**Continuous Integration (*CI*)**

Continuous Integration (***CI***) is a software development practice, where developers will ***commit*** (integrate) their code changes to a shared repository frequently.

* Enabling quick verification of the changes at an early stage of the development cycle, rather than waiting to detect bugs after the code is completely developed.
* ***Changes*** will be typically **small**.
* ***Errors*** can be ***detected quickly***.
* ***Pointing*** out the ***change*** that introduced an ***error*** can be ***done quickly***.

CI Services

Continuous Integration includes the following:

* Source Code Control
* Code Compile
* Integrate Database Changes
* Run Tests
* Code Inspection

***version control system*** (VCS).

***Source code control*** is the heart of Continuous Integration.  
Source code must be managed using a ***version control system*** (VCS).

The different types are:

* ***Local***
* ***Centralized***
* ***Distributed***

**Mainline Branch**

* Mainline branch must be stable always, so that code is in Ready to deploy state.
* Ready to deploy implies, that the code has successfully passed tests like integration, regression etc.
* Code in mainline is deployed to user or production environments.

**Release Branch**

* Change done on release branch must flow back to mainline.
* Release branch must never receive a change from mainline.
* Release branch must be closed after a release from the mainline.

**Work Branch**

* Work or development branch is where the developer compiles the code, integrates and runs tests.
* Stable changes in work branches are published to the mainline.

**Release Branch Problem**

* A high priority bug got detected post user-release, **what has to be done now?**
* Create a release branch from the mainline based on the time it was released.
* Fix the bug on the release branch.
* Merge the changes from the release branch to the mainline.

**Stream Based Version Control**

* Developers develop code in their own workspaces.
* Changes are promoted to streams once they are ready.
* Branch is replaced by streams.
* A change applied to a stream will be automatically inherited by the downstream streams.
* Merge problems addressed by this automatic inheritance.

**Stream based version control is helpful in below scenarios:**

* Applying a bug fix to several versions of the application.
* Adding new version of a third-party library to the codebase.

**How is it done?**

* Promote the changes in your stream to the common ancestor of all the streams that needs the change.
* IBM Clearcase and AccuRev are popular stream based version control systems.

**What is a Build?**

Activities performed to **generate, test, inspect and deploy software.**

**Build Scripts**

As soon as a change is moved in to version control, it is a recommended CI practice to build immediately.

* Create build scripts (use tools like Maven, Ant)
* Execute build scripts from an IDE or command line or CI tool.
* Build scripts should not be dependent on the IDE which means it must be executable from the CI tool, if used.

Use CI tools to automate the trigger and execution of build scripts on detecting a change.

**Build is of 3 types:**

* Private Build
* Integration Build
* Release Build

**Private Build**

Run by the developer before committing the code changes to the local version control or work branch.

**Sequence of activities done by a developer:**

* Check out the code from work branch.
* Do the required code changes.
* Get the latest system changes (like database changes) from central repository.
* Run build jobs that includes execution of unit test cases.
* On successful build, commit changes to the work branch.

**Integration Build**

* Integrates changes committed on the work branch with the mainline.
* Ideal to run integration build on a dedicated machine.
* Code compilation, unit test, component test, system and performance test along with inspections are executed as part of this build.

**Release Build**

* Deploys code to production or end-user.
* Includes extensive load and performance tests along with user acceptance tests.
* Release build will be triggered by code changes on mainline or release branch.

**Build Mechanism**

* On-Demand: Build initiated manually.
* Scheduled: Build triggered based on time.
* Poll for Changes: Build runs after a change is detected by a CI tool.
* Event-Driven: Build triggered by version control tool based on a change.

**Build for any environment**

In order to enable software deployment to different environments, say, for testing purposes, code must be deployed to various environments.

* Build scripts should remain the same.
* Configuration files (like .properties or .include) helps in differentiating the environments.

**Run Fast Builds**

Rapid feedback is a key factor of CI.

If the build runs for a longer duration, it may delay the reporting and fixing of errors. So stage the builds (split builds logically).

**Create separate build scripts:**

* To fetch changes from version control and compile.
* Run unit tests.
* Run automated processes, like, integrate the database changes, component tests or system tests and code inspection.

Do incremental builds (compile only the components changed) when compilation takes a longer duration. But, use it judiciously.

**Integrate Database Changes**

Any time a database is changed, such as:

* New objects are created.
* Existing objects are altered.
* Objects are dropped or removed.

Build the components that are using the database.

As a recommended practice:

* Incorporate database integration as part of build.
* Use a local sandbox to test database changes.
* Share the artifacts, like scripts to create, modify, delete, schema etc., using a central repository.

**Test Types**

You might have come across this quote: A code that cannot be tested is flawed.

As an important CI practice, you need to execute:

* Unit test (use tools like Junit, Nunit, PHPunit as appropriate).
* Integration or Component test, to verify how certain changes interact with the rest of the system (use tools like Junit, Nuint, DbUnit).
* System test, to completely test a software (Use tools like JWebunit).
* Functional test, to test functionality of a software from user perspective (tools like Selenium are used).

Automate the tests using the tools provided.

**Testing Strategy**

* Categorize your tests (unit, system, and component).
* Create test scripts and include as part of build.
* Ensure proper test code coverage.
* Schedule the build based on the test category. Different intervals needs to be planned for slow running tests.
* Prioritize and run faster tests.

**Inspection Activities**

Code review plays a crucial role in maintaining overall quality of a code base. This must be part of the build.

* Leverage automated inspectors like JavaNCSS or CCMetrics to identify piece of the code that is highly complex.

Complexity of code is determined by Cyclomatic Complexity Number (CCN)

It is the measure of :

* number of linearly independent paths (or)
* Number of ways present to traverse a piece of code.

It determines minimum number of test inputs required to check all the ways of executing a program (done using control flow graph).

* Report coding standards violation (using PMD, FxCop etc.).
* Identify amount of duplicate code (Use tools like Simian, CPD).
* Assess code coverage, identify percentage of code executed on running a test (using tools like NCover, Cobertura, and Clover).
* Determine if a package is highly dependent on other packages, measured using

\*\*\*Afferent and Efferent coupling\*\*\*.