SOLID

- Single responsibility
- Open/closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Single responsibility

- every class or method in your program should have only a single reason to change
    - can use higher order functions (lambdas) to limit methods and enforce single responsibility

The Open/Closed Principle

- existing entities should be open for extension but closed for modification
    - think command/strategy/visitor and/or adapter/bridge/mediator/facade
    - for functional paradigms, use lambdas as template arguments in C++ to extend functionality
    - immutability:
        - observable immutability means that from the perspective of any other object, a class is immutable
        - implementation immutability means that the object never mutates
        - (relates to the C++ idea of logical constness)
        - immutable objects implement the open/closed principle -> their internal state can't be modified so it is safe to add new methods

The Liskov Substitution Principle

- formally:
    - if q(x) is a provable property about objects x of type T then q(y) should be true for objects y of type S where S is a subtype of T
- simply: descendant classes should maintain invariants established by ancestor classes
- preconditions cannot be strengthened in a subtype
    - the child should also work when a parent would work

- postconditions cannot be weakened in a subtype
    - the child should cause side effects where a parent caused side effects
- invariants of the supertype must be preserved in a subtype
    - the child should maintain any suppositions/related requirements of a parent
- legacy rule: disallow state changes not permitted by the parent
    - e.g. a mutable type cannot inherit from an immutable type
- functional programming avoids inheritance thus avoiding problems Liskov substitution is meant to solve
- OO now promotes composite reuse, i.e. prefer composition over inheritance

The Interface-Segregation Principle

- the dependency of one class on another one should be limited to the smallest possible interface
- this relates to subtyping in Java and C++
- nominal subtyping -> a class must explicitly inherit clearly in code
- subtyping causes the inheriting class to be
- alternative is structural subtyping:
    - calling a method on an object instance only requires that the method exists -> there is no runtime or compile-time environment enforcing this
    - this is the case in C++ templates
    - also the case with duck typing in dynamically typed languages like Ruby and Python
- structural subtyping removes the need for interface segregation (according to author)
- more realistically (my note) the number of methods called in implementation relates to interface-segregation and reducing this decreases interdependencies

The Dependency-Inversion Principle

- abstractions should not depend on details -> details should depend on abstractions
- higher level abstractions should only interact with abstractions rather than explicit resources
- for example, reading and writing can be done through streams rather than specific file access
- callbacks/higher-order functions can be used to decouple these interactions