Software Engineering – Lecture 9
Configuration management

Adapted after ©**Ian Sommerville**
Software Engineering, 2010, Configuration management ;
Engineering Software Products, 2019, chapter **10**

Topics covered

- **Software configuration management**
- Version management
- System building
- Change management
- Release management

Software Configuration Management (SCM)

Rationale: New versions of software systems are created as they change:

- For different machines/OS;
- Offering different functionality;
- Tailored for particular user requirements.

$\Downarrow$

Software changes frequently $\Rightarrow$ software systems can be thought of as a set of *versions*, each of which has to be maintained and managed.

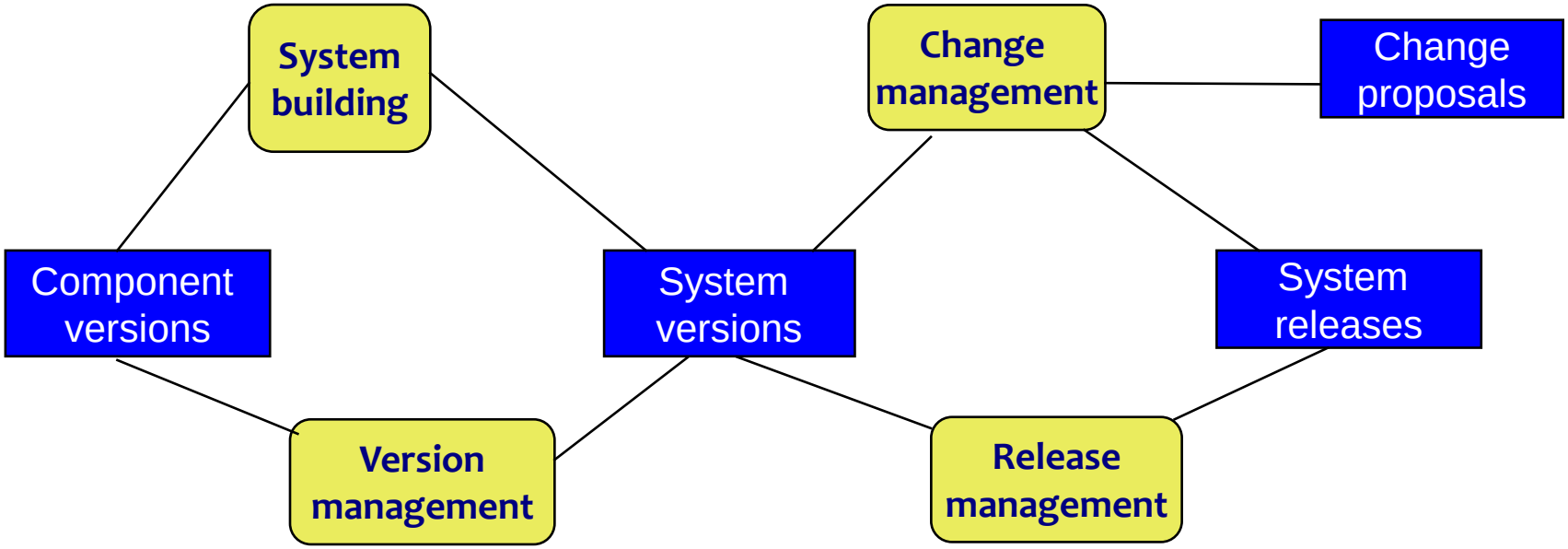
SCM is concerned with *policies*, *processes* and *tools* for managing *changing software systems*.

SCM allows keeping track of what *changes* and *component versions* have been incorporated into each *system version.*

SCM activities

The process of creating a complete, executable system by compiling and linking the *system components*, *external libraries*, *configuration files*, and other *related artifacts* implied in the specific system.

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.



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.

Preparing software for external release and keeping *track* of the system versions that have been released for customer use.

Development phases

Development - the development team is responsible for managing the software configuration and new functionality is being added to the software.

System testing - a version of the system is released internally for testing.

- No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.

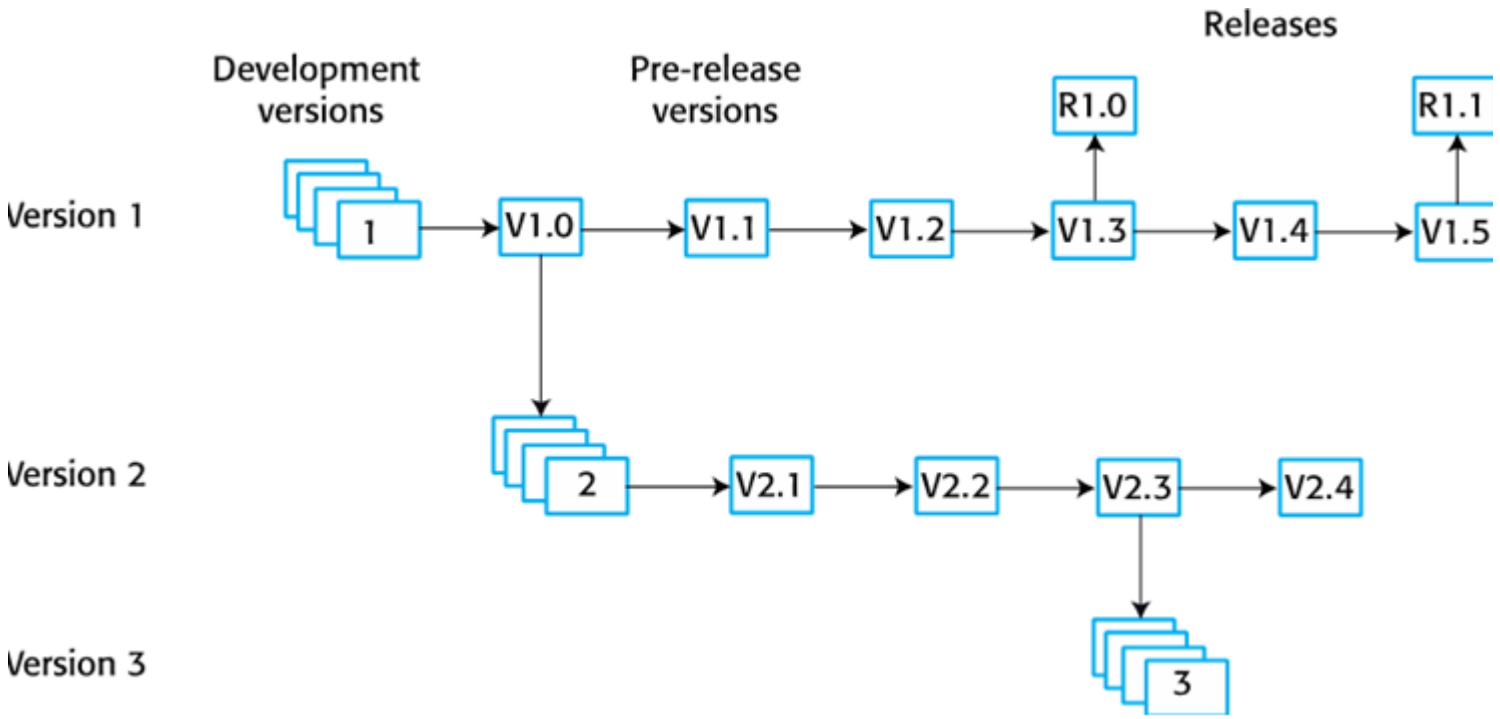
Release - the software is released to customers for use.

- New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

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.
- several teams may be involved in the development of different system versions.



SCM terminology (1)

Term	Explanation
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.*
Configuration 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.
Version	An *instance of a configuration item* that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
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 should always be possible to recreate a baseline from its constituent components.
Codeline	A codeline is a *set of versions of a software component* and other configuration items on which that component depends.

SCM terminology (2)

Term	Explanation
Mainline	A *sequence of baselines* representing different versions of a system.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Workspace	A *private work area* where software can be modified without affecting other developers who may be using or modifying that software.
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.
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.
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.

SCM process

SCM goal – to assure that all source code is compiled and linked to form a release package easy to install and execute.

- Discipline :
  - Clear records of all versions of source materials $\Rightarrow$ the possibility to use them anytime in order to build the correspondent release.
  - Posibly, the need for a SCM administrator

SCM is a natural extension of the software process.

Desiderate:

- Clear defined software process
- Well defined set of artifacts generated during the activities of the software process
- Integrated SCM system
- Team motivation and training with SCM

SCM plan

- Defines the *types of documents* to be managed and a document *naming scheme*.
- Defines who takes *responsibility* for the CM procedures and creation of *baselines*.
- Defines *policies* for change control and version management.
- Defines the SCM *records* which must be maintained.
- Describes the *tools* which should be used to assist the SCM process and any limitations on their use.
- Defines the *process* of tool use.
- Defines the SCM *database* used to record *configuration information*.
- May include information such as the configuration management of external software, process auditing, etc.

Artifacts

Categories of artifacts:

- Source code and executable code
- Code descriptions : requirements, models, user guide, etc.
- Data : contained in the program or external.

- Tipical artifacts:
  - Requirements specifications
  - Design specifications
  - Source code (business logic, DB tables, user screen scripts)
  - Executable code
  - Test cases (test scenarios, test scripts and associated test data)

SCM Tools

Levels:

Version and change control

- Example (mostly oriented on source code): RCS, CSSC, CVS, PRCS, Subversion

Including build functionality

- Example: Make, Odin, Cons, SCons, Ant, SourceForge

Integrating SCM activities with software process activities

- Example - only comercial tools and only partial integration: ClearCase, PVCS, Visual SourceSafe, AccuRev

For complete SCM (all artifacts of the software process and their relationships) – more tools, selected on different criteria (ex. artifacts that must be managed, execution infrastructure, cost, ...), are used.

Formative evaluation

1. Select what version management, as activity of the software configuration management process, means:
   a) keeping track of requests for changes to the software, cost and impact of changes analysis and deciding which changes should be implemented.
   b) 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.
   c) creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, and other related artifacts implied in the specific system.

2. Enumerate the artifacts, realized during the software development process, placed under version control.

https://forms.gle/LALX3BJZh5a7tqc36

Topics covered

- Software configuration management
- **Version management**
- System building
- Change management
- Release management

Example : a version management problem

Alice and Bob worked for a company called *FinanceMadeSimple* and were team members involved in developing a *personal finance product*. Alice discovered a bug in a *module called TaxReturnPreparation*. The bug was *that a tax return was reported as filed but, sometimes, it was not actually sent to the tax office*. She edited the module to fix the bug. Bob was working on the user interface for the system and was also working on TaxReturnPreparation. Unfortunately, he *took a copy before Alice had fixed the bug* and, after making his changes, he saved the module. This *overwrote Alice's changes* but she was not aware of this.

The product tests did not reveal the bug as it was an *intermittent failure* that depended on the sections of the tax return form that had been completed. The product was launched with the bug. For most users, everything worked OK. However, for a small number of users, their tax returns were not filed and *they were fined by the revenue service.* The subsequent investigation showed the software company was negligent. This was widely publicised and, as well as a fine from the tax authorities, *users lost confidence* in the software. Many switched to a rival product. FinanceMade Simple failed and both Bob and Alice lost their jobs

Version management

Artifacts implied in creating and running a software product are

- files (hundreds) containing lines of product code and of automated tests code (tens of thousands).
- libraries (dozens)
- several, different programs (tools)

- Version management (VM) is the process keeping *track of the different versions* of system components or configuration items and the systems in which these components are used.
- It is used to *manage* an *evolving codebase*.
- 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*.

Version management fundamentals

Version management systems provide a set of *features* that support four general *areas*:

Code transfer Developers take code into their personal file store to work on it, then return it to the shared code management system.

Version storage and retrieval Files may be stored in several different versions and specific versions of these files can be retrieved.

Merging and branching Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.

Version information Information about the different versions maintained in the system may be stored and retrieved

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* - a single master repository that maintains all versions of the software components that are being developed.

Example : Subversion

- *Distributed* - multiple versions of the component repository exist at the same time.

Example : Git

Functions of version management systems (tools)

Version and release identification

Versions are assigned unique identifiers when they are submitted to the system and can be retrieved using their identifier and other file attributes.

Storage management

To reduce the storage space required by multiple versions of components, only differences between versions are stored.

Change history recording

The reasons of all of the changes made to the code of a system or component are recorded and maintained.

Independent development support

The version management system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere. When a piece of code on which several developers work is submitted to the code management system, a new version is created so that files are never overwritten by later changes.

Project support

A version management system may support the development of several projects. All of the files associated with a project may be checked out at the same time.

Artifact storage and access model

Functions:

- *Create* artifact
- *Delete* artifact

Facilities:

- *View* (`check-out` for read-only)
- *Modify* (`check-out` for edit)

has impact on consistency ⇒ controlled retrieval: the artifact can only be viewed and not modified by another user.

- *Return* (`check-in`)

paired with modify - stores the modified artifact and re-enables modification on it; may include automated incrementation of the returned artifact version.

Public repository and private workspaces

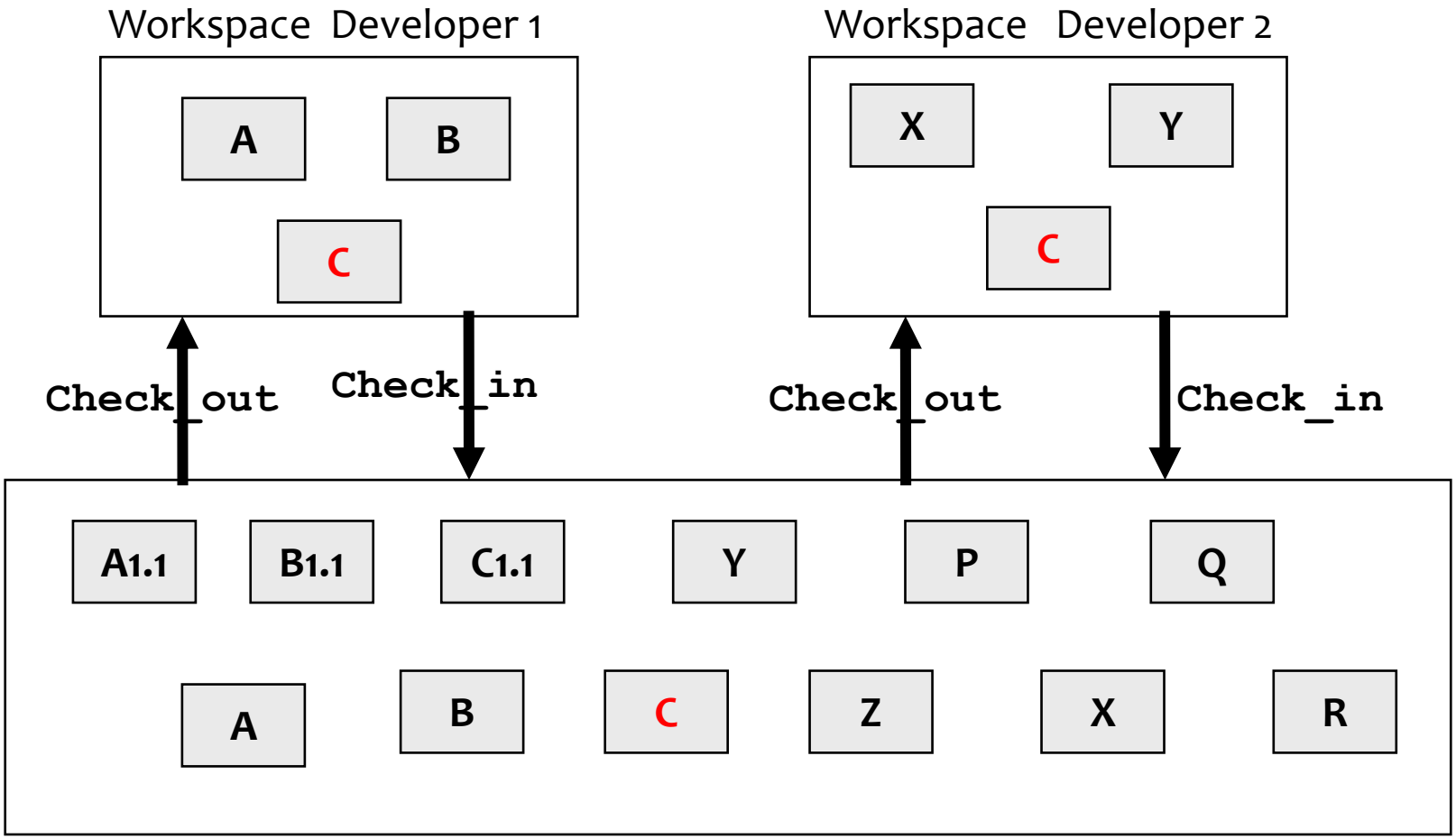
To support independent development without interference, version control systems use the concepts of *project repository* and *private workspace*.

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 `(checked-in)` to the repository.

Centralized version control
`Check-out` and `check-in` with a version repository



Version management system

Element C is marked as *shared,* and when `check-in,` version management system ensures that file copies are not in conflict.

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 they are responsible for 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.

Distributed version control
Repository cloning

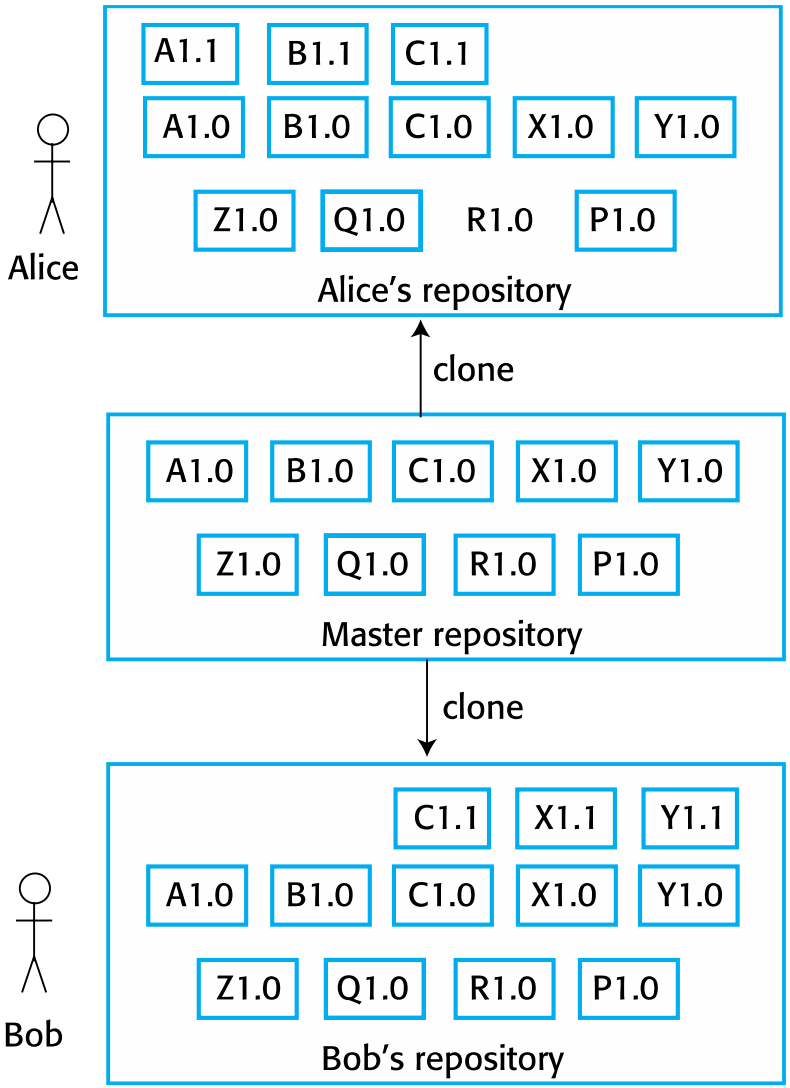


Figure 10.5 Repository cloning
Ian Sommerville –Software Engineering, ed.10

Distributed version management system

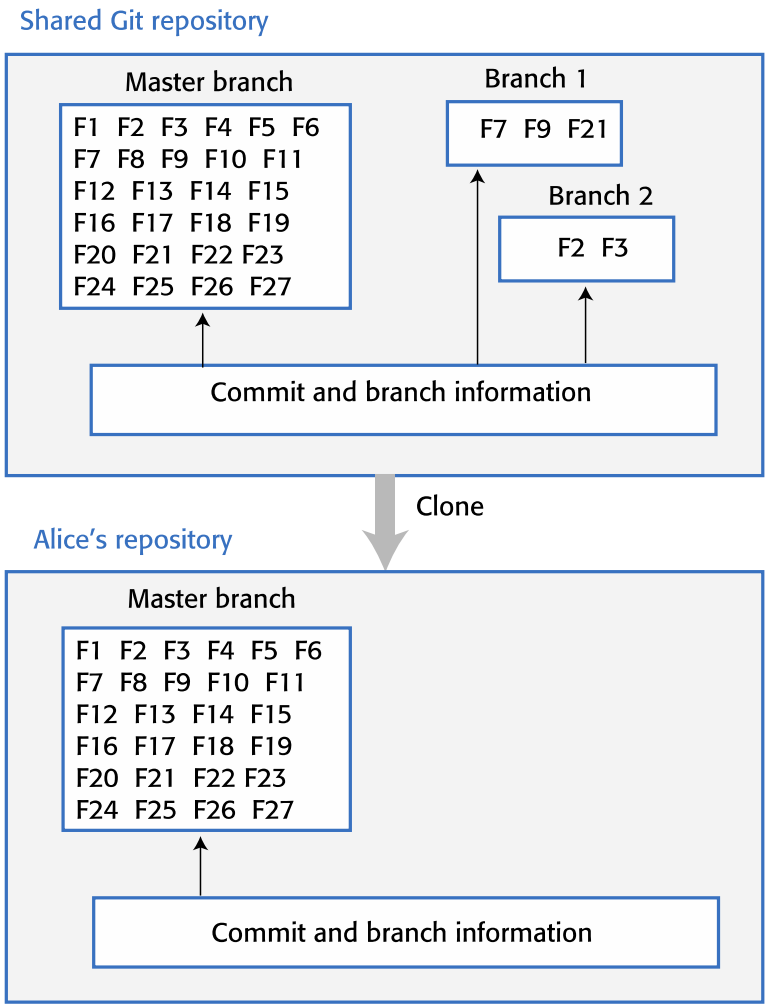
Example: Git

In 2005, Linus Torvalds, the developer of Linux, revolutionized source code management by developing a *distributed version control system* (DVCS) called Git to manage the code of the Linux kernel.

This was geared to supporting large-scale open source development. It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.

Instead of only keeping the copies of the files that users are working on, Git maintains a clone of the repository on every user's computer

Figure 10.5 Repository cloning in Git
Ian Sommerville – Engineering Software Products



Benefits of distributed version control

Resilience

Everyone working on a project has their own copy of the repository. If the *shared repository is damaged* or subjected to a *cyberattack*, work can continue, and the *clones can be used to restore the shared repository*. People can work *offline* if they don't have a network connection.

Speed

Committing changes to the repository is a *fast, local operation* and does not need data to be transferred over the network.

Flexibility

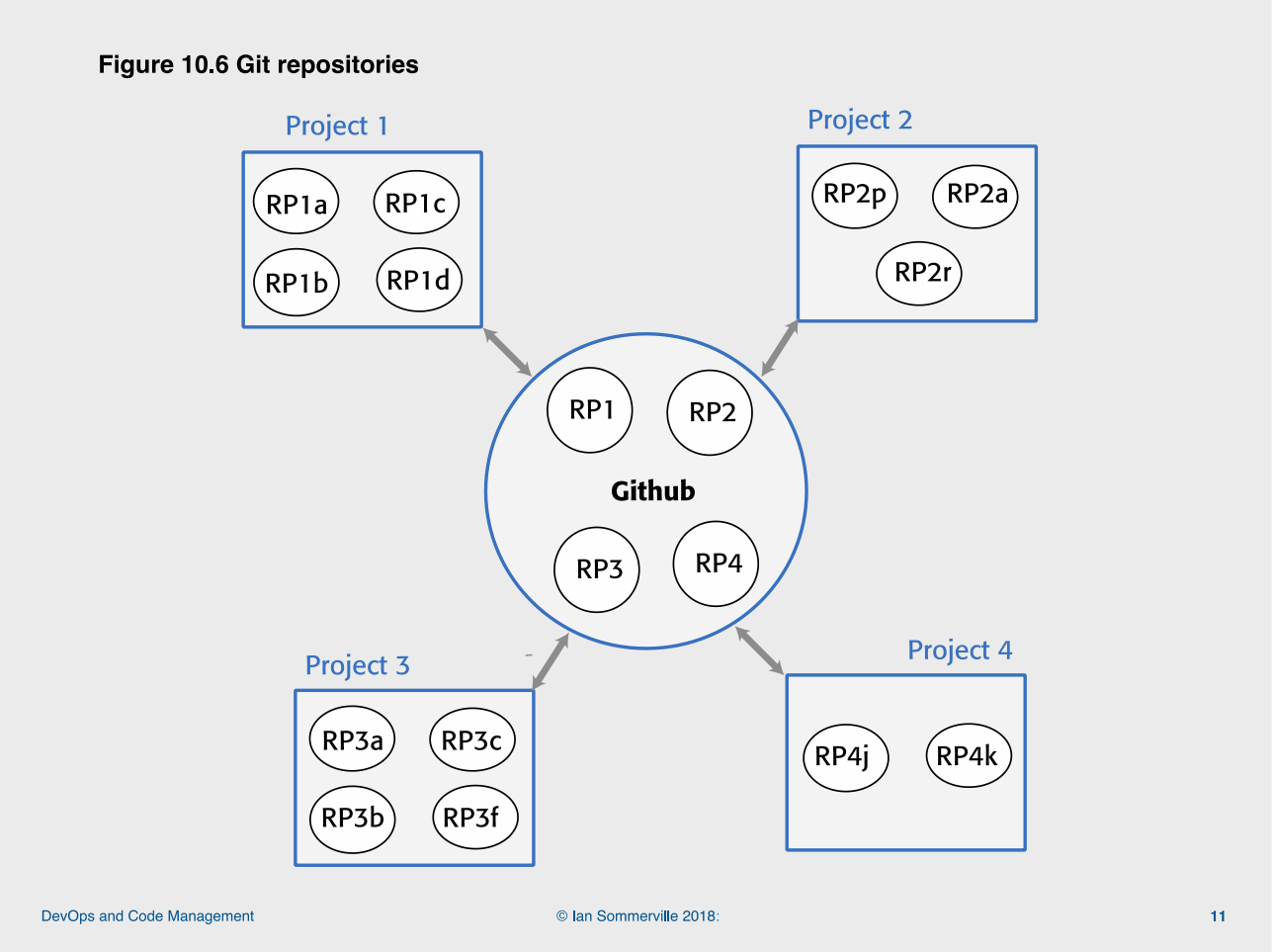
Local experimentation is much simpler. Developers can safely experiment and try different approaches without exposing these to other project members. With a centralized system, this may only be possible by working outside the code management system.

Distributed Git repositories

Figure 10.6 Git repositories
Ian Sommerville – Engineering Software Products

Variants for storing the central Git repository (repo):

- On a local server of the company.
- At Git repository hosting companies (ex. Github or Gitlab).



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.

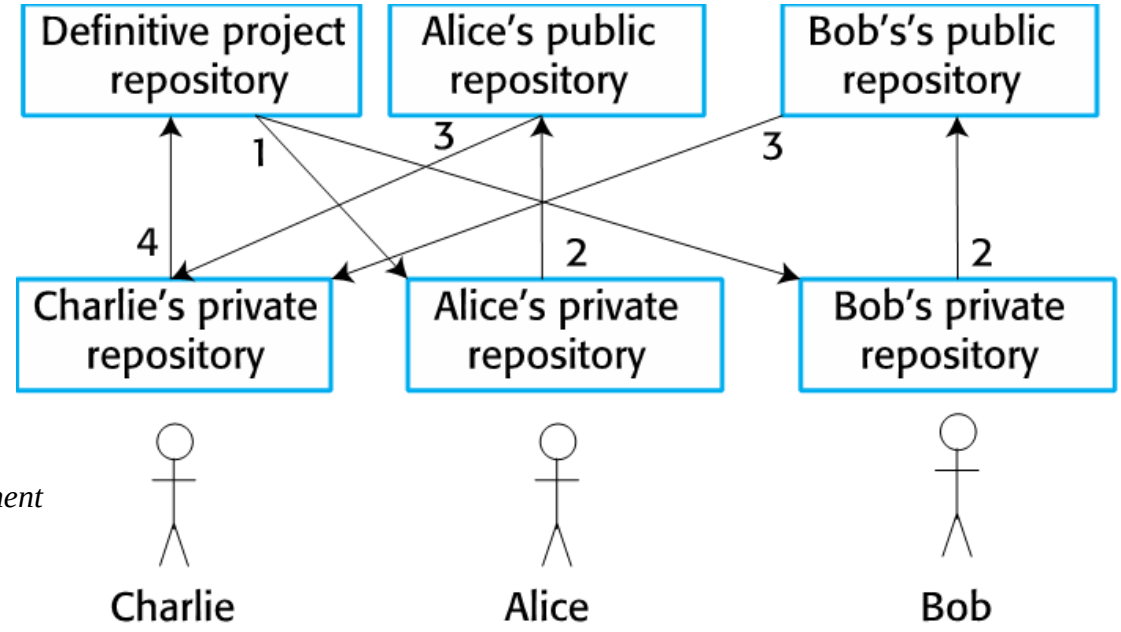


Figure 10.8 Using Github for open source development

Ian Sommerville – Engineering Software Products

Codelines and baselines

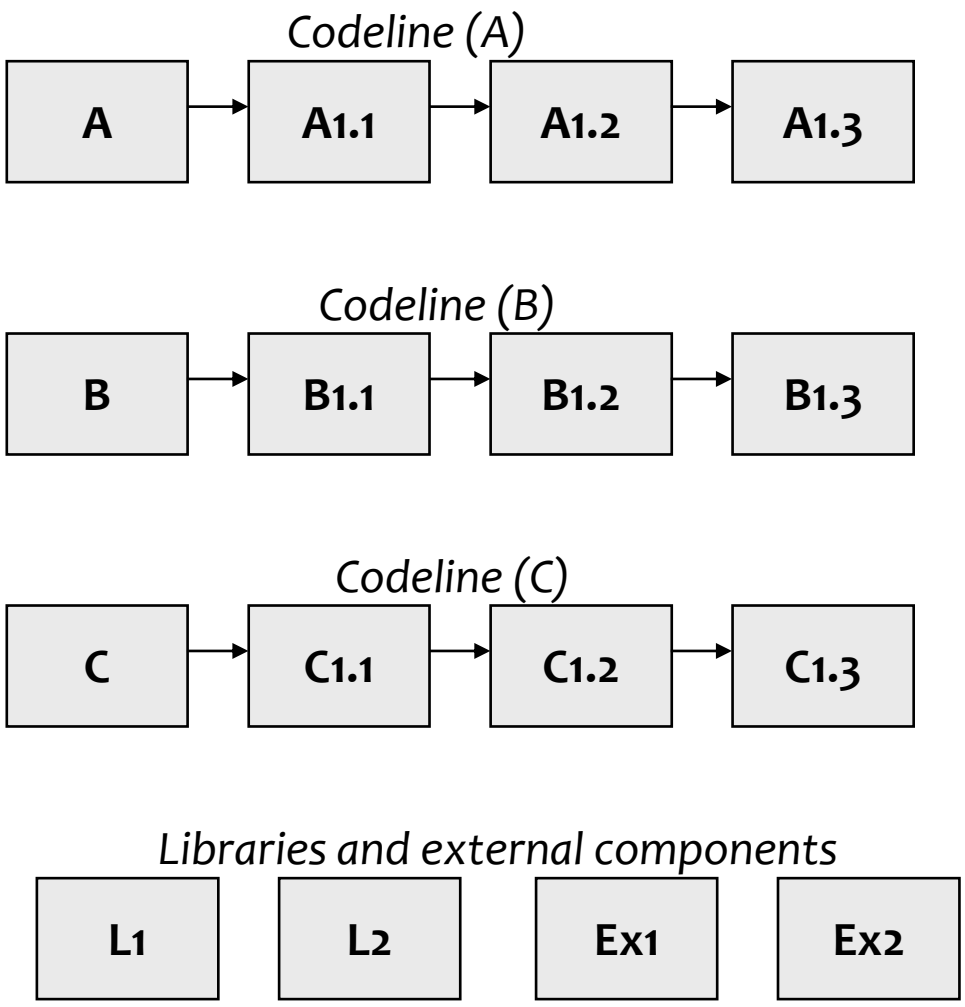
Def. *Codeline* = a sequence of versions of source code with later versions in the sequence derived from earlier versions.

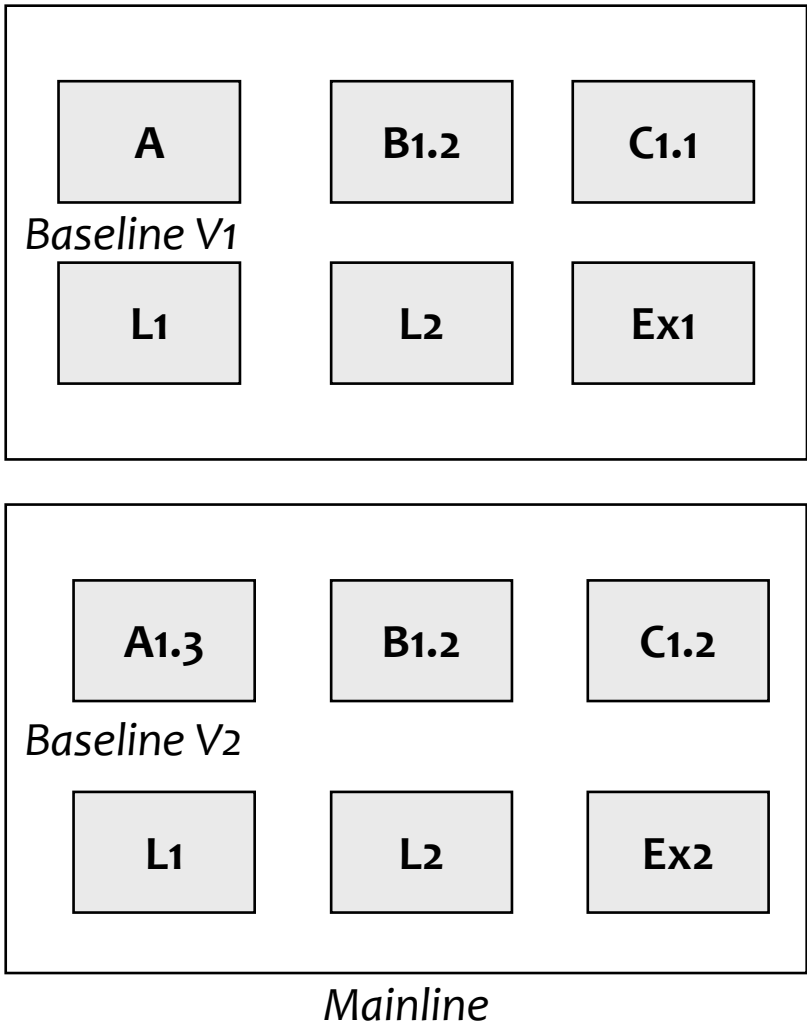
- Codelines normally apply to components of systems so that there are different versions of each component.

Def. *Baseline* = a definition of the composition 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, and other related artifacts implied in the specific system.

Codelines and baselines





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.

Branching and merging

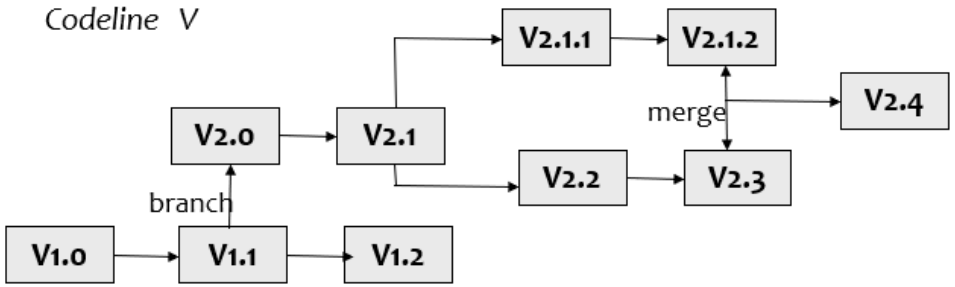
Rather than a linear sequence of versions that reflect changes to the component over time, there may be *several independent sequences*.

- This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.

A branch is an independent, *stand-alone version* that is created when a developer wishes to change a file.

At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.

- If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.
- For changes that imply the same part of the code, collaboration of the correspondent developers is necessary.



Branching and merging
Example

The repository ensures that branch files that have been changed cannot overwrite repository files without a merge operation.

- If Alice or Bob make mistakes on the branch they are working on, they can easily revert to the master file.
- If they commit changes, while working, they can revert to earlier versions of the work they have done. When they have finished and tested their code, they can then replace the master file by merging the work they have done with the master branch.

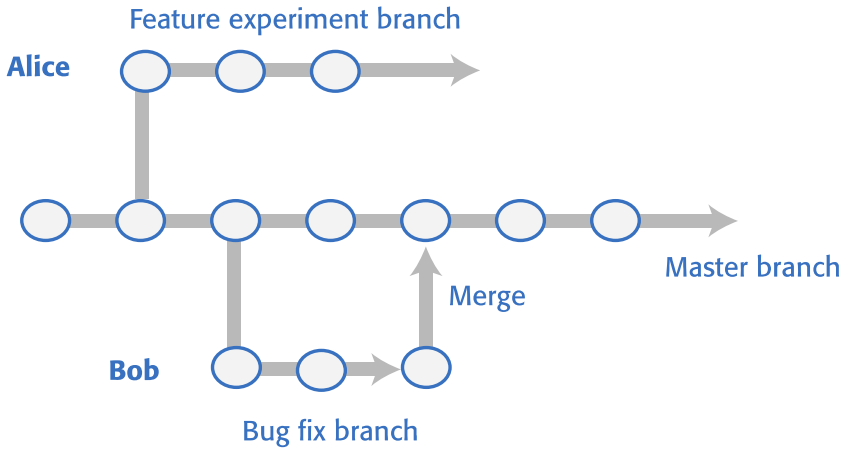
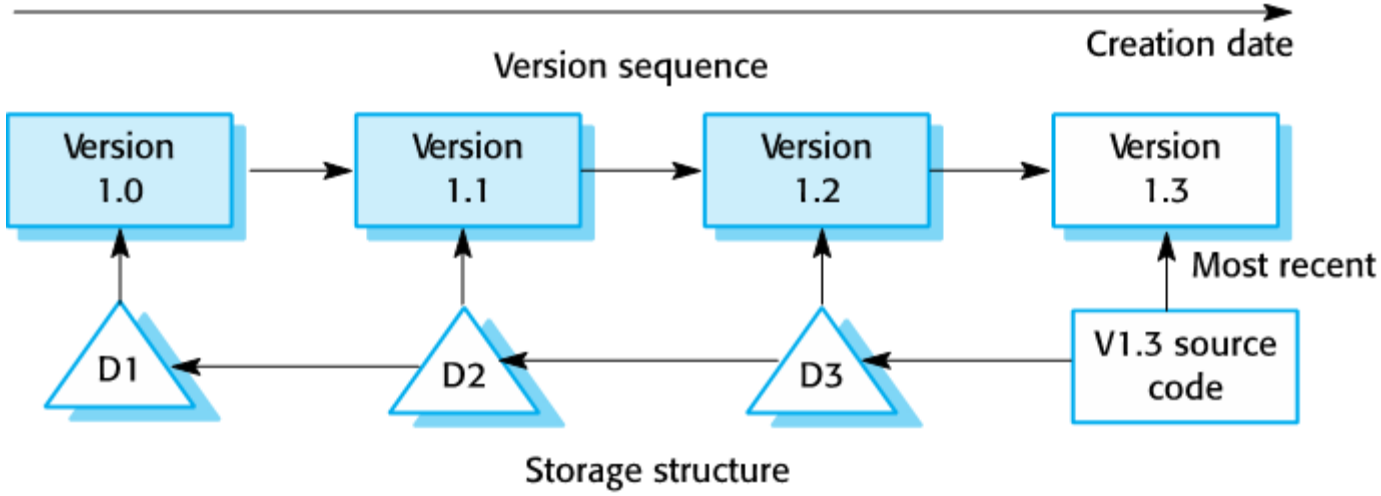


Figure 10.7 Branching and merging
Ian Sommerville – Engineering Software Products

Storage management

To spare expensive disk space, instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another.



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.

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.

Formative evaluation

1. Describe the operations “view”, “modify” and “return” provided by a version management system. Consider a project where 3 developers (John, Alice and Dan) colaborate and work in parallel. If at one moment John must read module X and Alice and Dan must modify it, describe a sequence of such operations executed by the 3 developers when using a centralized version control system.

2. Realize a comparison between centralized version management systems and the distributed ones. (similarities and differences).

https://forms.gle/XPu8XYWAeV7Fvbbc6

Topics covered

- Software configuration management
- Version management
- **System integration (building)**
- Change management
- Release management

System building

System integration (system building) implies gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system.

Typical activities that are part of the system integration process include:

- Installing database software and setting up the database with the appropriate schema.
- Loading test data into the database.
- Compiling the files that make up the product.
- Linking the compiled code with the libraries and other components used.
- Checking that external services used are operational.
- Deleting old configuration files and moving configuration files to the correct locations.
- Running a set of system tests to check that the integration has been successful.

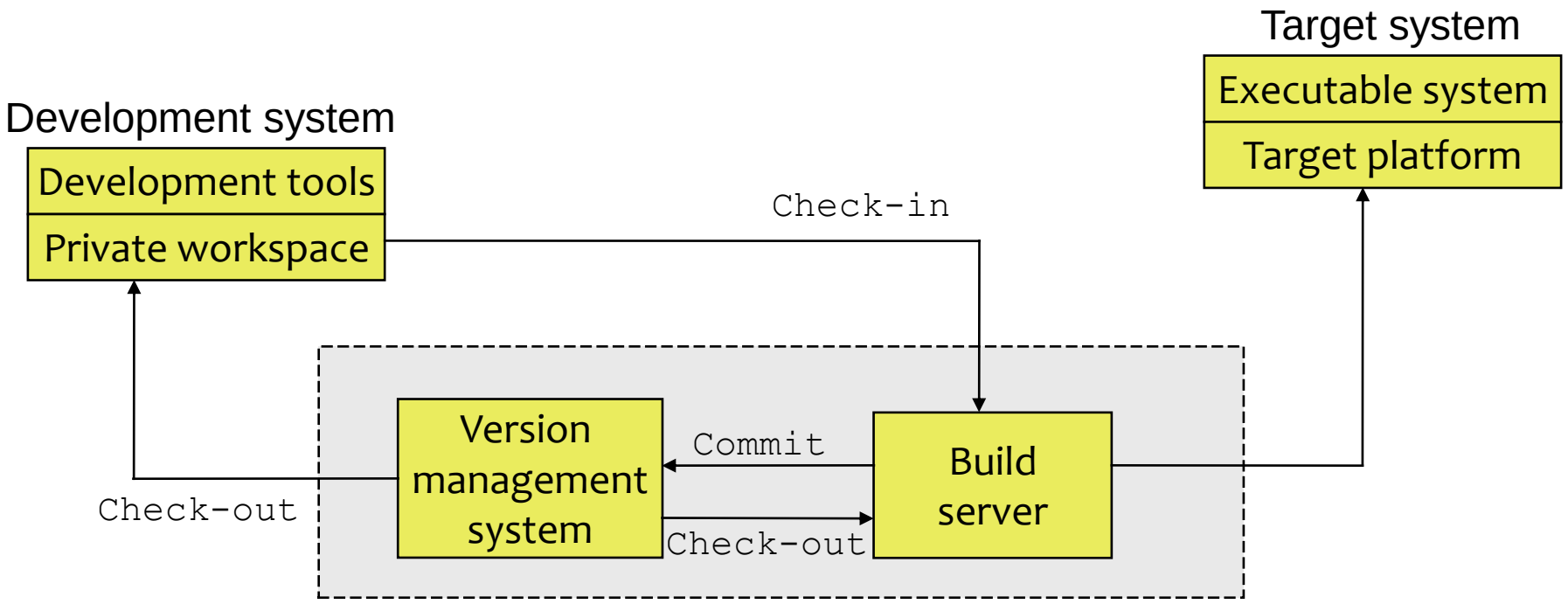
System building

System building is the process of creating a complete, executable system by compiling and linking the *system components*, *external libraries*, *configuration files*, and other *related artifacts* implied in the specific system.

System building tools and version management tools must communicate because

- 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.

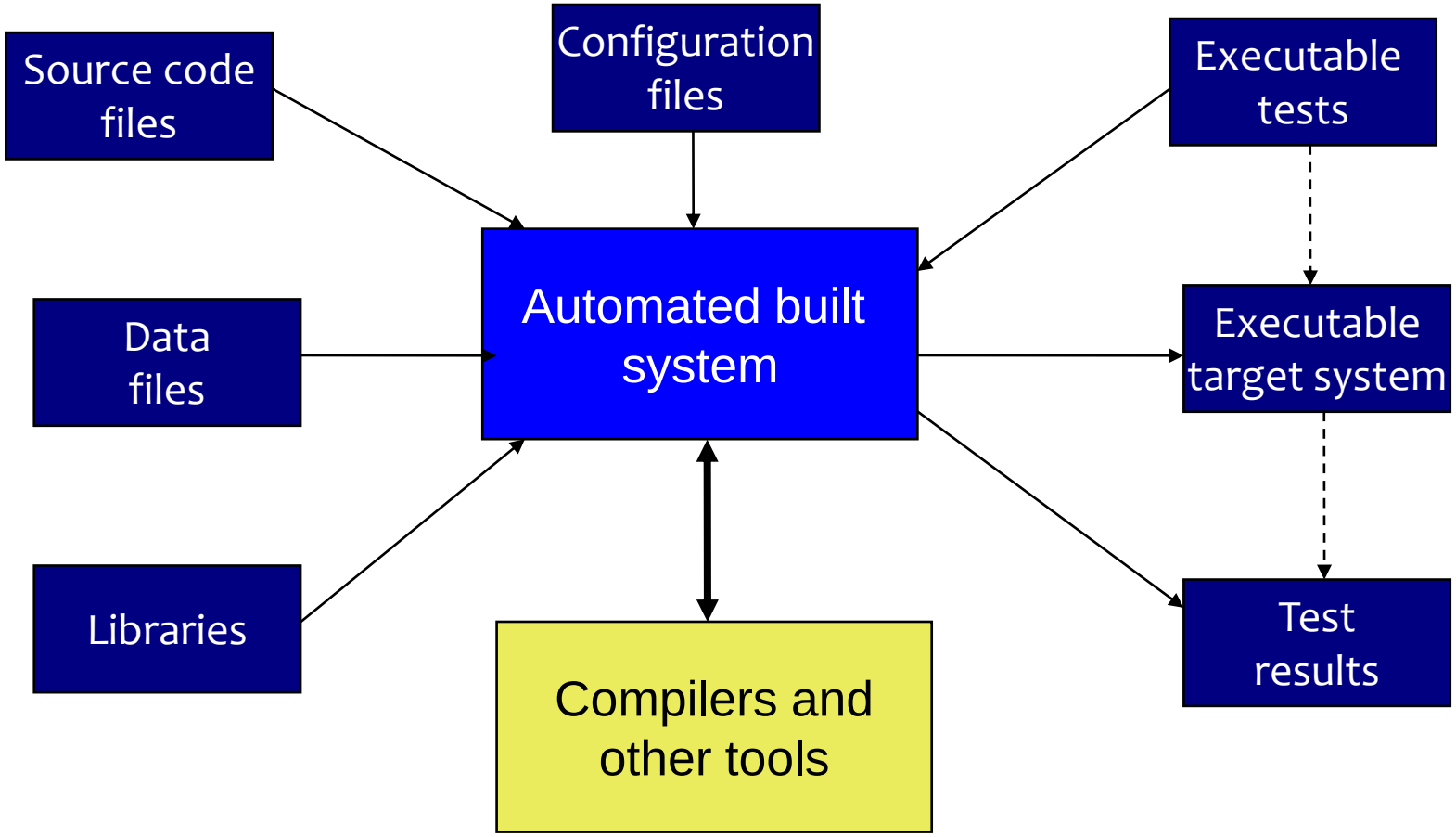
Development, building and execution platforms



Developers check out code from the version management system into a private workspace before making changes to the system.

Build server: used to build definitive, executable versions of the system.

System building



Functionality of a building tool

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

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.

Continous integration

Continuous integration means that an integrated version of the system is created and tested *every time a change is pushed* to the system's *shared repository*.

On completion of the push operation, the repository sends a message to an integration server to build a new version of the product

The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system.

If after a small change some system tests fail, the problem almost certainly lies in the new code that have been pushed to the project repo.

This is the code to focus on in order to find the bug that's causing the problem.

Continous integration

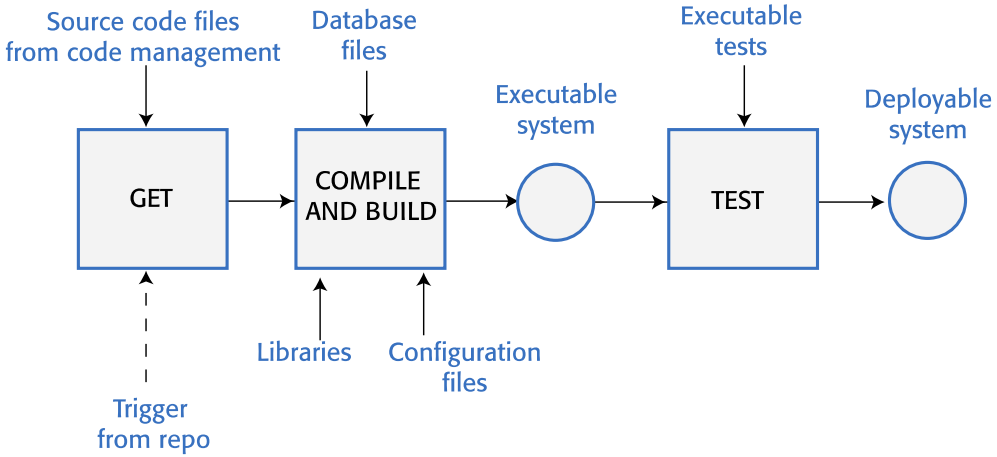


Figure 10.9 Continous integration
Ian Sommerville – Engineering Software Products

Problem : Breaking the build – after pushing code to the project repository, when integrated, causes some of the system tests to fail.

Solution : 'Integrate twice' approach to system integration.

Integration and test is done on local computer, before pushing code to the project repository to trigger the integration server.

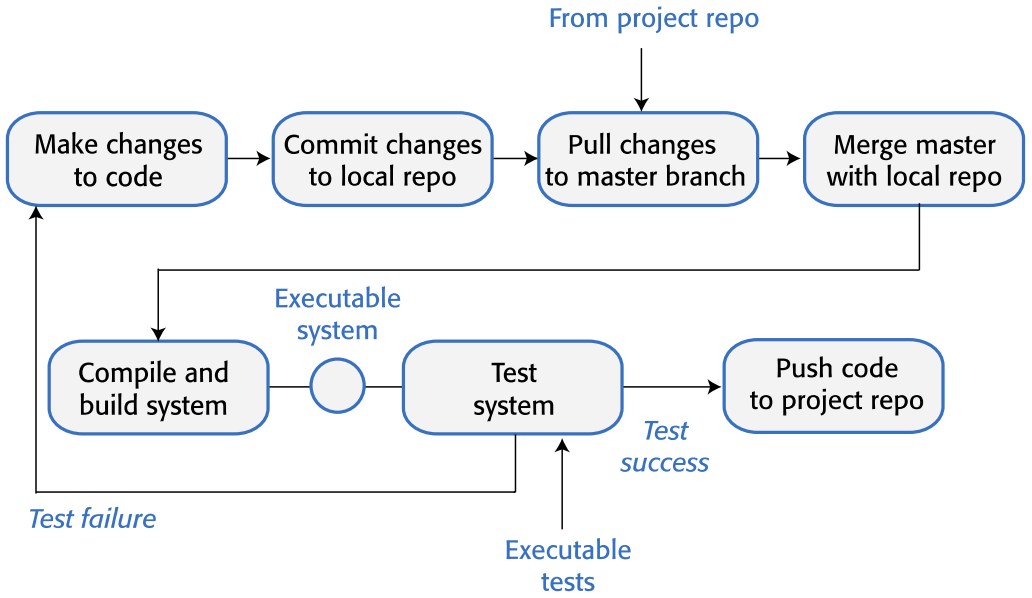


Figure 10.10 Local integration
Ian Sommerville – Engineering Software Products

System building

Continuous integration is only effective if the integration *process is fast* and developers do not have to wait for the results of their tests of the integrated system.

However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow.

It is therefore essential to have an *automated build process* that minimizes the time spent on these activities.

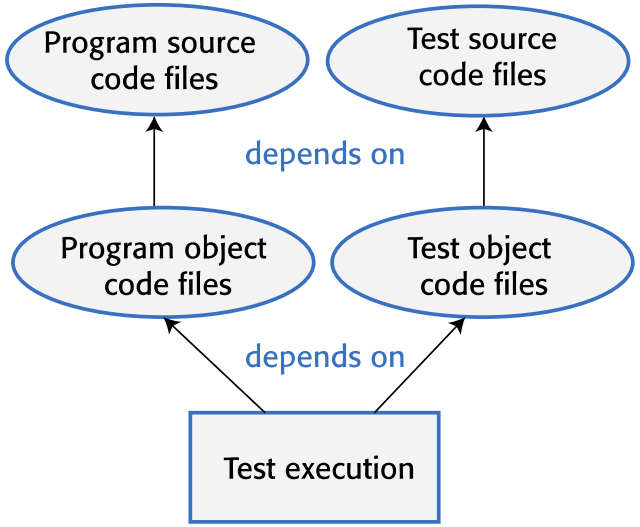
Fast system building is achieved using a process of *incremental building*, where *only those parts of the system that have been changed* are rebuilt.

System building

Figure 10.11 A dependency model
Ian Sommerville – Engineering Software Products

Running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests.

In turn, these depend on the source code for the system and the tests that are compiled to create the object code.



An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file ***modification timestamp*** to decide if a source code file has been changed.

- The modification date of the compiled code is after the modification date of the source code ⇒ The build system infers that no changes have been made to the source code and does nothing.
- The modification date of the compiled code is before the modification date of the source code ⇒ The build system recompiles the source and replaces the existing file of compiled code with an updated version.

System building

A lower-level dependency model shows the dependencies involved in creating the object code for a source code files called `Mycode`.

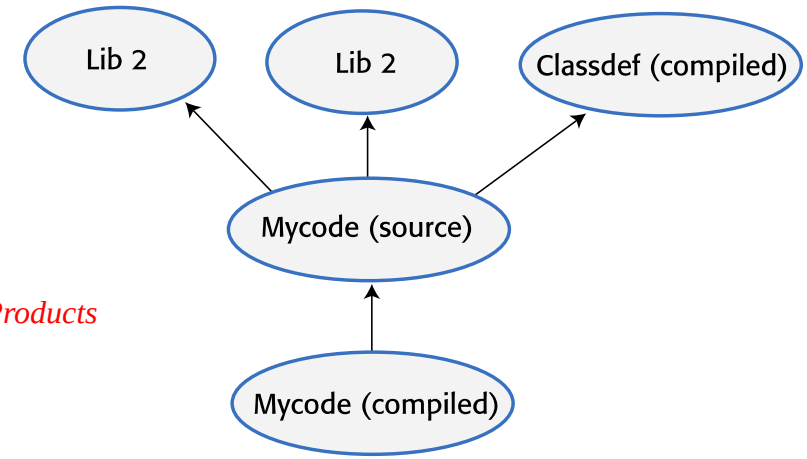


Figure 10.12 File dependencies
Ian Sommerville – Engineering Software Products

An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file ***modification timestamp*** to decide if a source code file has been changed.

- The modification date of the compiled code is after the modification date of the source code. However, the modification date of `Classdef` is after the modification date of the source code of `Mycode`. ⇒ `Mycode` has to be recompiled to incorporate these changes.

Continuous delivery and deployment (CD)

- Continuous integration (CI) means creating an executable version of a software system whenever a change is made to the repository. The CI tool *builds the system and runs tests on the development computer* or project integration server.

- However, the real environment in which software runs will inevitably be different from the development system.

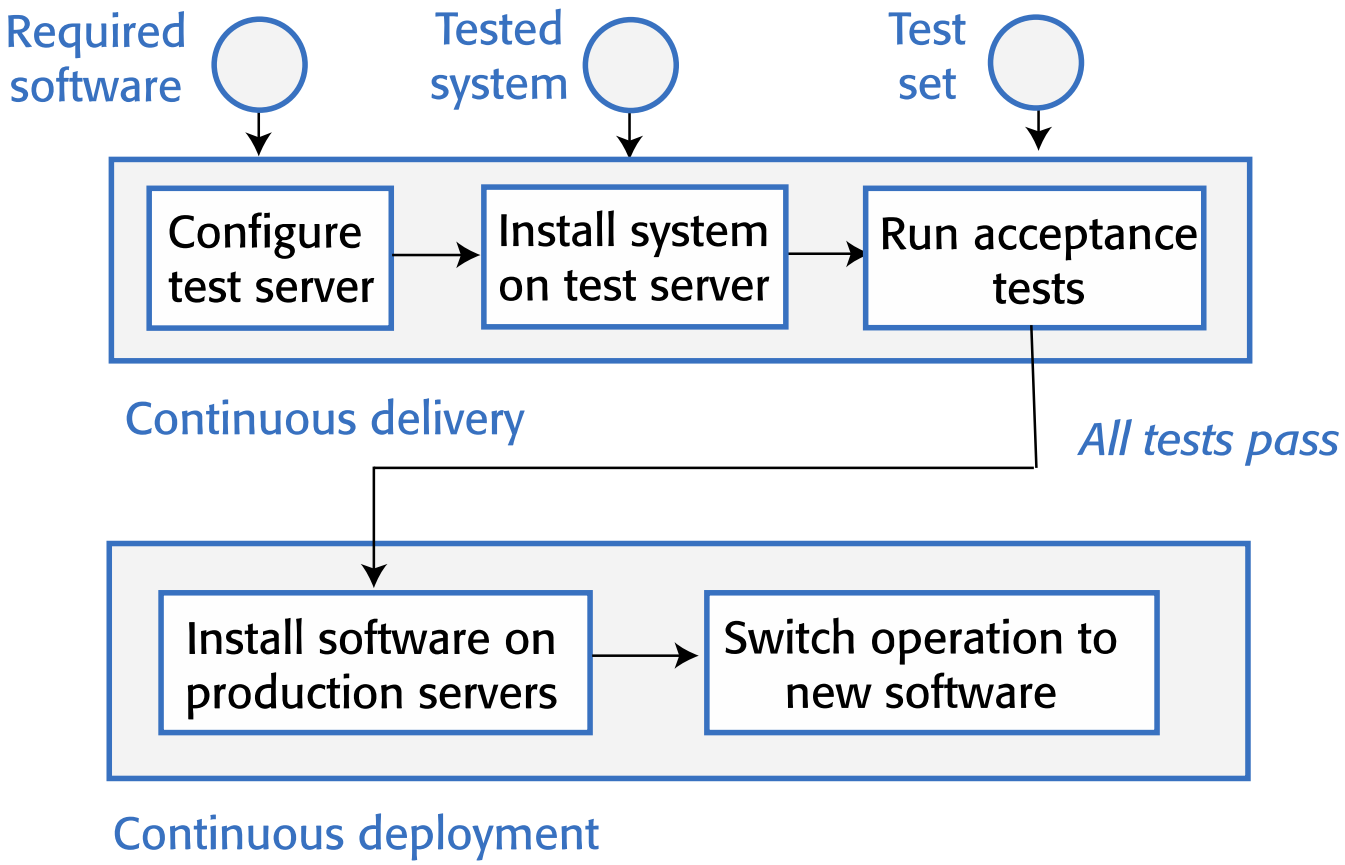
- When the software runs in its real, operational environment bugs may be revealed that did not show up in the test environment.

- Continuous delivery (CD) means that, after making changes to a system, one ensures that the changed system is ready for delivery to customers.

- This means that *the system has to be tested it in a production environment* to make sure that environmental factors do not cause system failures or slow down its performance.

Continuous delivery and deployment

Figure 10.13 Continous delivery and deployment
Ian Sommerville – Engineering Software Products



The deployment pipeline

- Initial integration testing
- Creating a *staged test environment* (a replica of the actual production environment in which the system will run).
- Running the system validation tests (functionality, load and performance) to check that the software works as expected.
- Testing and working until all of these tests pass
- Installing the changed on the production servers.
    - stop momentarily all new requests for service and the older version is left to process the outstanding transactions.
    - switch to the new version of the system and restart processing.

Benefits of continuous deployment

Reduced costs
Together with continuous deployment, *investment* in a completely automated deployment pipeline is needed. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and time-consuming, but these *costs can be recovered* if regular updates are made to the product.

Faster problem solving
If a *problem* occurs, it will probably only *affect a small part of the system* and it will be obvious what the source of that problem is.

Faster customer feedback
When new features are ready, they can be deployed for customer use. *User feedback* helps to identify improvements that are needed.

Formative evaluation

1. How must be done system building so that continous integration to be efficient ?

2. How is installed on the production servers a new version of a software system which has passed all validation tests ?

https://forms.gle/44wZ4svHhXdT6PHL9

Topics covered

- Software configuration management
- Version management
- System building
- **Change management**
- Release management

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.

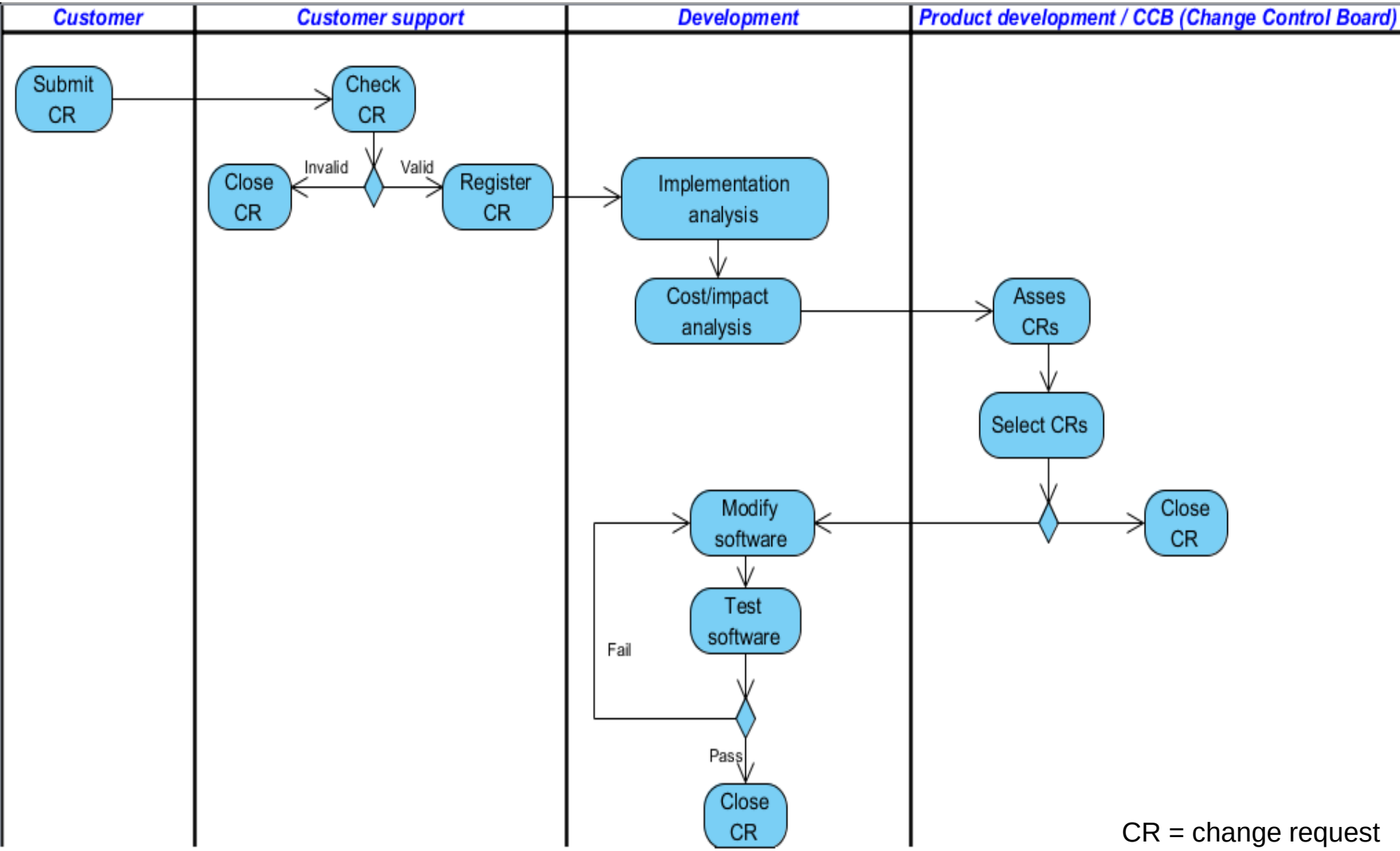
- *Priority* is given to the most *urgent* and *cost-effective* changes.

- Main activities:
  - Analyse the costs and benefits of proposed changes
  - Approve changes that are worthwhile
  - Give priorities to the changes to be done
  - Track which components in the system have been changed

Change management requirements

- Some *means* for users and developers to suggest required system changes
- A *process* to decide if changes should be included in a system
- *Software to keep track* of suggested changes and their status
- *Software* support for *managing* changing system *configurations* and *building* new systems

Change management process



CR = change request

Change request form

The definition of a change request form is part of the change management planning process.

Initially this form records

- *change* proposed
- *requestor* of change
- *reason* why change was suggested
- *urgency* of change

Further it records

- change *evaluation*
- *impact analysis*
- change *cost* and *recommendations*

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/01/09
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:** 25/01/09
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

...

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: 28/01/09 **CCB decision date:** 30/01/09
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:

Change approval

- Changes should be reviewed by an *external group* (CCB – change control board) who decides whether or not they are cost-effective from a *strategic and organizational viewpoint* rather than a technical viewpoint.

Factors in change analysis

- 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

Change management and agile methods

- In some agile methods, *customers* are *directly involved* in change management.
- They *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 development activities* 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.

Topics covered

- Software configuration management
- Version management
- System building
- Change management
- **Release management**

Release management

Preparing software for external release and keeping *track* of the system versions that have been released for customer use.

Def. *Release* = a version of a software system that is distributed to customers.

- For mass market software - types of release
  - *major releases* which deliver significant new functionality
  - *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 individual customer.

Release components

- the executable cod of the system
- configuration files - defining how the release should be configured for particular installations;
- data files (such as files of error messages) - are needed for successful system operation;
- an installation program - used to help install the system on target hardware;
- electronic and paper documentation - describing the system;
- packaging and associated publicity - designed for that release.

Objective : actual release for using.

Factors influencing system release planning

Factor	Description
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.
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.
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.

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 reproduction

- In the event of a problem, it may be necessary to *reproduce exactly* the software that has been delivered to a particular customer.

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.

- 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.

Objective : re-creating the executables.

Release planning

Preparing advertising and publicity material and put in place marketing strategies to convince customers to buy the new release of the system.

- Release timing:
  - Releases too frequent or requiring hardware upgrades $\Rightarrow$ customers may not move to the new release, especially if they have to pay for it.
  - Too infrequent releases $\Rightarrow$ market share may be lost as customers move to alternative systems.

Release management for SaaS

Software as a service (SaaS) is a *software distribution model* in which a *cloud provider hosts applications* and makes them *available to end users over the internet*.

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*.

Formative evaluation

1. In the change management process, what is the role of cost and impact analysis of the software system change requests ?

2. What categories of information must be kept, by the development team, for each release of a software system and for what are these necessary ?

https://forms.gle/vrxULPFeZMdq4CSz5

Key points (1)

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 version management, change management, system building and release management.

Version management involves keeping track of the different versions of software components as changes are made to them, aiming to avoid changes made by different developers interfering with each other.

All version management systems are based around a shared code repository with a set of features that support code transfer, version storage and retrieval, branching and merging and maintaining version information.

Git is a distributed code management system that is the most widely used system for software product development. Each developer works with their own copy of the repository which may be merged with the shared project repository.

Key points (2)

System building is the process of *assembling* system components into an *executable* program to run on a target computer system.

Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.

Continuous integration means that as soon as a change is committed to a project repository, it is integrated with existing code and a new version of the system is created for testing.

Automated system building tools reduce the time needed to compile and integrate the system by only recompiling those components and their dependents that have changed.

Key points (3)

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.