Software Engineering Principles Cheatsheet

A quick reference guide covering fundamental software engineering principles, methodologies, and best practices. This cheat sheet is designed to help software engineers at all levels in designing, developing, and maintaining high-quality software.

Core Principles

SOLID Principles

S - Single Responsibility Principle (SRP):
A class should have only one reason to change.

Example:
Instead of a class handling both database connections and business logic, separate these into distinct classes.

O - Open/Closed Principle (OCP):
Software entities should be open for extension, but closed for modification.

Example:
Use inheritance or composition to add new functionality without altering existing code.

L - Liskov Substitution Principle (LSP):
Subtypes must be substitutable for their base types without altering the correctness of the program.

Example:
If a function takes an object of type 'Animal', it should also work correctly with objects of type 'Dog' or 'Cat'.

I - Interface Segregation Principle (ISP):
Clients should not be forced to depend on methods they do not use.

Example:
Instead of one large interface, create multiple smaller interfaces specific to client needs.

D - Dependency Inversion Principle (DIP):
Depend upon abstractions, not concretions. High-level modules should not depend on low-level modules. Both should depend on abstractions.

Example:
Use dependency injection to inject dependencies into classes rather than creating dependencies within the class.

DRY Principle

Don't Repeat Yourself (DRY):
Avoid duplication of code and logic.

Benefits:

- Improved maintainability
- Reduced risk of errors
- Easier refactoring

Example:
Create a reusable function or class instead of copying and pasting code.

KISS Principle

Keep It Simple, Stupid (KISS):
Design systems to be as simple as possible.

Benefits:

- Easier to understand
- Easier to maintain
- Reduced complexity

Example:
Avoid over-engineering a solution when a simpler solution is sufficient.

YAGNI Principle

You Ain't Gonna Need It (YAGNI):
Avoid adding functionality until deemed necessary.

Benefits:

- Reduced complexity
- Faster development
- Avoidance of unnecessary code

Example:
Do not implement a feature 'just in case' it might be needed in the future.

Design Patterns

Creational Patterns

Singleton	Ensures only one instance of a class is created and provides a global point of access to it.
Factory Method	Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.
Abstract Factory	Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
Prototype	Specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Structural Patterns

Adapter	Allows incompatible interfaces to work together. Converts the interface of a class into another interface clients expect.
Bridge	Decouples an abstraction from its implementation so that the two can vary independently.
Composite	Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.
Decorator	Dynamically adds responsibilities to an object. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Uses sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provides a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Chain of Responsibility	Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template Method	Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Software Development Methodologies

Agile Methodologies

Overview:
Agile methodologies emphasize iterative development, collaboration, and flexibility to adapt to changing requirements.

Key Principles:

- Customer satisfaction through early and continuous delivery
- Welcoming changing requirements
- Frequent delivery of working software
- Close collaboration between business stakeholders and developers
- Self-organizing teams
- Continuous attention to technical excellence

Popular Agile Frameworks:

- Scrum
- Kanban
- Extreme Programming (XP)

Scrum

Overview:
An iterative and incremental framework for managing complex projects.

Key Components:

- **Roles:** Product Owner, Scrum Master, Development Team
- **Events:** Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective
- **Artifacts:** Product Backlog, Sprint Backlog, Increment

Kanban

Overview:
A visual system for managing workflow, limiting work in progress (WIP), and improving flow.

Key Principles:

- Visualize the workflow
- Limit WIP
- Manage flow
- Make process policies explicit
- Implement feedback loops
- Improve collaboratively, evolve experimentally

Waterfall Methodology

Overview:
A sequential, linear approach to software development where each phase must be completed before the next phase can begin.

Phases:

1. Requirements
2. Design
3. Implementation
4. Verification
5. Maintenance

Limitations:

- Inflexible to changes
- Not suitable for complex or evolving projects

Code Quality and Testing

Code Quality Metrics

Cyclomatic Complexity	Measures the number of linearly independent paths through a program's source code. Lower values indicate simpler, more testable code.
Code Coverage	Measures the extent to which the source code of a program has been tested. Higher coverage generally indicates better testing.
Maintainability Index	Calculates an index value that represents the relative ease of maintaining the code. Higher values are better.
Lines of Code (LOC)	A simple measure of the size of a program. Can indicate complexity and effort required for maintenance.

Testing Types

Unit Testing	Testing individual units or components of a software application. Focuses on verifying that each part of the system works as expected.
Integration Testing	Testing the interaction between different units or components to ensure they work together correctly.
System Testing	Testing the entire system to ensure it meets the specified requirements. Conducted after integration testing.
Acceptance Testing	Testing conducted by end-users or stakeholders to determine whether the system meets their needs and expectations.

Test-Driven Development (TDD)

Overview:
A software development process in which tests are written before the code. This helps ensure that the code is testable and meets the specified requirements.

Steps:

1. Write a test
2. Run the test and see it fail
3. Write the minimal code to pass the test
4. Run all tests and ensure they pass
5. Refactor the code

Code Review Best Practices

Key Considerations:

- Focus on code correctness, clarity, and maintainability
- Provide constructive feedback
- Ensure code adheres to coding standards
- Check for potential bugs and security vulnerabilities
- Automate code reviews where possible