Software configuration management

Whenever software is built, there is always scope for improvement and those improvements bring picture changes.

Changes may be required to modify or update any existing solution or to create a new solution for a problem.

Requirements keep on changing daily so we need to keep on upgrading our systems based on the current requirements and needs to meet desired outputs.

Changes should be analyzed before they are made to the existing system, recorded before they are implemented, reported to have details of before and after, and controlled in a manner that will improve quality and reduce error.

System Configuration Management (SCM) When we develop software, the product (software) undergoes many changes in their maintenance phase; we need to handle these changes effectively.

Several individuals (programs) works together to achieve these common goals. This individual produces several work product (SC Items) e.g., Intermediate version of modules or test data used during debugging, parts of the final product.

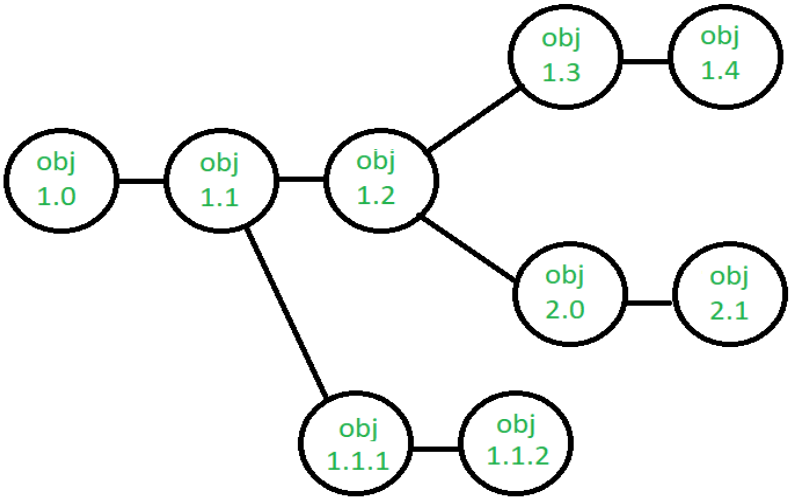
The elements that comprise all information produced as a part of the software process are collectively called a software configuration.

As software development progresses, the number of Software Configuration elements (SCI's) grow rapidly.

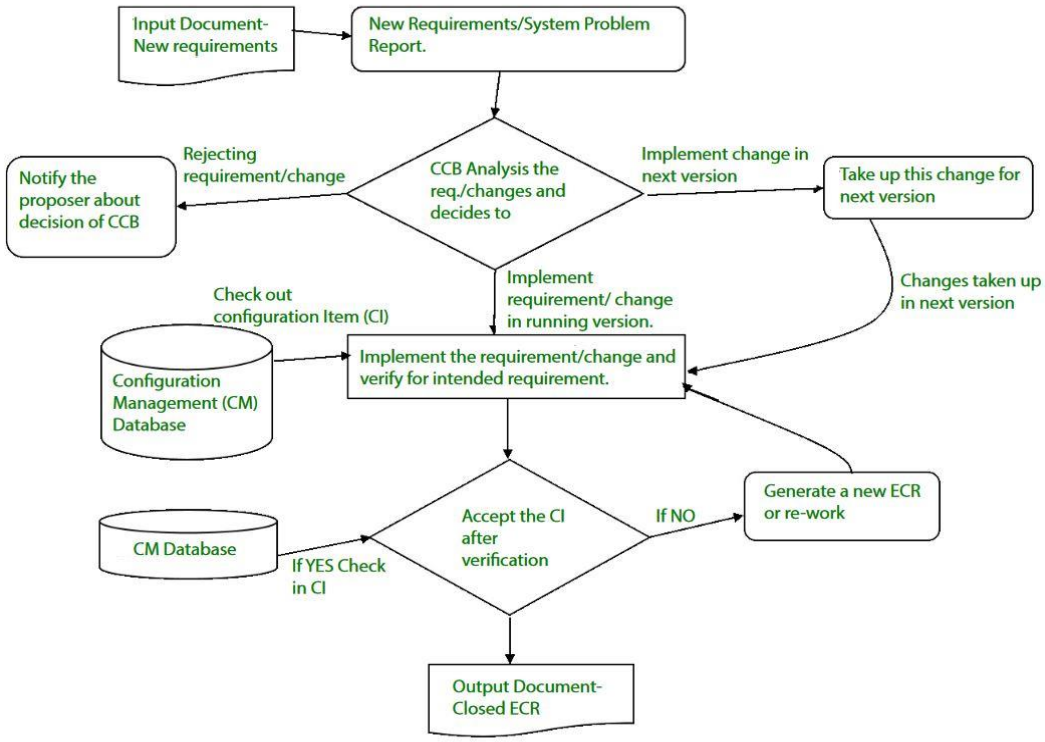
Processes involved in SCM – Configuration management provides a disciplined environment for smooth control of work products. It involves the following activities:

1. **Identification and Establishment –** Identifying the configuration items from products that compose baselines at given points in time (a baseline is a set of mutually consistent Configuration Items, which has been formally reviewed and agreed upon, and serves as the basis of further development). Establishing relationships among items, creating a mechanism to manage multiple levels of control and procedure for the change management system.
2. **Version control –** Creating versions/specifications of the existing product to build new products with the help of the SCM system. A description of the

version is given below:



3. Suppose after some changes, the version of the configuration object changes from 1.0 to 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The development of object 1.0 continues through 1.3 and 1.4, but finally, a noteworthy change to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.
3. **Change control –** Controlling changes to Configuration items (CI). The change control process is explained in Figure below:



4. **A** change request (CR) is submitted and evaluated to assess technical merit, potential side effects, the overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the

evaluation are presented as a change report, which is used by a change control board (CCB) —a person or group who makes a final decision on the status and priority of the change. An engineering change Request (ECR) is generated for each approved change. Also, CCB notifies the developer in case the change is rejected with proper reason. The ECR describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is "checked out" of the project database, the change is made, and then the object is tested again. The object is then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.

4. **Configuration auditing –** A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified. The audit confirms the completeness, correctness, and consistency of items in the SCM system and tracks action items from the audit to closure.
5. **Reporting –** Providing accurate status and current configuration data to developers, testers, end users, customers, and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guides, etc.

System Configuration Management (SCM) is a software engineering practice that focuses on managing the configuration of software systems and ensuring that software components are properly controlled, tracked, and stored. It is a critical aspect of software development, as it helps to ensure that changes made to a software system are properly coordinated and that the system is always in a known and stable state.

SCM involves a set of processes and tools that help to manage the different components of a software system, including source code, documentation, and other assets. It enables teams to track changes made to the software system, identify when and why changes were made, and manage the integration of these changes into the final product.

Importance of Software Configuration Management

1. **Effective Bug Tracking**: Linking code modifications to issues that have been reported, makes bug tracking more effective.
2. **Continuous Deployment and Integration**: SCM combines with continuous processes to automate deployment and testing, resulting in more dependable and timely software delivery.
3. **Risk management**: SCM lowers the chance of introducing critical flaws by assisting in the early detection and correction of problems.
4. **Support for Big Projects**: Source Code Control (SCM) offers an orderly method to handle code modifications for big projects, fostering a well-organized development process.
5. **Reproducibility:** By recording precise versions of code, libraries, and dependencies, source code versioning (SCM) makes builds repeatable.

6. **Parallel Development**: SCM facilitates parallel development by enabling several developers to collaborate on various branches at once.

Need for System configuration management

1. **Replicability**: Software version control (SCM) makes ensures that a software system can be replicated at any stage of its development. This is necessary for testing, debugging, and upholding consistent environments in production, testing, and development.
2. **Identification of Configuration**: Source code, documentation, and executable files are examples of configuration elements that SCM helps in locating and labelling. The management of a system's constituent parts and their interactions depend on this identification.
3. **Effective Process of Development**: By automating monotonous processes like managing dependencies, merging changes, and resolving disputes, SCM simplifies the development process. Error risk is decreased and efficiency is increased because of this automation.

Key objectives of SCM

1. **Control the evolution of software systems:** SCM helps to ensure that changes to a software system are properly planned, tested, and integrated into the final product.
2. **Enable collaboration and coordination:** SCM helps teams to collaborate and coordinate their work, ensuring that changes are properly integrated and that everyone is working from the same version of the software system.
3. **Provide version control:** SCM provides version control for software systems, enabling teams to manage and track different versions of the system and to revert to earlier versions if necessary.
4. **Facilitate replication and distribution:** SCM helps to ensure that replication and distribution occurred in a software product.

.....

Product Metrics in Software Engineering

In software engineering, product metrics are quantitative measures used to assess the characteristics of software products. These metrics help in evaluating various aspects such as quality, performance, maintainability, and complexity of the software. By providing objective data, product metrics enable developers and managers to make informed decisions about the software development process. Common product metrics include lines of code (LOC), cyclomatic complexity, depth of conditional nesting, and readability measures like the Fog Index.

What are Product Matrices?

Product metrics are software product measures at any stage of their development, from requirements to established systems. Product metrics are related to software features only. **Product metrics fall into two classes:**

1. Dynamic metrics are collected by measurements made from a program in execution.
2. Static metrics are collected by measurements made from system representations such as design, programs, or documentation.

Dynamic metrics help assess a program's efficiency and reliability while static metrics help understand, understand, and maintain the complexity of a software system. **Dynamic metrics** are usually quite closely related to software quality attributes. It is relatively easy to measure the execution time required for particular tasks and to estimate the time required to start the system. These are directly related to the efficiency of the system failures and the type of failure can be logged and directly related to the reliability of the software. On the other hand, static matrices have an indirect relationship with quality attributes. A large number of these matrices have been proposed to try to derive and validate the relationship between complexity, understand ability, and maintainability—several static metrics that have been used for assessing quality attributes.

Software Product Metrics

Some common product metrics are:

Common Product Metrics

- Fan-in/Fan-out
- Length of Code
- Cyclomatic complexity
- Length of Identifiers
- Depth of conditional nesting
- Fog index

1. Fan-in/Fan-out

Fan-in is a measure of the number of functions that call some other function (say X). Fan-out is the number of functions which are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of the control logic needed to coordinate the called components.

High Fan-in: Indicates that a module is widely reused, which might suggest it's a critical piece of functionality. However, it can also mean that changes to this module can have wide-ranging impacts. High Fan-out:

Indicates that a module depends on many others, which can make it complex and potentially more fragile, as it is affected by changes in all the modules it calls.
In simple word Fan-in measures how many other modules depend on a given module. Whereas Fan-out measures how many other modules a given module depends on.

2. Length of Code

Length of code is measure of the size of a program. Generally, the large the size of the code of a program component, the more complex and error-prone that component is likely to be. Length of code can be measure in various ways. The most common metric is Lines of Code (LOC) .
Lines of Code (LOC) are a measure of the number of lines in a program's source code.

There are two types of Lines of Code (LOC)

1. **Physical LOC:** Physical LOC counts all the lines in the source code file, including blank lines and comments.
2. **Logical LOC:** Logical LOC counts only the lines that contain executable statements or declarations, excluding comments and blank lines.

Importance of Lines of Code (LOC)

Following are the importance of Lines of Code (LOC):

- **Estimate Effort and Cost**: LOC can help estimate the effort and cost required for development, maintenance, and testing.
- **Measure Productivity**: Comparing LOC written over a period can give a rough measure of productivity.
- **Understand Complexity**: More lines of code can indicate more complexity, which might lead to more bugs and higher maintenance costs.

3. Cyclomatic complexity

The cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the Control Flow Graph of the program. The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if the second command might immediately follow the first command.

Use of Cyclomatic Complexity

- Determining the independent path executions thus proven to be very helpful for Developers and Testers.
- It can make sure that every path has been tested at least once.
- Thus help to focus more on uncovered paths.
- Code coverage can be improved.
- Risks associated with the program can be evaluated.

- These metrics being used earlier in the program help in reducing the risks.

4. Length of Identifiers

Length of Identifiers is a measure of the average length of distinct identifier in a program. The longer the identifiers, the more understandable the program. The length of identifiers metric helps in assessing the readability and maintainability of the code. Longer, descriptive names generally indicate better readability and understanding.

Importance of Length of Identifiers

Following are the importance of Length of Identifiers:

- **Readability:** Descriptive identifiers make the code easier to read and understand, which is crucial for maintenance and debugging.
- **Maintainability:** Well-named identifiers make it easier for future developers to understand the code without extensive documentation.
- **Consistency:** Consistent use of appropriately long identifiers reflects good coding practices and improves overall code quality.

5. Depth of conditional nesting

Depth of Nesting Condition is a measure of the depth of nesting of if statements in a program. Deeply nested statements are hard to understand and are potentially error-prone. It indicates how many levels deep the nested conditional statements (such as `if`, `else`, `while`, `for`, etc.) go. This metric is important because deeply nested code can be difficult to read, understand, and maintain.

Example of Depth of Conditional Nesting

```
if condition1:
if condition2:
if condition3:
# Code block
```

Importance of Conditional Nesting

- **Readability:** Deeply nested conditionals can make code hard to follow.
- **Maintainability:** Complex nesting increases the likelihood of introducing errors during code changes.
- **Testing:** More nested conditions can lead to an exponential increase in the number of test cases required to achieve thorough test coverage.
- **Refactoring:** Indicates areas of code that might benefit from refactoring to improve simplicity and readability.

6. Fog index

Fog Index is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document may be to understand.

Formula of calculating Fog Index:

Fog Index = 0.4 × (Number of Words / Number of Sentences + 100×Number of Complex Words / Number of Words)
where:

- ***Number of Words****: Total words in the text.*
- ***Number of Sentences****: Total sentences in the text.*
- ***Number of Complex Words****: Words with three or more syllables, not including proper nouns, familiar jargon, or compound words.*

Importance of Fog Index

Following are the importance of Fog Index

- **Documentation**: Ensuring that documentation is easy to read and understand is crucial for effective communication among developers, especially in large teams or open-source projects.
- **Code Comments**: Readable comments help developers understand the code quickly, which is essential for maintenance and debugging.
- **User Manuals**: For end-user documentation, a lower Fog Index ensures that the material is accessible to a broader audience.

Conclusion

Product metrics play a crucial role in software engineering by offering valuable insights into the code and overall software quality. They help identify potential issues early, guide improvements, and ensure that the software meets desired standards and performance criteria.