# Software Configuration Management

- Configuration management is a term that is widely used, often as a synonym for version control.
- **Configuration management** refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified.
- Your configuration management strategy will determine how you manage all of the changes that happen within your project.

- In Software Engineering, **Software Configuration Management**(SCM) is a process to systematically manage, organize, and control the changes in the documents, codes, and other entities during the Software Development Life Cycle.
- The primary goal is to increase productivity with minimal mistakes.
- The objective is to maintain software integrity and traceability throughout the software life cycle.

Configuration management is the process of preparing and maintaining all necessary elements required to build, deploy, test, and release an application. This involves two key aspects:

1. **Version Control and Dependency Management:** Ensuring all application code, related artifacts, and dependencies are organized, tracked, and versioned.
2. **Environment Configuration Management:** Managing the complete environment in which the application runs, including software, hardware, infrastructure, and dependencies such as operating systems, application servers, databases, and other supporting tools or commercial off-the-shelf (COTS) software.

- This approach ensures consistency, reliability, and repeatability across all stages of the software development lifecycle

# Processes involved in SCM

**Identification and Establishment**

Identifying the configuration itemsfrom products that compose baselines at given points intime(a base line 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 relationship among items, creating a mechanism to manage multiple level of control and procedure for change management system.

**Version control**

Creating versions/specifications of the existing product to build new products from the help of SCM system.

## Change control

A change request (CR) is submitted and evaluated to assess technical merit, potential side effects, 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.

## Configuration auditing

A software configuration audit complementstheformal technical review of the process and product. It focuses onthetechnicalcorrectness of the configuration object that has been modified. Theauditconfirms the completeness, correctness and consistency of items intheSCMsystem and track action items from the audit to closure.

## Reporting

Providing accurate status and current configuration data to developers, tester, end users, customers and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guide etc .

# Using Version Control

- **Version control systems**, also known as **source control**, **source code management systems**, or **revision control systems**, are a mechanism for keeping multiple versions of your files, so that when you modify a file you can still access the previous revisions.
- The aim of a version control system is twofold: **First**, it retains, and provides access to, every version of every file that has ever been stored in it. **Second**, it allows teams that may be distributed across space and time to collaborate.

***Keep Absolutely Everything in Version Control :-***

- The objective is to have everything that can possibly change at any point in the life of the project stored in a controlled manner.
- This allows you to recover an exact snapshot of the state of the entire system, from development environment to production environment, at any point in the project's history.
- It is even helpful to keep the configuration files for the development team's development environments in version control since it makes it easy for everyone on the team to use the same settings.

- Analysts should store requirements documents.
- Testers should keep their test scripts and procedures in version control.
- Project managers should save their release plans, progress charts, and risk logs here.
- In short, every member of the team should store any document or file related to the project in version control.

## Check In Regularly to Trunk

**Objective:**
To ensure continuous integration and maintain a functional, up-to-date version of the software.

**Frequent Commits:**

- Commit incremental changes to the trunk (main branch) regularly.
- Avoid holding large changes; integrate small, tested changes frequently.

**Benefits:**

- **Prevents Merge Conflicts:** Smaller, frequent updates reduce the risk of complex conflicts.
- **Ensures Integration:** Problems are identified and fixed immediately.
- **Higher Quality:** Continuous integration ensures the trunk is always in a working state.

**Automated Testing:**

- Continuous Integration (CI) servers run automated tests on the trunk after each check-in, ensuring stability.

**Example:**
When developing a new feature, break it into smaller chunks and commit changes after each functional step, such as completing the user interface or backend logic.

***Use Meaningful Commit Messages:-***

- Every version control system has the facility to add a description to your commit.
- It is easy to omit these messages, and many people get into the bad habit of doing so.
- The most important reason to write descriptive commit messages is so that, when the build breaks, you know who broke the build and why.
- A commit message explaining what the person was doing when they committed that change can save you hours of debugging.

# Version Control Workflow

1.All team members commit changes to the trunk (main branch).

2.CI server automatically builds and tests each commit.

3.Any failed tests or integration issues are flagged immediately.

**Example Workflow:**

- **Day 1:** Tester commits updated test cases.
- **Day 2:** Developer commits new feature code.
- **Day 3:** CI server identifies a configuration mismatch.
- **Outcome:** Issue fixed before deployment.

# Managing Dependencies

- The most common external dependencies within your application are the third party libraries it uses and the relationships between components or modules under development by other teams within your organization.
- ***Managing External Libraries:-*** External libraries usually come in binary form, unless you’re using an interpreted language. Even with interpreted languages, external libraries are normally installed globally on your system by a package management system.

- It is better to keep copies of your external libraries somewhere locally.
- This is essential if you have to follow compliance regulations, and it also makes getting started on a project faster.
- we emphasize that your build system should always specify the exact version of the external libraries that you use.
- If you don’t do this, you can’t reproduce your build.
- Whether you keep external libraries in version control or not involves some trade-offs. It makes it much easier to correlate versions of your software with the versions of the libraries that were used to build them.

# Managing Software Configuration

- Configuration is one of the three key parts that comprise an application, along with its binaries and its data.
- Configuration information can be used to change the behavior of software at build time, deploy time, and run time.
- Delivery teams need to consider carefully what configuration options should be available, how to manage them throughout the application’s life, and how to ensure that configuration is managed consistently across components, applications, and technologies.
- you should treat the configuration of your system in the same way you treat your code: Make it subject to proper management and testing.

- **Configuration and Flexibility**
- Everyone wants flexible software. But flexibility usually comes at a cost.
- Most applications they are designed for a specific purpose, but within the bounds of that purpose they will usually have some ways in which their behavior can be modified.
- The desire to achieve flexibility may lead to the common antipattern of “ultimate configurability” which is, all too frequently, stated as a requirement for software projects.
- It is at best unhelpful, and at worst, this one requirement can kill a project.
- In short, While flexibility is desirable, it should be purposeful and limited to the application's scope. Avoid the trap of making everything configurable, as it can lead to excessive complexity, unhelpful features, and even project failure.

- Configurable software is not always the cheaper solution it appears to be. It’s almost always better to focus on delivering the high-value functionality with little configuration and then add configuration options later when necessary.
- Most configuration information is free-form and untested.
- Configuration is not inherently evil. But it needs to be managed carefully and consistently.
- Modern computer languages have evolved all sorts of characteristics and techniques to help them reduce errors. . These protections are often absent in configuration files, making them more prone to mistakes

- In most cases, these protections do not exist for configuration information, and more often than not there are not even any tests in place to verify that your software has been configured correctly in testing and production environments.
- Deployment smoke tests are one way to mitigate this problem and should always be used.
- Smoke tests focus only on the **core functionalities** of the software, not every single feature.
- These tests confirm that the main parts of the software are functioning properly and are ready for more detailed testing.
- **Example**: A smoke test for an e-commerce website might verify that users can log in, search for products, and add items to their cart.

## *Types of Configuration*

Configuration information can be injected into your application at several points in your build, deploy, test, and release process, and it's usual for it to be included at more than one point.

- Your build scripts can pull configuration in and incorporate it into your binaries at **build time.**
- Your packaging software can inject configuration at **packaging time**, such as when creating assemblies, ears, or gems.
- Your deployment scripts or installers can fetch the necessary information or ask the user for it and pass it to your application at **deployment time** as part of the installation process.
- Your application itself can fetch configuration at **startup time or run time.**

- Whatever mechanism you choose, it is recommended that, as far as practically possible, you should try and supply all configuration information for all the applications and environments in your organization through the same mechanism.
- While this may not always be feasible, when it is, it ensures that all configuration is managed in one centralized location, making it easier to update, track changes, version-control, and override if needed
- In organizations where this practice isn’t followed, people regularly spend hours tracking down the source of some particular setting in one of their environments.

# Managing Your Environments (build and deployment)

- No application is an island.
- Every application depends on hardware, software, infrastructure, and external systems in order to work – refered as application’s environment.
- The principle to bear in mind when managing the environment that your application runs in is that the configuration of that environment is as important as the configuration of the application.
- For example, if your application depends on a messaging bus, the bus needs to be configured correctly or the application will not work.

- The problems can be summed up as follows:

1. One small change can break the whole application or severely degrade its performance.
2. Once it is broken, finding the problem and fixing it takes an indeterminate amount of time and requires senior personnel.
3. It is extremely difficult to precisely reproduce manually configured environments for testing purposes.
4. The collection of configuration information is very large.
5. It is difficult to maintain such environments without the configuration, and hence behavior, of different nodes drifting apart.

- In order to reduce the cost and risk of managing environments, it is essential to turn our environments into mass-produced objects whose creation is repeatable and takes a predictable amount of time.
- The key to managing environments is to make their creation a fully automated process.
- It should always be cheaper to create a new environment than to repair an old one.

- Being able to reproduce your environments is essential for several reasons:-

1. It removes the problem of having random pieces of infrastructure around whose configuration is only understood by somebody who has left the organization and cannot be reached. When such things stop working, you can usually assume a significant downtime. This is a large and unnecessary risk.
2. Fixing one of your environments can take many hours. It is always better to be able to rebuild it in a predictable amount of time so as to get back to a known good state.
3. It is essential to be able to create copies of production environments for testing purposes. In terms of software configuration, testing environments should be exact replicas of the production ones, so configuration problems can be found early.

- The kinds of environment configuration information you should be concerned about are:

1. The various operating systems in your environment, including their versions, patch levels, and configuration settings.
2. The additional software packages that need to be installed on each environment to support your application, including their versions and configuration.
3. The networking topology required for your application to work.
4. Any external services that your application depends upon, including their versions and configuration.
5. Any data or other state that is present in them (for example, production databases)

- There are **two principles** that form the basis of an effective configuration management strategy:

1. Keep binary files independent from configuration information, and
2. keep all configuration information in one place.

- Applying these fundamentals to every part of your system will pave the way to the point where creating new environments, upgrading parts of your system, and rolling out new configurations without making your system unavailable becomes a simple, automated process.

- When evaluating third-party products and services, start by asking the following questions:

1. Can we deploy it?

2. Can we version its configuration effectively?

3. How will it fit into our automated deployment strategy?

- Maintain a baseline representing the properly deployed state of your environment.
- Version control any changes to the environment and associate them with application versions for consistency.
- **Best Practices:**
- Treat environments like code: incrementally change, version control, and test continuously
- Perform integration and testing early and often

# Benefits of Configuration Management

Configuration Management provides the following benefits:

1. Allowing configuration to be version controlled
2. Detecting and correcting configuration drift
3. Treating infrastructure as flexible resource
4. Facilitating automation
5. Enabling automated scale-up and scale-out
6. Providing environment consistency