Week 10: Configuration Management

- 10.1 Version management
- 10.2 System building
- 10.3 Change management
- 10.4 Release management
- 10.5 Subversion
- 10.6 Tutorial: Subversion

(Reading: Chapter 25, Software Engineering, Ian Sommerville, 10th edition, 2016)

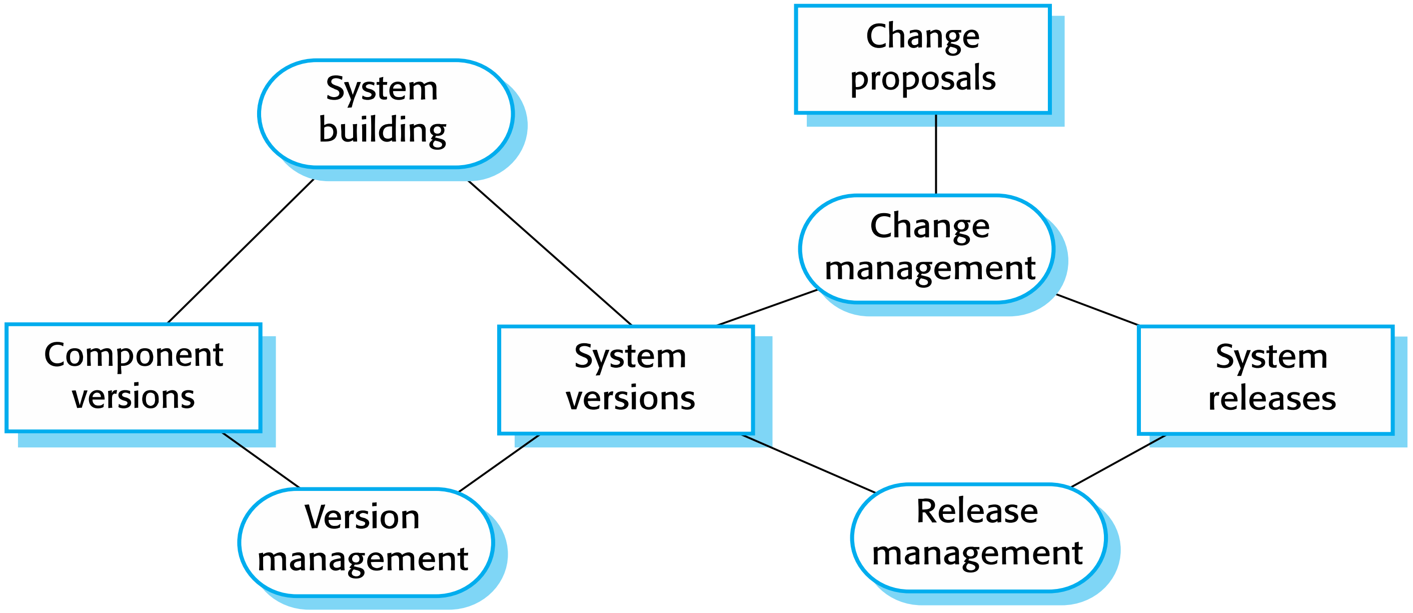
Configuration management

- Software systems are constantly changing during development and use.
- *Configuration management (CM)* is concerned with the policies, processes and tools for managing changing software systems.
- You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- CM is essential for team projects to control changes made by different developers

CM activities

- Version management (Version Control)
  - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- System building
  - The process of assembling program components, data and libraries, then compiling these to create an executable system.
- Change management
  - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- Release management
  - Preparing software for external release and keeping track of the system versions that have been released for customer use.

Configuration management activities



- CM tools are used to store versions of system components, build systems from these components, track the releases of system versions to customer, and keep track of change proposals.

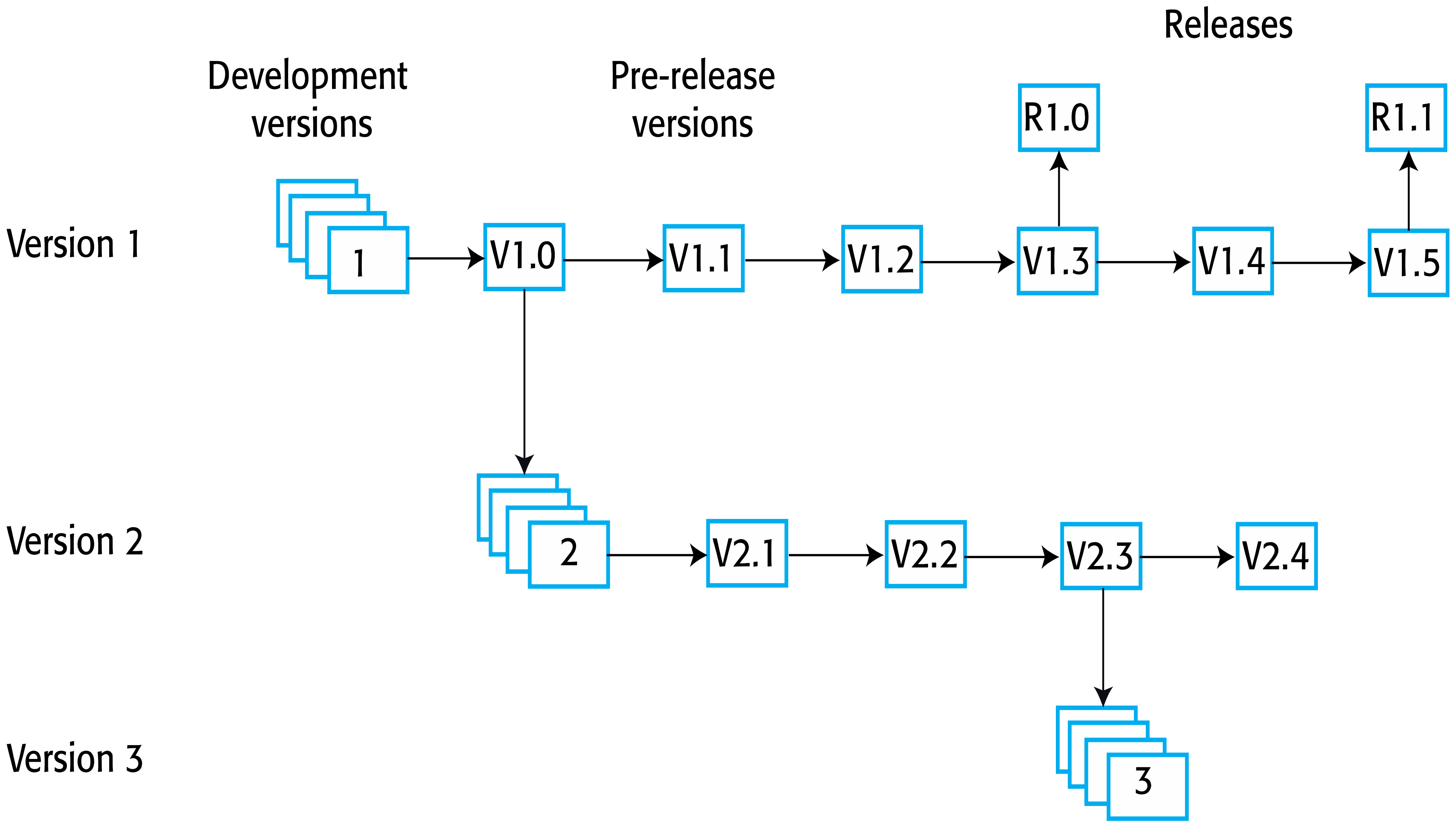
Agile development and CM

- Agile development, where components and systems are changed several times per day, is impossible without using CM tools.
- The definitive versions of components are held in a *shared project repository* and developers copy these into their own *workspace*.
- They make changes to the code then use *system building* tools to create a new system on their own computer for testing.
- Once they are happy with the changes made, they return the modified components to the project repository.

Multi-version systems

- For large systems, there is never just one ‘working’ version of a system.
- There are always several versions of the system at different stages of development.
- There may be several teams involved in the development of different system versions.

Multi-version system development



CM terminology

Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Mainline	A sequence of baselines representing different versions of a system.

CM terminology

Term	Explanation
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

CM terminology

Term	Explanation
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

10.1 Version management

- *Version management (VM)* is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- Therefore version management can be thought of as the process of managing *codelines* and *baselines*.

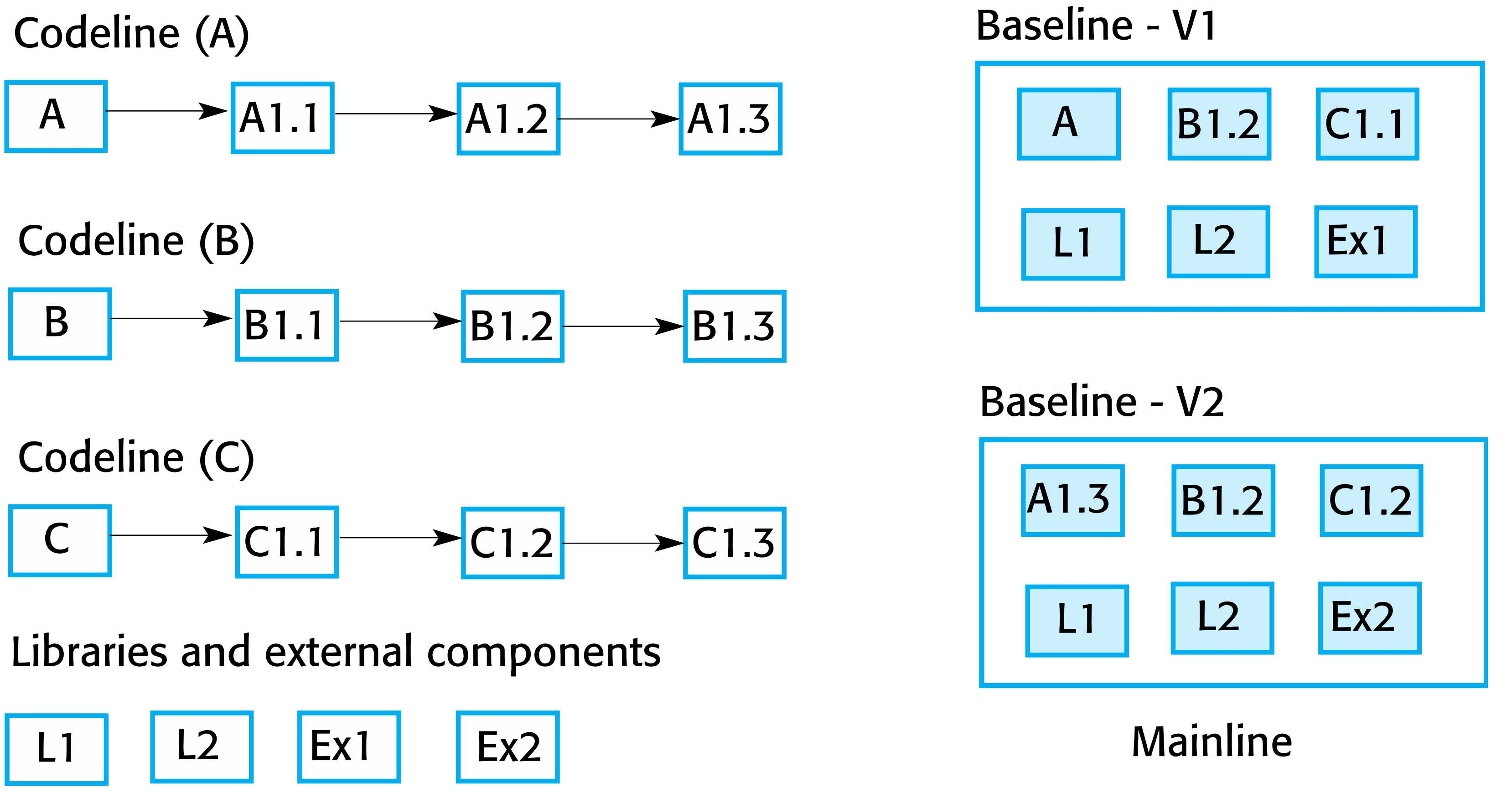
Codelines and Baselines

- A *codeline* is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- *Codelines* normally apply to components of systems so that there are different versions of each component.
- A *baseline* is a definition of a specific system.
- The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

Baselines

- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- Baselines are important because you often have to recreate a specific version of a complete system.
  - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

Codelines and Baselines



Version control systems

- *Version control (VC)* systems identify, store and control access to the different versions of components. There are two types of modern version control system
  - *Centralized systems*, where there is a single master repository that maintains all versions of the software components that are being developed. ***Subversion*** is a widely used example of a centralized VC system.
  - *Distributed systems*, where multiple versions of the component repository exist at the same time. ***Git*** is a widely-used example of a distributed VC system.

Key features of version control systems

- Version and release identification
  - Unique identifiers
- Change history recording
  - Tagging components with keywords
- Support for independent development
- Support the development of several projects
- Storage management
  - Store a list of differences between one version and another

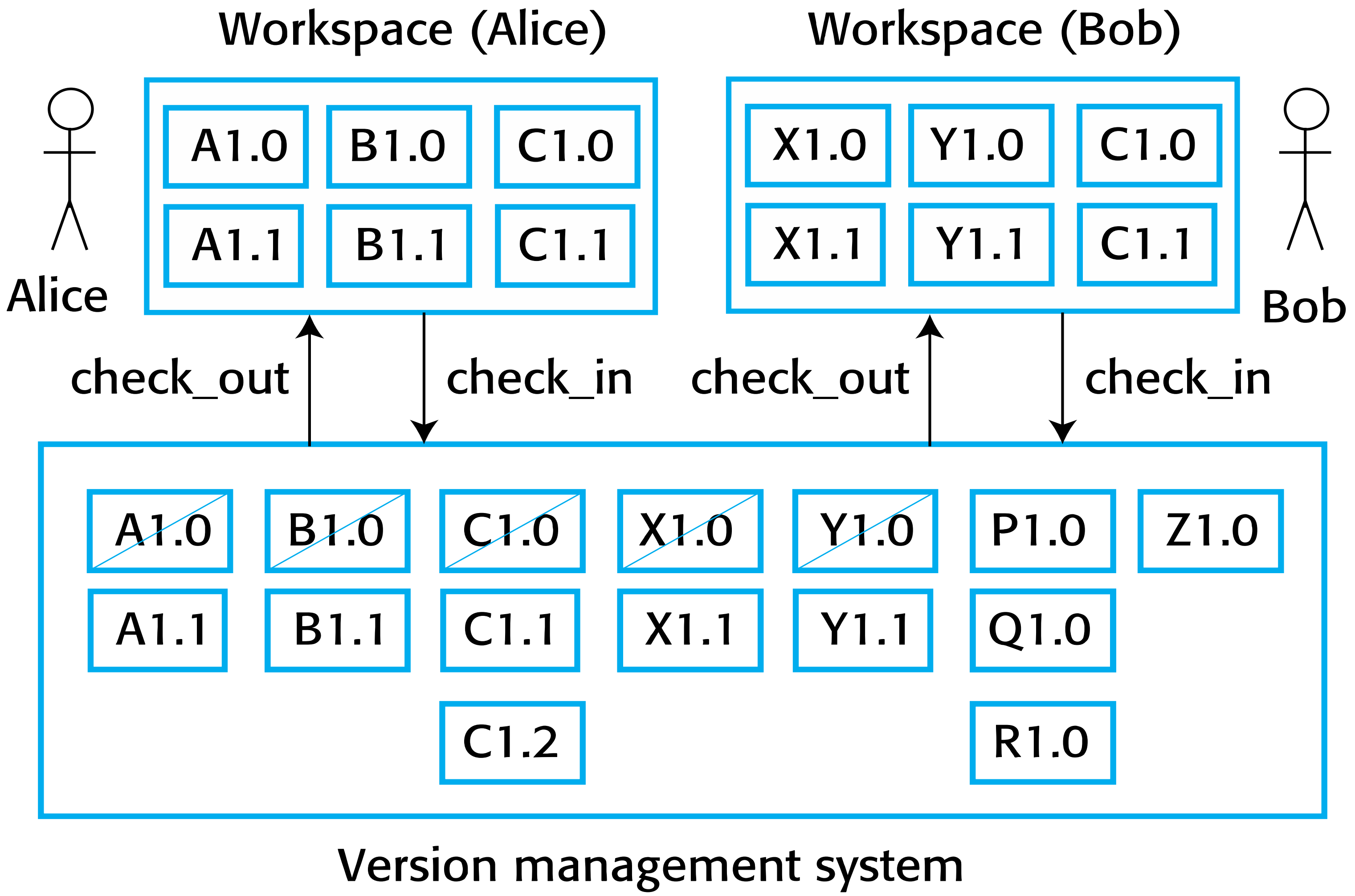
Public repository and private workspaces

- To support independent development without interference, version control systems use the concept of a *project repository* and a *private workspace*.
- The project repository maintains the *'master'* version of all components. It is used to create baselines for system building.
- When modifying components, developers copy (*check-out*) these from the repository into their workspace and work on these copies.
- When they have finished their changes, the changed components are returned (*check-in*) to the repository.

Centralized version control

- Developers *check out* components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- When their changes are complete, they *check-in* the components back to the repository.
- If several people are working on a component at the same time, each check it out from the repository.
- If a component has been *checked out*, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

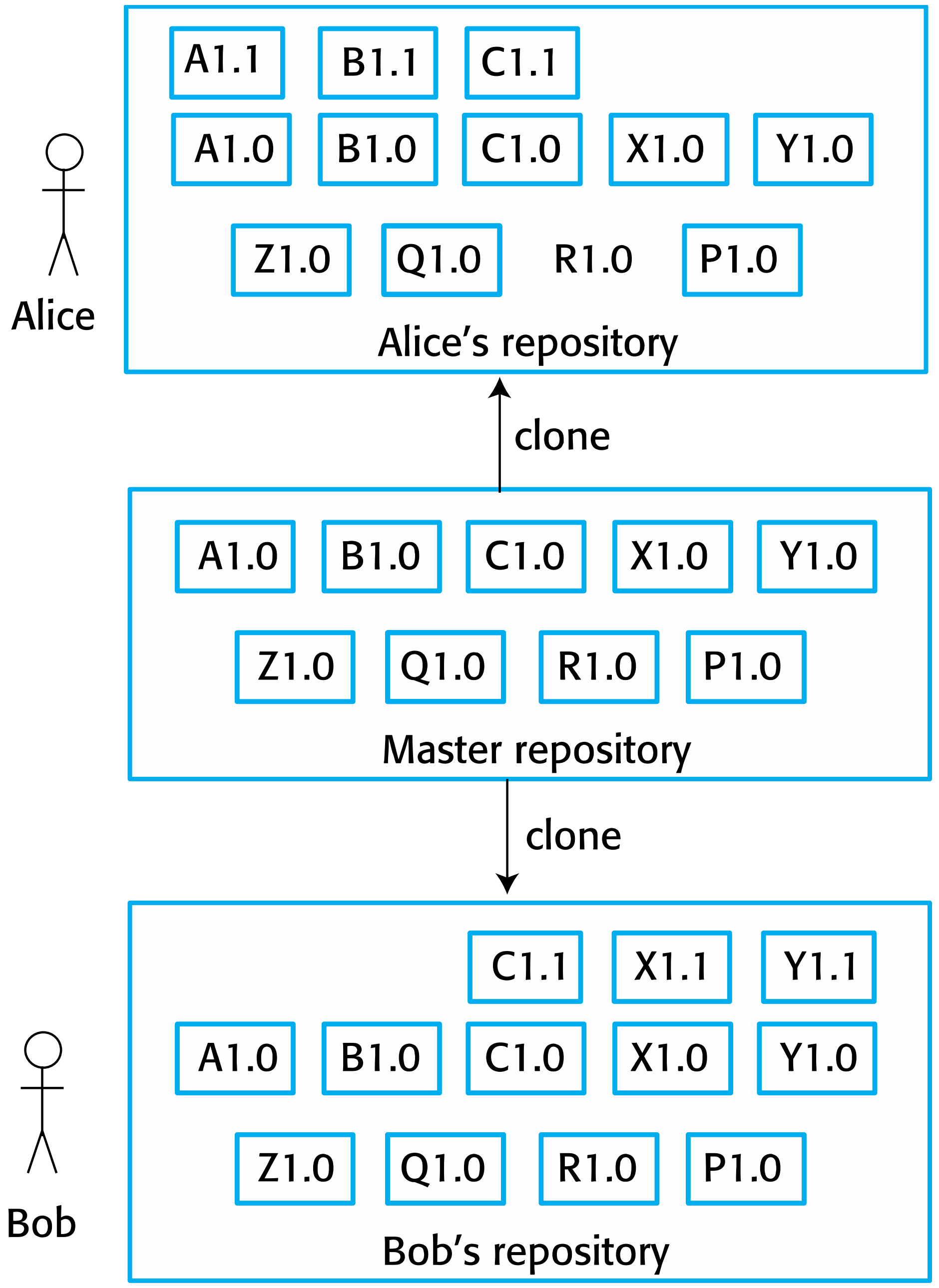
Repository Check-in/Check-out



Distributed version control

- A *'master'* repository is created on a server that maintains the code produced by the development team.
- Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
- Developers work on the files required and maintain the new versions on their private repository on their own computer.
- When changes are done, they *'commit'* these changes and update their private server repository. They may then *'push'* these changes to the project repository.

Repository cloning



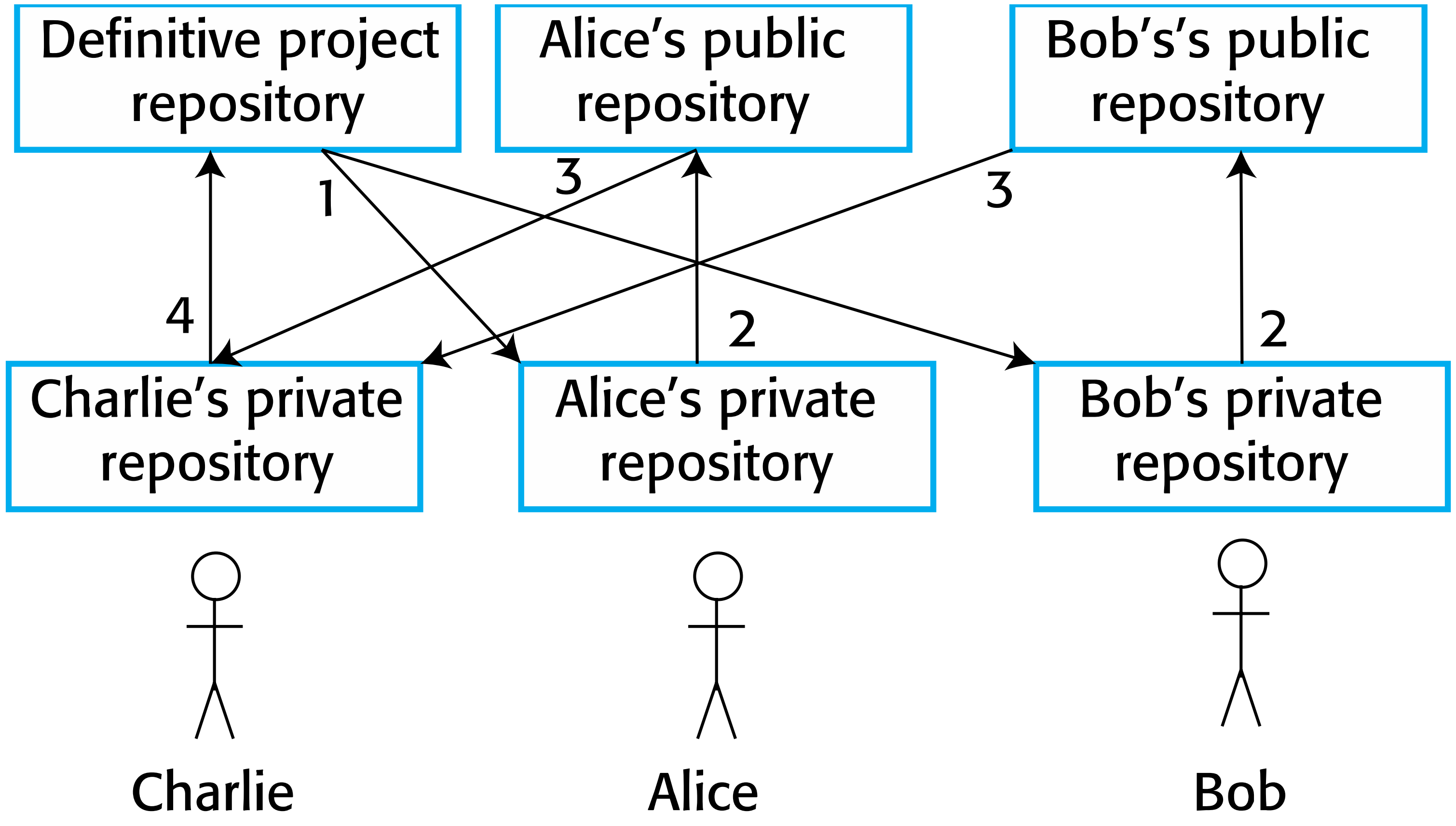
Benefits of distributed version control

- It provides a backup mechanism for the repository.
  - If the repository is corrupted, work can continue and the project repository can be restored from local copies.
- It allows for off-line working so that developers can commit changes if they do not have a network connection.
- Project support is the default way of working.
  - Developers can compile and test the entire system on their local machines and test the changes that they have made.

Open source development

- Distributed version control is essential for open source development.
  - Several people may be working simultaneously on the same system without any central coordination.
- As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
  - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

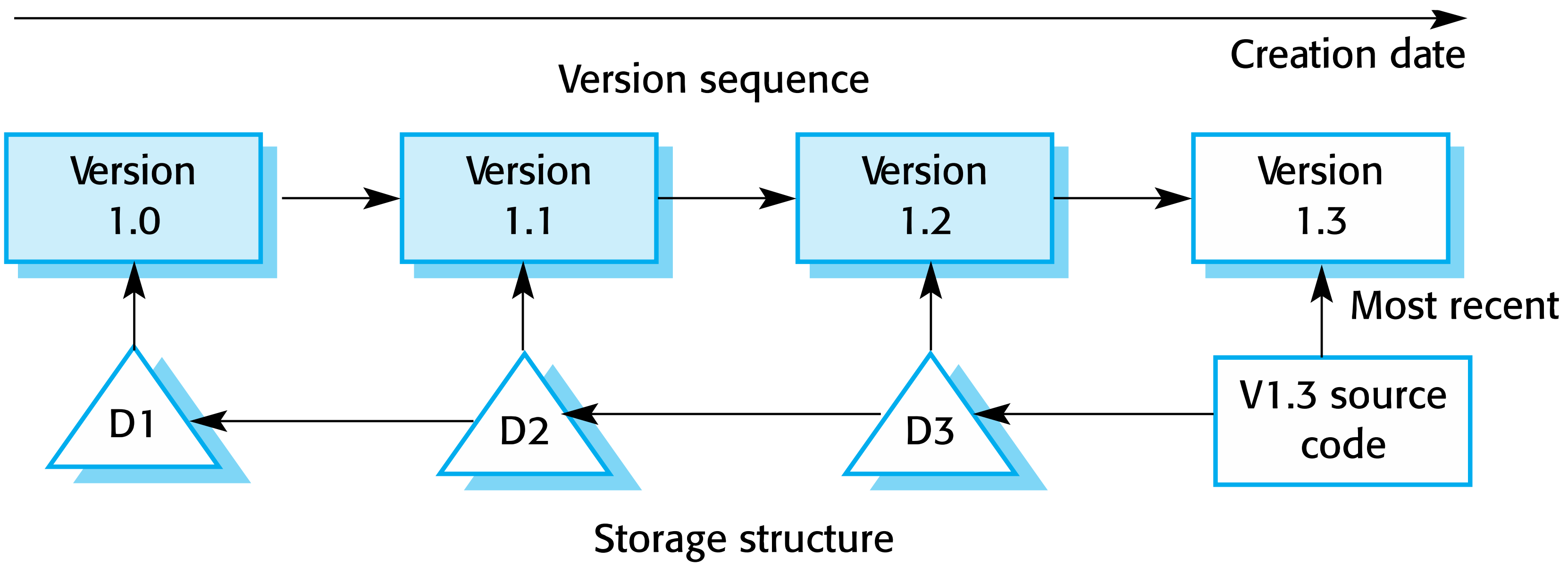
Open source development


Definitive project repository
Alice's public repository
Bob's's public repository
1
3
3
4
2
2
Charlie's private repository
Alice's private repository
Bob's private repository
Charlie
Alice
Bob

Storage management

- When version control systems were first developed, storage management was one of their most important functions.
- Disk space was expensive and it was important to minimize the disk space used by the different copies of components.
- Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another.
  - By applying these to a master version (usually the most recent version), a target version can be recreated.

Storage management using deltas



Storage management in Git

- As disk storage is now relatively cheap, *Git* uses an alternative, faster approach.
- *Git* does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information.
- It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- Git also uses the notion of *packfiles* where several smaller files are combined into an indexed single file.

10.2 System building

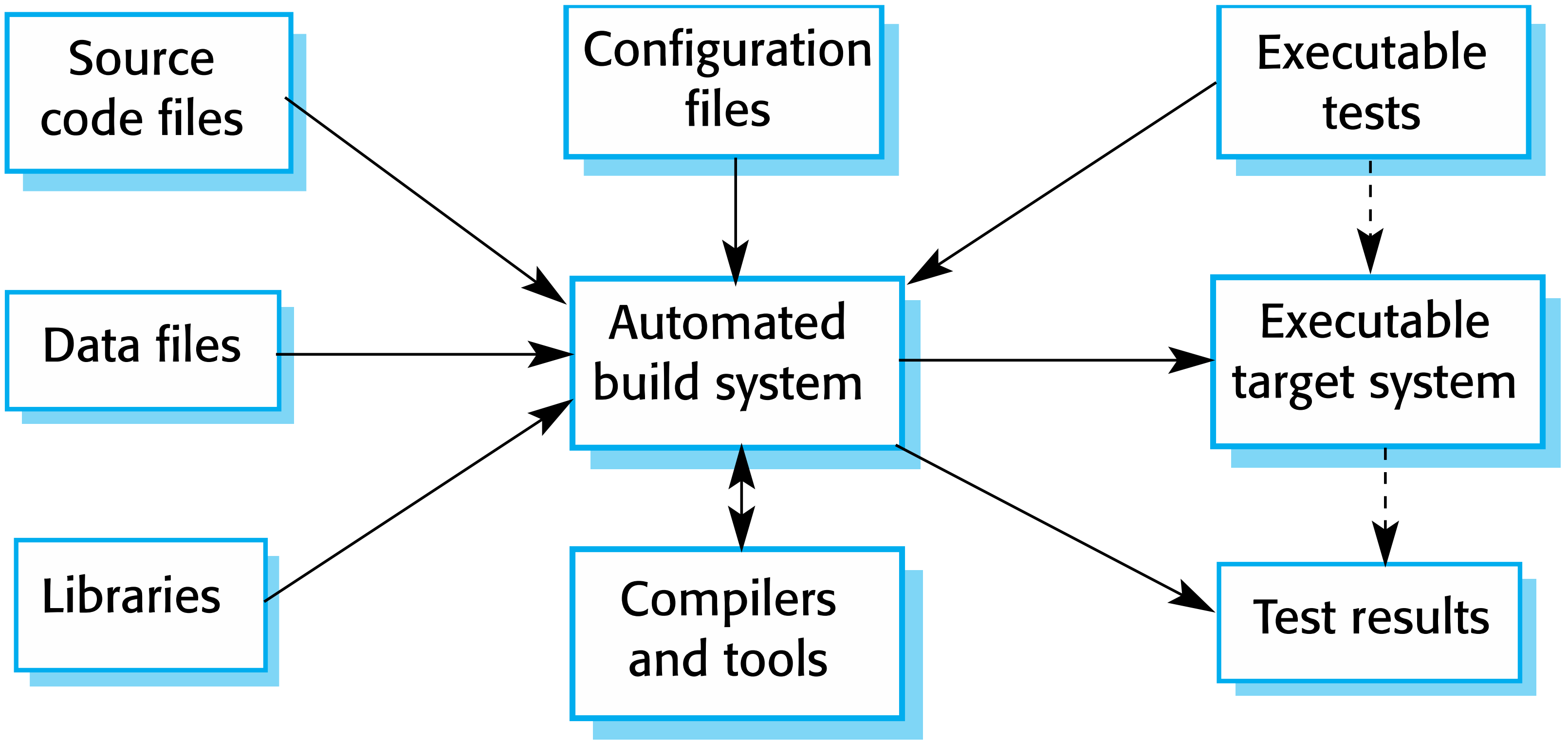
- *System building* is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- *System building* tools and *version management* tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- The configuration description used to identify a baseline is also used by the system building tool.

Build platforms

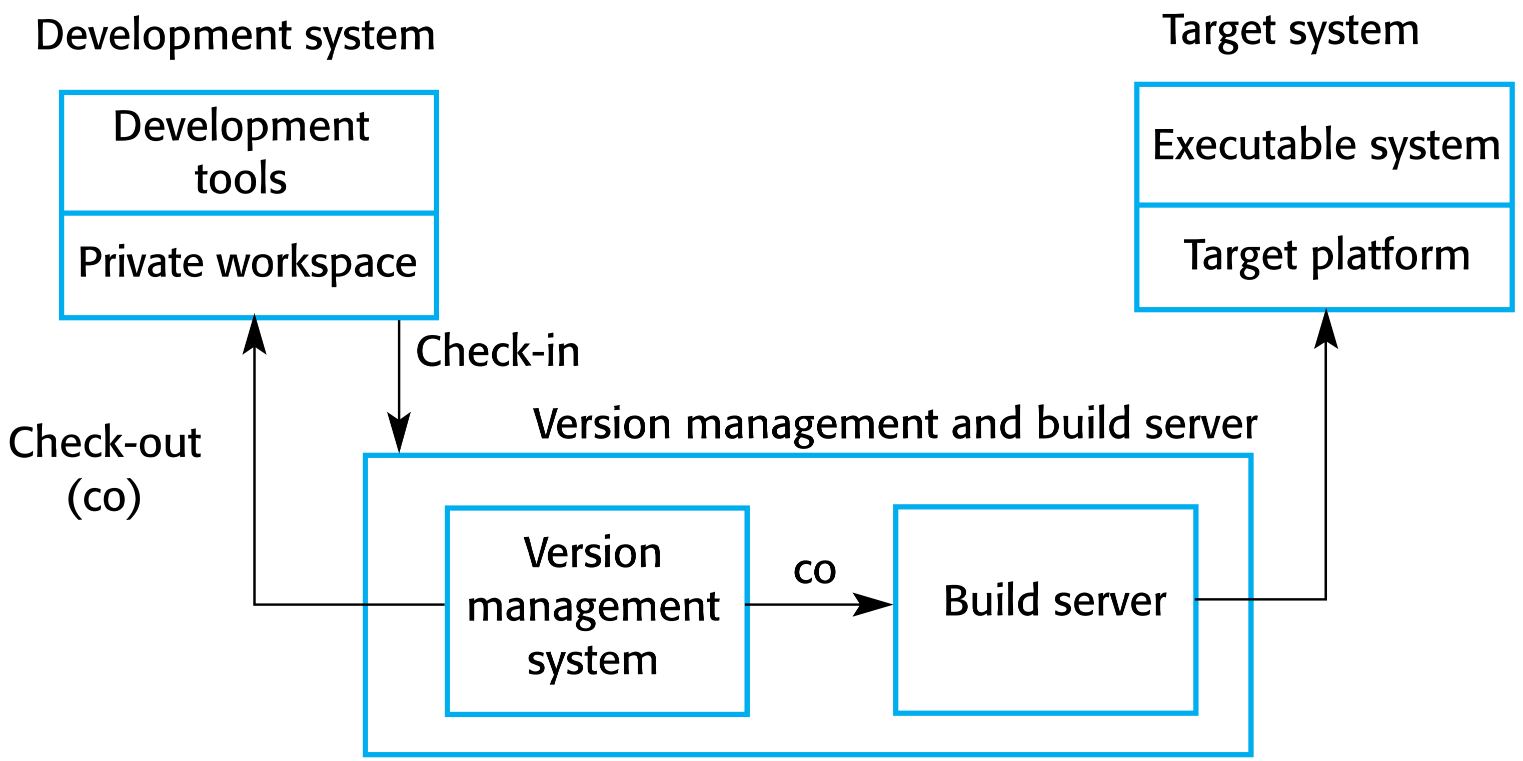
- The *development system*, which includes development tools such as compilers, source code editors, etc.
  - Developers check out code from the version management system into a private workspace before making changes to the system.
- The *build server*, which is used to build definitive, executable versions of the system.
  - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.

- The *target environment*, which is the platform on which the system executes.
  - For real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g. a cell phone)

System building



Development, build, and target platforms



Build system functionality

- Build script generation
- Version management system integration
- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

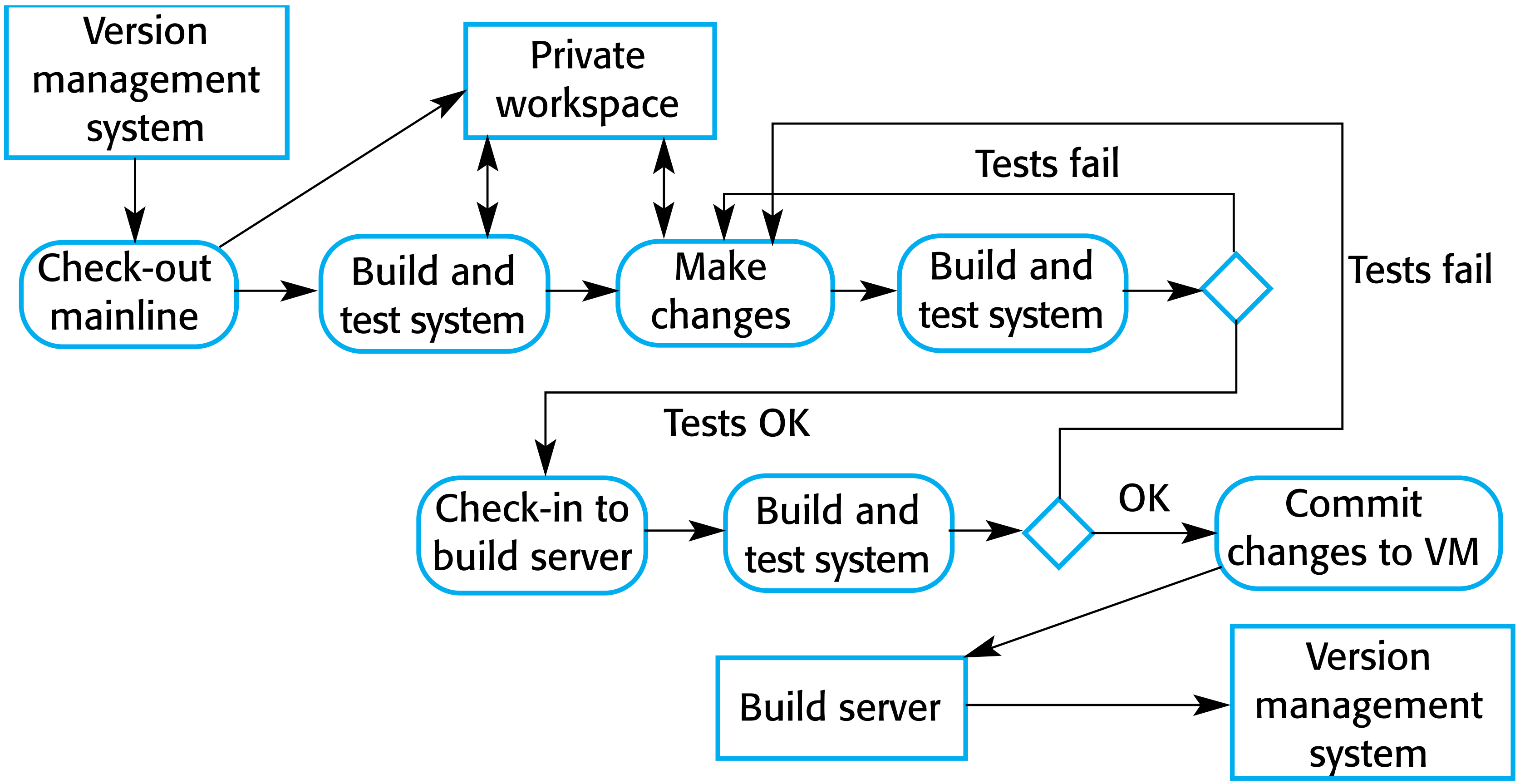
Agile building

- Check out the mainline system from the version management system into the developer’s private workspace.
- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- Make the changes to the system components.
- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

Agile building

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.
- Tool such as ***Jenkins*** support continuous integration.

Continuous integration



Pros and cons of continuous integration

- Pros: The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible. The most recent system in the mainline is the definitive working system.
- Cons:
  - If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved.
  - If the development platform is different from the target platform, it may not be possible to run system tests in the developer’s private workspace.

Daily building

- The development organization sets a delivery time (say 2 p.m.) for system components.
  - If developers have new versions of the components that they are writing, they must deliver them by that time.
  - A new version of the system is built from these components by compiling and linking them to form a complete system.
  - This system is then delivered to the testing team, which carries out a set of predefined system tests
  - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

Minimizing recompilation

- Tools to support system building are usually designed to minimize the amount of compilation that is required.
- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- A unique signature identifies each source and object code version and is changed when the source code is edited.
- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

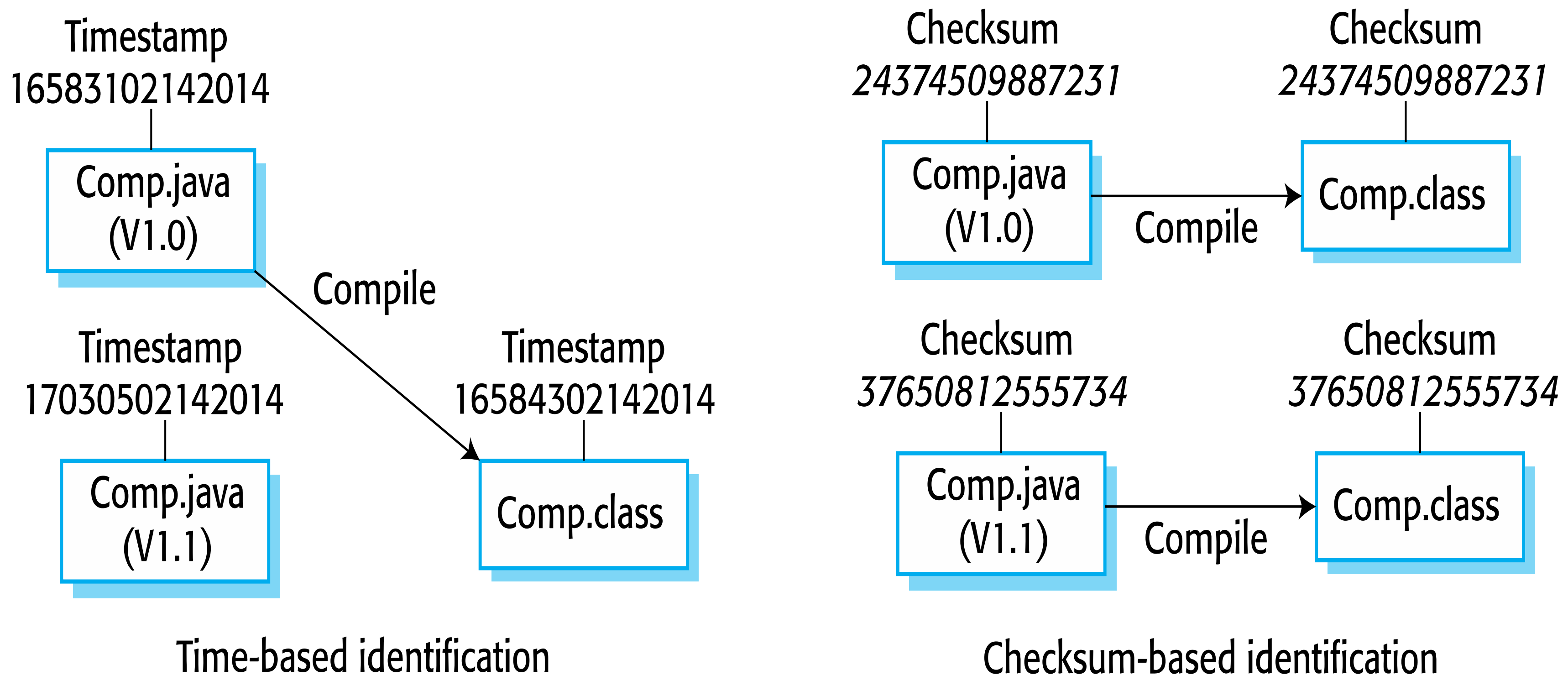
File identification

- Modification timestamps
  - The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- Source code checksums
  - The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

Timestamps vs checksums

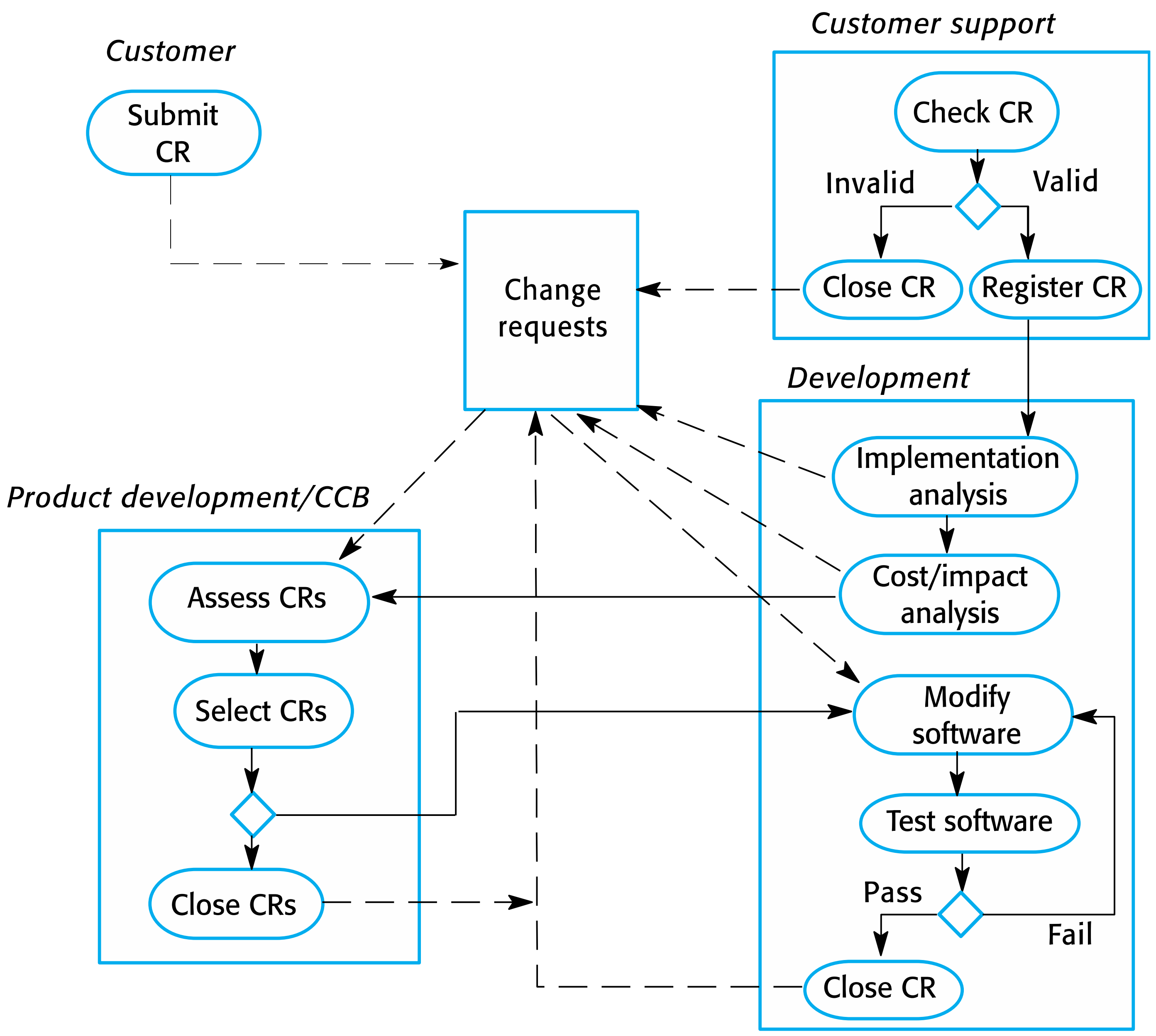
- Timestamps
  - Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.
- Checksums
  - When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

Linking source and object code



10.3 Change management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.



The change management process

A partially completed change request form (a)

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

A partially completed change request form (b)

Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments:

CCB = Change Control Board (or product development group)

Factors influence the decision to implement a change

- The consequences of not making the change
- The benefits of the change
- The number of users affected by the change
- The costs of making the change
- The product release cycle

Derivation history

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2012
//
// © St Andrews University 2012
//
// Modification history
// Version Modifier   Date          Change        Reason
// 1.0     J. Jones   11/11/2009    Add header    Submitted to CM
// 1.1     R. Looek   13/11/2012    New field     Change req. R07/02
```

Change management and agile methods

- In some agile methods, customers are directly involved in change management.
- The propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

10.4 Release management

- A system release is a version of a software system that is distributed to customers.
- For mass market software, it is usually possible to identify two types of release: *major releases* which deliver significant new functionality, and *minor releases*, which repair bugs and fix customer problems that have been reported.
- For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

Release components

- As well as the the executable code of the system, a release may also include:
  - configuration files defining how the release should be configured for particular installations;
  - data files, such as files of error messages, that are needed for successful system operation;
  - an installation program that is used to help install the system on target hardware;
  - electronic and paper documentation describing the system;
  - packaging and associated publicity that have been designed for that release.

Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

Release creation

- The executable code of the programs and all associated data files must be identified in the version control system.
- Configuration descriptions may have to be written for different hardware and operating systems.
- Update instructions may have to be written for customers who need to configure their own systems.
- Scripts for the installation program may have to be written.
- Web pages have to be created describing the release, with links to system documentation.
- When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

Release tracking

- In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
  - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

Release reproduction

- To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- You must keep copies of the source code files, corresponding executables and all data and configuration files.
- You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

Release planning

- As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- Release timing
  - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
  - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

Software as a Service (SaaS)

- Delivering software as a service (SaaS) reduces the problems of release management.
- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.
- Tool such as *Puppet* developed for pushing new software to servers.

Summary

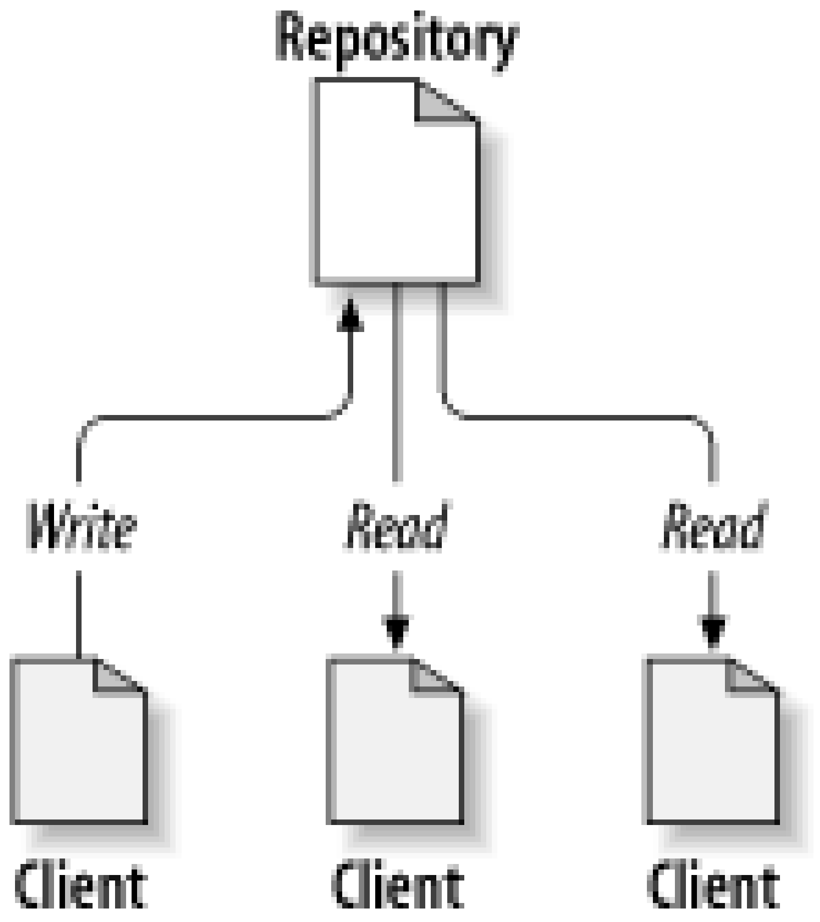
- *Configuration management* is the management of an evolving software system. When maintaining a system, a *CM* team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- The main configuration management processes are concerned with *version management, system building, change management, and release management.*
- *Version management* involves keeping track of the different versions of software components as changes are made to them.

- *System building* is the process of assembling system components into an executable program to run on a target computer system.
- *Change management* involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- System releases include executable code, data files, configuration files and documentation. *Release management* involves making decisions on system release dates, preparing all information for distribution and documenting each system release.

10.5 Subversion: *Basic Version-Control Concepts*

- *Subversion* is a centralized system for sharing information. At its core is a *repository*, which is a central store of data.
- The repository stores information in the form of a *filesystem tree* - a typical hierarchy of files and directories.
- Any number of *clients* connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

Figure 1. A Typical Client/Server System



- So why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository *is* a kind of file server, but it's not your usual breed.

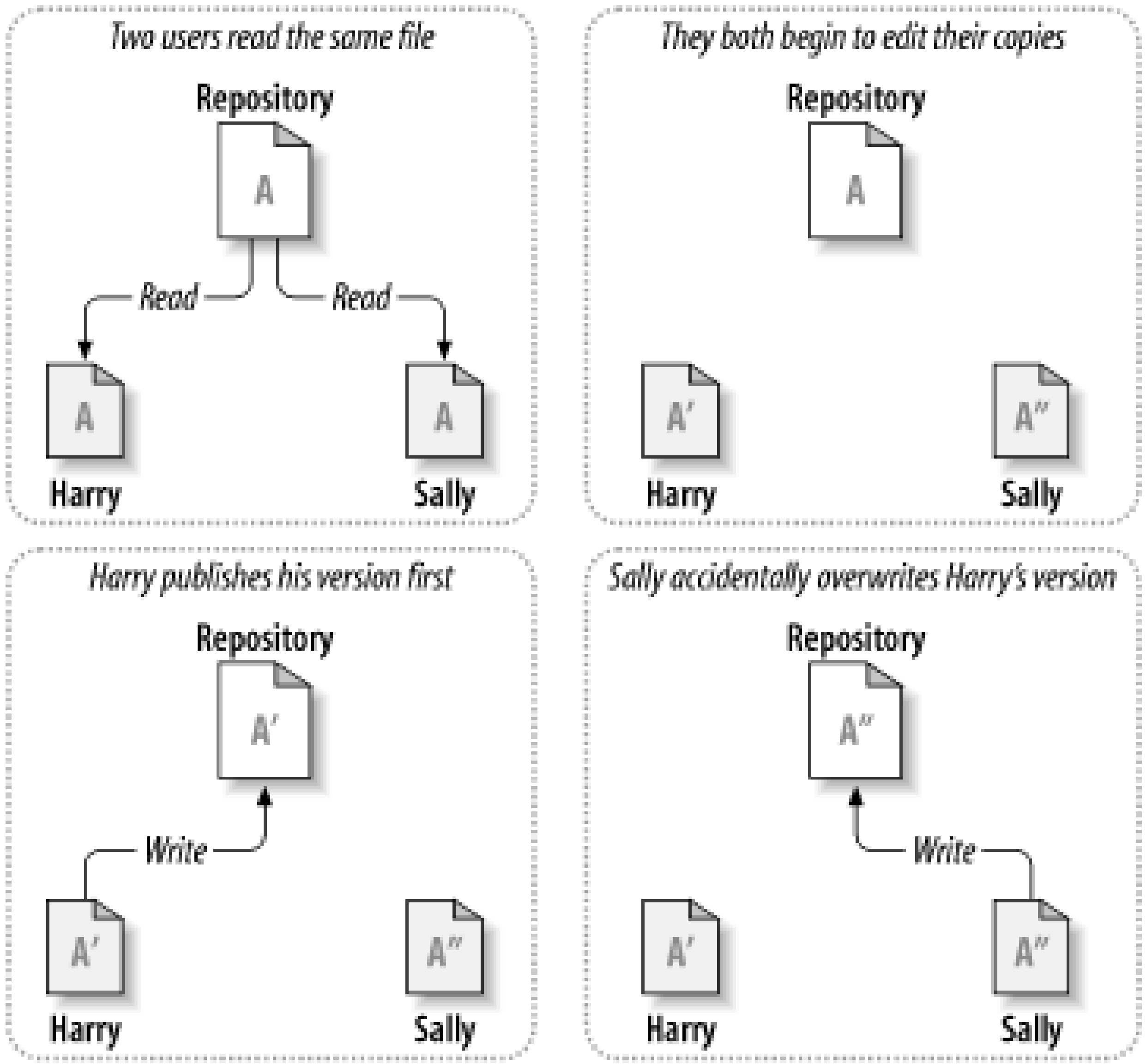
- What makes the Subversion repository special is that *it remembers every change* ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

- When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view *previous* states of the filesystem.
- For example, a client can ask historical questions like, “ what did this directory contain last Wednesday? ”, or “ who was the last person to change this file, and what changes did they make? ”
- These are the sorts of questions that are at the heart of any *version control system*: systems that are designed to record and track changes to data over time.

The Problem of File-Sharing

- Consider this scenario: suppose we have two co-workers, *Harry and Sally.* They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file.
- While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost - or at least missing from the latest version of the file - and probably by accident. This is definitely a situation we want to avoid!

Figure 2. The Problem to Avoid

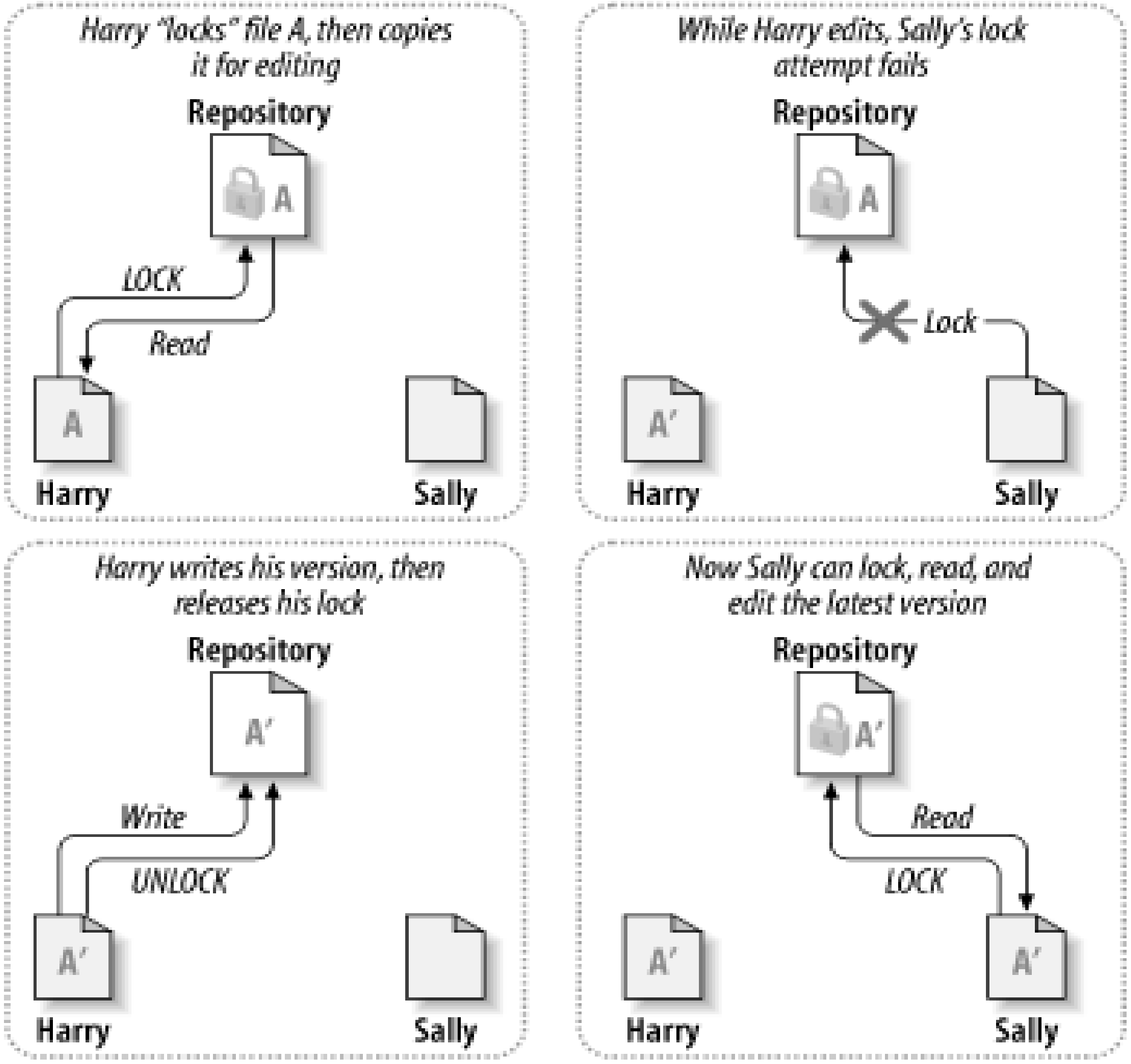


Two users read the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Harry publishes his version first
Repository
A′
Write
A′
A″
Harry
Sally
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
A″
Harry
Sally

The Lock-Modify-Unlock Solution

- Many version control systems use a *lock-modify-unlock* model to address this problem, which is a very simple solution.
- In such a system, the repository allows only one person to change a file at a time. First Harry must *lock* the file before he can begin making changes to it.
- After Harry unlocks the file, his turn is over, and now Sally can take her turn by locking and editing.

Figure 3. The Lock-Modify-Unlock Solution


Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A′
Harry
Sally
Harry writes his version, then releases his lock
Repository
A′
Write
UNLOCK
A′
Harry
Sally
Now Sally can lock, read, and edit the latest version
Repository
A′
Read
LOCK
A′
A′
Harry
Sally

The Copy-Modify-Merge Solution

- *Subversion*, *CVS (Concurrent Versions System)*, and other version control systems use a *copy-modify-merge* model as an alternative to locking.
- In this model, each user's client reads the repository and creates a personal working copy of the file or project.
- Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version.
- The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

- Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same *file A* within their copies.
- Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his *file A* is out-of-date.
- In other words, that *file A* in the repository has somehow changed since he last copied it. So Harry asks his client to merge any new changes from the repository into his working copy of *file A*.
- Chances are that Sally's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository.

Figure 4. The Copy-Modify-Merge Solution

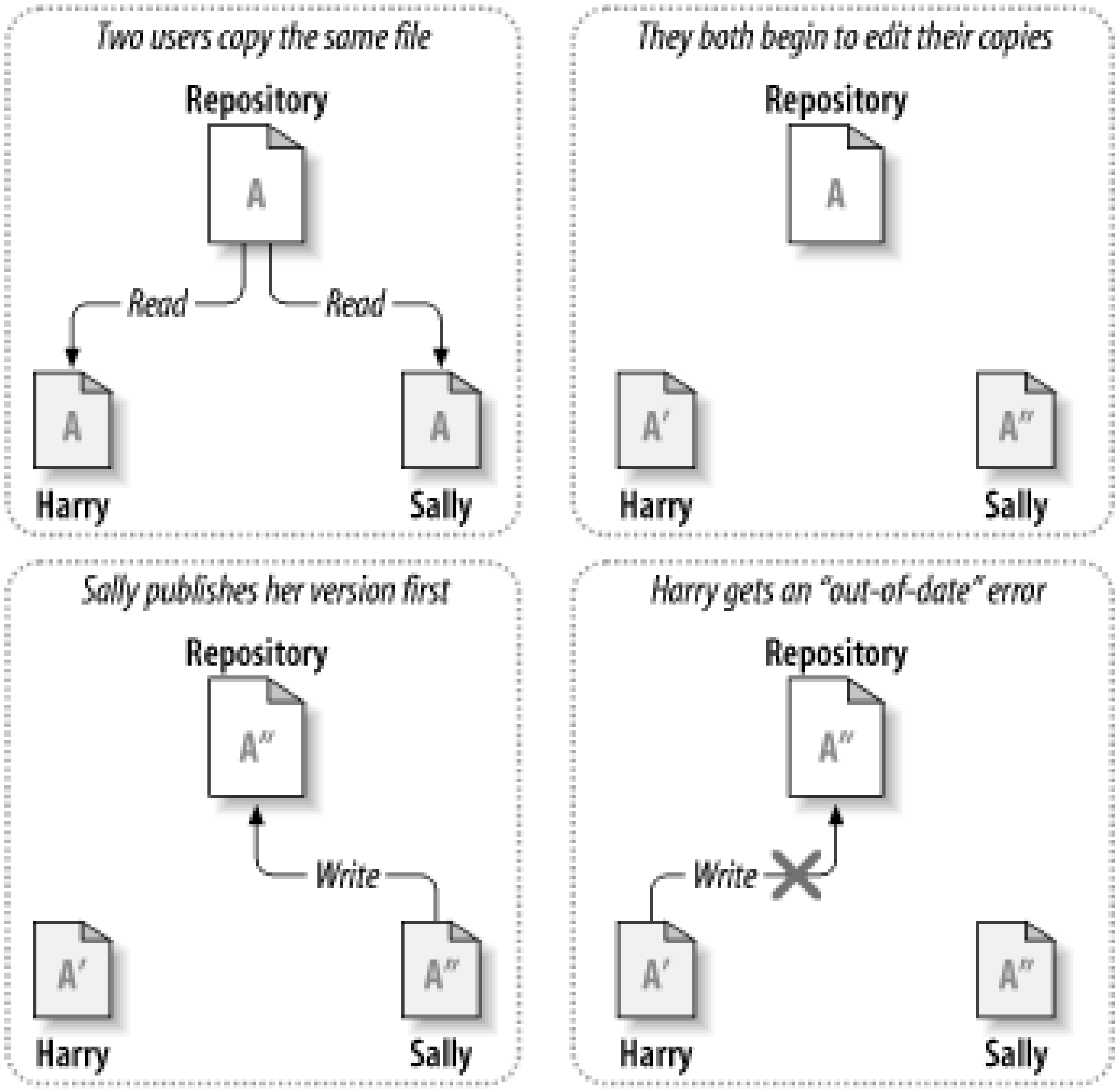
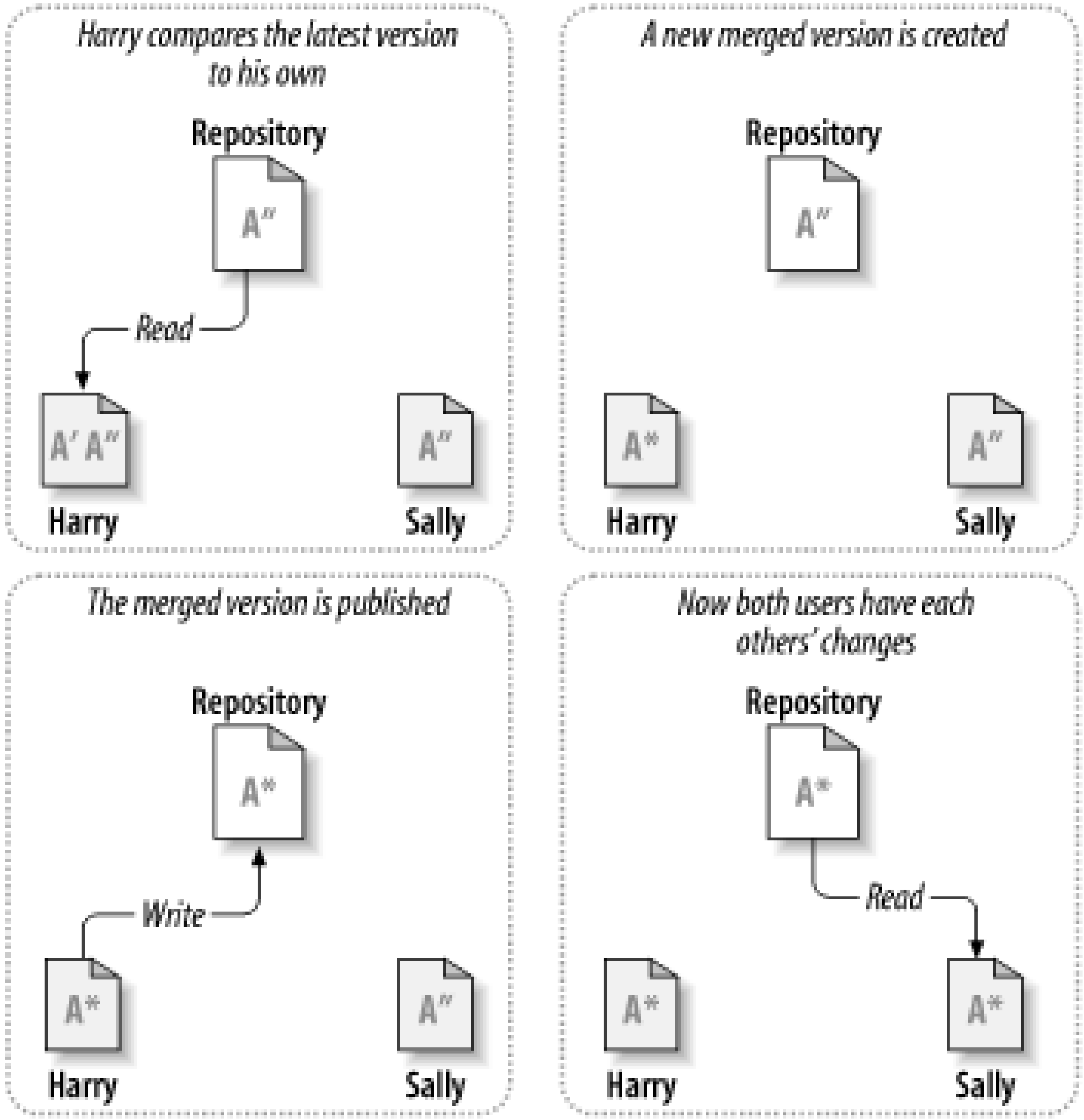

Two users copy the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Sally publishes her version first
Repository
A″
Write
A′
A″
Harry
Sally
Harry gets an "out-of-date" error
Repository
A″
Write
A′
A″
Harry
Sally

Figure 5. …Copy-Modify-Merge Continued

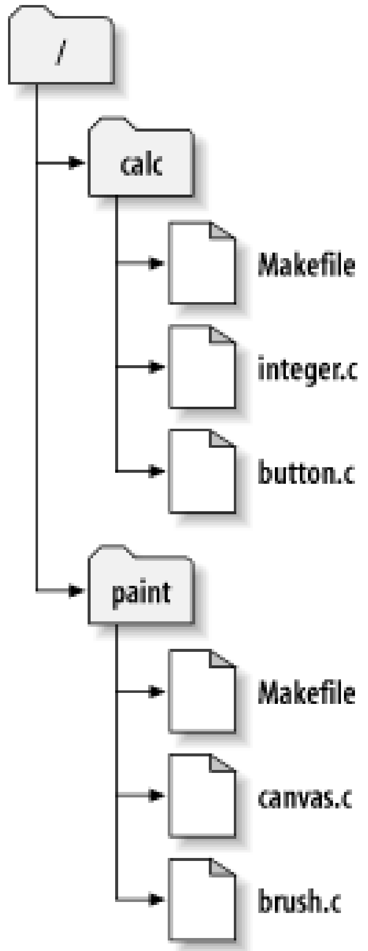

Harry compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Harry
Sally
A new merged version is created
Repository
A″
A*
A″
Harry
Sally
The merged version is published
Repository
A*
Write
A*
A″
Harry
Sally
Now both users have each others' changes
Repository
A*
Read
A*
A*
Harry
Sally

Working Copies

- A *Subversion* working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they're source code files, you can compile your program from them in the usual way.
- Your working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so.
- After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to *publish* your changes to the other people working with you on your project (by writing to the repository).
- If other people publish their own changes, Subversion provides you with commands to merge those changes into your working directory (by reading from the repository).

Figure 6. The Repository's Filesystem

- A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular subtree of the repository.
- For example, suppose you have a repository that contains two software projects.



- In other words, the repository's root directory has two subdirectories: *paint* and *calc*
- To get a working copy, you must *check out* some subtree of the repository.
- Suppose you make changes to *button.c*. Since the .svn directory remembers the file's modification date and original contents, *Subversion* can tell that you've changed the file.
- However, Subversion does not make your changes public until you explicitly tell it to. The act of publishing your changes is more commonly known as *committing* (or *checking in*) changes to the repository.
- To publish your changes to others, you can use Subversion's ***commit*** command.

- Now your changes to *button.c* have been committed to the repository; if another user checks out a working copy of /calc, they will see your changes in the latest version of the file.

- Suppose you have a collaborator, Sally, who checked out a working copy of /calc at the same time you did. When you commit your change to *button.c*, Sally's working copy is left unchanged; Subversion only modifies working copies at the user's request.

- To bring her project up to date, Sally can ask Subversion to *update* her working copy, by using the Subversion ***update*** command. This will incorporate your changes into her working copy, as well as any others that have been committed since she checked it out.

- Note that Sally didn't need to specify which files to update; Subversion uses the information in the .svn directory, and further information in the repository, to decide which files need to be brought up to date.

Repository URLs

- Subversion repositories can be accessed through many different methods - on local disk, or through various network protocols. A repository location, however, is always a URL. The URL schema indicates the access method:

Table: Repository Access URLs

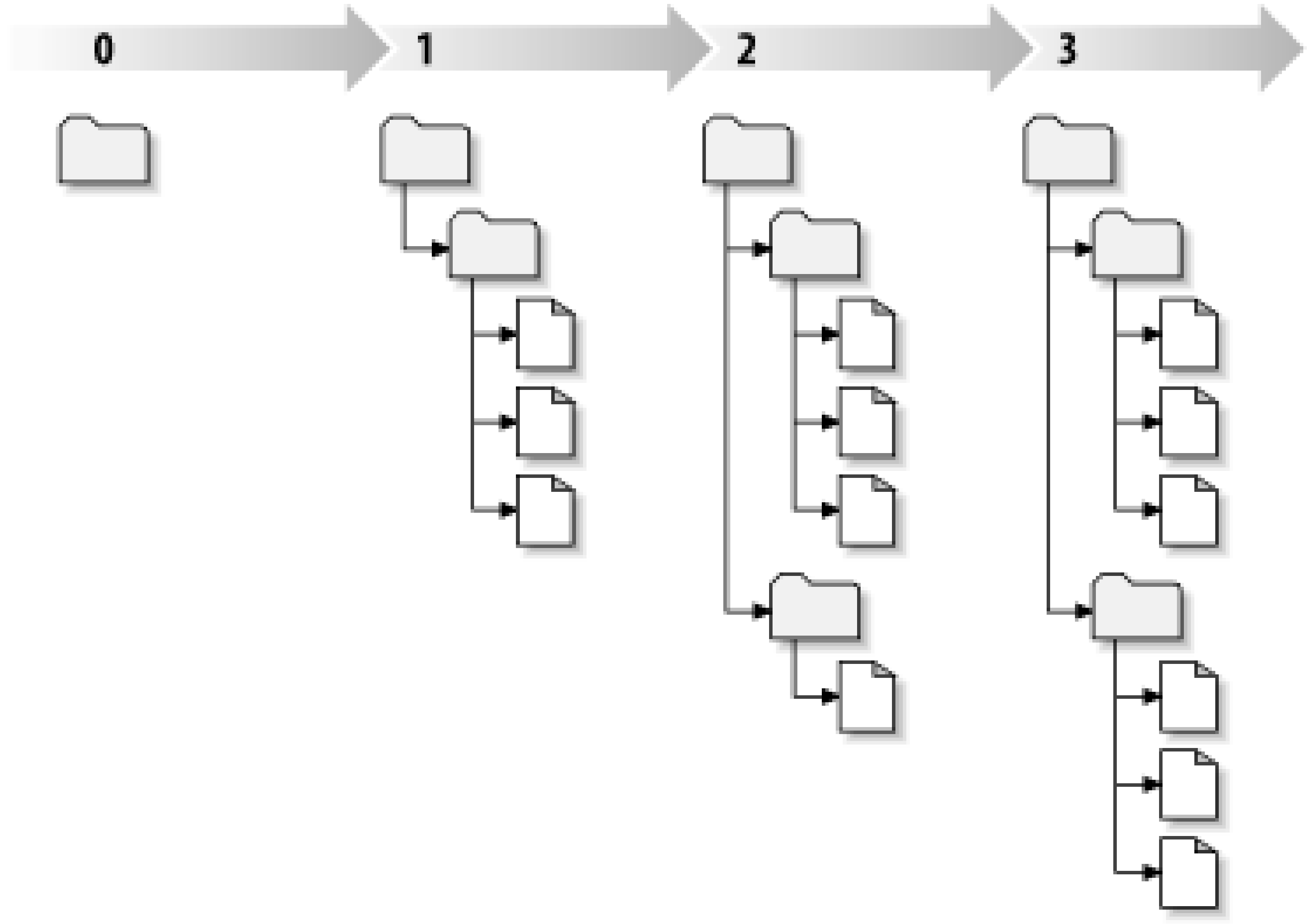
Schema	Access Method
file://	Direct repository access on local or network drive.
http://	Access via WebDAV protocol to Subversion-aware Apache server.
https://	Same as http://, but with SSL encryption.
svn://	Unauthenticated TCP/IP access via custom protocol to a svnserve server.
svn+ssh://	authenticated, encrypted TCP/IP access via custom protocol to a svnserve server.

Revisions

- A ***svn commit*** operation can publish changes to any number of files and directories as a single atomic transaction.
- In your working copy, you can change files' contents, create, delete, rename and copy files and directories, and then commit the complete set of changes as a unit.
- In the repository, each commit is treated as an atomic transaction: either all the commits changes take place, or none of them take place. Subversion retains this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

- Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number of the previous revision. The initial revision of a freshly created repository is numbered zero, and consists of nothing but an empty root directory.
- A nice way to visualize the repository is as a series of trees. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a “snapshot” of the way the repository looked after each commit.

Figure 7. The Repository



- It's important to note that working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working copy from a repository whose most recent revision is 4:

```
    calc/Makefile:4   integer.c:4        button.c:4
```

- At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to *button.c*, and commit that change. Assuming no other commits have taken place, your commit will create revision 5 of the repository, and your working copy will now look like this:

```
    calc/Makefile:4   integer.c:4        button.c:5
```

- Suppose that, at this point, Sally commits a change to *integer.c*, creating revision 6. If you use *svn update* to bring your working copy up to date, then it will look like this:

```
calc/Makefile:6   integer.c:6       button.c:6
```

- Sally's changes to *integer.c* will appear in your working copy, and your change will still be present in *button.c*.
- In this example, the text of Makefile is identical in revisions 4, 5, and 6, but Subversion will mark your working copy of Makefile with revision 6 to indicate that it is still current.
- So, after you do a clean update at the top of your working copy, it will generally correspond to exactly one revision in the repository.

How Working Copies Track the Repository

- For each file in a working directory, Subversion records two essential pieces of information in the .svn/ administrative area:
  - what revision your working file is based on (this is called the file's working revision), and
  - a timestamp recording when the local copy was last updated by the repository.
- Given this information, by talking to the repository, Subversion can tell which of the following four states a working file is in:

Unchanged, and current

- The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. A commit of the file will do nothing, and an update of the file will do nothing.

Locally changed, and current

- The file has been changed in the working directory, and no changes to that file have been committed to the repository since its base revision. There are local changes that have not been committed to the repository, thus a commit of the file will succeed in publishing your changes, and an update of the file will do nothing.

Unchanged, and out-of-date

- The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the public revision. A commit of the file will do nothing, and an update of the file will fold the latest changes into your working copy.

Locally changed, and out-of-date

- The file has been changed both in the working directory, and in the repository. A commit of the file will fail with an out-of-date error. The file should be updated first; an update command will attempt to merge the public changes with the local changes. If Subversion can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

Summary - Subversion

- We've covered a number of fundamental Subversion concepts
- We've introduced the notions of the central repository, the client working copy, and the array of repository revision trees.
- We've seen some simple examples of how two collaborators can use *Subversion* to publish and receive changes from one another, using the '*copy-modify-merge'* model.
- We've talked a bit about the way Subversion tracks and manages information in a working copy.

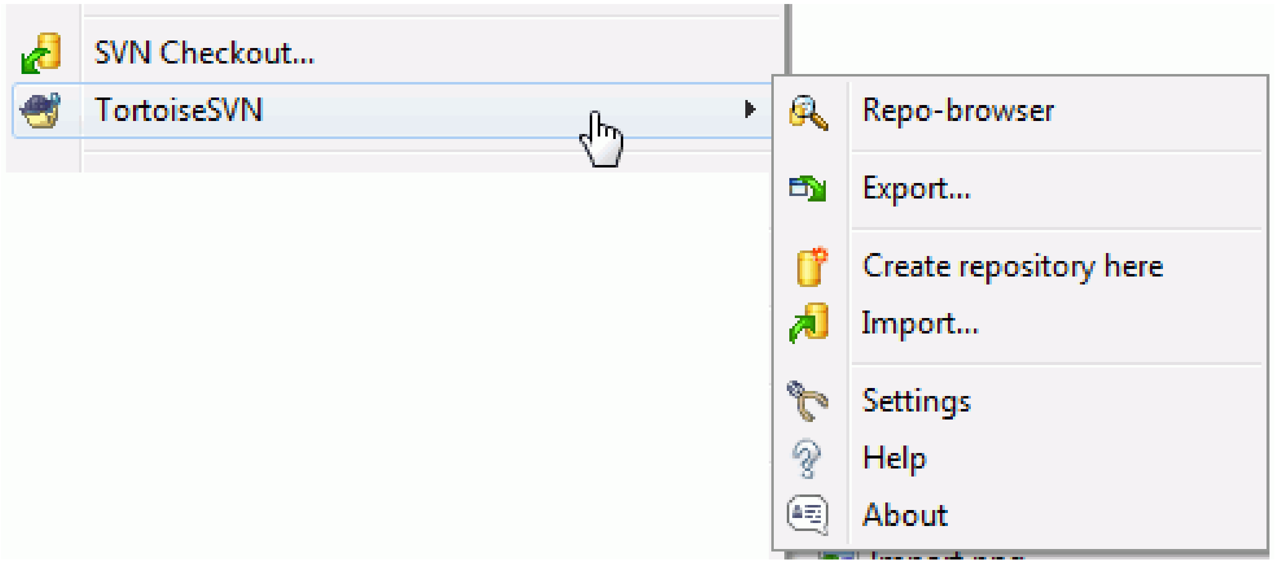
10.6 Tutorial: Subversion

- https://tortoisesvn.net/docs/release/TortoiseSVN_en/index.html

What is TortoiseSVN?

- TortoiseSVN is a free open-source Windows client for the *Apache™ Subversion®* version control system. That is, TortoiseSVN manages files and directories over time.
- Files are stored in a central *repository*. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories.
- This allows you to recover older versions of your files and examine the history of how and when your data changed, and who changed it. This is why many people think of Subversion and version control systems in general as a sort of “time machine”.

- You also need to know where to find TortoiseSVN because there is not much to see from the Start Menu. This is because TortoiseSVN is a Shell extension, so first of all, start Windows Explorer. Right click on a folder in Explorer and you should see some new entries in the context menu like this:

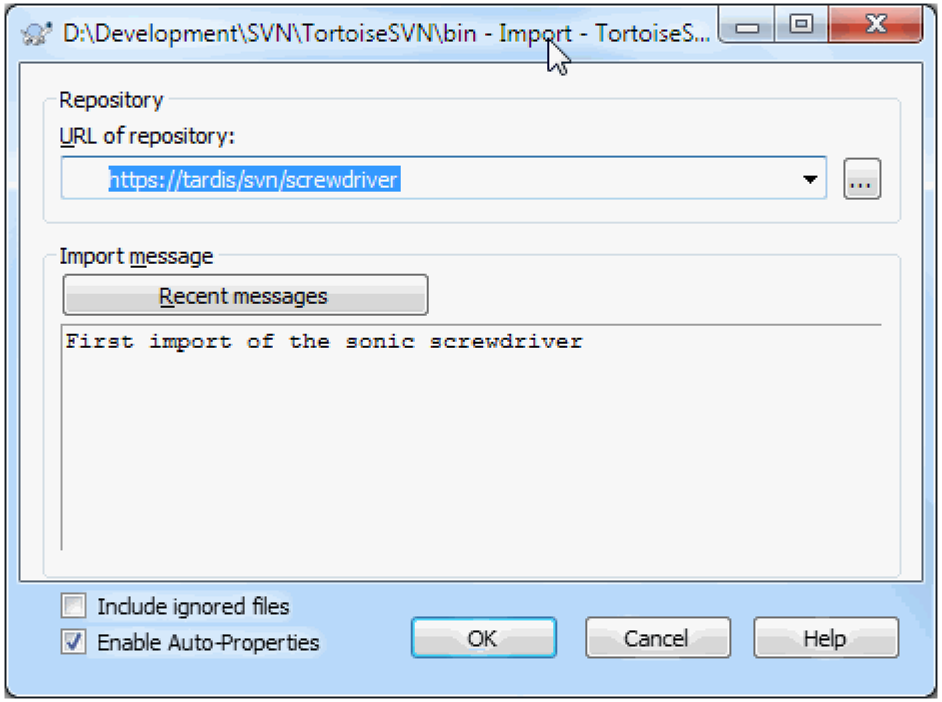


Creating a Repository

- For a real project you will have a repository set up somewhere safe and a *Subversion* server to control it. For the purposes of this tutorial we are going to use Subversion's local repository feature which allows direct access to a repository created on your hard drive without needing a server at all.
- First create a new empty directory on your PC. It can go anywhere, but in this tutorial we are going to call it *C:\svn_repos*.
- Now right click on the new folder and from the context menu choose TortoiseSVN → Create Repository here.... The repository is then created inside the folder, ready for you to use. We will also create the default internal folder structure by clicking the <u>Create folder structure</u> button.

Importing a Project

- Now we have a repository, but it is completely empty at the moment. Let's assume we have a set of files in *C:\Projects\Widget1* that we would like to add. Navigate to the Widget1 folder in Explorer and right click on it. Now select TortoiseSVN → Import... which brings up a dialog



D:\Development\SVN\TortoiseSVN\bin - Import - TortoiseS...

Repository

URL of repository: https://tardis/svn/screwdriver

Import message

Recent messages

```
First import of the sonic screwdriver
```

☐ Include ignored files

☑ Enable Auto-Properties

OK Cancel Help

- A Subversion repository is referred to by URL, which allows us to specify a repository anywhere on the Internet. In this case we need to point to our own local repository which has a URL of *file:///c:/svn_repos/trunk*, and to which we add our own project name *Widget1*. Note that there are 3 slashes after file: and that forward slashes are used throughout.
- The other important feature of this dialog is the Import Message box which allows you to enter a message describing what you are doing. When you come to look through your project history, these commit messages are a valuable guide to what changes have been made and why. In this case we can say something simple like "*Import the Widget1 project*".
- Click on OK and the folder is added to your repository.

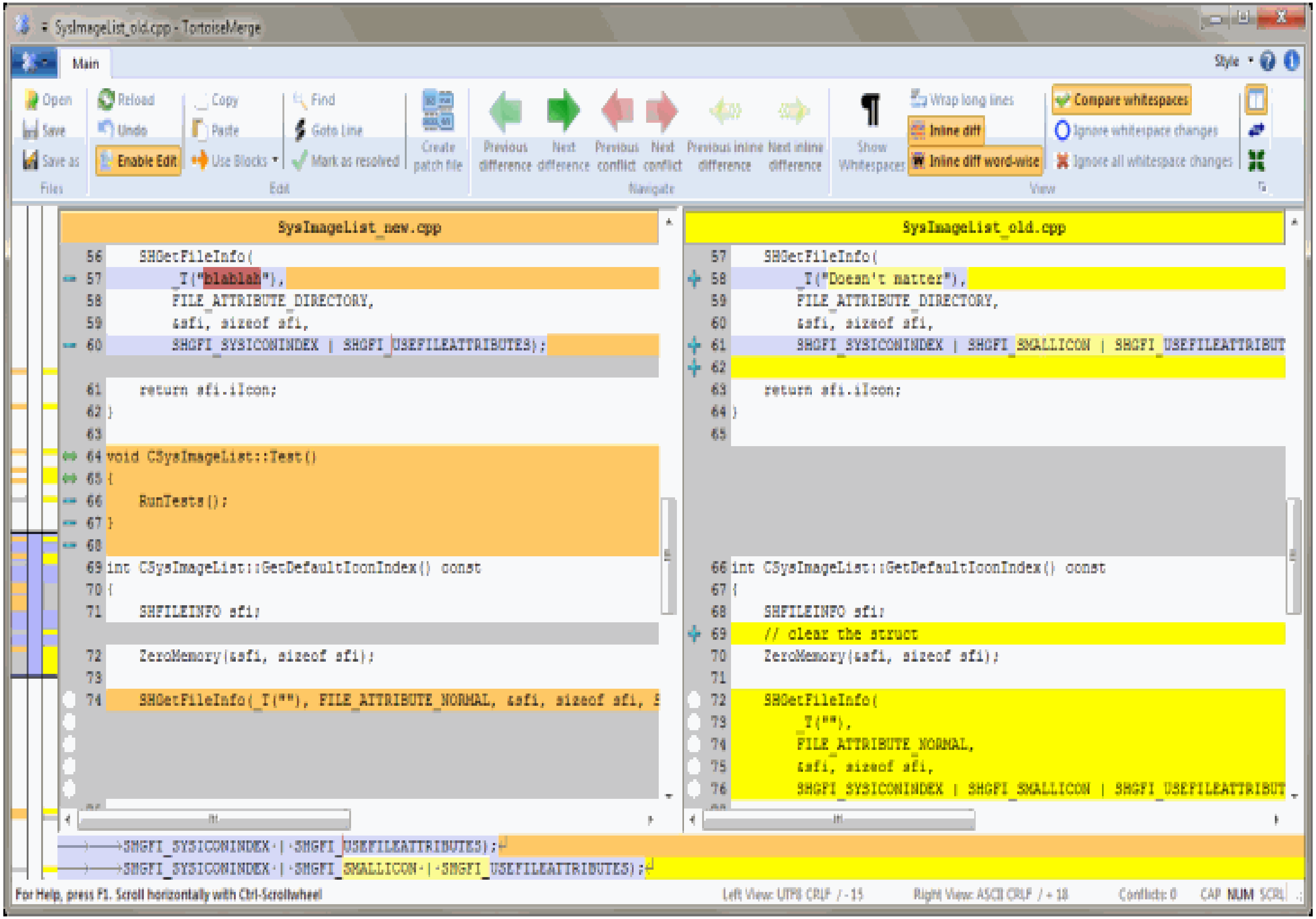
Checking out a Working Copy

- Now that we have a project in our repository, we need to create a working copy to use for day-to-day work. Note that the act of importing a folder does not automatically turn that folder into a working copy. The Subversion term for creating a fresh working copy is *Checkout*.
- We are going to checkout the *Widget1* folder of our repository into a development folder on the PC called *C:\Projects\Widget1-Dev*.
- Create that folder, then right click on it and select TortoiseSVN → Checkout.... Then enter the URL to checkout, in this case *file:///c:/svn_repos/trunk/Widget1* and click on OK. Our development folder is then populated with files from the repository.

Making Changes

- Time to get to work. In the *Widget1-Dev* folder we start editing files - let's say we make changes to *Widget1.c* and *ReadMe.txt*. Notice that the icon overlays on these files have now changed to red, indicating that changes have been made locally.

- But what are the changes? Right click on one of the changed files and select *TortoiseSVN → Diff.* TortoiseSVN's file compare tool starts, showing you exactly which lines have changed.



SysImageList_new.cpp

```
    SHGetFileInfo(
        _T("blablah"),
        FILE_ATTRIBUTE_DIRECTORY,
        &sfi, sizeof sfi,
        SHGFI_SYSICONINDEX | SHGFI_USEFILEATTRIBUTES);

    return sfi.iIcon;
}

void CSysImageList::Test()
{
    RunTests();
}

int CSysImageList::GetDefaultIconIndex() const
{
    SHFILEINFO sfi;

    ZeroMemory(&sfi, sizeof sfi);

    SHGetFileInfo(_T(""), FILE_ATTRIBUTE_NORMAL, &sfi, sizeof sfi, S
```

SysImageList_old.cpp

```
    SHGetFileInfo(
        _T("Doesn't matter"),
        FILE_ATTRIBUTE_DIRECTORY,
        &sfi, sizeof sfi,
        SHGFI_SYSICONINDEX | SHGFI_SMALLICON | SHGFI_USEFILEATTRIBUT

    return sfi.iIcon;
}

int CSysImageList::GetDefaultIconIndex() const
{
    SHFILEINFO sfi;
    // clear the struct
    ZeroMemory(&sfi, sizeof sfi);

    SHGetFileInfo(
        _T(""),
        FILE_ATTRIBUTE_NORMAL,
        &sfi, sizeof sfi,
        SHGFI_SYSICONINDEX | SHGFI_SMALLICON | SHGFI_USEFILEATTRIBUT
```

```
→→SHGFI_SYSICONINDEX·|·SHGFI_USEFILEATTRIBUTES);↵
→→SHGFI_SYSICONINDEX·|·SHGFI_SMALLICON·|·SHGFI_USEFILEATTRIBUTES);↵
```

For Help, press F1. Scroll horizontally with Ctrl-Scrollwheel   Left View: UTF8 CRLF / - 15   Right View: ASCII CRLF / + 18   Conflicts: 0   CAP NUM SCRL

- OK, so we are happy with the changes, let's update the repository. This action is referred to as a *Commit* of the changes.
- Right click on the *Widget1-Dev* folder and select *TortoiseSVN → Commit*. The commit dialog lists the changed files, each with a checkbox.
- You might want to choose only a subset of those files, but in this case we are going to commit the changes to both files.
- Enter up a message to describe what the change is all about and click on OK. The progress dialog shows the files being uploaded to the repository and you're done.

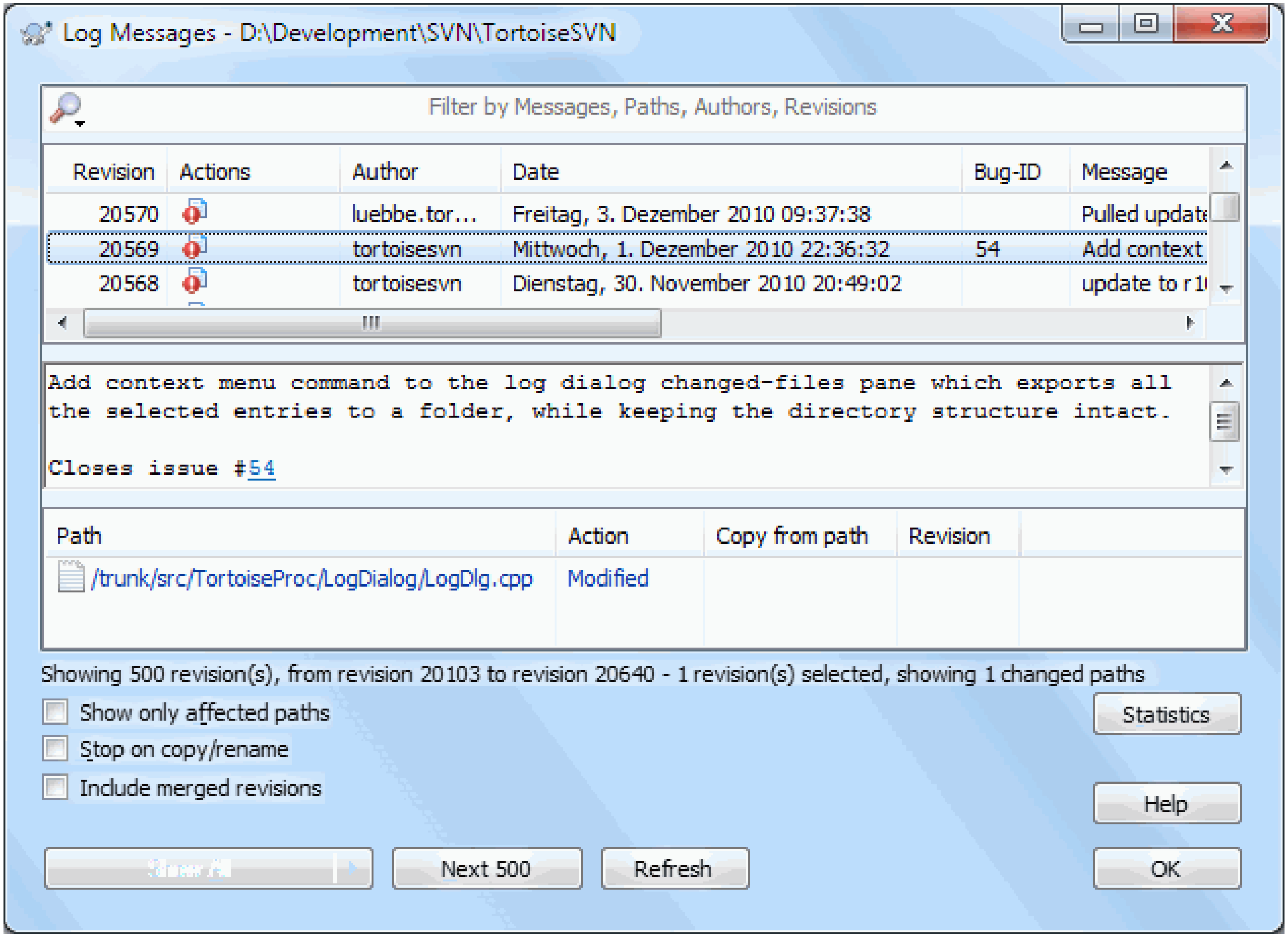
Adding More Files

- As the project develops you will need to add new files - let's say you add some new features in *Extras.c* and add a reference in the existing *Makefile*.
- Right click on the folder and *TortoiseSVN → Add*. The Add dialog now shows you all unversioned files and you can select which ones you want to add. Another way of adding files would be to right click on the file itself and select *TortoiseSVN → Add*.
- Now when you go to commit the folder, the new file shows up as *Added* and the existing file as *Modified*. Note that you can double click on the modified file to check exactly what changes were made.

Viewing the Project History

- One of the most useful features of TortoiseSVN is the *Log dialog*. This shows you a list of all the commits you made to a file or folder, and shows those detailed commit messages that you entered (you did enter a commit message as suggested? If not, now you see why this is important).

- The top pane shows a list of revisions committed along with the start of the commit message. If you select one of these revisions, the middle pane will show the full log message for that revision and the bottom pane will show a list of changed files and folders.



Log Messages - D:\Development\SVN\TortoiseSVN

Filter by Messages, Paths, Authors, Revisions

Revision	Actions	Author	Date	Bug-ID	Message
20570		luebbe.tor...	Freitag, 3. Dezember 2010 09:37:38		Pulled updat
20569		tortoisesvn	Mittwoch, 1. Dezember 2010 22:36:32	54	Add context
20568		tortoisesvn	Dienstag, 30. November 2010 20:49:02		update to r1

```
Add context menu command to the log dialog changed-files pane which exports all
the selected entries to a folder, while keeping the directory structure intact.

Closes issue #54
```

Path	Action	Copy from path	Revision
/trunk/src/TortoiseProc/LogDialog/LogDlg.cpp	Modified		

Showing 500 revision(s), from revision 20103 to revision 20640 - 1 revision(s) selected, showing 1 changed paths

Undoing Changes

- If you want to get rid of changes that you have not yet committed and reset your file to the way it was before you started editing, *TortoiseSVN → Revert* is your friend. This discards your changes (to the Recycle bin, just in case) and reverts to the committed version you started with.
- If you want to get rid of just some of the changes, you can use *Tortoise → Merge* to view the differences and selectively revert changed lines.
- If you want to undo the effects of a particular revision, start with the *Log dialog* and find the offending revision. Select *Context Menu → Revert* changes from this revision and those changes will be undone.

Typical working cycle

You will need to learn to work using the following regular pattern:

1) Update your working copy.

2) Make your changes.

3) Review your changes

(use the status and diff commands)

4) Happy? No? Fix your mistakes (used diff or revert).

5) Commit, if there are conflicts, resolve them and then commit again.

6) Repeat at regular intervals