MODULE 4.3

Software Project Management - Risk management, Managing people, Teamwork. Project Planning, Software pricing, Plan-driven development, Project scheduling, Agile planning. Estimation techniques, COCOMO cost modeling. **Configuration management, Version management, System building, Change management, Release management**, Agile software management - SCRUM framework. Kanban methodology and lean approaches.

Configuration management

- Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems.
- You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems.
- Several versions may be under development and in use at the same time.
- If you don't have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, delivering the wrong version of a system to customers, or forgetting where the software source code for a particular version of the system or component is stored.
- Configuration management is useful for individual projects as it is easy for one person to forget what changes have been made.
- It is essential for team projects where several developers are working at the same time on a software system.
- The configuration management system provides team members with access to the system being developed and manages the changes that they make to the code

Configuration management

- The configuration management of a software system product involves four closely related activities (Figure 1):

1. Version control: This involves 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.
2. System building: This is the process of assembling program components, data, and libraries, then compiling and linking these to create an executable system.
3. Change management: This involves keeping track of requests for changes to delivered software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
4. Release management: This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Configuration management

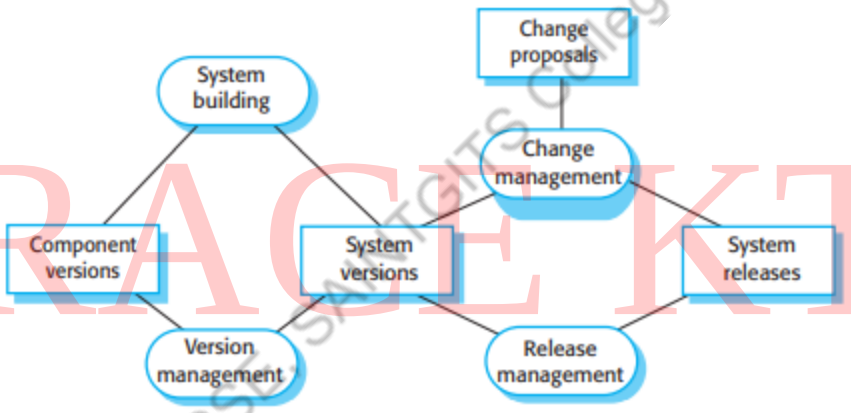


Figure 1: Configuration management activities

Configuration management

- Because of the large volume of information to be managed and the relationships between configuration items, tool support is essential for configuration management.
- Configuration management tools are used to store versions of system components, build systems from these components, track the releases of system versions to customers, and keep track of change proposals.
- CM tools range from simple tools that support a single configuration management task, such as bug tracking, to integrated environments that support all configuration management activities.
- Agile development, where components and systems are changed several times a day, is impossible without using CM tools.
- The definitive versions of components are held in a shared project repository, and developers copy them into their own workspace.
- They make changes to the code and 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. This makes the modified components available to other team members.

Configuration management

- The development of a software product or custom software system takes place in three distinct phases:

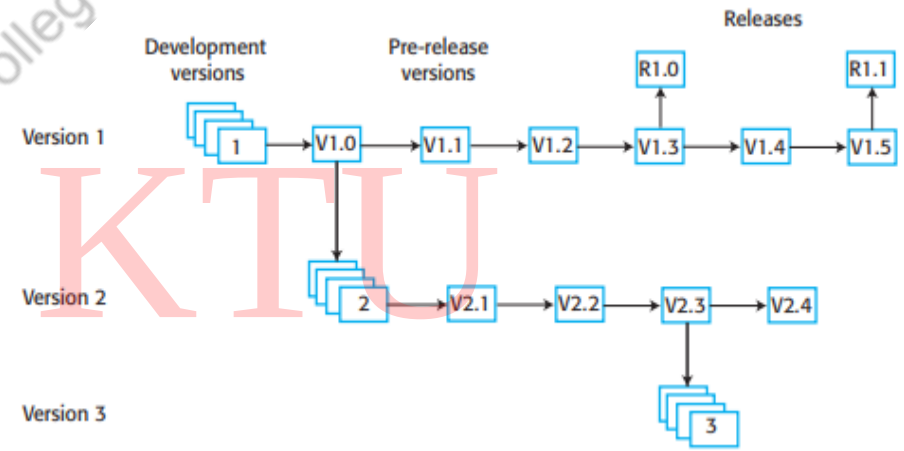
1. A **development phase** where the development team is responsible for managing the software configuration and new functionality is being added to the software. The development team decides on the changes to be made to the system.
2. A **system testing phase** where a version of the system is released internally for testing. This may be the responsibility of a quality management team or an individual or group within the development team. At this stage, no new functionality is added to the system. The changes made at this stage are bug fixes, performance improvements, and security vulnerability repairs. There may be some customer involvement as beta testers during this phase.
3. A **release phase** where the software is released to customers for use. After the release has been distributed, customers may submit bug reports and change requests. New versions of the released system may be developed to repair bugs and vulnerabilities and to include new features suggested by customers.

Configuration management

- 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. Figure 2 shows situations where three versions of a system are being developed:

1. Version 1.5 of the system has been developed to repair bug fixes and improve the performance of the first release of the system. It is the basis of the second system release (R1.1).
2. Version 2.4 is being tested with a view to it becoming release 2.0 of the system. No new features are being added at this stage.
3. Version 3 is a development system where new features are being added in response to change requests from customers and the development team. This will eventually be released as release 3.0.

These different versions have many common components as well as components or component versions that are unique to that system version. The CM system keeps track of the components that are part of each version and includes them as required in the system build



Configuration management

- In large software projects, configuration management is sometimes part of software quality management.
- The **quality manager** is responsible for both quality management and configuration management.
- When a pre-release version of the software is ready, the development team hands it over to the **quality management team**.
- The QM team checks that the system quality is acceptable. If so, it then becomes a controlled system, which means that all changes to the system have to be agreed on and recorded before they are implemented.
- Many **specialized terms** are used in configuration management. Unfortunately, these are not standardized.
- Military software systems were the first systems in which software CM was used, so the terminology for these systems reflected the processes and terminology used in hardware configuration management.
- Commercial systems developers did not know about military procedures or terminology and so often invented their own terms.
- Agile methods have also devised new terminology in order to distinguish the agile approach from traditional CM methods

Configuration management

- The definition and use of configuration management standards are essential for quality certification in both ISO 9000 and the SEI's capability maturity model.
- CM standards in a company may be based on generic standards such as IEEE 828-2012, an IEEE standard for configuration management.
- These standards focus on CM processes and the documents produced during the CM process (IEEE 2012).
- Using the external standards as a starting point, companies may then develop more detailed, company-specific standards that are tailored to their specific needs.
- However, agile methods rarely use these standards because of the documentation overhead involved.

CM Terminology

Term	Explanation
Baseline	A collection of component versions that make up a system. Baselines are controlled, which means that the component versions used in the baseline cannot be changed. It is always possible to re-create 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 set of versions of a software component and other configuration items on which that component depends.
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. Configuration items always have a unique identifier.
Mainline	A sequence of baselines representing different versions of a system.
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.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions should always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Version Management

Version management

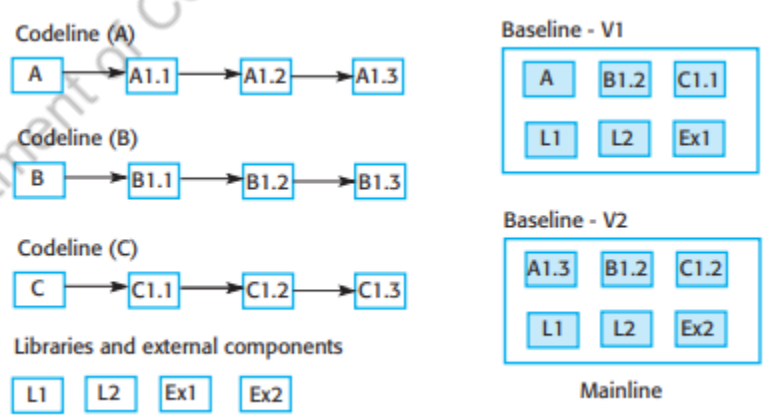
- Version management is the **process of keeping track of different versions of software components and the systems in which these components are used.**
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- In other words, version management is the process of managing **codelines and baselines.**

Version management

- 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 specifies the component versions that are included in the system and identifies the libraries used, configuration files, and other system information.

Version management

- In the figure different baselines use different versions of the components from each codeline.
- In the diagram, boxes representing components are shaded in the baseline definition to indicate that these are actually references to components in a codeline.
- The mainline is a sequence of system versions developed from an original baseline



Version management

- Baselines may be specified using a configuration language in which you define what components should be included in a specific version of a system.
- It is possible to explicitly specify an individual component version (X.1.2, say) or simply to specify the component identifier (X).
- If you simply include the component identifier in the configuration description, the most recent version of the component should be used.
- Baselines are important because you often have to re-create an individual version of a system.

Version management

- **Version control (VC) systems** identify, store, and control access to the different versions of components.
- There are **two types of modern version control system**:

1. **Centralized systems**, where a single master repository maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
2. **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.

Version management

- **Centralized and distributed VC systems** provide comparable functionality but implement this functionality in different ways. **Key features** of these systems include:

1. Version and release identification: Managed versions of a component are assigned unique identifiers when they are submitted to the system. These identifiers allow different versions of the same component to be managed, without changing the component name. Versions may also be assigned attributes, with the set of attributes used to uniquely identify each version.
2. Change history recording: The VC system keeps records of the changes that have been made to create a new version of a component from an earlier version.
3. Independent development: Different developers may be working on the same component at the same time. The version control 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.
4. Project support: A version control system may support the development of several projects, which share components. It is usually possible to check in and check out all of the files associated with a project rather than having to work with one file or directory at a time.
5. Storage management: Rather than maintain separate copies of all versions of a component, the version control system may use efficient mechanisms to ensure that duplicate copies of identical files are not maintained. Where there are only small differences between files, the VC system may store these differences rather than maintain multiple copies of files. A specific version may be automatically re-created by applying the differences to a master version

Version management

- Most software development is a team activity, so several team members often work on the same component at the same time.
- It's important to avoid situations where changes interfere with each other.
- The project repository maintains the "master" version of all components, which 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 completed their changes, the changed components are returned (checked-in) to the repository.
- However, **centralized and distributed VC systems support independent development of shared components in different ways.**
- In **centralized** systems (FIGURE 3**)**, 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. This creates a new component version that may then be shared.
- If two or more people are working on a component at the same time, each must check out the component from the repository.
- If a component has been checked out, the version control system warns other users wanting to check out that component that it has been checked out by someone else.
- The system will also ensure that when the modified components are checked in, the different versions are assigned different version identifiers and are stored separately.

Version management

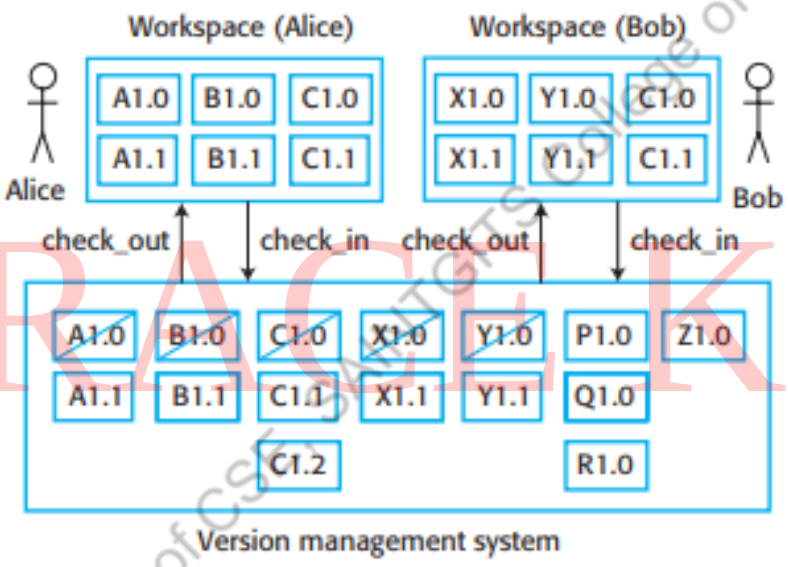


Fig 3: Check-in and check-out from a centralized version repository

Version management

- In a **distributed VC system**, such as Git, a different approach is used.
- A “master” repository is created on a server that maintains the code produced by the development team.
- Instead of simply checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on his or her computer.
- Developers work on the files required and maintain the new versions on their private repository on their own computer.
- When they have finished making changes, they “commit” these changes and update their private server repository.
- They may then “push” these changes to the project repository or tell the integration manager that changed versions are available.
- He or she may then “pull” these files to the project repository (see Figure 4). In this example, both Bob and Alice have cloned the project repository and have updated files.
- They have not yet pushed these back to the project repository.

Version management

- This model of development has a number of advantages:

1. 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.
2. It allows for offline working so that developers can commit changes if they do not have a network connection.
3. Project support is the default way of working. Developers can compile and test the entire system on their local machines and test the changes they have made.

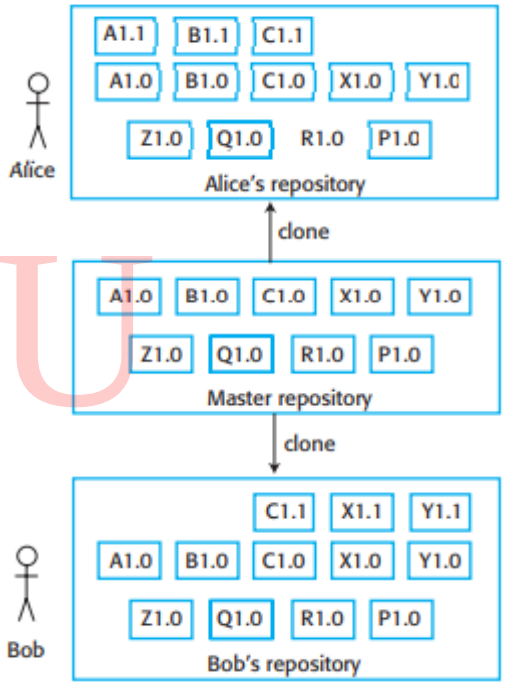


Figure 4: Repository cloning

Version management

- Distributed version control is essential for open-source development where several people may be working simultaneously on the same system without any central coordination.
- There is no way for the open-source system "manager" to know when changes will be made.
- In this case, 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.
- This organization is shown in figure 5.

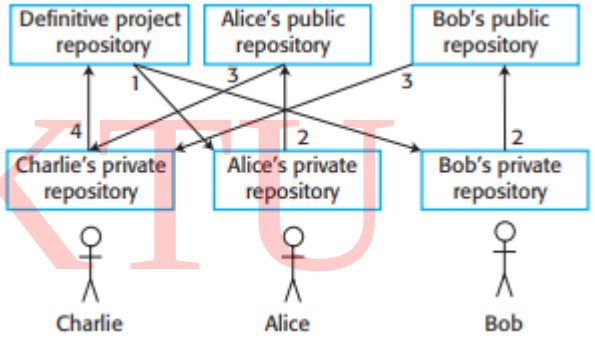


Figure 5

Version management

- A consequence of the independent development of the same component is that **codelines may branch**.
- Rather than a linear sequence of versions that reflect changes to the component over time, there may be **several independent sequences**, as shown in Figure 6.
- This is normal in system development, where different developers work independently on different versions of the source code and change it in different ways.
- It is generally recommended when working on a system that a new branch should be created so that changes do not accidentally break a working system.
- 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.
- This is also shown in Figure 6, where component versions 2.1.2 and 2.3 are merged to create version 2.4.
- If the changes made involve completely different parts of the code, the component versions may be merged automatically by the version control system by combining the code changes.
- This is the normal mode of operation when new features have been added.
- These code changes are merged into the master copy of the system. However, the changes made by different developers sometimes overlap.
- The changes may be incompatible and interfere with each other. In this case, a developer has to check for clashes and make changes to the components to resolve the incompatibilities between the different versions.

Version management

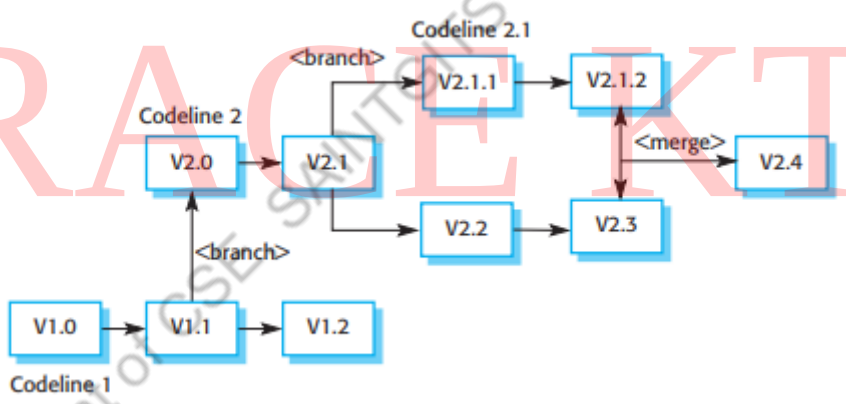
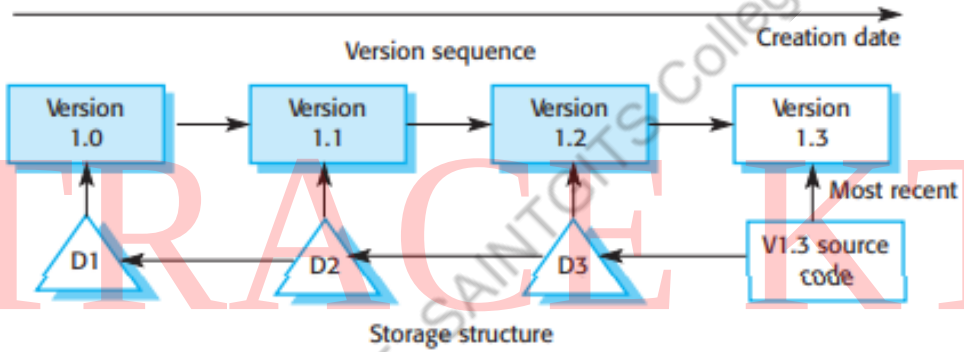


Figure 6: Branching and Merging

Version 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 re-created. This is illustrated in Figure 7.
- When a new version is created, the system simply stores a delta, a list of differences, between the new version and the older version used to create that new version.
- In Figure 7, the shaded boxes represent earlier versions of a component that are automatically re-created from the most recent component version.
- Deltas are usually stored as lists of changed lines, and, by applying these automatically, one version of a component can be created from another.
- As the most recent version of a component will most likely be the one used, most systems store that version in full. The deltas then define how to re-create earlier system versions.
- One of the problems with **a delta-based** approach to storage management is that it can take a long time to apply all of the deltas.
- 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. This reduces the overhead associated with lots of small files. Deltas are used within packfiles to further reduce their size



Storage management using deltas

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 information.
- System-building tools and version control tools must be integrated as the build process takes component versions from the repository managed by the version control system.
- System building involves assembling a large amount of information about the software and its operating environment.
- Therefore, it always makes sense to use an automated build tool to create a system build (Figure 8).
- source code files that are involved in the build are not enough. You may have to link these with externally provided libraries, data files (such as a file of error messages), and configuration files that define the target installation.
- You may have to specify the versions of the compiler and other software tools that are to be used in the build. Ideally, you should be able to build a complete system with a single command or mouse click.

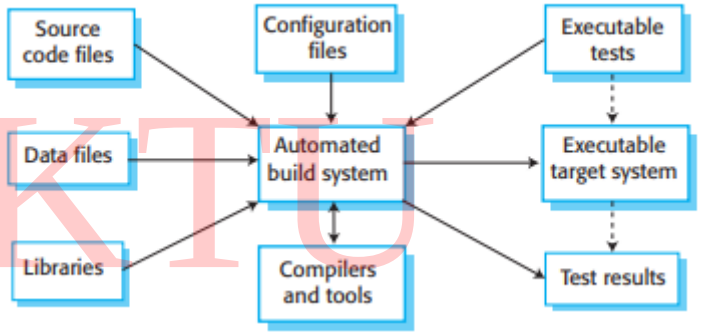


Figure 8: System building

System building

- Tools for system integration and building include some or all of the following features:

1. Build script generation: The build system should analyze the program that is being built, identify dependent components, and automatically generate a build script (configuration file). The system should also support the manual creation and editing of build scripts.
2. Version control system integration: The build system should check out the required versions of components from the version control system.
3. Minimal recompilation: The build system should work out what source code needs to be recompiled and set up compilations if required.
4. Executable system creation: The build system should link the compiled object code files with each other and with other required files, such as libraries and configuration files, to create an executable system.
5. Test automation: Some build systems can automatically run automated tests using test automation tools such as JUnit. These check that the build has not been "broken" by changes.
6. Reporting: The build system should provide reports about the success or failure of the build and the tests that have been run.
7. Documentation generation: The build system may be able to generate release notes about the build and system help pages.

System building

- The build script is a definition of the system to be built.
- It includes information about components and their dependencies, and the versions of tools used to compile and link the system.
- The configuration language used to define the build script includes constructs to describe the system components to be included in the build and their dependencies.

System building

- Building is a complex process, which is potentially error-prone, as three different system platforms may be involved (Figure 9):

1. **The development system**, which includes development tools such as compilers and source code editors. Developers check out code from the version control system into a private workspace before making changes to the system. They may wish to build a version of a system for testing in their development environment before committing changes that they have made to the version control system.
2. The **build server**, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system. All of the system developers check in code to the version control system on the build server for system building.
3. The **target environment**, which is the platform on which the system executes. This may be the same type of computer that is used for the development and build systems. However, for real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g., a cell phone). For large systems, the target environment may include databases and other application systems that cannot be installed on development machines. In these situations, it is not possible to build and test the system on the development computer or on the build server

System building

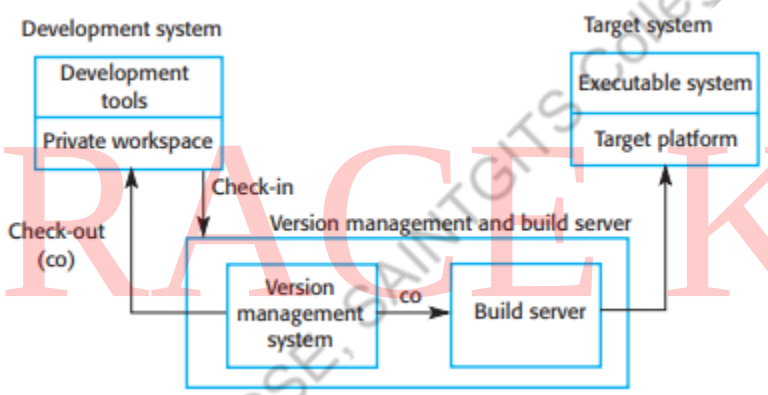


Figure 9: Development, build, and target platforms

System building

- Agile methods recommend that very frequent system builds should be carried out, with automated testing used to discover software problems. Frequent builds are part of a process of continuous integration as shown in Figure 10.
- In keeping with the agile methods notion of making many small changes, **continuous integration** involves rebuilding the mainline frequently, after small source code changes have been made.
- The **steps in continuous integration** are:

1. Extract the mainline system from the VC system into the developer's private workspace.
2. 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. He or she is responsible for repairing the problem.
3. Make the changes to the system components.
4. Build the system in a private workspace and rerun system tests. If the tests fail, continue editing.
5. Once the system has passed its tests, check it into the build system server but do not commit it as a new system baseline in the VC system.
6. Build the system on the build server and run the tests. Alternatively, if you are using Git, you can pull recent changes from the server to your private workspace. 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.
7. 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.

System building

- Tools such as Jenkins are used to support continuous integration.
- These tools can be set up to build a system as soon as a developer has completed a repository update.
- 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.
- However, although continuous integration is a good idea, it is not always possible to implement this approach to system building:
    1. 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. It may be impractical to build the system being developed several times per day.
    2. 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. There may be differences in hardware, operating system, or installed software. Therefore, more time is required for testing the system.

System building

- For large systems or for systems where the execution platform is not the same as the development platform, continuous integration is usually impossible. In those circumstances, frequent system building is supported using a **daily build system**:

1. 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. Components may be incomplete but should provide some basic functionality that can be tested.
2. A new version of the system is built from these components by compiling and linking them to form a complete system.
3. This system is then delivered to the testing team, which carries out a set of predefined system tests.
4. 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.

The advantages of using frequent builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Frequent building encourages thorough unit testing of components.

Frequent building encourages thorough unit testing of components.

System building

- As compilation is a computationally intensive process, tools to support system building may be 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.
- Therefore, there has to be a way of unambiguously linking the source code of a component with its equivalent object code.
- This linking is accomplished by associating a unique signature with each file where a source code component is stored.
- The corresponding object code, which has been compiled from the source code, has a related signature.
- The signature identifies each source 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 component was used to generate the object code component

System building

- Two types of signature may be used.(figure 10)

1. 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.
- [For example, say components Comp.java and Comp.class have modification signatures of 17:03:05:02:14:2014 and 16:58:43:02:14:2014, respectively. This means that the Java code was modified at 3 minutes and 5 seconds past 5 on the 14th of February 2014 and the compiled version was modified at 58 minutes and 43 seconds past 4 on the 14th of February 2014. In this case, the system would automatically recompile Comp.java because the compiled version has an earlier modification date than the most recent version of the component.]

2. 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 one character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.
- The checksum is assigned to the source code just before compilation and uniquely identifies the source file.
- The build system then tags the generated object code file with the checksum signature.
- If there is no object code file with the same signature as the source code file to be included in a system, then recompilation of the source code is necessary

System building

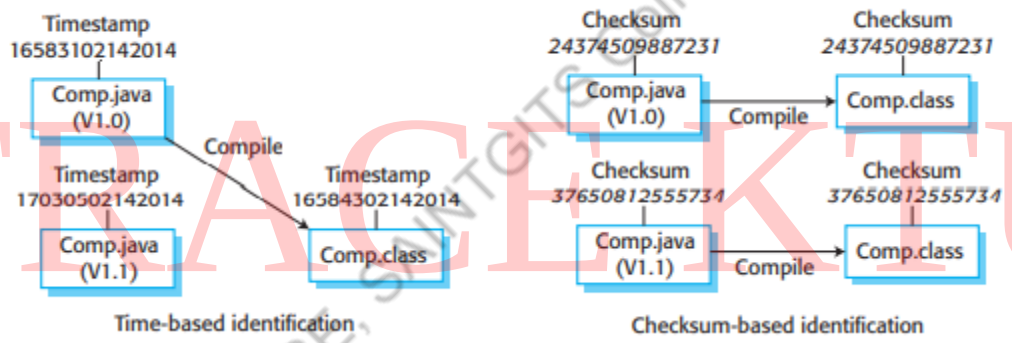


Figure 10: Linking source and object code

System building

- As object code files are not normally versioned, the first approach(modification timestamps) means that only the most recently compiled object code file is maintained in the system.
- This is normally related to the source code file by name; that is, it has the same name as the source code file but with a different suffix. Therefore, the source file Comp.Java may generate the object file Comp.class.
- Because source and object files are linked by name, it is not usually possible to build different versions of a source code component into the same directory at the same time.
- The compiler would generate object files with the same name, so only the most recently compiled version would be available.
- The checksum approach has the advantage of allowing many different versions of the object code of a component to be maintained at the same time.
- The signature rather than the filename is the link between source and object code. The source code and object code files have the same signature. Therefore, 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.

Change management

- Change is a fact of life for large software systems. 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.
- To ensure that the changes are applied to the system in a controlled way, you need a set of tool-supported, change management processes.
- **Change management is intended to ensure that the evolution of the system is controlled and that the most urgent and cost-effective changes are prioritized.**
- **Change management is the process of analyzing the costs and benefits of proposed changes, approving those changes that are cost-effective, and tracking which components in the system have been changed.**
- Figure 11 is a model of a change management process that shows the main change management activities. This process should come into effect when the software is handed over for release to customers or for deployment within an organization

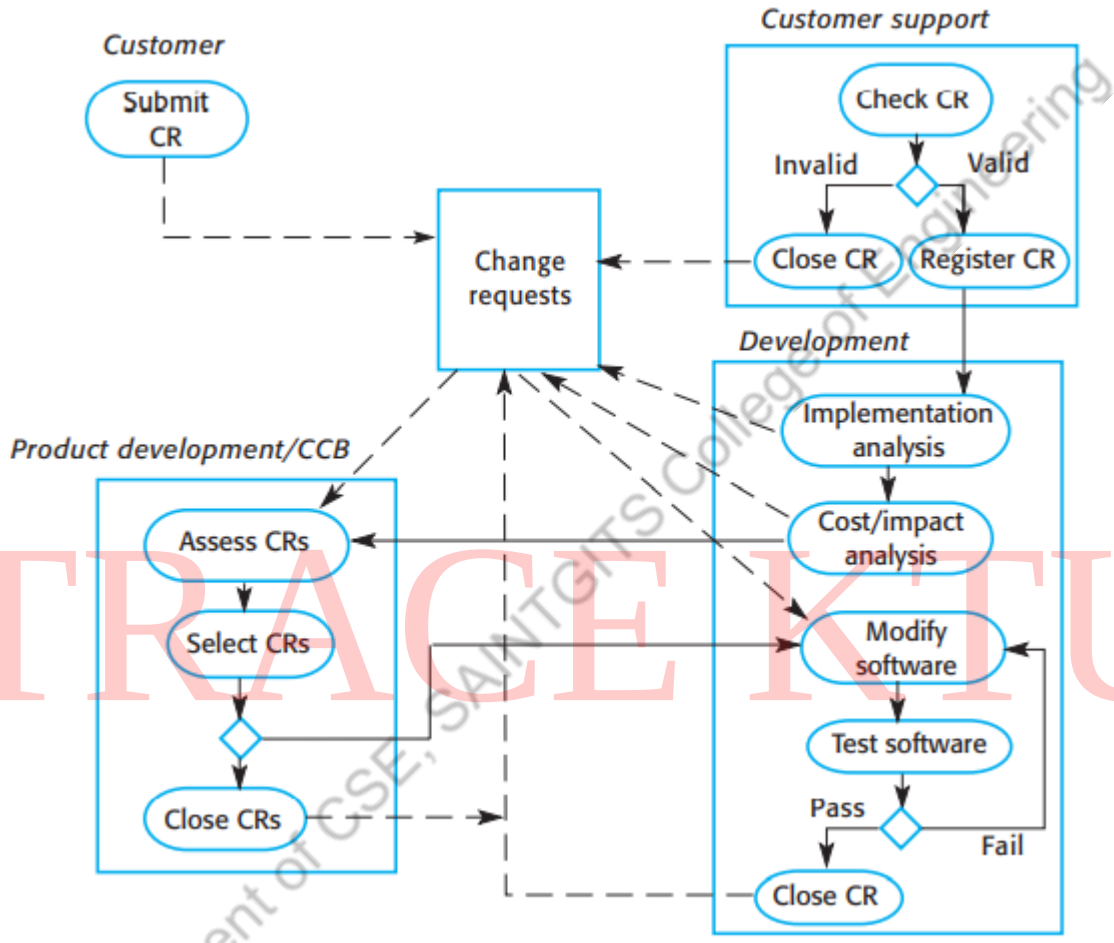


Figure 11: The change management process

Change management

- [Many variants of this process are in use depending on whether the software is a custom system, a product line, or an off-the-shelf product. The size of the company also makes a difference—small companies use a less formal process than large companies that are working with corporate or government customers.]
- All change management processes should include some way of checking, costing, and approving changes.
- Tools to support change management may be relatively simple issue or bug tracking systems or software that is integrated with a configuration management package for large-scale systems, such as Rational Clearcase.
- Issue tracking systems allow anyone to report a bug or make a suggestion for a system change, and they keep track of how the development team has responded to the issues.
- More complex systems are built around a process model of the change management process. They automate the entire process of handling change requests from the initial customer proposal to final change approval and change submission to the development team.

Change management

- **The change management process is initiated when a system stakeholder completes and submits a change request describing the change required to the system.**
- This could be a bug report, where the symptoms of the bug are described, or a request for additional functionality to be added to the system.
- Change requests may be submitted using a **change request form (CRF).**
- Stakeholders may be system owners and users, beta testers, developers, or the marketing department of a company.
- Electronic change request forms record information that is shared between all groups involved in change management.
- As the change request is processed, information is added to the CRF to record decisions made at each stage of the process.
- At any time, it therefore represents a snapshot of the state of the change request.
- In addition to recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change, and the dates when the change was requested, approved, implemented, and validated.
- The CRF may also include a section where a developer outlines how the change may be implemented. Again, the degree of formality in the CRF varies depending on the size and type of organization that is developing the system.

Change management

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

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 QM: **QM decision:**
Date submitted to CM:
Comments:

Change management

- System developers decide how to implement the change and estimate the time required to complete the change implementation.
- **After a change request has been submitted, it is checked to ensure that it is valid**.
- The checker may be from a customer or application support team or, for internal requests, may be a member of the development team. **The change request may be rejected at this stage**.
- If the change request is a bug report, the bug may have already been reported and repaired.
- Sometimes, what people believe to be problems are actually misunderstandings of what the system is expected to do.
- On occasions, people request features that have already been implemented but that they don't know about.
- **If any of these features are true(ie. the change is not valid), the issue is closed and the form is updated with the reason for closure**.
- **If it is a valid change request, it is then logged as an outstanding request for subsequent analysis**.
- **For valid change requests, the next stage of the process is change assessment and costing**.
- This function is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change.

Change management

- The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the components affected by the change.
- If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost of change implementation.
- Next, the required changes to the system modules are assessed.
- Finally, the cost of making the change is estimated, taking into account the costs of changing related components
- Following this analysis, **a separate group decides if it is cost-effective for the business to make the change to the software**.
- For military and government systems, this group is often called the **change control board (CCB**).
- In industry, it may be called something like a "**product development group**" responsible for making decisions about how a software system should evolve.
- This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, web pages, or documents.
- These small requests should be passed to the development team for immediate implementation. The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view.
- It decides whether the change in question is economically justified, and it prioritizes accepted changes for implementation.
- **Accepted changes are passed back to the development group**; **rejected change requests are closed and no further action is taken**.

Change management

- The factors that influence the decision on whether or not to implement a change include:

1. The consequences of not making the change: When assessing a change request, you have to consider what will happen if the change is not implemented.[If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the system to crash, this is very serious, and failure to make the change may disrupt the operational use of the system. On the other hand, if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly. The change should therefore have a low priority.]
2. The benefits of the change: Will the change benefit many users of the system, or will it only benefit the change proposer?
3. The number of users affected by the change: If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it means that the majority of system users have to adapt to it.
4. The costs of making the change If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected.
5. The product release cycle If a new version of the software has just been released to customers, it may make sense to delay implementation of the change until the next planned release

Change management

- **Change management for software products (e.g., a CAD system product), rather than custom systems specifically developed for a certain customer, are handled in a different way.**
- In software products, the customer is not directly involved in decisions about system evolution, so the relevance of the change to the customer's business is not an issue.
- Change requests for these products come from the customer support team, the company marketing team, and the developers themselves. These requests may reflect suggestions and feedback from customers or analyses of what is offered by competing products.
- The customer support team may submit change requests associated with bugs that have been discovered and reported by customers after the software has been released.
- Customers may use a web page or email to report bugs. A bug management team then checks that the bug reports are valid and translates them into formal system change requests.
- Marketing staff may meet with customers and investigate competitive products.
- They may suggest changes that should be included to make it easier to sell a new version of a system to new and existing customers.
- The system developers themselves may have some good ideas about new features that can be added to the system.

Change management

- During development, when new versions of the system are created through daily (or more frequent) system builds, there is no need for a formal change management process.
- Problems and requested changes are recorded in an issue tracking system and discussed in daily meetings.
- Changes that only affect individual components are passed directly to the system developer, who either accepts them or makes a case for why they are not required.
- However, an independent authority, such as the system architect, should assess and prioritize changes that cut across system modules that have been produced by different development teams.
- In some agile methods, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system.
- However, changes that involve software improvement are left to the discretion of 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.
- **As the development team changes software components, they should maintain a record of the changes made to each component. This is sometimes called the derivation history of a component**.
- A good way to keep the derivation history is in a **standardized comment at the beginning of the component source** code (Figure 12). This comment should reference the change request that triggered the software change. These comments can be processed by scripts that scan all components for the derivation histories and then generate component change reports.
- For documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document.

Change management

```
// 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   13/11/2009   New field   Change req. R07/02
```

Figure 12: Derivation History

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.
- A software product release is not just the executable code of the system.
- The release may also include:

■ configuration files defining how the release should be configured for particular installations;

■ data files, such as files of error messages in different languages, 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

Release management

- Preparing and distributing a **system release for mass-market products is an expensive process**.
- In addition to the technical work involved in creating a release distribution, advertising and publicity material have to be prepared.
- Marketing strategies may have to be designed to convince customers to buy the new release of the system.
- Careful thought must be given to **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 infrequent, market share may be lost as customers move to alternative systems.

Release management

- The various technical and organizational factors that you should take into account when deciding on when to release a new version of a software product are shown in Figure 13.

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. For marketing reasons, it may be necessary to include new features in a system so that users can be persuaded to upgrade from a previous release.
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 that affect the way in which many customers use the system, it may be necessary to correct them in a new system release. Minor system faults may be repaired by issuing patches, distributed over the Internet, which can be applied to the current release of the system.

Figure 13: Factors influencing system release planning

Release management

- **Release creation is the process of creating the collection of files and documentation that include all components of the system release**.
- This process involves several steps:

1. The executable code of the programs and all associated data files must be identified in the version control system and tagged with the release identifier.
2. Configuration descriptions may have to be written for different hardware and operating systems.
3. Updated instructions may have to be written for customers who need to configure their own systems.
4. Scripts for the installation program may have to be written.
5. Web pages have to be created describing the release, with links to system documentation.
6. Finally, 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 management

- For custom software or software product lines, the complexity of the system release management process depends on the number of system customers.
- Special releases of the system may have to be produced for each customer.
- Individual customers may be running several different releases of the system at the same time on different hardware.
- Where the software is part of a complex system of systems, different variants of the individual systems may have to be created.
- A software company may have to manage tens or even hundreds of different releases of their software.
- Their configuration management systems and processes have to be designed to provide information about which customers have which releases of the system and the relationship between releases and system versions.
- In the event of a problem with a delivered system, you have to be able to recover all of the component versions used in that specific system

Release management

- 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 military systems and those that control complex machines. These systems may have a long lifetime—30 years in some cases.
- Customers may use a single release of these systems for many years and may require specific changes to that release long after it has been superseded.
- 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.
- It may be necessary to keep copies of older operating systems and other support software because they may still be in operational use.

Release management

- You should also record the versions of the operating system, libraries, compilers, and other tools used to build the software.
- These tools may be required in order to build exactly the same system at some later date.
- Accordingly, you may have to store copies of the platform software and the tools used to create the system in the version control system, along with the source code of the target system.

Release management

- When planning the installation of new system releases, you cannot assume that customers will always install new system releases. Some system users may be happy an existing system and may not consider it worthwhile to absorb the cost of changing to a new release.
- New releases of the system cannot, therefore, rely on the installation of previous releases.
- One benefit of delivering software as a service (SaaS) is that it avoids all of these problems.
- 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, which is made available to all customers at the same time.
- However, this approach requires that all servers running the services be updated at the same time. To support server updates, specialized distribution management tools such as Puppet have been developed for “pushing” new software to servers.