

Go Code Simplified: Best Practices, Design Patterns, Clean Code Principles & Package Management



SONU RAJ · [Follow](#)

9 min read · May 26, 2024



35



Introduction

Design patterns are general reusable solutions to commonly occurring problems in software design. They are not specific to any programming language but rather provide a way to structure code and organize components to improve maintainability, flexibility, and scalability. While the examples in this documentation are written in Go, the underlying principles and design pattern concepts apply to any programming language.



Golang Clean Code Guide - Part 1



<https://golang.withcodeexample.com/>

Contents

1. Principles of Clean Code
2. Handling Packages in Go
3. Common Design Patterns in Go
 - Singleton Pattern
 - Factory Pattern
 - Observer Pattern
 - Strategy Pattern
4. Aggressive Use of Interfaces
5. Composition and Embedding in Go
6. Practical Examples
7. Using Go's Standard Library for Design Patterns
8. Real-World Examples: Go Libraries Using Design Patterns
9. References

1. Principles of Clean Code

1.1. Meaningful Names

- **Use descriptive names:** Choose names that clearly describe the purpose of variables, functions, and types.
- **Avoid abbreviations:** Use full words to avoid confusion.
- **Consistent naming conventions:** Follow Go's naming conventions, such as using camelCase for variables and functions.

1.2. Functions

- **Small and focused:** Functions should do one thing and do it well.
- **Descriptive names:** Function names should clearly indicate their purpose.
- **Limit parameters:** Keep the number of parameters to a minimum.

1.3. Comments

- **Use sparingly:** Code should be self-explanatory; use comments to explain why, not what.
- **Update regularly:** Ensure comments are updated to reflect changes in the code.

1.4. Error Handling

- **Check errors:** Always check and handle errors.
- **Use custom error types:** Create custom error types for more descriptive error handling.

1.5. Formatting

- **Consistent style:** Use `gofmt` to format your code consistently.
- **Organize imports:** Group standard library imports separately from third-party imports.

2. Handling Packages in Go

2.1. Organizing Code with Packages

- **Purpose:** Organize code into reusable and maintainable units.
- **Structure:** Follow a logical structure for your project, such as separating domain logic, services, and utilities.

2.2. Creating and Using Packages

- **Creating a Package:** Define a new package by creating a directory and adding Go files with the `package` keyword.
- **Using a Package:** Import the package using the `import` statement and access its exported functions, types, and variables.

Project Structure

```
example/
├─ main.go
├─ models/
│   └─ product.go
├─ repositories/
│   └─ product_repository.go
├─ services/
│   └─ product_service.go
├─ handlers/
│   └─ product_handler.go
├─ patterns/
│   └─ factory/
│       └─ product_factory.go
│   └─ strategy/
│       └─ pricing_strategy.go
│   └─ observer/
│       └─ config_observer.go
```

3. Common Design Patterns in Go

3.1. Singleton Pattern

- **Purpose:** Ensure a class has only one instance and provide a global point of access to it.
- **Implementation:** Use a package-level variable and a ``sync.Once`` to ensure thread safety.

```
package singleton

import (
    "sync"
)

type Singleton interface {
    DoSomething()
}

type singletonImpl struct{}
```

```
func (s *singletonImpl) DoSomething() {
    // Implementation
}

var instance Singleton
var once sync.Once

func GetInstance() Singleton {
    once.Do(func() {
        instance = &singletonImpl{}
    })
    return instance
}
```

3.2. Factory Pattern

- **Purpose:** Create objects without specifying the exact class of object that will be created.
- **Implementation:** Define an interface and create a factory function to instantiate the objects.

```
package factory

type Shape interface {
    Draw() string
}

type Circle struct{}

func (c Circle) Draw() string {
    return "Drawing Circle"
}

type Square struct{}

func (s Square) Draw() string {
    return "Drawing Square"
}

func ShapeFactory(shapeType string) Shape {
    if shapeType == "circle" {
        return Circle{}
    }
    if shapeType == "square" {
        return Square{}
    }
    return nil
}
```

3.3. Observer Pattern

- **Purpose:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

- **Implementation:** Use interfaces to define the subject and observers.

```
package observer

type Observer interface {
    Update(string)
}

type Subject interface {
    Register(Observer)
    Deregister(Observer)
    NotifyAll()
}

type ConcreteSubject struct {
    observers []Observer
    state     string
}

func (s *ConcreteSubject) Register(o Observer) {
    s.observers = append(s.observers, o)
}

func (s *ConcreteSubject) Deregister(o Observer) {
    for i, observer := range s.observers {
        if observer == o {
            s.observers = append(s.observers[:i], s.observers[i+1:]...)
            break
        }
    }
}

func (s *ConcreteSubject) NotifyAll() {
    for _, observer := range s.observers {
        observer.Update(s.state)
    }
}

func (s *ConcreteSubject) SetState(state string) {
    s.state = state
    s.NotifyAll()
}
```

3.4. Strategy Pattern

- **Purpose:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Implementation:** Define an interface for the strategy and create concrete implementations for different strategies.

```

package strategy

import "ecommerce/models"

// PricingStrategy defines the interface for pricing strategies
type PricingStrategy interface {
    CalculatePrice(product models.Product) float64
}

// RegularPricingStrategy is a simple implementation of PricingStrategy
type RegularPricingStrategy struct{}

func (s *RegularPricingStrategy) CalculatePrice(product models.Product) float64
    return product.Price
}

// DiscountPricingStrategy is an implementation of PricingStrategy with a discount
type DiscountPricingStrategy struct {
    DiscountRate float64
}

func (s *DiscountPricingStrategy) CalculatePrice(product models.Product) float64
    return product.Price * (1 - s.DiscountRate)
}

```

4. Aggressive Use of Interfaces

- **Interfaces over Concrete Types:** Prefer using interfaces over concrete types to decouple components and increase flexibility.
- **Interface Segregation Principle:** Separate interfaces into smaller, cohesive interfaces to avoid bloated interfaces.
- **Dependency Inversion Principle:** High-level modules should not depend on low-level modules; both should depend on abstractions (interfaces).
- **Composition over Inheritance:** Favor composition over inheritance to achieve code reuse and flexibility.

```

package example

type Reader interface {
    Read([]byte) (int, error)
}

type Writer interface {
    Write([]byte) (int, error)
}

type ReadWriter interface {
    Reader
    Writer
}

```

```

}

type File struct {
    // Implementation
}

func (f *File) Read(p []byte) (int, error) {
    // Implementation
}

func (f *File) Write(p []byte) (int, error) {
    // Implementation
}

func ProcessData(rw ReadWriter) error {
    // Use rw.Read() and rw.Write() to process data
}

```

5. Composition and Embedding in Go

Go does not support traditional class-based inheritance like some other object-oriented languages (e.g., Java or C++), it does support composition, which is often considered a more flexible and powerful alternative. In Go, you can achieve similar functionality to inheritance through embedding and interfaces.

5.1. Composition Over Inheritance

- **Purpose:** Achieve code reuse and flexibility by composing objects with other objects rather than inheriting from a base class.
- **Implementation:** Use struct embedding to include the fields and methods of one struct within another.

```

package main

import "fmt"

// Base struct
type Animal struct {
    Name string
}

func (a Animal) Speak() {
    fmt.Println(a.Name, "makes a sound")
}

// Derived struct using embedding
type Dog struct {
    Animal
    Breed string
}

```



```

func (d Dog) Speak() {
    fmt.Println(d.Name, "barks")
}

func main() {
    dog := Dog{
        Animal: Animal{Name: "Buddy"},
        Breed:   "Golden Retriever",
    }
    dog.Speak()           // Buddy barks
    dog.Animal.Speak()    // Buddy makes a sound
}

```

5.2. Interfaces and Composition

- **Purpose:** Use interfaces to define behavior and achieve polymorphism.
- **Implementation:** Define interfaces and implement them in different structs.

```

package main

import "fmt"

// Speaker interface
type Speaker interface {
    Speak()
}

// Animal struct
type Animal struct {
    Name string
}

func (a Animal) Speak() {
    fmt.Println(a.Name, "makes a sound")
}

// Dog struct
type Dog struct {
    Animal
    Breed string
}

func (d Dog) Speak() {
    fmt.Println(d.Name, "barks")
}

func main() {
    var s Speaker

    dog := Dog{
        Animal: Animal{Name: "Buddy"},
        Breed:   "Golden Retriever",
    }

    s = dog
    s.Speak() // Buddy barks
}

```

```
s = dog.Animal
s.Speak() // Buddy makes a sound
}
```

6. Practical Examples

1. Using Singleton Pattern in a Configuration Manager

```
package main

import (
    "fmt"
    "sync"
)

type Config interface {
    GetSetting(key string) string
    SetSetting(key, value string)
}

type configImpl struct {
    settings map[string]string
}

func (c *configImpl) GetSetting(key string) string {
    return c.settings[key]
}

func (c *configImpl) SetSetting(key, value string) {
    c.settings[key] = value
}

var configInstance Config
var once sync.Once

func GetConfigInstance() Config {
    once.Do(func() {
        configInstance = &configImpl{
            settings: make(map[string]string),
        }
    })
    return configInstance
}

func main() {
    config := GetConfigInstance()
    config.SetSetting("app_name", "MyApp")
    fmt.Println(config.GetSetting("app_name"))
}
```

2. Using Factory Pattern for Shape Creation

```

package main

import (
    "fmt"
    "path/to/your/factory"
)

func main() {
    shape1 := factory.ShapeFactory("circle")
    fmt.Println(shape1.Draw())

    shape2 := factory.ShapeFactory("square")
    fmt.Println(shape2.Draw())
}

```

3. Using Observer Pattern for Event Notification

```

package main

import (
    "fmt"
    "path/to/your/observer"
)

type ConcreteObserver struct {
    id string
}

func (co *ConcreteObserver) Update(state string) {
    fmt.Printf("Observer %s: State changed to %s\n", co.id, state)
}

func main() {
    subject := &observer.ConcreteSubject{}

    observer1 := &ConcreteObserver{id: "1"}
    observer2 := &ConcreteObserver{id: "2"}

    subject.Register(observer1)
    subject.Register(observer2)

    subject.SetState("New State")
}

```

4. Using Strategy Pattern for Pricing in E-commerce System

```

package main

import (
    "ecommerce/handlers"
    "ecommerce/models"
    "ecommerce/patterns/factory"
)

```

```

    "ecommerce/patterns/observer"
    "ecommerce/patterns/strategy"
    "ecommerce/repositories"
    "ecommerce/services"
    "fmt"
    "net/http"
)

func main() {
    // Initialize repository
    repo := repositories.NewInMemoryProductRepository()

    // Initialize factory
    productFactory := factory.NewSimpleProductFactory()

    // Create products using factory
    product1 := productFactory.CreateProduct("Product 1", 100.0)
    product2 := productFactory.CreateProduct("Product 2", 200.0)
    repo.Create(product1)
    repo.Create(product2)

    // Initialize pricing strategy
    regularPricing := &strategy.RegularPricingStrategy{}
    discountPricing := &strategy.DiscountPricingStrategy{DiscountRate: 0.1}

    // Initialize service with repository and pricing strategy
    service := services.NewProductService(repo, regularPricing)

    // Initialize handler with service
    handler := handlers.NewProductHandler(service)

    // Set up HTTP routes
    http.HandleFunc("/products", handler.GetAllProducts)
    http.HandleFunc("/product", handler.GetProductByID)
    http.HandleFunc("/product/add", handler.AddProduct)

    // Watch configuration changes
    configObserver := &observer.ViperConfigObserver{}
    observer.WatchConfig(configObserver)

    // Start HTTP server
    fmt.Println("Starting server on :8080")
    http.ListenAndServe(":8080", nil)

    // Change pricing strategy dynamically
    service.SetPricingStrategy(discountPricing)
}

```

7. Using Go's Standard Library for Design Patterns

Singleton Pattern

- `sync.Once`: The `sync` package provides the `once` type, which ensures that a piece of code is executed only once.

```

package singleton

import (
    "sync"
)

type Singleton interface {
    DoSomething()
}

type singletonImpl struct{}

func (s *singletonImpl) DoSomething() {
    // Implementation
}

var instance Singleton
var once sync.Once

func GetInstance() Singleton {
    once.Do(func() {
        instance = &singletonImpl{}
    })
    return instance
}

```

Factory Pattern

- **fmt.Stringer:** The `fmt` package provides the `stringer` interface, which can be used to create factory methods for different types that implement the `string` method.

```

package factory

import "fmt"

type Shape interface {
    fmt.Stringer
}

type Circle struct{}

func (c Circle) String() string {
    return "Circle"
}

type Square struct{}

func (s Square) String() string {
    return "Square"
}

func ShapeFactory(shapeType string) Shape {
    switch shapeType {
    case "circle":
        return Circle{}
    }
}

```



{

Observer Pattern

- **fsnotify:** The `fsnotify` package can be used to watch for file system changes, which is useful for implementing the Observer pattern.

```
package observer

import (
    "fmt"
    "github.com/fsnotify/fsnotify"
    "github.com/spf13/viper"
)

// ConfigObserver defines the interface for configuration observers
type ConfigObserver interface {
    OnConfigChange(e fsnotify.Event)
}

// ViperConfigObserver is an implementation of ConfigObserver using Viper
type ViperConfigObserver struct{}

func (o *ViperConfigObserver) OnConfigChange(e fsnotify.Event) {
    fmt.Println("Config file changed:", e.Name)
    // Handle the configuration change (e.g., reload settings)
}

func WatchConfig(observer ConfigObserver) {
    viper.WatchConfig()
    viper.OnConfigChange(observer.OnConfigChange)
}
```

Strategy Pattern

- **sort.Interface:** The `sort` package provides the `Interface` type, which can be used to implement different sorting strategies.

```
package strategy

import "sort"

type SortStrategy interface {
    Sort(data sort.Interface)
}

type AscendingSort struct{}
```

```
func (s AscendingSort) Sort(data sort.Interface) {
    sort.Sort(data)
}

type DescendingSort struct{}

func (s DescendingSort) Sort(data sort.Interface) {
    sort.Sort(sort.Reverse(data))
}
```

8. Real-World Examples: Go Libraries Using Design Patterns

8.1. Singleton Pattern: `database/sql`

The `database/sql` package in Go uses the Singleton pattern to manage database connections. The `sql.DB` object is a singleton that manages a pool of connections to a database.

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "log"
)

func main() {
    db, err := sql.Open("mysql", "user:password@dbname")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // Use the db object to interact with the database
    var version string
    db.QueryRow("SELECT VERSION()").Scan(&version)
    fmt.Println("Database version:", version)
}
```

8.2. Factory Pattern: `net/http`

The `net/http` package uses the Factory pattern to create HTTP handlers. The `http.NewServeMux` function creates a new `ServeMux` object, which is a request multiplexer.

```

package main

import (
    "fmt"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, world!")
    })

    http.ListenAndServe(":8080", mux)
}

```

8.3. Observer Pattern: fsnotify

The `fsnotify` package is a cross-platform file system notification library that implements the Observer pattern. It watches for changes to files and directories and notifies registered observers.

```

package main

import (
    "fmt"
    "github.com/fsnotify/fsnotify"
    "log"
)

func main() {
    watcher, err := fsnotify.NewWatcher()
    if err != nil {
        log.Fatal(err)
    }
    defer watcher.Close()

    done := make(chan bool)
    go func() {
        for {
            select {
            case event, ok := <-watcher.Events:
                if !ok {
                    return
                }
                fmt.Println("event:", event)
                if event.Op&fsnotify.Write == fsnotify.Write {
                    fmt.Println("modified file:", event.Name)
                }
            case err, ok := <-watcher.Errors:
                if !ok {
                    return
                }
                fmt.Println("error:", err)
            }
        }
    }()
}

```



```
}()
```

```
err = watcher.Add("/path/to/watch")
```

9. References:

Books:

- “Clean Code: A Handbook of Agile Software Craftsmanship” by Robert C. Martin
- “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- “The Go Programming Language” by Alan A. A. Donovan and Brian W. Kernighan

Online Resources:

- [Effective Go](#)
- [Go Design Patterns](#)
- [Go by Example](#)
- [Go Modules](#)
- [Interface Segregation Principle](#)
- [Dependency Inversion Principle](#)

By following these guidelines and examples, you can write clean, maintainable, and efficient Go code using well-established design patterns, effective package management techniques, and the aggressive use of interfaces to promote code flexibility and decoupling.

Enjoyed the content? Support me by following on [Medium](#), [Twitter](#) and connecting on [LinkedIn](#) for more such content. Show your appreciation with claps.
Thanks for reading!



Written by SONU RAJ

27 Followers · 57 Following

Follow



Decoding System Architecture | Backend Developer | Java | Python | Go | Swift |
Passionate about Tech and Fitness | <https://www.linkedin.com/in/srajsonu/>

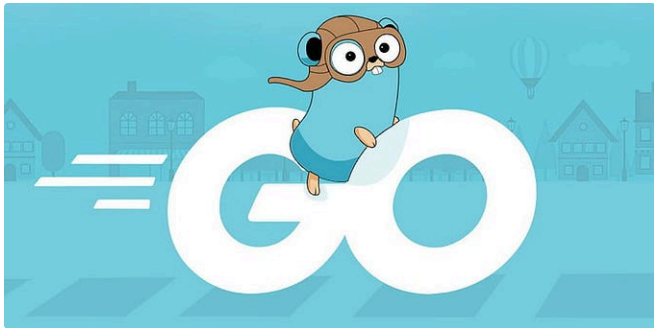
No responses yet



What are your thoughts?

Respond

More from SONU RAJ



SONU RAJ

Efficient Debugging with pprof in Go

Introduction

Jul 14, 2024 17



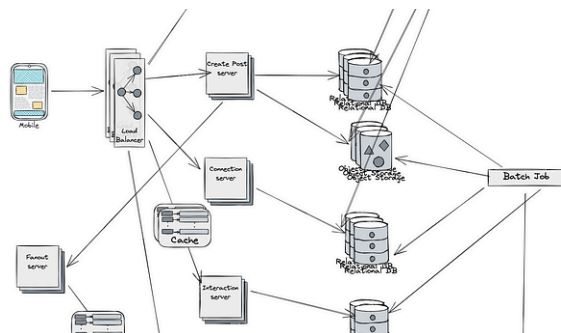
SONU RAJ

Understanding and Preventing Goroutine Leaks in Go

Checkout my previous blog for Part 2 if you haven't visited it yet: Concurrency in Go:...

Jun 8, 2024 24



 SONU RAJ


Designing a Micro-blogging Platform—Scalable, Available, an...

Introduction: Micro-blogging platforms have become a crucial part of our online social...

+

Recommended from Medium



 In Stackademic by Cheikh seck

Generic Structs with Go

Process HTTP API Responses safely.



Lists



General Coding Knowledge

20 stories · 1853 saves



Stories to Help You Grow as a Software Developer

19 stories · 1543 saves



Coding & Development

11 stories · 963 saves



Good Product Thinking

13 stories · 794 saves



ZhangJie (Kn)

Design Patterns in Go: Builder

Creational patterns address problems related to creating objects and defining their...

🌟 Oct 19, 2024 🖱️ 29



Kenzy Limon

Best Practices when developing Golang Backend Database APIs....

Welcome back to the thrilling finale of our Golang backend development series! 🎉 By...

Jan 2 🖱️ 85 💬 1



In ITNEXT by Mehran

The Facade Design Pattern in Golang

A Hidden Gem in Software Engineering

🌟 Jun 23, 2023 🖱️ 234 💬 3



In Nerd For Tech by Samuel Martins

Understanding Go's Concurrency: A Dive into Goroutines and...

Concurrency is an essential concept in modern programming, allowing software to...

🌟 Dec 8, 2024 🖱️ 11



See more recommendations