.
 - Resolve conflicts
(e.g., when somebody else updated the same file).
4. Get the changes of the colleagues from the repository.

Repository based on merging.

General Workflow of a Developer

(2)

1. Get an initial version from the repository.
2. Lock the files you want to work on
 - o Complain to people who block the desired files with locks. Wait for the locks to be released.
3. Write code (create changes)
4. Upload the changes into the repository.
5. Unlock (often happens automatically)
6. Get the changes of the colleagues from the repository.

Repository based on locking.

General Workflow of a Deployer

1. Get the latest changes from the repository.
2. Name the current version of the system.
 - The name is usually related to what you call the shipped software (e.g., V2_0_0 for release 2.0.0)
3. Create a branch that will serve as future bug fix branch.
4. Bundle the software and ship it.

Non-central Repository Systems

- o There also are repository systems that do not require a central server with the code repository
- o Well suited for distributed development.
- o Development in a closed team usually does not need them.
- o Example products:
 - n Bitkeeper
 - n Monotone