Solid basics:

The **S**ingle Responsibility Principle

The **O**pen-Closed Principle

The **L**iskov Substitution Principle

The **I**nterface Segregation Principle

The **D**ependency Inversion Principle

1. Single Responsibility Principle

"A class should have only one reason to change."

- A class should **do one thing and do it well**
- It keeps code **modular** and **easier to understand/test**

- because many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules.
- It makes version control easier. For example, say we have a persistence class that handles database operations, and we see a change in that file in the GitHub commits. By following the SRP, we will know that it is related to storage or database-related stuff.
- Merge conflicts -> They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

2. Open/Closed Principle

"Software entities should be open for extension, but closed for modification."

- You should be able to **add new features** without **changing existing code**
- a class can be both open (for extension) and closed (for modification) at the same time
- Use **abstraction** (**interfaces/abstract classes**) and **polymorphism**
- Strategy pattern, plugins, inheritance, dependency injection

Strategy pattern:

Strategy design pattern allows you to change the strategy or algorithm for a program as simply as possible, without modifying the code that uses it. You define the strategies and the client program will choose the appropriate one at runtime.

It allows the behavior of a class to be selected dynamically.

Structure:

Uses a strategy to perform a task

Common interface for all algorithms

Different implementations of the strategy

When to use it?

You have many variations of an algorithm -> Keep logic modular instead of using if/else

You want to switch behavior at runtime -> Choose strategy dynamically

You want to adhere to Open/Closed Principle -> Add new strategies without changing existing code

Cool things about strategy pattern:

Removes if-else/switch-case hell

Adds behavior via composition not inheritance

Keeps code clean, flexible, and testable

Works great with dependency injection

3. Liskov Substitution Principle

"Subtypes must be substitutable for their base types."

- Any subclass should **work perfectly in place of its parent** class.
- Prevents **unexpected behaviors** when using inheritance.

This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down.

When a class does not obey this principle, it leads to some nasty bugs that are hard to detect.

4. Interface Segregation Principle

> "Clients should not be forced to depend on methods they do not use."

- **Prefer small, focused interfaces** over large, general-purpose ones.
- Encourages **cohesion** and **decoupling**.

Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces

The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need

5. Dependency Inversion Principle

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

- Use **abstractions** (**interfaces**) instead of direct dependencies.
- Inversion of control -> facilitates **testing**, **modularity**, and **flexibility**.

SOLID in go examples:

https://youtu.be/o_yTAosQUGc?si=M3djQFVy6plQjUIB

Code: https://github.com/plutov/packagemain...

1.single repository principle:

survey.go > (*Survey).Save

```go
package survey

type Survey struct {
    Title     string
    Questions []string
}

func (s *Survey) GetTitle() string {
    return s.Title
}

func (s *Survey) Validate() bool {
    return len(s.Questions) > 0
}

func (s *Survey) Save() error {
    // save survey to the db
    return nil
}
```

save() is problematic here because we are mixing db functions with simple functions that handle survey logic, in the same struct.

So to fix it we can to add an interface (repository) to handles this save and other db funcs in another package and file:

```go
type Repository interface {
    Save(s *Survey) error
}

type InMemoryRepository struct {
    surveys []*Survey
}

func (r *InMemoryRepository) Save(s *Survey) error {
    r.surveys = append(r.surveys, s)
    return nil
}

func SaveSurvey(s *Survey, r Repository) error {
    return r.Save(s)
}
```

2. open/closed principle:

```go
// export.go > ExportSurvey
package survey

import (
  "fmt"
)

func ExportSurvey(s *Survey, dst string) error {
  switch dst {
  case "s3":
    // export to aws s3
    return nil
  case "gcs":
    // export to gcs
    return nil
  default:
    return fmt.Errorf("unsupported dst: %s", dst)
  }
}
```

Problems: if we wanna add another export destination, we should come to the original file and change this function. We might cause bugs and lose the availability of the already fine function that works correctly for other exporting cases

```go
// export.go > Exporter
package survey

func ExportSurvey(s *Survey, exporter Exporter) error {
  return exporter.Export(s)
}

type Exporter interface {
  Export(s *Survey) error
}

type S3Exporter struct{}

func (e *S3Exporter) Export(s *Survey) error {
  return nil
}

type GCSExporter struct{}

func (e *GCSExporter) Export(s *Survey) error {
  return nil
}
```

Now the first func receives one as a parameter (exporter Exporter). That means whoever calls ExportSurvey decides whether to use S3Exporter or GCSExporter.

The function depends on the Exporter interface:

```
type Exporter interface {
	Export(*Survey) error
}
```

And both of these structs implement the interface:

```
type S3Exporter struct{}
func (e *S3Exporter) Export(s *Survey) error {
	return nil
}
type GCSExporter struct{}
func (e *GCSExporter) Export(s *Survey) error {
	return nil
```

Example for Choosing the Exporter:

```
s := &Survey{}

var exporter Exporter

// choose one based on condition
if useS3 {
	exporter = &S3Exporter{}
} else {
	exporter = &GCSExporter{}
}

// pass it in
err := ExportSurvey(s, exporter)
```