Mastering Golang: Design Patterns & Refactoring for Clean, Scalable Code

Robert Benyamin · Follow
8 min read · Nov 30, 2024

In the complex world of software development, design patterns are not just academic concepts — they are powerful tools that transform chaos into elegant, maintainable code. This comprehensive guide will explore the most critical design patterns in Golang, providing you with a robust toolkit for creating scalable, efficient, and clean software architectures.

1. Why Refactoring Matters

Imagine your codebase as a garden. Over time, without careful tending, it can become overgrown, tangled, and difficult to navigate. Refactoring is the careful pruning that keeps your code healthy, vibrant, and productive.

Refactoring involves restructuring code without altering its external behavior — think of it as an internal renovation that doesn't change the building's facade. In Golang, with its minimalist and pragmatic philosophy, the emphasis on simplicity makes refactoring both an art and a necessity.

Key Benefits:

- **Improved Readability:** Clean code is easier to understand and review.
- **Easier Debugging:** Simplified code reduces the time spent fixing bugs.
- **Scalability:** Refactored code is easier to extend and maintain.

2. Design Patterns: Building Blocks of Good Code

Design patterns are reusable solutions to common problems in software design. They help you structure code in a way that promotes flexibility, readability, and maintainability. Let's explore some of the most useful patterns in Golang.

a. Data Transfer Object (DTO) Pattern

The **DTO pattern** helps transfer data between layers or systems. By using DTOs, you can decouple internal models from external clients.

Example:

```
type UserDTO struct {
    ID    string `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
}
```

Best Practice: Use DTOs to protect your internal data models from being exposed to clients directly.

b. Repository Pattern

The **Repository pattern** abstracts data access, making it easier to swap out or mock the data source.

Example:

```
type UserRepository interface {
    FindByID(id string) (*User, error)
    Save(user *User) error
}

type userRepositoryImpl struct {
    db *sql.DB
}

func (r *userRepositoryImpl) FindByID(id string) (*User, error) {
    // Database query logic here
}
```

Best Practice: Define interfaces for your repositories to make unit testing easier and allow flexibility in changing data sources.

c. Service Layer Pattern

The **Service layer** encapsulates business logic, separating it from controllers and repositories.

Example:

```
type UserService struct {
    repo UserRepository
}

func (s *UserService) GetUser(id string) (*UserDTO, error) {
    user, err := s.repo.FindByID(id)
    if err != nil {
        return nil, err
    }
    return &UserDTO{ID: user.ID, Name: user.Name, Email: user.Email}, nil
}
```

Best Practice: Keep business logic in services to maintain a clear separation of concerns.

d. Controller Layer Pattern

Controllers handle HTTP requests and delegate tasks to services.

Example:

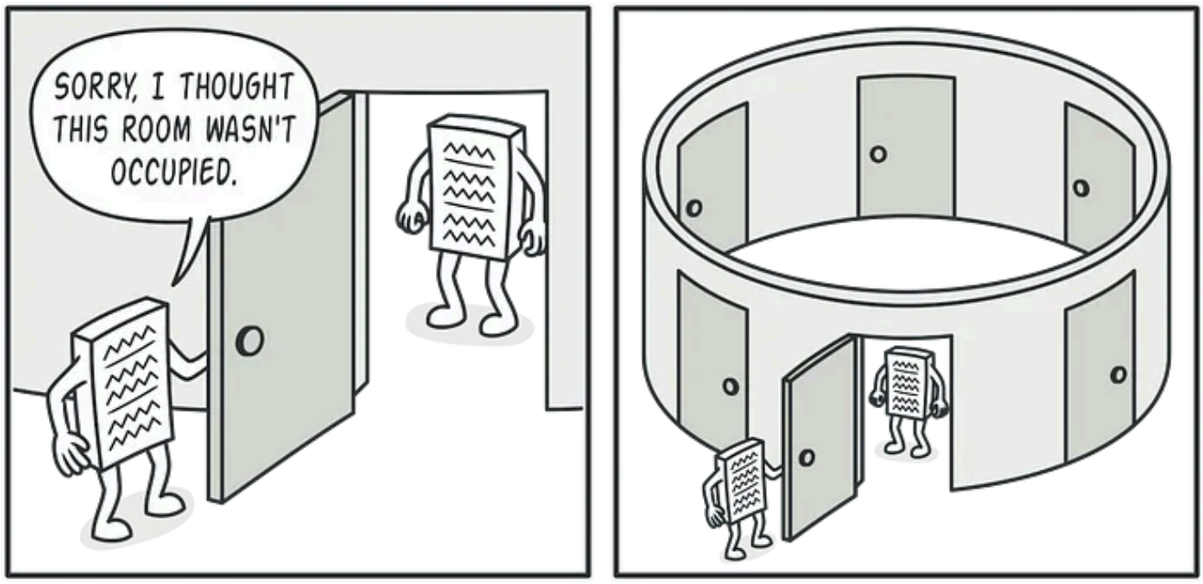
```
func (c *UserController) GetUserHandler(w http.ResponseWriter, r *http.Request)
    id := r.URL.Query().Get("id")
    user, err := c.service.GetUser(id)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    json.NewEncoder(w).Encode(user)
}
```

Best Practice: Keep controllers thin — focus only on request handling and response formatting.

3. More Design Patterns in Golang

Here are more design patterns that can help you write clean, scalable Golang code:

a. Singleton Pattern



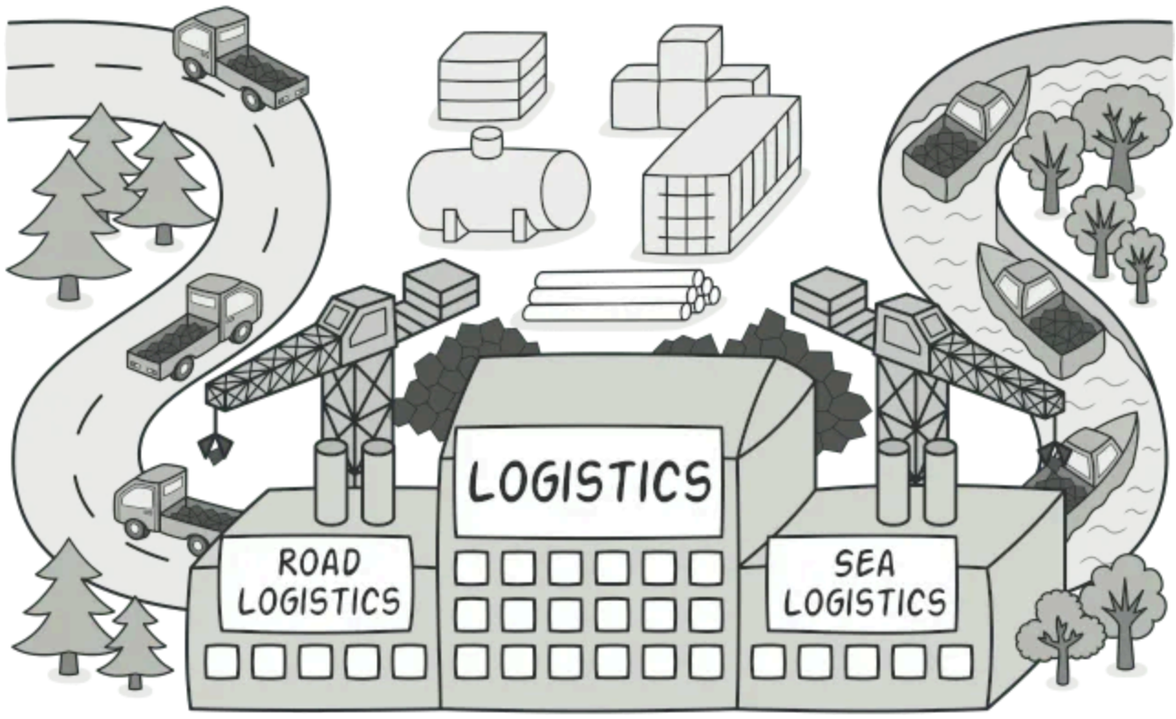
source: refactoring.guru

Ensures a single instance of a struct is created. Useful for managing configuration, logging, or database connections.

```
var instance *Config
var once sync.Once

func GetConfig() *Config {
    once.Do(func() {
        instance = &Config{Setting: "Default"}
    })
    return instance
}
```

b. Factory Pattern

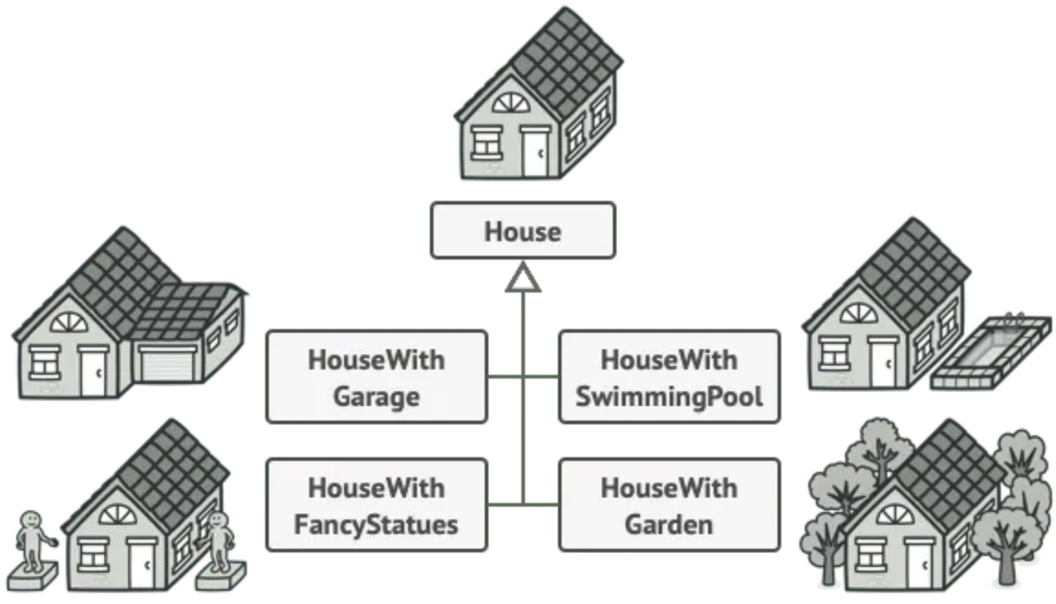


source: refactoring.guru

Creates objects without specifying their concrete type.

```
func GetAnimal(animalType string) Animal {
    switch animalType {
    case "dog":
        return Dog{}
    case "cat":
        return Cat{}
    default:
        return nil
    }
}
```

c. Builder Pattern



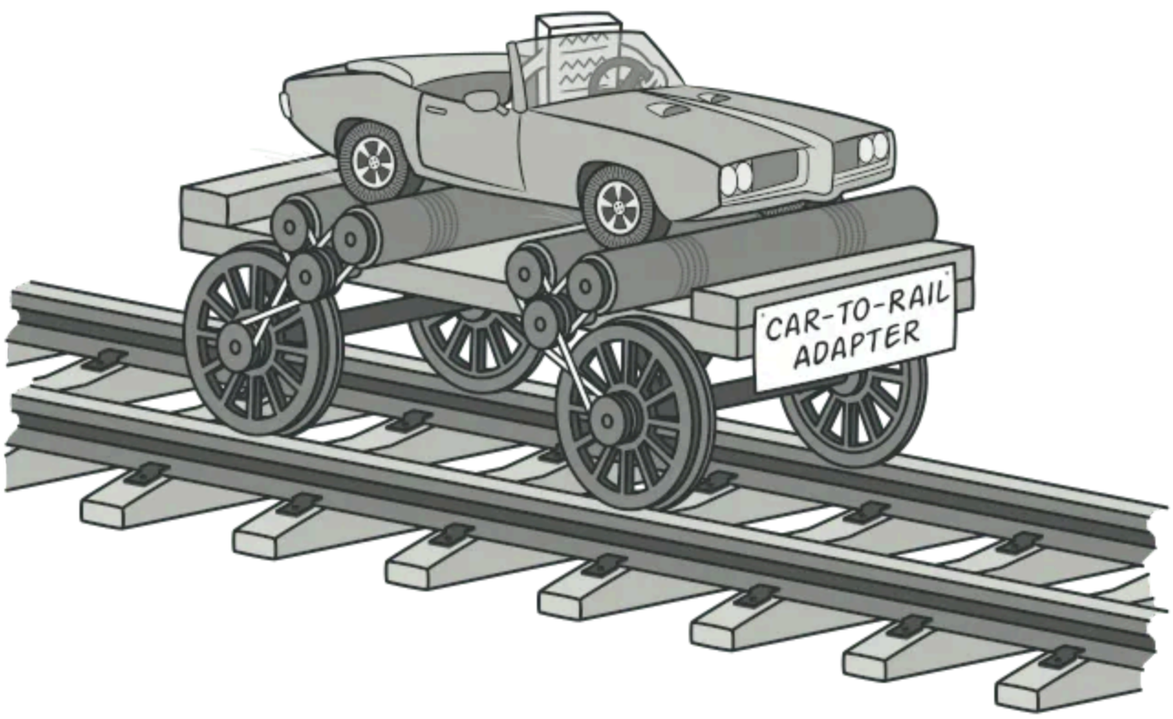
source: refactoring.guru

Simplifies the construction of complex objects.

```
type HouseBuilder struct {
    house House
}

func (b *HouseBuilder) SetWindows(w string) *HouseBuilder {
    b.house.Windows = w
    return b
}
```

d. Adapter Pattern



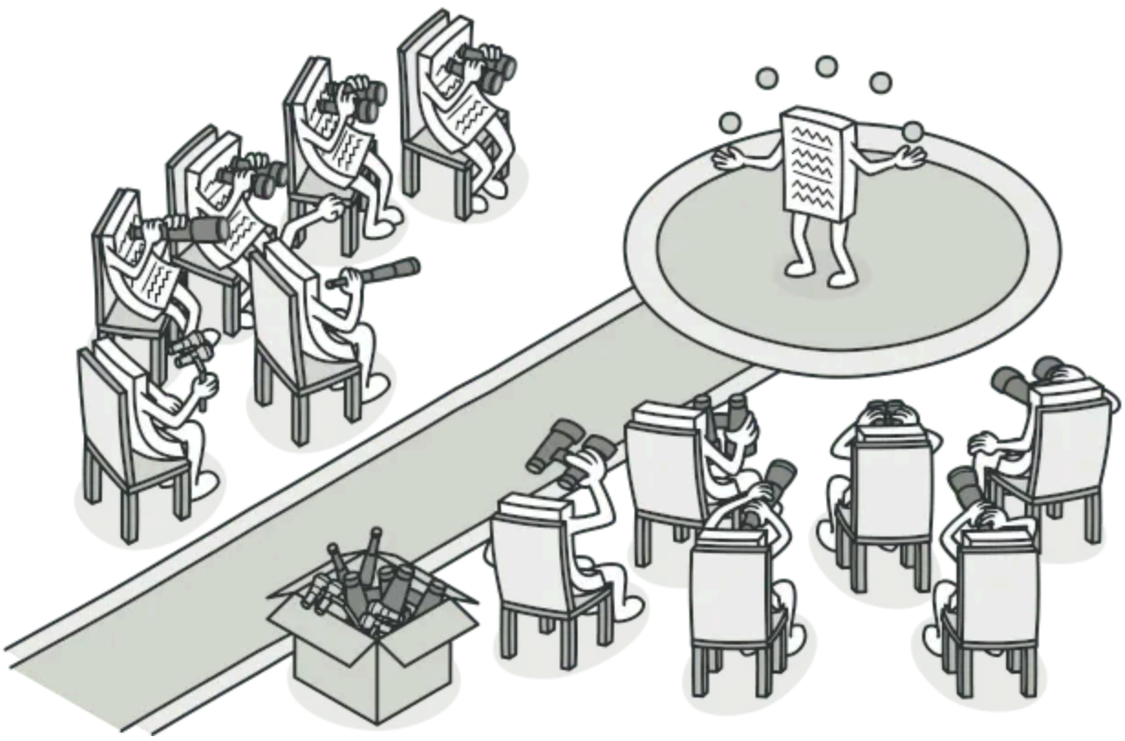
source: refactoring.guru

Allows incompatible interfaces to work together.

```
type PrinterAdapter struct {
    OldPrinter OldPrinter
}

func (p *PrinterAdapter) Print(msg string) {
    p.OldPrinter.PrintMessage(msg)
}
```

e. Observer Pattern



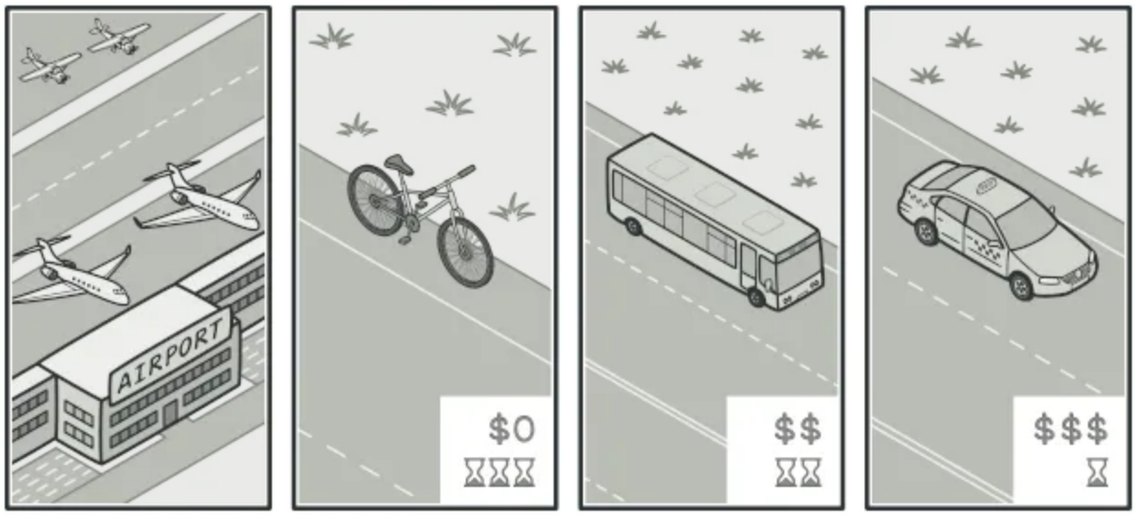
source: refactoring.guru

Notifies multiple observers about events or state changes.

```
type Observer interface {
    Update(string)
}

func (s *Subject) NotifyObservers(msg string) {
    for _, o := range s.observers {
        o.Update(msg)
    }
}
```

f. Strategy Pattern



Various strategies for getting to the airport. source: refactoring.guru

Allows you to define interchangeable algorithms.

```
type PaymentContext struct {
    strategy PaymentStrategy
}

func (p *PaymentContext) ExecutePayment(amount int) {
    p.strategy.Pay(amount)
}
```

4. Anti-Patterns to Avoid

While design patterns are useful, **anti-patterns** can lead to messy, unmaintainable code. Here are examples of common **anti-patterns** in Golang and how to refactor them into better solutions:

a. God Object Anti-Pattern

A **God Object** handles too many responsibilities, leading to tight coupling and difficult maintenance.

Anti-Pattern Example:

```
type GodObject struct {
    UserRepository UserRepository
    OrderRepository OrderRepository
    PaymentService  PaymentService
}

func (g *GodObject) ProcessOrder(userID string, orderID string) error {
    user, err := g.UserRepository.FindByID(userID)
```

```
    if err != nil {
        return err
    }
    order, err := g.OrderRepository.FindByID(orderID)
    if err != nil {
        return err
    }
    return g.PaymentService.ProcessPayment(user, order)
}
```

Solution: Break the God Object into smaller, cohesive components.

Refactored Example:

```
type OrderProcessor struct {
    UserService    UserService
    PaymentService PaymentService
}

func (o *OrderProcessor) ProcessOrder(userID string, orderID string) error {
    user, err := o.UserService.GetUser(userID)
    if err != nil {
        return err
    }
    return o.PaymentService.ProcessPayment(user, orderID)
}
```

b. Spaghetti Code Anti-Pattern

Spaghetti code lacks structure, making it hard to read and maintain.

Anti-Pattern Example:

```
func HandleRequest(w http.ResponseWriter, r *http.Request) {
    id := r.URL.Query().Get("id")
    db, err := sql.Open("mysql", "user:password@/dbname")
    if err != nil {
        http.Error(w, "Database error", http.StatusInternalServerError)
        return
    }
    defer db.Close()
    row := db.QueryRow("SELECT name FROM users WHERE id = ?", id)
    var name string
    err = row.Scan(&name)
    if err != nil {
        http.Error(w, "User not found", http.StatusNotFound)
        return
    }
```

```
    fmt.Fprintf(w, "Hello, %s!", name)
}
```

Solution: Separate concerns into different layers and functions.

Refactored Example:

```
type UserController struct {
    UserService UserService
}

func (c *UserController) HandleRequest(w http.ResponseWriter, r *http.Request) {
    id := r.URL.Query().Get("id")
    user, err := c.UserService.GetUserByID(id)
    if err != nil {
        http.Error(w, "User not found", http.StatusNotFound)
        return
    }
    fmt.Fprintf(w, "Hello, %s!", user.Name)
}

type UserService struct {
    Repo UserRepository
}

func (s *UserService) GetUserByID(id string) (*User, error) {
    return s.Repo.FindByID(id)
}
```

c. Over-Engineering Anti-Pattern

Over-engineering occurs when unnecessary abstractions make code overly complex.

Anti-Pattern Example:

```
type PaymentMethod interface {
    Pay(amount float64)
}

type CreditCardPayment struct {}
func (c CreditCardPayment) Pay(amount float64) { fmt.Println("Paid with credit c

type PayPalPayment struct {}
func (p PayPalPayment) Pay(amount float64) { fmt.Println("Paid with PayPal") }

type PaymentProcessor struct {
    Method PaymentMethod
```

```
}
func (p PaymentProcessor) Process(amount float64) {
    p.Method.Pay(amount)
}
```

Solution: Keep it simple when only one method is required.

Refactored Example:

```
type PaymentService struct {}

func (p PaymentService) Pay(amount float64) {
    fmt.Println("Paid", amount)
}
```

d. Hardcoding Configuration

Hardcoding configurations makes your code inflexible and difficult to maintain.

Anti-Pattern Example:

```
func ConnectToDB() (*sql.DB, error) {
    return sql.Open("mysql", "user:password@tcp(127.0.0.1:3306)/dbname")
}
```

Solution: Use environment variables or configuration files.

Refactored Example:

```
func ConnectToDB() (*sql.DB, error) {
    dsn := os.Getenv("DB_DSN")
    return sql.Open("mysql", dsn)
}
```

e. Tight Coupling

Tightly coupled components depend heavily on each other, making changes difficult.

Anti-Pattern Example:

```
type OrderService struct {
    db *sql.DB
}

func (o *OrderService) CreateOrder(order Order) error {
    _, err := o.db.Exec("INSERT INTO orders ...")
    return err
}
```

Solution: Use dependency injection to decouple components.

Refactored Example:

```
type OrderService struct {
    Repo OrderRepository
}

func (o *OrderService) CreateOrder(order Order) error {
    return o.Repo.Save(order)
}
```

Refactoring in Action: Design Patterns in the Indonesian Student Survey Project

In this section, we'll explore how design patterns improve code structure, maintainability, and scalability using real-world refactoring examples from the **Indonesian Student Survey Project**. The primary focus will be on refactoring the `UpdateQuestion` function and its related logic to adopt the **Strategy** and **Single Responsibility** patterns.

Problem: Handling Question Updates with Complex Logic

The original `UpdateQuestion` function was a **monolithic block** of logic, tightly coupling multiple responsibilities like:

- Validating IDs.

- Handling different types of answers (text, multiple-choice).
- Managing updates and deletions of answers based on question type changes.

Before Refactoring (Anti-Pattern: God Function):

```
func (s *questionService) UpdateQuestion(ctx context.Context, id string, questio
    // Validate Section and Question IDs
    _, err := s.sectionRepo.GetSectionByID(ctx, question.SectionID.Hex())
    if err != nil {
        return nil, ErrInvalidSectionID
    }

    objID, err := primitive.ObjectIDFromHex(id)
    if err != nil {
        return nil, ErrInvalidQuestionID
    }

    // Fetch and handle updates based on question type
    existingQuestion, err := s.questionRepo.GetQuestionByID(ctx, objID.Hex())
    if err != nil {
        return nil, ErrQuestionNotFound
    }

    if existingQuestion.Type != question.Type {
        // Delete old answers and create new ones
        oldAnswerHandler, _ := NewAnswerHandler(existingQuestion.Type, s.answerR
        oldAnswerHandler.DeleteAnswer(ctx, objID.Hex())
        newAnswerHandler, _ := NewAnswerHandler(question.Type, s.answerRepo)
        newAnswerHandler.CreateAnswer(ctx, question, listOfOptions)
    } else {
        // Update existing answers
        answerHandler, _ := NewAnswerHandler(question.Type, s.answerRepo)
        answerHandler.UpdateAnswer(ctx, objID.Hex(), question, listOfOptions)
    }

    return s.questionRepo.UpdateQuestion(ctx, objID.Hex(), question)
}
```

Refactoring Solution: Strategy Pattern & Separation of Concerns

Key Changes:

- **Strategy Pattern:** Used to abstract different answer handling logic based on the question type.
- **Helper Functions:** Extracted `validateSectionAndQuestionIDs` and `handleAnswerUpdate` to make `UpdateQuestion` cleaner.

- **Single Responsibility Principle:** Each method now focuses on a single task, improving readability and testability.


```
func (s *questionService) UpdateQuestion(ctx context.Context, id string, questio
    if err := s.validateSectionAndQuestionIDs(ctx, id, question); err != nil {
        return nil, err
    }

    existingQuestion, err := s.questionRepo.GetQuestionByID(ctx, id)
    if err != nil {
        return nil, ErrQuestionNotFound
    }

    if err := s.handleAnswerUpdate(ctx, id, existingQuestion, question, listOfOp
        return nil, err
    }

    return s.questionRepo.UpdateQuestion(ctx, id, question)
}
```

Strategy Pattern in Action: AnswerHandler Interface

We introduced an **`AnswerHandler` interface** to manage different answer types (text or multiple-choice) dynamically.

```
type AnswerHandler interface {
    CreateAnswer(ctx context.Context, question *models.Question, listOfOptions [
    UpdateAnswer(ctx context.Context, questionID string, question *models.Questi
    DeleteAnswer(ctx context.Context, questionID string) error
}

func NewAnswerHandler(answerType string, repo repositories.AnswerRepository) (An
    switch answerType {
    case "text_answer":
        return &textAnswerHandler{repo: repo}, nil
    case "multiple_answer":
        return &multipleAnswerHandler{repo: repo}, nil
    default:
        return nil, ErrInvalidAnswerType
    }
}
```

Benefits of Refactoring with Design Patterns:

- **Maintainability:** Each function and handler now has a clearly defined purpose.

- **Scalability:** Adding new answer types only requires creating a new handler without touching existing logic.
- **Readability:** Cleaner, modular code is easier to understand and modify.

This example highlights the importance of design patterns like the **Strategy Pattern** in simplifying complex logic while adhering to best practices.

Conclusion

Designing software is an art of creating intelligent, adaptable systems. The design patterns and refactoring techniques we've explored in Golang are practical strategies that transform chaotic codebases into elegant, maintainable solutions.

Great code is not about perfection, but continuous improvement. Each refactoring is a step towards a more robust system. Embrace the iterative nature of software design, challenge your existing solutions, and always seek cleaner, more modular approaches.

Golang provides a powerful toolkit for building clean architectures. Its minimalist philosophy doesn't limit you — it empowers you to create truly remarkable software.

Keep learning, keep refactoring, and never stop growing as a software craftsman. 🚀

Meme For Us To Refreshen Our Day :)

Design Patterns | Refactoring | Golang

R

Written by Robert Benyamin

0 Followers · 1 Following

Follow

No responses yet

What are your thoughts?

Respond

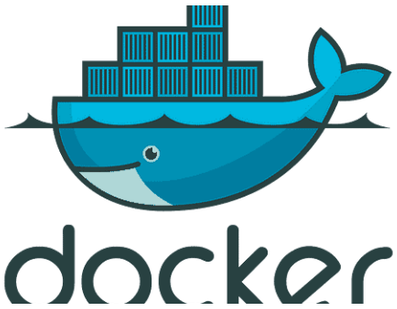
More from Robert Benyamin

Robert Benyamin

A Complete Guide to Data Migration and Seeding with Golan...

Introduction

Dec 2, 2024



Robert Benyamin

Containerization Unleashed: Revolutionizing Software...

Imagine you're a chef preparing a gourmet meal. Traditionally, you'd have to ensure your...

Dec 5, 2024

Robert Benyamin

Monitoring Unveiled: The Digital Heartbeat of Modern Technology

Imagine driving a car without a dashboard. No speedometer, no fuel gauge, no warning...

Dec 5, 2024

Robert Benyamin

SOLID Principles in Go: A Practical Guide with Real-World Examples

Ever found yourself drowning in a sea of spaghetti code? Or perhaps you've inherited...

Oct 28, 2024 1

See all from Robert Benyamin

Recommended from Medium

In Stackademic by Cheikh seck

Generic Structs with Go

Process HTTP API Responses safely.

5d ago 4

In Level Up Coding by Matt Bentley

My Top 3 Tips for Being a Great Software Architect

My top tips for being a great Software Architect and making the best decisions for...

6d ago 1.2K 16

Lists

General Coding Knowledge

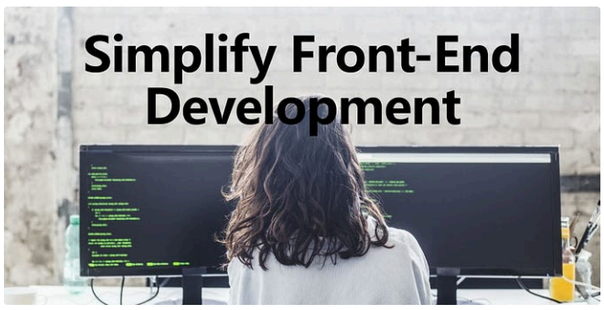
20 stories · 1853 saves

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 The Ordinary Programmer by huizhou92

HTMX First Experience

Is htmx just another JavaScript framework?

Jan 2 139

ZhangJie (Kn)

Design Patterns in Go: Builder

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

Oct 19, 2024 29

In CodeX by Rahul Sharma

Django REST Framework vs FastAPI

A Detailed Comparison With Code Exam

5d ago 298 2

See more recommendations