***Configuration Management***

Configuration management refers to the process by which all artifacts relevant to

the project, and the relationships between them, are stored, retrieved, uniquely

identified, and modified.

Configuration management strategy will determine how we manage all of the changes that happen within the project. It thus records the evolution of our systems and applications. It will also govern how your team collaborates.

**Advantages of Configuration Management**

1. We can exactly reproduce any of the environments, including the version of the operating system, its patch level, the network configuration, the software stack, the applications deployed into it, and their configuration.
2. We can easily make an incremental change to any of these individual items and deploy the change to any, and all, of my environments.
3. We can easily see each change that occurred to a particular environment and trace it back to see exactly what the change was, who made it, and when they made it.
4. We can satisfy all of the compliance regulations that are present

Configuration Management can be achieved by

1. ***Using Version Control***

2. ***Managing Dependencies***

3. ***Managing Software Configurations***

1. ***Managing Environments***
2. ***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. They are also a mechanism through which people involved in software

delivery collaborate.

The first popular version control system was a proprietary UNIX tool called

SCCS (Source Code Control System). This was superseded by RCS, the Revision Control System, and later CVS, Concurrent Versions System. All three of these systems are still in use today. Subversion, Mercurial and Git are open source systems that we use now.

The aim of a version control system is twofold:

1.It retains, and provides access to, every version of every file that has ever been stored in it. Such systems also provide a way for **metadata**—that is, information that describes the data stored—to be attached to single files or collections of files.

2.it allows teams that may be distributed across space and time to collaborate.

To make the effective use of version control we have to do the following:

1. ***Keep Absolutely Everything in Version Control***

* Version control isn’t just for source code. Every single artifact related to the creation of your software should be under version control.
* Developers should use it for source code, of course, but also for tests, database scripts, build and deployment scripts, documentation, libraries and configuration files for your application, your compiler and collection of tools, and so on—so that a new member of your team can start working from scratch.
* 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.

2. ***Check In Regularly to Trunk***

* + We have to enter contents into version control so that we can step back to a recent known-good version of your artifacts, it is important to check in frequently.
  + Once you check your changes into version control, they become public, instantly available to everybody else on the team.
  + Since checking in is a form of publication, it is important to be sure that your work, whatever it may be, is ready for the level of publicity that a check-in implies. This applies to developers in particular who, given the nature of their work, need to be cautious about the effects of their check-ins.
  + A solution that some people use to resolve this dilemma is to create a separate branch within the version control system for new functionality. At some point, when the changes are deemed satisfactory, they will be merged into the main development branch.
  + To ensure you aren’t going to break the application when you check in, two practices are useful. One is to run your commit test suite before the check-in. This is a quick-running (less than ten minutes) but relatively comprehensive set of tests which validate that you haven’t introduced any obvious regressions. Many continuous integration servers have a feature called pretested commit which allows you to run these tests on a production-like environment before you check in.
  + The second is to introduce changes incrementally. commit changes to the version control system at the conclusion of each separate incremental change or refactoring.

***3.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. A couple of medium-to-long sentences with an overview of what you were doing will often save you many

times the effort later on.

We can use multi paragraph commit message in which the first paragraph is a summary and the following paragraphs add more detail.

1. ***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***

We can 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. If you keep it in version control, it makes it much easier to correlate versions of your software with the versions of the libraries that were used to build them.

***Managing Components***

Splitting application into components reduces regression bugs, encourages reuse and enables a much more efficient development process on large projects. If there is only few components checked out it should be relatively straightforward to make changes in some of them and run a single command that rebuilds the necessary bits in the right order, creates the appropriate binaries, and runs relevant tests.

1. ***Managing Software Configuration***

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.

***Configuration and Flexibility***

All applications 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.“Configurability” is stated as a requirement for software projects. Changing software configuration is very difficult to handle. There are not even any tests in place to verify that software has been configured correctly in testing and production environments. It is almost always better to focus on delivering the high-value functionality with little configuration and then add configuration options later when necessary.

***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.

It is usually important to be able to configure your application at deployment time so that you can tell it where the services it depends upon (such as database, messaging servers, or external systems) belong. For example, if the runtime configuration of your application is stored in a database, you may want to pass the database’s connection parameters to the application at deployment time so it can retrieve it when it starts up.

We should supply all configuration information for all the applications and environments in your organization through the same mechanism.

***Managing Application Configuration***

Configuration information is often modeled as a set of name-value strings.

Sometimes it is useful to use types in your configuration system and to organize

it hierarchically.

Version control is used to store configuration information .We can just check in your configuration file, and we get the history of your configuration over time for free. We should keep a list of the available configuration options for your application in the same repository as its source code.

It is often important to keep the actual configuration information specific to each of your application’s testing and production environments in a repository separate from your source code. This information generally changes at a different rate to other version-controlled artifacts. If we do this route, you will have to be careful to track which versions of configuration information match with which versions of the application. This separation is particularly relevant for security-related configuration elements, such as passwords and digital certificates, to which access should be restricted.

***Modeling Configuration***

Each configuration setting can be modeled as a tuple, so the configuration for an application consists of a set of tuples. The set of the tuples available and their values typically depend on three things:

1. The application
2. The version of the application
3. The environment it runs in

Eg: version 1.0 of your reporting application will have a set of tuples different from version 2.2, or from version 1.0 of your portfolio management application. The values of those tuples will, in turn, vary depending on the environment they are deployed into.

***Testing System Configuration***

In the same way that your application and build scripts need testing, so do your configuration settings. There are two parts to testing configuration.

1. The first stage is to ensure that references to external services in your configuration settings are good. You should, as part of your deployment script, ensure that the messaging bus you are configured to use is actually up and running at the address configured, and that the mock order fulfillment service your application expects to use in the functional testing environment is working.
2. The second stage is to actually run some **smoke tests** once your application is installed to make sure it is operating as expected. This should involve just a few tests exercising functionality that depends on the configuration settings being correct. Ideally, these tests should stop the application and fail the installation or deployment process if the results are not as expected.

***Smoke Test:*** It checks the core functionality of a program, to ensure that the program is ready for further testing. This prevents a QA team from attempting to run a full test of software that can't complete basic functions.

***Managing Configuration across Applications***

The problem of managing configuration is particularly complex in medium and

large organizations where many applications have to be managed together. One of the most important tasks is to keep a catalogue of all the configuration options that each of your applications has, where they are stored, what their lifecycle is, and how they can be changed. It is especially important to have access to this information on a real-time basis when your applications depend on each other and deployments must be orchestrated.

***Principles of Managing Application Configuration***

1. Consider where in your application’s lifecycle it makes sense to inject a

particular piece of configuration—at the point of assembly where you are Managing Software Configuration **47** packaging your release candidate, at deployment or installation time, atstartup time, or at run time. Speak to the operations and support team towork out what their needs are.

1. Keep the available configuration options for your application in the same

repository as its source code, but keep the values somewhere else. Configuration settings have a lifecycle completely different from that of code, while passwords and other sensitive information should not be checked to version control at all.

1. Configuration should always be performed by automated processes using

values taken from your configuration repository, so that you can always identify the configuration of every application in every environment.

4. Your configuration system should be able to provide different values to your application (including its packaging, installation, and deployment scripts) based on the application, its version, and the environment it is being deployed into. It should be easy for anyone to see what configuration options are available for a particular version of an application across all environments it will be deployed into.

5. Use clear naming conventions for your configuration options. Avoid obscure

or cryptic names. Try to imagine someone reading the configuration file without a manual—it should be possible to understand what the configuration properties are.

6. Ensure that your configuration information is modular and encapsulated so that changes in one place don’t have knock-on effects for other, apparently unrelated, pieces of configuration.

7. Use the DRY (don’t repeat yourself) principle. Define the elements of your

configuration so that each concept has only one representation in the set of configuration information.

8. Be minimalist: Keep the configuration information as simple and as focused

as possible. Avoid creating configuration options except where there is a requirement or where it makes sense to do so.

9. Avoid over engineering the configuration system. Keep it as simple as you can.

10. Ensure that you have tests for your configuration that are run at deployment or installation time. Check that the services your application depends upon are available, and use smoke tests to assert that any functionality depending on your configuration settings works as it should.

***5.Managing Your Environments***

Every application depends on hardware, software, infrastructure, and external systems in order to work. 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. If, for example, your application depends on a messaging bus, the bus needs to be configured correctly or the application will not work. Your operating system’s configuration is also important. For example, you may have an application that relies on a large number of file descriptors being available. If the operating system defaults to a lower limit for the number of file descriptors, your application won’t work.

**Problems in configuration management:**

1. The collection of configuration information is very large.
2. One small change can break the whole application or severely degrade its

performance.

1. Once it is broken, finding the problem and fixing it takes an indeterminate

amount of time and requires senior personnel.

1. It is extremely difficult to precisely reproduce manually configured

environments for testing purposes.

1. It is difficult to maintain such environments without the configuration, and

hence behavior, of different nodes drifting apart.

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.

The kinds of environment configuration information we should be concerned

about are:

• The various operating systems in your environment, including their versions, patch levels, and configuration settings

• The additional software packages that need to be installed on each environment

to support your application, including their versions and configuration

• The networking topology required for your application to work

• Any external services that your application depends upon, including their versions and configuration

• Any data or other state that is present in them (for example, production

databases)

**There are two principles** that, as we have found, form the basis of an effective configuration management strategy:

1. Keep binary files independent from configuration information
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.

An environment that is in a properly deployed state is known as a ***baseline***in configuration management terminology. Your automated environment provisioning

system should be able to establish, or reestablish, any given baseline that has existed in the recent history of your project. Any time you change any aspect of the host environment of your applications, you should store the change, creating a new version of the baseline and associating that version of the application with the new version of the baseline. We should treat your environment the same way you treat your code—changing it incrementally and checking the changes into version control. Every change should be tested to ensure that it doesn’t break any of the applications that run in the new version of the environment.

***Tools to Manage Environments***

Puppet and CfEngine are two examples of tools that make it possible to manage operating system configuration in an automated fashion.

Virtualization can also improve the efficiency of the environment management

process. Instead of creating a new environment from scratch using an automated

process, you can simply take a copy of each **box** in your environment and store

it as a baseline. Then it is trivial to create new environments—it can be done by

clicking a button.

***Managing the change process***

A production environment should be completely locked down. It should not be possible for *anybody* to make a change to it without going through your organization’s change management process. Even a tiny change could break it. A change must be tested before it goes into production, and for that it should be scripted and checked into version control.

Then, once the change has been approved, it can be rolled out to the production

environments in an automated fashion.

A change to the environment is just like a change to software. It has to go through your build, deploy, test, and release process in exactly the same way as a change to the application’s code. Testing environments should be treated the same as production environments. They should be managed, deployed to, and configured by the same mechanisms.