Design Principles and Coding Standards

Vadivel Murugesan
Tech Lead

A consolidated guide to build a better application

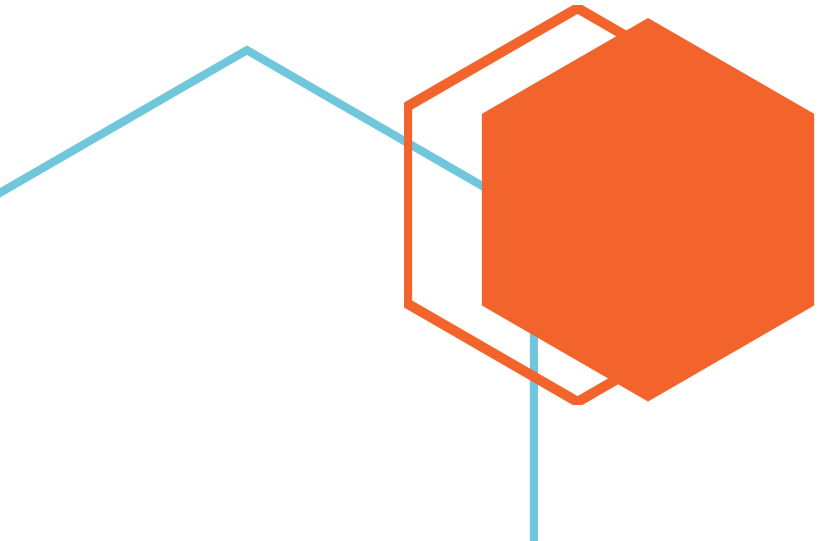

Table of Contents

Design Principles:

This guide is a consolidated document of all the findings related to the design principles, which helps engineers design and develop the application, primarily by mastering important related principles – fundamental ideas that they repeatedly use.

When we design the application, we consider the following properties to ensure the quality of the application.

1. **Usability** – is it easy for the client to use?
2. **Completeness** – does it satisfy all the client's needs?
3. **Robustness** – will it deal with unusual situations gracefully and avoid crashing.
4. **Efficiency** – will it perform the necessary computations in a reasonable amount of time using practical memory and other resources.
5. **Scalability** – will it still perform correctly and efficiently when the problems grow by several orders of magnitude?
6. **Readability** – is it easy for another developer to read and understand the design and code?
7. **Reusability** – can it be reused in a completely different setting?
8. **Simplicity** – is this design and implementation unnecessarily complicated?
9. **Maintainability** – can defects be found and fixed easily without adding new defects?
10. **Extensibility -** can be easily enhanced by adding new features or removing existing features without breaking the code.
11. **Security** – is this application protected against known security vulnerabilities?
12. **Testability** – is it easy to test the components using automated unit tests?

The driving principles:

This section covers principles of the essential design concepts

- **Complexity**
- **Cohesion**
- **Coupling**

A highly cohesive and loosely coupled design is the ultimate objective when we design the application.

Manage Complexity (Mac) a.k.a Simplicity:

Minimize accidental complexity and manage essential complexity

Complexity is a direct indicator of quality and costs. If the complexity of code is high, then the quality of that code will be lower, and it will cost more to manage it.

There are two types

Essential Complexity: It is the Inherent complexity of the program to be solved. It cannot be eliminated but managed by correctly applying many design principles like Abstraction, Separation of Concerns, Modularity, etc.

Accidental Complexity: The developer created the problem as part of the design and development of the solution. It can be eliminated or minimized by emphasizing the creation of simple and readable code rather than brilliant.

The metrics to measure the complexity of code

- Cyclomatic complexity
- Halstead Metrics
- Coupling Metrics
- Lines of Code
- Interface Complexity and
- Maintainability Index.

The factors would increase the complexity of the application

- Spaghetti Code
- God Class
- Proliferation of Classes
- Duplicated Code
- Contrived Complexity
- Large Class
- Large Method
- Feature Envy
- Excessive use of literals
- Too many parameters
- Inconsistent names
- Conditional Complexity
- Combination Explosion
- Oddball Solution
- Lack of Standards and Conversion

High Cohesion (HIC) a.k.a Maximize Cohesion

Cohesion is a measure of how strongly-related to focused the responsibilities of a single module are. It measures the single-mindedness of a class, object, or method in a system.

Method Cohesion – A method should do only one thing. A method that performs more than one task is complicated to understand.

Class Cohesion – It is about the level of cohesion between the attributes and methods of a class. A class should represent only one thing. All attributes of the class must be required to mean the thing.

Generalization/Specialization Cohesion – It is about how the classes in a hierarchy are related.

Low cohesion leads to the following issues

- High Complexity – Difficult to understand what belongs to where and why
- Poor Reuse

- Poor Readability

Symptoms that indicate low cohesion

- A class looks like a random set of functions
- Methods that don't interact with the rest of the class.
- Fields that are used by only one method
- Classes that change together (an indication that we need to refactor the classes into a separate class with high cohesion)

Also, note that maximizing cohesion should not lead to an adverse effect on minimizing coupling. i.e., please don't go overboard by making everything a separate module to maximize cohesion; it will increase the number of dependencies between modules and, hence, lead to strong coupling.

This Principle helps to improve flexibility, reusability, testability, maintainability, and readability. It also reduces complexity.

The metrics to measure the cohesion

- Lack of Cohesion of Methods (LCOM)
- Information flow-based cohesion
- Tight class cohesion
- Loose class cohesion

The following factors would decrease cohesion

- Big Ball of Mud
- God Class
- Divergent Change
- Large Class
- Mixed Abstractions
- Swiss Army knife.

Loose Coupling (LCO) a.k.a Minimize Coupling

This Principle is to minimize the coupling between components.

Minimizing coupling makes a component or module as independent as possible. Low coupling between modules indicates a well-partitioned system. The coupling can be reduced by

- Eliminating unnecessary relationships
- Reducing the number of necessary relationships
- Easing the "tightness" of the necessary relationships.

Loose coupling reduces the risk that a change made within one element will create unanticipated changes within other components.

Minimizing interconnections can help isolate the problems when things go wrong and simplify testing, maintenance, and troubleshooting procedures.

This Principle helps to improve flexibility, reusability, testability, maintainability. It also reduces complexity and coupling.

The metrics to measure the coupling

- Afferent Coupling (Ca)
- Efferent Coupling (Ce)
- Coupling Between Objects (CBO)
- Data Abstraction Coupling (DAC)
- Message Passing Coupling (MPC)
- And Coupling Factor (CF).

The factors would increase the coupling

- Cyclic Dependency
- Solution Sprawl
- Temporal Coupling
- Dense Object Network
- Hardcoded dependencies
- Hardcoded Resources
- Method chains
- Inappropriate Intimacy
- Violated Layering
- Large Class
- Large Package
- Unnecessary Dependencies
- Swiss Army knife.

Core Principles

This section describes a set of fundamental design principles that every developer should be familiar with

Abstraction

This Principle is a model that reduces real-world entities' characteristics into a set of objects with essential elements (state, behavior, and relationships) relevant to the context.

Information Hiding

This Principle hides design decisions subject to change within a module (function or procedure). It provides access to the module through well-defined interfaces that reveal only the required information to use the module.

Encapsulation

This Principle groups related ideas into a single unit and control hide/protect the visibility of attributes, methods, classes, and modules from clients to avoid dependency on things that might change.

Inheritance:

This Principle is a model Is-A Relationship using inheritance.

Polymorphism:

This Principle uses to provide a single interface for accessing multiple related behaviors.

Don't Repeat Yourself (DRY) a.k.a. avoid duplication, once and only once, duplication is evil (DIE) Principle

Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system.

There are two types of duplication:

Mechanical Duplication: Copy & Paste code is the most common form of mechanical duplication and may help get things up and running quickly, but it makes code fragile and hard to change. It is easy to identify and fix.

Knowledge Duplication: Reintroduce existing functionality by writing new code (usually not immediately recognized as a duplication) with a different name, abstractions, and maybe better implementation. It is difficult both to identify and fix.

Duplication can happen in code (both production and test code) configuration, data model, documentation, and any other artifact used to develop and maintain the system.

Recognizing and eliminating duplication through proper abstraction and other best practices will produce a flexible and maintainable system.

This Principle helps to improve reusability, testability, and maintainability. It also reduces complexity.

The metrics to measure the duplication

- Code Duplication (SonarQube helps us in identifying these duplicate code)

The following factors would cause the duplication

- Copy Paste Reuse
- Duplicated Code
- Data Redundancy
- Derived Value
- Repeated Value
- Alternative Modules with Different Interfaces
- Magic Number

Program to an Interface (P2I)

Always program to an interface, not an implementation.

It is easy to add a dependency to concreate implementation, but getting rid of an unwanted dependency could potentially lead to refactoring and hinder the reuse of code in another context. When you depend only on interfaces, you are automatically decoupled from the implementation, and you can add new behavior without breaking the clients.

Composite Reuse principle

Favor object composition over class inheritance

Inheritance and composite are two mechanisms supported by object-oriented programming to extend existing code and reuse functionality. Each of the mechanisms has its advantages and disadvantages.

Here are some of the disadvantages of inheritance:

- Difficult to change the behavior at runtime
- Hard to gain knowledge about all behaviors of the subtypes
- Leads to the class explosion
- Challenging to write unit tests
- Changes to the base type can unintentionally affect some of the subtypes.

The composition is an alternative to inheritance. The composition can be used to overcome all of these disadvantages. Composition allows for dynamically switching an implementation by delegating calls to appropriate objects at runtime. It's also referred to as dynamic/simulated inheritance. Favoring composition over inheritance will reduce maintainability problems and provide the flexibility to change the behavior at runtime.

It is important to note that composition and inheritance are complementary, and we use them together to achieve better designs.

Strategic Pattern is a classic example that uses composition and delegation to change its behavior without touching it.

Encapsulate what varies

Changes in requirements usually lead to redesign. It is impossible to predict future needs' specifics but more comfortable determining what aspects are likely to change. So instead of focusing on the cause for redesign, consider what you want to be able to change without redesign,

During the software design, look for the aspects most likely to change and make provisions to support them to protect the rest of the system from that change called encapsulating variation.

Modularity Principle a.k.a Modularization Principle

Decompose an extensive system into several small manageable modules.

Modularity helps to achieve the following objectives:

- Manage complexity and improve comprehensibility
- Enable parallel work; and
- Accommodate future change.

Language constructs as Classes and Packages can be considered as modules as part of logical design. Physical design issues like determining which classes below in which deployable units and managing the relationships between deployable units also need to be addressed to ensure the dependencies between the physical entities are optimal to realize the benefits modularity promises.

Closed Layered Architecture Principle

We prefer closed layered architecture over open layered architecture. Allow each layer to communicate only with the layer immediately below it. It minimizes the dependencies between layers.

Separation of Concerns:

SoC ensures changes to one part of the program do not impact other parts of the program by localizing the changes.

Concerns can be classified into the following categories:

Core Concern: Generally, a functional requirement –specific functionality to be included in a system.
Crosscutting Concern: Also referred to as an aspect is a non-functional behavior of the system. In general, those non-functional things apply to the system as a whole rather than individual requirements. E.g., Security, Logging, Caching.

Inversion of Control a.k.a Hollywood principle, dependency injection

It allows one to observe an object's state and act (as and when required) in a well-defined and unobtrusive manner.

E.g., Spring Framework.

SOLID PRINCIPLES

SOLID is an acronym for five design principles:

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle a.k.a class consistency principle

All the features of a class must pertain to a single, well-identified abstraction. Or A class should have only one reason to change.

The single responsibility principle encourages us to design a class in such a way that there is only one reason for it to change. When a class has more than one reason to change, it has multiple responsibilities. A class with various responsibilities should be refactored to smaller classes, each with only one reason (responsibility) to change.

It usually leads to smaller classes and hence improving the readability and maintainability of the system.

Open Closed Principle a.k.a Protected Variations

Software entities (classes, modules, and functions) should be open for extension, but closed for modification.

Here are some of the most common indicators that your code violates this Principle:

- Using conditional code with if/else or switch/case to determine the strategy.
- Hardcoded references to other class names, type checking, and casting.
- Using a new operator to create class objects

Strategy, Template Method, and Abstract Factory are good examples of patterns that implement the open-closed Principle.

Liskov Substitution Principle a.k.a Principle of type conformance (or substitutability)

Subtypes must be substitutable for their base types.

LSP is all about implementing behavioral subtyping (polymorphism) without altering any of the program's desirable properties like correctness, the task performed. Etc.,

Subtypes must follow the below rules to comply with this Principle:

- The subtype has all the attributes of the base type
- The class invariant of the subtype is equal to or more potent than that of the base type.
- For each of the operation that is overridden and redefined by subtype:
    - The name of the operation must be the same as the base type.
    - The signature must correspond to the base type operation's signature
    - Precondition must be equal or weaker than base type operation. Input parameter type should be the same or its base type (i.e., the input parameter can be widened/upcasted – the contravariant argument is allowed)
    - Postcondition must be equal to or more potent than the base type operation. The return type can be narrowed/down casted – covariant return type is allowed)
    - Exceptions can be narrowed, but no new exceptions can be thrown
    - **History Constraint** – subtype should not allow state changes that are not permissible in the base type.

Interface Segregation Principle

The client should not be forced to depend on the methods that they do not use. Interfaces belong to clients, not to hierarchies.

Interface bloat happens when an interface has too many operations, but most clients can't perform or require all the functions. It is also known as a fat interface. Fat interfaces are not cohesive and include unnecessary dependencies that clients do not need.

Hence many client-specific interfaces are better than one general-purpose interface.

Dependency Inversion Principle:

"High-level modules should not depend on low-level modules. Both should depend on the abstractions.

Abstractions should not depend on the details. Details should depend on the abstractions."

An object should not control creating its dependencies. It should allow the caller to provide the dependencies through a dependency injection mechanism. Spring makes it easy to implement this Principle.

Developer Tools

To ensure the quality of code during the development in eclipse IDE, we use the following plugins

1. Check style
2. Spot Bugs
3. PMD
4. Sonar Lint
5. UDDetector

References

Martin Fowler - Refactoring: Improving the design of existing code

Larry L. Constantine - Structured Design: Fundamentals of a discipline of computer program and systems design

Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson - Design Patterns: Elements of Reusable Object-Oriented Software is a software engineering book describing software design patterns.