Single Responsibility Principle (SRP)

This principle states that there should never be more than one reason for a class to change. This means that you should design your classes in such a way that each class should have a single purpose.

Example - An Account class is responsible for managing Current and Saving Account but a CurrentAccount and a SavingAccount classes would be responsible for managing current and saving accounts respectively. Hence both are responsible for single purpose only. Hence we are moving towards specialization.

Open/Closed Principle (OCP)

This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. The "closed" part of the rule states that once a module has been developed and tested, the code should only be changed to correct bugs. The "open" part says that you should be able to extend existing code in order to introduce new functionality.

Example – A PaymentGateway base class contains all basic payment related properties and methods. This class can be extended by different PaymentGateway classes for different payment gateway vendors to achieve theirs functionalities. Hence it is open for extension but closed for modification.

Liscov Substitution Principle (LSP)

This principle states that functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Example - Assume that you have an inheritance hierarchy with Person and Student. Wherever you can use Person, you should also be able to use a Student, because Student is a subclass of Person.

Interface Segregation Principle (ISP)

This principle states that Clients should not be forced to depend upon interfaces that they don’t use. This means the number of members in the interface that is visible to the dependent class should be minimized.

Example - The service interface that is exposed to the client should contains only client related methods not all.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle states that:

1. High level modules should not depend upon low level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

It helps us to develop loosely couple code by ensuring that high-level modules depend on abstractions rather than concrete implementations of lower-level modules. The Dependency Injection pattern is an implementation of this principle

Example - The Dependency Injection pattern is an implementation of this principle

DRY (Don’t Repeat Yourself)

This principle states that each small pieces of knowledge (code) may only occur exactly once in the entire system. This helps us to write scalable, maintainable and reusable code.

Example – Asp.Net MVC framework works on this principle.

KISS (Keep it simple, Stupid!)

This principle states that try to keep each small piece of software simple and unnecessary complexity should be avoided. This helps us to write easy maintainable code.

YAGNI (You ain't gonna need it)

This principle states that always implement things when you actually need them never implements things before you need them.

https://adevait.com/software/solid-design-principles-the-guide-to-becoming-better-developers

The 7 Basic System Design Principles in Software Engineering

Abstraction

All modern software techniques predominantly involve working with abstractions of various types. This general approach is called *object-oriented software development*.

Using abstractions means hiding the coding complexities and redundant details behind high-level abstractions and not delving into them until absolutely required.

Thanks to abstraction-based design concepts, you can decrease irrelevant data, speed up the development process, and improve the general quality of your programming outcomes.

Refinement

This means getting rid of structural impurities by moving from higher levels of software design (abstractions) to lower levels in a step-by-step manner.

According to this idea, refinement is an incremental process in which a team of software engineers drills down to acquire more technical details at each increment. In this way software design is consistently elaborated without wasting time on irrelevant or side matters.

Modularity

Dividing a complex project or system into smaller components helps to better understand and manage the product. Separate modules can be created independently and assembled together and/or reused and recombined, if required. This also facilitates easier scalability in software systems.

Learn more: HMS Modules: How to Select the Best Configuration for Your Hospital Management Software

Architecture

Everything within your software system should be pre-planned and approved with the help of engineering assessment methods. Serious systems flaws must be avoided at the beginning.

Interactions between different system components (e.g., modules and abstractions) should be the focus of architectural efforts—all of which should be seamlessly arranged within a solid software structure. Their interrelationship should be described in detail.

Patterns

Delivering pattern-based solutions is one of the most important techniques that allows software developers to achieve system predictability while saving a great deal of time. This also makes it possible to quickly deal with typical issues and apply pattern-based solutions to fix them in no time.

Data Protection

Data must be protected from unauthorized access. Therefore, secure software development life-cycle principles should be applied and propagated throughout the entire software structure.

For example, information accessible via one software module should not be accessible via another module unless it is explicitly allowed and regulated by the software architecture plan.

Refactoring

Refactoring is the continuous process of bringing improvements to internal software structure without affecting its behavior or functions.

In fact, refactoring practice is a part of the perpetual software maintenance process, and involves regular review of and improvement to the code in order to make it more effective and lightweight.

The 7 Modern Design Principles of Successful Software Development

Principle #1. Avoid Tunnel Vision

While moving through the design process, it's easy to fall victim to *tunnel vision:* focusing on a very narrow perspective without taking into account side effects and other factors.

It is necessary to consider and evaluate all available alternatives and align your performance with the most rational scenarios instead of just "moving through the tunnel."

Principle #2. Don't Reinvent the Wheel

Your team should not waste time and energy on building solutions that are already available.

If you're not a newcomer to software development, you should know that all modern programming takes place in the object-oriented model.

Today, you can easily find tons of free libraries, ready-made modules, and API-based solutions to significantly speed up your software project.

Principle #3. Respect the Traceability Model

Make sure that your software system design is available for traceability analysis.

This implies tracking changes introduced to the product, thereby maintaining high-quality documentation in which all product components are represented through integral interrelationships with their predecessors and successors.

Principle #4. Minimize the Distance between Solution and Problem

The lesser the gap between a real-life problem and the software solution you offer, the better it works. In this way, it becomes more popular and more in demand. Enough said. Purify functionality and simplify user experience to the maximum, until your software turns into a pure solution without excessive detail.

Principle #5. Fault-tolerance

This is also called graceful degradation. Even if the software generates an error or an incident occurs during the process of program execution, the software should keep working smoothly instead of shutting down. This could cause user data loss and/or changes, which you want to avoid at all costs.

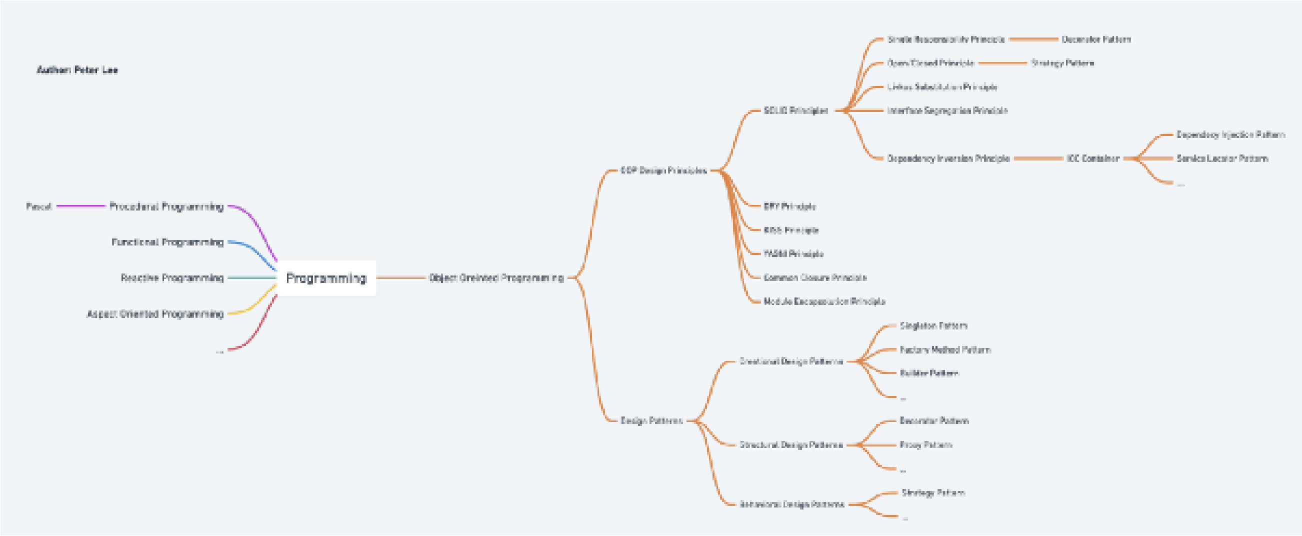
Principle #6. Quality Assessment and Assurance

Quality planning and testing of software products should be one of the primal focuses of any development team as these processes ensure that your product features the necessary characteristics of good software design.

Principle #7. Accommodation of Change

Software structure should be designed to tolerate and painlessly accept change. Frequent change requests can be generated by customers (product owners) or dictated by market disposition.

This basically refers to principles of modular software, completeness, and traceability. A system developed on those principles is always ready for scaling, adjustments, and improvements.


Author: Peter Lee
Programming
Pascal
Procedural Programming
Functional Programming
Reactive Programming
Aspect Oriented Programming
...
Object Oriented Programming
OOP Design Principles
SOLID Principles
Single Responsibility Principle
Decorator Pattern
Open/Closed Principle
Strategy Pattern
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle
IOC Container
Dependency Injection Pattern
Service Locator Pattern
...
DRY Principle
KISS Principle
YAGNI Principle
Common Closure Principle
Module Encapsulation Principle
Design Patterns
Creational Design Patterns
Singleton Pattern
Factory Method Pattern
Builder Pattern
...
Structural Design Patterns
Decorator Pattern
Proxy Pattern
...
Behavioral Design Patterns
Strategy Pattern
...