Single responsibility principle (SRP) SOLID

A class should have only one responsibility and reason to change. Minimizes code duplication, centralizes behaviours simplifying changes.

## Open closed principle (OCP)

A class should be open for extension but closed for modification. (Changes occur in class file). Achieved by following SRP and/or abstraction.

## Liskov Substitution Principle (LSP)

Subtypes should be substitutable for their base types. (Inheritance)

## Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. (Small Interfaces). Interfaces with many behaviours are difficult to maintain and evolve, should be avoided.

## Dependency inversion principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. (logic involving classes should depend on the interface not the implementation)

Loose Coupling: components have minimal or no knowledge of the internal workings or implementation details of other components. Changes in one component have little impact on others, and components can be easily replaced or modified without affecting the entire system.

Abstraction: hiding the details of a class, encapsulates inner workings, minimum public accessibility.

Polymorphism: many forms, a class extends/implemenets many different classes, e.g. a child class overrides/implements concretely its parent/interfaces methods.

Dependency Injection: relationships between classes determine how dependent they are. Related to the Dependency Inversion Principle (uses abstraction to avoid duplicate code etc).

If a class creates another class it depends upon it, instead this dependency can be injected (created outside of this class)

Construction Injection: Parse into a class an abstract type (e.g Food), class sets its field during construction

Setter injection: Use setClassA(ClassA classA), field set using a setter instead.

Field injection: class uses a public Abstract class that is mutated externally.

 DRY (Don't Repeat Yourself): Avoid duplication of code or logic. Instead, strive for code reuse by abstracting common functionality into reusable components.

 KISS (Keep It Simple, Stupid): Keep software designs and implementations as simple as possible. Avoid unnecessary complexity and strive for simplicity, clarity, and ease of understanding.

 YAGNI (You Ain't Gonna Need It): Do not implement functionality until it is necessary. Avoid adding unnecessary features or code that may never be used.

 Composition over Inheritance: Favour composition (building objects by combining simpler components) over inheritance (extending existing classes). Composition provides greater flexibility, maintainability, and code reuse.

 Separation of Concerns (SoC): Divide software systems into distinct components that address separate concerns or responsibilities. Each component should focus on a specific aspect of the system's functionality.

 Loose Coupling and High Cohesion: Aim for loosely coupled components that have minimal dependencies on each other. Encourage high cohesion within each component, meaning that its internal elements are strongly related and focused on a single purpose.

 Design Patterns: Utilize proven design patterns to solve common software design problems and promote code reusability, flexibility, and maintainability.

 Test-Driven Development (TDD): Write tests before writing the actual code. TDD ensures that code is developed with a focus on meeting specific requirements and helps maintain code quality through automated testing.

 Continuous Integration and Continuous Delivery (CI/CD): Implement automated processes for integrating code changes, running tests, and deploying software frequently and reliably.

 Agile and Iterative Development: Embrace agile methodologies and iterative development approaches to enable frequent feedback, adaptability, and incremental delivery of software.

 Documentation: Document code, designs, and system architecture to aid understanding, maintenance, and collaboration among team members.

Waterfall (build it twice): requirements->design->implement+test->integrate+sys test->devop

Spiral (no upfront analysis): iterative. get objectives ->identify/resolve risks->dev+test->get..

AGILE (framework implemented by LEAN/KANBAN/SCRUM): Individuals+Interactions over processes, working software over documentation, customer collaboration over contract negotiations, responding to change over following plan. 12 Principles: satisfy customer through continuous delivery, changing requirements on demand, deliver software fast, business people + devs work together daily, support and trust team members, face-to-face comms, working software is the primary measure of progress, sustainable development to maintain a constant pace indefinitely, attention to technical excellence and good design, Simplicity by minimizing work, self organizing teams, reflects on how to be more effective and adjusts accordingly.

SCRUM: fast product dev method from “1986 new product dev game”, adapted to Software (AGILE). Values: Openness, Focus, Respect, Courage, Commitment. Sprints: planning, development, review, retrospective. Initial backlog (user stories + acceptance criteria), groom the backlog to decouple commonalities, priorities of tasks stated by the PO (project owner). User stories broken into tasks by team, estimations set collaboratively (bubble/tree hierarchy approach) (or planning poker, group of people choose a relative estimate). Roles: PO= translates user requirements to user stories, maintains backlog, negoatiates time estimates with team, SM (Scrum Master): acts as coach, does out of team comms, represents management but protects team members, Team (devs): no hierarchy, self organising, collective responsibility.

LEAN: Optimise big picture, customer focus, energise workers, avoid over-engineering, deliver fast, move on after completion (focus on delivering more results instead of results)

KANBAN: minimize upfront planning, limit WIP to avoid bottlenecking, measures: effiency by WIP and TODO tasks, tasks\_completed / time\_unit, delta\_time of delivery vs customer demand, Cycletime = WIP/task\_rate = time\_unit\*WIP / tasks\_completed.