Moi Programming

Friday, 1 March 2019

Industry practices and tools 2

Discuss the importance of maintaining the quality of the code, explaining the different aspects of the code quality

Why Is Good Quality Code So Important? Good quality code is an essential property of a software because it could lead to financial losses or waste of time needed for further maintenance, modification or adjustments if code quality is not good enough.

Characteristics Of Good Quality Code

- ¬ **Efficiency**
- ¬ **Reliability**
- ¬ **Robustness**
- ¬ **Portability**
- ¬ **Maintainability**
- ¬ **Readable**

EFFICIENCY o Directly related to the performance and speed of running the software. WHY IMPORTANT ? o The quality of the software can be evaluated with the efficiency of the code used. o No one likes to use a software that takes too long to perform an action. HOW? o Remove unnecessary or redundant code o Write reusable code o Reduce resource consumption o Use appropriate data types, functions and looping in appropriate place

RELIABILITY o Ability to perform consistent and failure-free operations every time it runs. WHY IMPORTANT ? o The software would be very less useful if the code function differently every time it runs even with the same input in same environment and if it breaks down often without throwing any errors. HOW? o Take much time to review and test the code carefully and thoroughly in every possible way. o Use proper error and exception handling.

ROBUSTNESS o Ability to cope with errors during program execution even under unusual condition. WHY IMPORTANT ? o Image how would you feel when you use a software that keep showing strange and unfamiliar message when you did something wrong. o Software is typically buggy and fragile but it should handle any errors encountered gracefully. HOW? o Test software from every condition; both usual and unusual. o Use proper error and exception handling. o Provide clear and understandable error messages to allow user to more easily debug the program.

PORTABILITY o Ability of the code to be run on as many different machines and operating systems as possible. WHY IMPORTANT ? o It would be a waste of time and energy for programmers to re-write the same code again when it transferred from one environment to another. HOW? o Start from the very beginning, write code that could work on every possible environment.

MAINTAINABILITY o Code that is easy to add new features, modify existing features or fix bugs with a minimum of effort without the risk of affecting other related modules . WHY IMPORTANT ? o Software always needs new features or bug fixes. So the written code must be easy to understand, easy to find what needs to be change, easy to make changes and easy to check that the changes have not introduced any bugs. HOW? o Good naming of variable, method and class names o Use of proper indentation and formatting style o Good technical documentation o Write appropriate comment or summary descriptions at the top of files, classes or functions

READABLE o The ability of allowing the code to be easily, quickly, and clearly understandable by someone new or someone that hasn't seen it in a while. oWHY IMPORTANT ? o Ensures that everyone can understand the code written by everyone else. o If the code is messy and badly written, it would be very hard to understand what the code does and where changes need to be made. o This could waste much time trying to figure out how it all fits together before making any action and even end up re-writing the again assuming that it is buggy and carelessly written. HOW? o Use proper variable, method and class names o Use consistent indentation and formatting style o Write appropriate comment or summary descriptions at the top of files, classes or functions

About Me

Moi Programming

View my complete profile

Blog Archive

March 2019 (1)

February 2019 (2)

Report Abuse

Search This Blog

Se

Home

Industry practices and tool

Popular Posts

Programming Applications and Frameworks Compare and contrast declarative and imperative paradigms

Declarative Programmin

Explain different approaches and measurements used to measure the quality of code

There's no one right or wrong way to measure code quality.

You could apply various metrics to the code. There are tools to report many of these metrics:

Count of open reported defects in a given product.

Defect density. Take the number of defects found per the number of source lines of code in the product. Lower is better. However, this metric does ignore defects that haven't been recorded.

Fan-in and Fan-out. Fan-in is the number of modules that a given module calls. Fan-out is the number of modules that call the module.

Coupling. Consider the number of inputs, outputs, global variables used, module, fan-in, and fan-out. Wikipedia provides a formula to compute coupling.

Cyclomatic complexity. This measures the number of paths through a given block of code. The cyclomatic complexity for a block is the upper bound of tests to achieve complete branch coverage. If all paths through the code are actually possible, then this is also the upper bound on test cases needed for path coverage.

Halstead complexity measures of program vocabulary, program length, calculated program length, volume, difficulty, and effort. The difficulty is especially useful as it is a representation of how complex the code is to understand. There is also a calculation for the estimated number of bugs in an implementation.

Count of open static or dynamic analysis findings. Various tools exist to examine the source code, binary files, and execution paths of software to find possible errors automatically. These findings can be reported as a measure of quality.

You can also take a qualitative approach. Sometimes, the best measure of code quality is to ask someone to look at it and comment on it. This is easiest if you also have a style guide and consistent rules for how you write your code (from formatting through naming conventions). If you want to know about how readable or maintainable your code is, sometimes the best thing to do is to just ask someone else.

Identify and compare some available tools to maintain the code quality

- **Collaborator**
- **Review Assistant**
- **Codebrag**
- **Gerrit**
- **Codestriker**
- **Rhodecode**
- **Phabricator**
- **Crucible**
- **Veracode**
- **Review Assistant**
- **Review Board**

Discuss the need for dependency/package management tools in software development?

Bower

The package management system Bower runs on NPM which seems a little redundant but there is a difference between the two, notably that NPM offers more features while Bower aims for a reduction in filesize and load times for frontend dependencies.

Some devs argue that Bower is basically obsolete since it runs on NPM, a service that can do almost everything Bower can do. Generally speaking this isn't wrong.

But devs should realize Bower can optimize the workflow specifically with frontend dependencies. I recommend Ben McCormick's article Is Bower Useful to learn more about the value offered from both package management tools.

RubyGems

RubyGems is a package manager for Ruby with a high popularity among web developers. The project is open source and inclusive of all free Ruby gems.

To give a brief overview for beginners, a "gem" is just some code that runs on a Ruby environment. This can lead to programs like Bundler which manage gem versions and keep everything updated. Rails developers will love this feature, but what about frontend packages?
Since Ruby is open source, developers can build projects like Bower for Rails. This brings frontend package management to the Ruby platform with a small learning curve.

RequireJS

There's something special about RequireJS in that it's primarily a JS toolset.
It can be used for loading JS modules quickly including Node modules.
RequireJS can automatically detect required dependencies based on what you're using so this might be akin to classic software programming in C/C++ where libraries are included with further libraries.

Jam

Browserbased package management comes in a new form with JamJS. This is a JavaScript package manager with automatic management similar to RequireJS.
All your dependencies are pulled into a single JS file which lets you add and remove items quickly. Plus these can be updated in the browser regardless of other tools you're using (like RequireJS).

Browserify

Most developers know of Browserify even if it's not part of their typical workflow. This is another dependency management tool which optimizes required modules and libraries by bundling them together.
These bundles are supported in the browser which means you can include and merge modules with plain JavaScript. All you need is NPM to get started and then Browserify to get moving.

Mantri

Still in its early stages of growth, MantriJS is a dependency system for midtohigh level web applications. Dependencies are managed through namespaces and organized functionally to avoid collisions and reduce clutter.

Volo

The project management tool volo is an open source NPM repo that can create projects, add libraries, and automate workflows.
Volo runs inside Node and relies on JavaScript for project management. A brief intro guide can be found on GitHub explaining the installation process and common usage. For example if you run the command volvo create you can affix any library like HTML5 Boilerplate

Explain the role of dependency/package management tools in software development

- Build tools are programs that automate the creation of executable applications from source code.
- In small projects, developers will often manually invoke the build process.This is not practical for larger projects, where it is very hard to keep track of what needs to build.
- In what sequences and what dependencies there are in the building process. Using an automation tool allows the build process to be more consistent.

Compare and contrast different dependency/package management tools used in industry

• **Gradle**

Gradle may be a project automation tool that builds upon the ideas of Apache ant and Apache maven and introduces a Groovy-based domain-specific language.

• **Azure DevOps**

The Azure DevOps Project presents a simplified expertise wherever you bring your existing code and git repository or make a choice from one among st the sample applications.

• **Maven**

Maven may be a build automation tool used primarily for Java comes. The word maven means that 'accumulator of knowledge' in German. maven addresses two aspects.

What is a build tool? Indicate the significance of using a build tool in large scale software development, distinguishing it from small scale software development

Building life cycle refers to the read of a building over the course of its entire life - in alternative words, viewing it not even as associate degree operational building, however additionally taking under consideration the look, construction, operation, demolition and waste treatment.[1] it's helpful to use this read once (attempting|trying|making associate degree attempt) to boost an operational feature of a building that's associated with however a building was designed. for instance, overall energy conservation. within most cases there's but comfortable effort place into coming up with a building to be energy economical and thus massive inefficiencies are incurred in the operational section. Current analysis is current in exploring ways of incorporating an entire life cycle read of buildings, instead of simply that specialize in the operational section as is that the current scenario.

• Validate: validate the project is correct and all necessary information is available.

• Compile: compile the source code of the project.

• Test: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.

• Package: take the compiled code and package it in its distributable format, such as a JAR.

• Integration-test: process and deploy the package if necessary, into an environment where integration tests can be run.

• Verify: run any checks to verify the package is valid and meets quality criteria.

• Install: install the package into the local repository, for use as a dependency in other projects locally.

• Deploy: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Explain the role of build automation in build tools indicating the need for build automation

Build Lifecycle Basics

Maven is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

There are three builtin build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's site documentation.

A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

validate validate the project is correct and all necessary information is available compile compile the source code of the project test test the compiled source code using a suitable unit testing framework. These tests should

not require the code be packaged or deployed

package take the compiled code and package it in its distributable format, such as a JAR.

verify run any checks on results of integration tests to ensure quality criteria are met

install install the package into the local repository, for use as a dependency in other projects

locally deploy done in the build environment, copies the final package to the remote repository for

sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

Usual Command Line Calls

In a development environment, use the following call to build and install artifacts into the local repository.

This command executes each default life cycle phase in order (validate, compile, package, etc.), before executing install. You only need to call the last build phase to be executed, in this case, install:

In a build environment, use the following call to cleanly build and deploy artifacts into the shared repository.

The same command can be used in a multimodule scenario (i.e. a project with one or more subprojects). Maven traverses into every subproject and executes clean, then executes deploy (including all of the prior build phase steps).

A Build Phase is Made Up of Plugin Goals

However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the plugin goals bound to those build phases.

A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation. The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked.

For example, consider the command below. The clean and package arguments are build phases, while the dependency:copy-dependencies is a goal (of a plugin).

If this were to be executed, the clean phase will be executed first (meaning it will run all preceding phases of the clean lifecycle, plus the clean phase itself), and then the dependency:copydependencies goal, before finally executing the package phase (and all its preceding build phases of the default lifecycle).

Moreover, if a goal is bound to one or more build phases, that goal will be called in all those phases.

Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals

(Note: In Maven 2.0.5 and above, multiple goals bound to a phase are executed in the same order as they are declared in the POM, however multiple instances of the same plugin are not supported. Multiple instances of the same plugin are grouped to execute together and ordered in Maven 2.0.11 and above).

Some Phases Are Not Usually Called From the Command Line

The phases named with hyphenatedwords (pre-*, post-*, or process-*) are not usually directly called from the command line. These phases sequence the build, producing intermediate results that are not useful outside the build. In the case of invoking integration-test, the environment may be left in a hanging state.

Code coverage tools such as Jacoco and execution container plugins such as Tomcat, Cargo, and Docker bind goals to the pre-integration-test phase to prepare the integration test container environment. These plugins also bind goals to the post-integration-test phase to collect coverage statistics or decommission the integration test container.

Failsafe and code coverage plugins bind goals to integration-test and verify phases. The net result is test and coverage reports are available after the verifyphase. If integration-test were to be called from the command line, no reports are generated. Worse is that the integration test container environment is left in a hanging state; the Tomcat webserver or Docker instance is left running, and Maven may not even terminate by itself

Compare and contrast different build tools used in industry

A Build Lifecycle is Made Up of Phases. Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle. … deploy – done in the build environment, copies the final package to the remote repository for sharing with other developers and projects

Explain the build life cycle, using an example (java, .net, etc…)

Maven configuration occurs at 3 levels:

Project most static configuration occurs in pom.xmlInstallation this is configuration added once for a Maven installationUser this is configuration specific to a particular userThe separation is quite clear - the project defines information that applies to the project, nomatter who is building it, while the others both define settings for the current environment.The installation and user configuration cannot be used to add shared project information forexample, setting <organization> or <distributionManagement> company wide.For this, you should have your projects inherit from a companywide parent pom.xml.

You can specify your user configuration in ${user.home}/.m2/settings.xml . A full reference to the configuration file is available. This section will show how to make some common configurations. Note that the file is not required defaults will be used if it is not found.

What is Maven, a dependency/package management tool or a build tool or something more?

Convention over configuration (also known as coding by convention) is a software design paradigm used by software frameworks that attempts to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility. The concept was introduced by David Heinemeier Hansson to describe the philosophy of the Ruby on Railsweb framework, but is related to earlier ideas like the concept of “sensible defaults” and the principle of least astonishment in user interface design.

Discuss how Maven uses conventions over configurations, explaining Maven’s approach to manage the configurations

Maven is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired

There are three built-in build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project’s site documentation.

on March 01, 2019

No comments:

Post a Comment