Design problem definition

- Good Code vs Bad Code
- Code reuse
- Modular programming
- Testability

Good Code vs Bad Code

Code is about people, not about computers.

Good

- Things are clever, but not too clever
- Algorithms are optimal, both in speed as well as in readability
- Classes, variables and functions are well named and make sense without having to think too much
- You come back to it after a weekend off, and you can jump straight in
- Things that will be reused are reusable
- Methods tend to be very short that ideally perform a single task.
- You have enough information to call methods without looking at the body of them.
- Unit tests are easy to write

Bad

- Long methods made up of 2-3 sub-tasks that are not broken out into other methods.
- If you change the implementation of one method (but not the interface) you need to change the implementation of other methods.
- Lots of comments, especially long winded comments.
- You have code that never runs in your application to provide "future flexibility"
- Large try/catch blocks
- You are having a hard time coming up with good method names or they contain the words "OR" and "AND"
- Identical or nearly identical blocks of code
- Use of public static variables

Before each push to the repo

- Have you run all tests?
- Have you added new tests for all new methods?
- Are your code written according to the coding policy?
- Have your code passed code review or at least are you ready for the code review?
- Are your commits made to solve issues with Id from bug-tracking system?
- Have you mentioned the Id of the issue in the message of the each commit?

Code reuse

Code reuse aims to save time and resources and reduce redundancy by taking advantage of assets that have already been created in some form within the software product development process.

The key idea in reuse is that parts of a computer program written at one time can be or should be used in the construction of other programs written at a later time.

Good examples of code reuse:

- Software libraries
- Design patterns
- Frameworks
- Higher-order function

Copy and paste programming is not code reuse.

Concerning motivation and driving factors, reuse can be:

Opportunistic – While getting ready to begin a project, the team realizes that there are existing components that they can reuse.

Planned – A team strategically designs components so that they'll be reusable in future projects.

Reuse can be categorized further:

Internal reuse – A team reuses its own components. This may be a business decision, since the team may want to control a component critical to the project.

External reuse – A team may choose to license a third-party component. Licensing a third-party component typically costs the team 1 to 20 percent of what it would cost to develop internally. The team must also consider the time it takes to find, learn and integrate the component.

Concerning form or structure of reuse, code can be:

Referenced – The client code contains a reference to reused code, and thus they have distinct life cycles and can have distinct versions.

Forked – The client code contains a local or private copy of the reused code, and thus they share a single life cycle and a single version.

Modular programming

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules.

Modular programming is the process of subdividing a computer program into separate sub-programs.

Each module must contains everything necessary to execute only one aspect of the desired functionality.

Modules in modular programming enforce logical boundaries between components and improve maintainability. They are incorporated through interfaces

Java packages are intended to be modules.

The benefits of using modular programming

- Less code has to be written.
- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Programs can be designed more easily because a small team deals with only a small part of the entire code.
- Modular programming allows many programmers to collaborate on the same application.
- The code is stored across multiple files.
- Code is short, simple and easy to understand.
- Errors can easily be identified, as they are localized to a subroutine or function.
- The same code can be used in many applications.
- The scoping of variables can easily be controlled.

Testability

A unit test is a method that instantiates a small portion of our application and verifies its behavior independently from other parts.

A typical unit test contains 3 phases:

- it initializes a small piece of an application it wants to test
- it applies some stimulus to the system under test
- it observes the resulting behavior.

So, how to write code which is easy to test?

SOLID

S - Single Responsibility Principle: An object should do exactly one thing, and should be the only object in the codebase that does that one thing.

O - Open/Closed Principle: A class should be open to extension, but closed to change.

L - Liskov Substitution Principle: Functions that use references to base classes must be able to use objects of derived classes without knowing it.

I - Interface Segregation Principle: An interface should have as few methods as is feasible to provide the functionality of the role defined by the interface.

D - Dependency Inversion Principle: Concretions and abstractions should never depend on other concretions, but on abstractions.