PART I.IV

Version Control

@huangyz0918

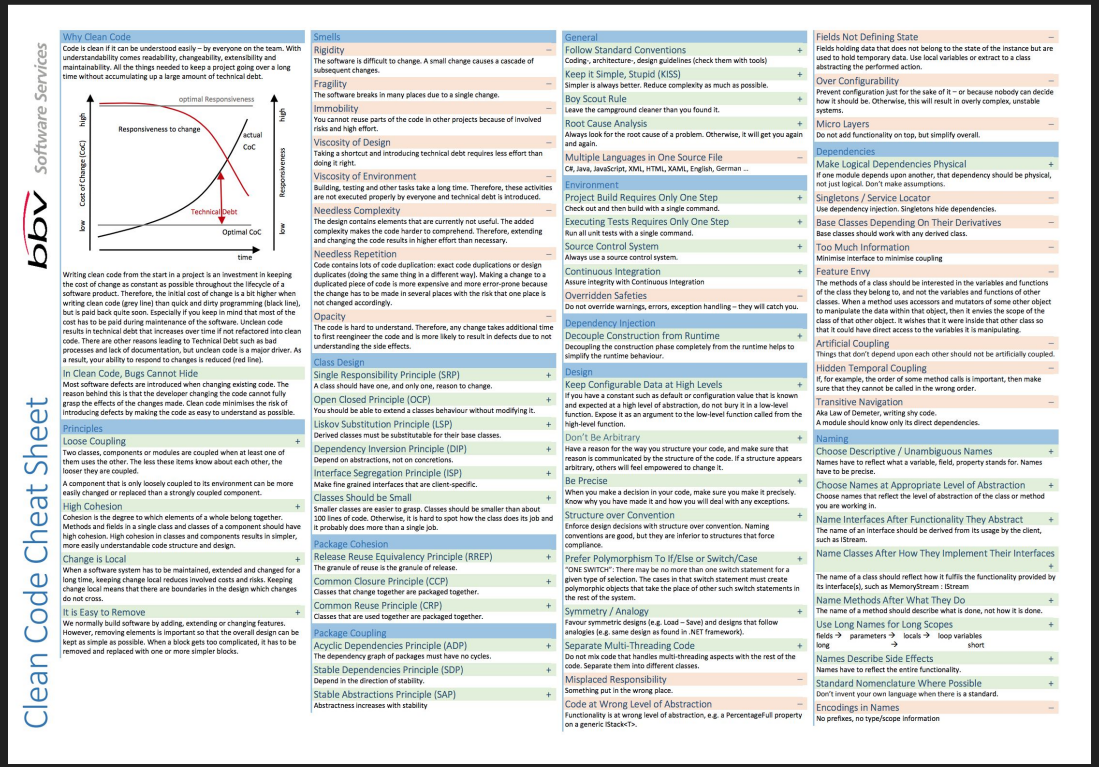
Git Basic Review

- Initializing a Repository in an Existing Directory
- Cloning an Existing Repository
- Recording Changes to the Repository
- Checking the Status of Your Files
- Tracking New Files
- Ignoring Files
- Committing Your Changes

Git Basic Review

- Working with Remotes
- Tagging
- Basic Branching and Merging
- Branch Management
- Git Workflow
- Remote Branches
- Rebasing

Clean Code Rules



Clean Code Cheat Sheet

bbv Software Services

Why Clean Code

Code is clean if it can be understood easily – by everyone on the team. With understandability comes readability, changeability, extensibility and maintainability. All the things needed to keep a project going over a long time without accumulating up a large amount of technical debt.

Writing clean code from the start in a project is an investment in keeping the cost of change as constant as possible throughout the lifecycle of a software product. Therefore, the initial cost of change is a bit higher when writing clean code (grey line) than quick and dirty programming (black line), but is paid back quite soon. Especially if you keep in mind that most of the cost has to be paid during maintenance of the software. Unclean code results in technical debt that increases over time if not refactored into clean code. There are other reasons leading to Technical Debt such as bad processes and lack of documentation, but unclean code is a major driver. As a result, your ability to respond to changes is reduced (red line).

In Clean Code, Bugs Cannot Hide

Most software defects are introduced when changing existing code. The reason behind this is that the developer changing the code cannot fully grasp the effects of the changes made. Clean code minimises the risk of introducing defects by making the code as easy to understand as possible.

Principles

Loose Coupling +

Two classes, components or modules are coupled when at least one of them uses the other. The less these items know about each other, the looser they are coupled.

A component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

High Cohesion +

Cohesion is the degree to which elements of a whole belong together. Methods and fields in a single class and classes of a component should have high cohesion. High cohesion in classes and components results in simpler, more easily understandable code structure and design.

Change is Local +

When a software system has to be maintained, extended and changed for a long time, keeping change local reduces involved costs and risks. Keeping change local means that there are boundaries in the design which changes do not cross.

It is Easy to Remove +

We normally build software by adding, extending or changing features. However, removing elements is important so that the overall design can be kept as simple as possible. When a block gets too complicated, it has to be removed and replaced with one or more simpler blocks.

Smells

Rigidity –

The software is difficult to change. A small change causes a cascade of subsequent changes.

Fragility –

The software breaks in many places due to a single change.

Immobility –

You cannot reuse parts of the code in other projects because of involved risks and high effort.

Viscosity of Design –

Taking a shortcut and introducing technical debt requires less effort than doing it right.

Viscosity of Environment –

Building, testing and other tasks take a long time. Therefore, these activities are not executed properly by everyone and technical debt is introduced.

Needless Complexity –

The design contains elements that are currently not useful. The added complexity makes the code harder to comprehend. Therefore, extending and changing the code results in higher effort than necessary.

Needless Repetition –

Code contains lots of code duplication: exact code duplications or design duplicates (doing the same thing in a different way). Making a change to a duplicated piece of code is more expensive and more error-prone because the change has to be made in several places with the risk that one place is not changed accordingly.

Opacity –

The code is hard to understand. Therefore, any change takes additional time to first reengineer the code and is more likely to result in defects due to not understanding the side effects.

Class Design

Single Responsibility Principle (SRP) +

A class should have one, and only one, reason to change.

Open Closed Principle (OCP) +

You should be able to extend a classes behaviour without modifying it.

Liskov Substitution Principle (LSP) +

Derived classes must be substitutable for their base classes.

Dependency Inversion Principle (DIP) +

Depend on abstractions, not on concretions.

Interface Segregation Principle (ISP) +

Make fine grained interfaces that are client-specific.

Classes Should be Small +

Smaller classes are easier to grasp. Classes should be smaller than about 100 lines of code. Otherwise, it is hard to spot how the class does its job and it probably does more than a single job.

Package Cohesion

Release Reuse Equivalency Principle (RREP) +

The granule of reuse is the granule of release.

Common Closure Principle (CCP) +

Classes that change together are packaged together.

Common Reuse Principle (CRP) +

Classes that are used together are packaged together.

Package Coupling

Acyclic Dependencies Principle (ADP) +

The dependency graph of packages must have no cycles.

Stable Dependencies Principle (SDP) +

Depend in the direction of stability.

Stable Abstractions Principle (SAP) +

Abstractness increases with stability

General

Follow Standard Conventions +

Coding-, architecture-, design guidelines (check them with tools)

Keep it Simple, Stupid (KISS) +

Simpler is always better. Reduce complexity as much as possible.

Boy Scout Rule +

Leave the campground cleaner than you found it.

Root Cause Analysis +

Always look for the root cause of a problem. Otherwise, it will get you again and again.

Multiple Languages in One Source File –

C#, Java, JavaScript, XML, HTML, XAML, English, German ...

Environment

Project Build Requires Only One Step +

Check out and then build with a single command.

Executing Tests Requires Only One Step +

Run all unit tests with a single command.

Source Control System +

Always use a source control system.

Continuous Integration +

Assure integrity with Continuous Integration

Overridden Safeties –

Do not override warnings, errors, exception handling – they will catch you.

Dependency Injection

Decouple Construction from Runtime +

Decoupling the construction phase completely from the runtime helps to simplify the runtime behaviour.

Design

Keep Configurable Data at High Levels +

If you have a constant such as default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to the low-level function called from the high-level function.

Don't Be Arbitrary +

Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it.

Be Precise +

When you make a decision in your code, make sure you make it precisely. Know why you have made it and how you will deal with any exceptions.

Structure over Convention +

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance.

Prefer Polymorphism To If/Else or Switch/Case +

"ONE SWITCH": There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.

Symmetry / Analogy +

Favour symmetric designs (e.g. Load – Save) and designs that follow analogies (e.g. same design as found in .NET framework).

Separate Multi-Threading Code +

Do not mix code that handles multi-threading aspects with the rest of the code. Separate them into different classes.

Misplaced Responsibility –

Something put in the wrong place.

Code at Wrong Level of Abstraction –

Functionality is at wrong level of abstraction, e.g. a PercentageFull property on a generic IStack<T>.

Fields Not Defining State –

Fields holding data that does not belong to the state of the instance but are used to hold temporary data. Use local variables or extract to a class abstracting the performed action.

Over Configurability –

Prevent configuration just for the sake of it – or because nobody can decide how it should be. Otherwise, this will result in overly complex, unstable systems.

Micro Layers –

Do not add functionality on top, but simplify overall.

Dependencies

Make Logical Dependencies Physical +

If one module depends upon another, that dependency should be physical, not just logical. Don't make assumptions.

Singletons / Service Locator –

Use dependency injection. Singletons hide dependencies.

Base Classes Depending On Their Derivatives –

Base classes should work with any derived class.

Too Much Information –

Minimise interface to minimise coupling

Feature Envy –

The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it envies the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating.

Artificial Coupling –

Things that don't depend upon each other should not be artificially coupled.

Hidden Temporal Coupling –

If, for example, the order of some method calls is important, then make sure that they cannot be called in the wrong order.

Transitive Navigation –

Aka Law of Demeter, writing shy code.
A module should know only its direct dependencies.

Naming

Choose Descriptive / Unambiguous Names +

Names have to reflect what a variable, field, property stands for. Names have to be precise.

Choose Names at Appropriate Level of Abstraction +

Choose names that reflect the level of abstraction of the class or method you are working in.

Name Interfaces After Functionality They Abstract +

The name of an interface should be derived from its usage by the client, such as IStream.

Name Classes After How They Implement Their Interfaces +

The name of a class should reflect how it fulfils the functionality provided by its interface(s), such as MemoryStream : IStream

Name Methods After What They Do +

The name of a method should describe what is done, not how it is done.

Use Long Names for Long Scopes +

fields → parameters → locals → loop variables
long → short

Names Describe Side Effects +

Names have to reflect the entire functionality.

Standard Nomenclature Where Possible +

Don't invent your own language when there is a standard.

Encodings in Names –

No prefixes, no type/scope information

Coding best practices (Clean Codes) are **a set of informal rules** that the software development community has learned over time which can help improve the quality of software.

Rules for Specific Language - Language C/C++

- **Formatting:** Formatting your source code.
- **Comments:** Commenting your work.
- **Syntactic Conventions:** Clean use of C constructs.
- **Names:** Naming variables, functions, and files.
- **System Portability:** Portability among different operating systems.
- **Character Set:** Use ASCII by default.
- **System Functions:** Portability and “standard” library functions.
- **Internationalization:** Techniques for internationalization.

For Example:

When you have an if-else statement nested in another if statement, always put braces around the if-else.

```
if (foo)
  if (bar)
    win ();
  else
    lose ();
```

Wrong

```
if (foo)
  {
    if (bar)
      win ();
    else
      lose ();
  }
```

Right !

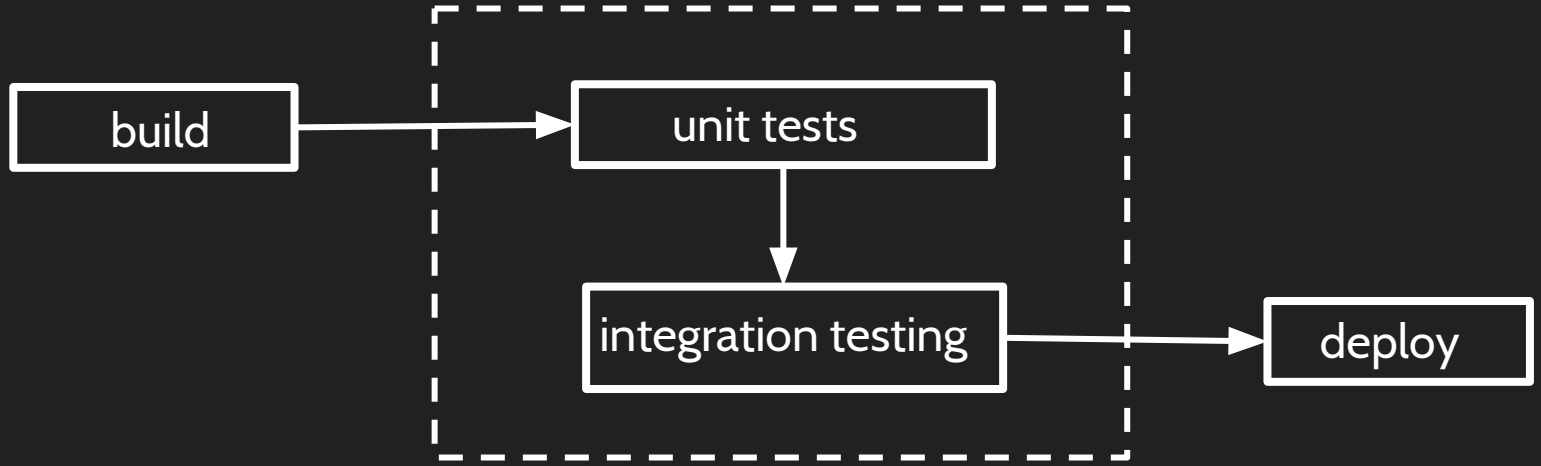
Code is Art.

Examples with Gradle

- Build Anything
- Automate Everything
- Deliver Faster

Bulid Java Project with Gradle

Continuous Integration

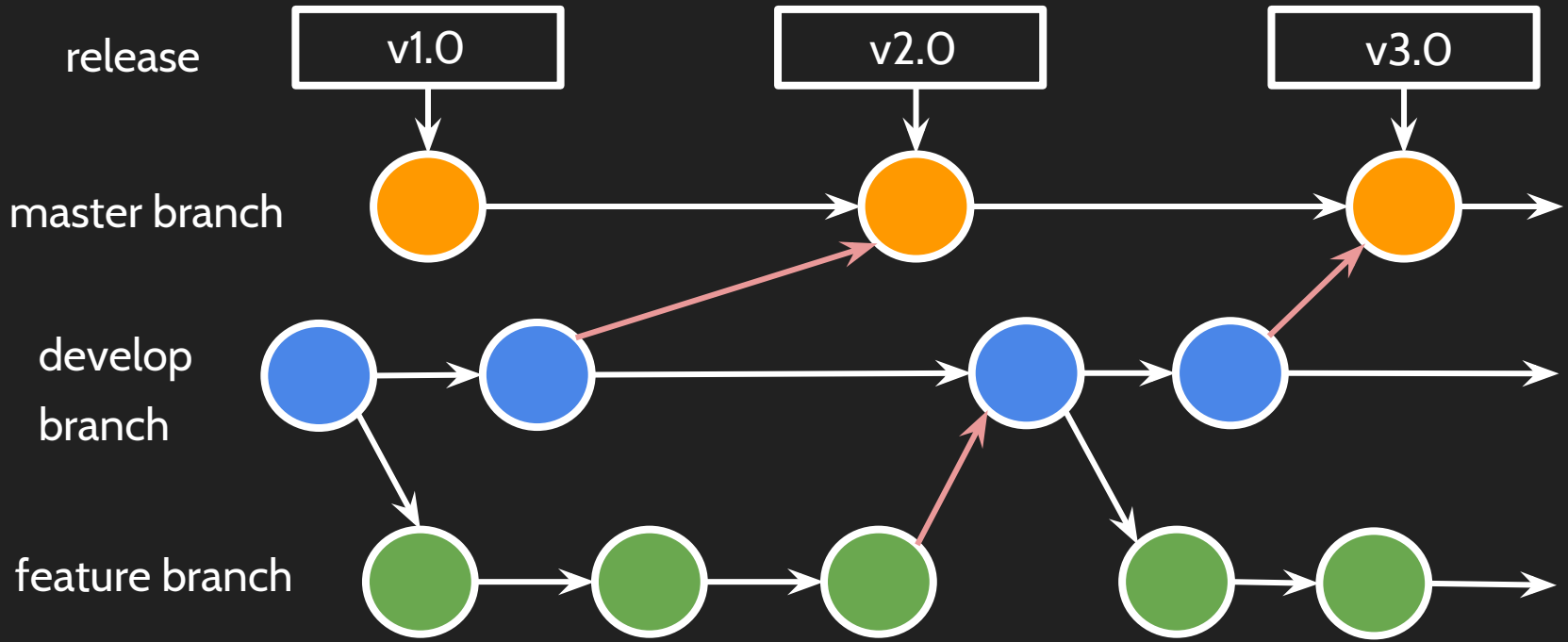


Continuous Integration

Continuous Integration for Android

- PMD
- Checkstyle
- FindBugs
- Lint
- Deploy & Releasing
- Environments
- Coverage

Git Workflow with CI



Practice

- Clone the Project Course-PR (or update as upstream).
- Create a new directory like: ***Course-PR/lesson/ [You name].***
- Create a new Java class: ***Sort.java.***
- Write a bubble sort algorithm in ***Sort.java.***
- Add changes to git history, commit and push to your fork.
- Open a pull request and fix the ci errors.
- Get your work review and merged.