**C H A P T E R 13**

CHAPTER 13

# *GOF Design Patterns of Go*

# Introduction

In this chapter, we look at design patterns like Creational, Structural, and Behavioural Design patterns based on Gang of Four (GOF) Design patterns. Code samples are presented for the Gang of Four Design patterns. Readers will understand object-oriented design principles using Go Language by learning GRASP principles.

## Structure

The chapter covers the following topics:

* Creational Design Patterns
* Structural Design Patterns
* Behavioural Design Patterns
* Object Oriented Design Patterns

Figure 13.1: Design Patterns – Go Lang

## Objectives

In this chapter, we are going to look at Design patterns in Go Lang. Design Patterns help in resolving issues with solutions which make the design easy, and the applications can scale. Gang of Four Patterns term is related to a book written by 4 different authors. This book was written in 1994 and title of the book was “Design Patterns: Elements of Reusable Object-Oriented Software”. Go Lang solutions can be built using these design patterns.

.

Let us first look at Creational Design Patterns.

# Creational Design Patterns

You can use creational design patterns for designing class creators and object instantiations. These patterns help in instantiating objects agnostic of the class for which the object belongs. Creational Design Patterns are:

* Abstract Factory
* Builder
* Factory Method
* Prototype
* Singleton

Let us first look at Abstract Factory

## Abstract Factory

You can use abstract factory pattern to create a group of objects which have commonalities. Interface is defined for instantiating the group of objects agnostic of the class for which the object belongs to. Let us look at an example to create Truck Objects. Truck has an interface which has NumOfTyres and GetModelName methods.

package main

import (

"fmt"

)

const (

CivilType = 1

)

const (

RuggedModel = 1

StandardModel = 2

)

type Truck interface {

NumOfTyres() int

GetModelName() string

}

TruckFactory is an interface to create truck objects. You can have CivilTruckFactory which implements the Truck Factory Interface.

type TruckFactory interface {

Create(t int) (Truck, error)

}

type CivilTruckFactory struct{}

type RuggedModelType struct{}

func (f \*RuggedModelType) NumOfTyres() int {

return 6

}

func (f \*RuggedModelType) GetModelName() string {

return "Rugged"

}

type StandardModelType struct {

}

func (f \*StandardModelType) NumOfTyres() int {

return 6

}

func (f \*StandardModelType) GetModelName() string {

return "Standard"

}

func (i \*CivilTruckFactory) Create(t int) (Truck, error) {

switch t {

case RuggedModel:

return new(RuggedModelType), nil

case StandardModel:

return new(StandardModelType), nil

}

return nil, fmt.Errorf("There are no trucks of type %d\n", t)

}

func CreateAbstractFactory(c int) (TruckFactory, error) {

switch c {

case CivilType:

return new(CivilTruckFactory), nil

default:

return nil, fmt.Errorf("There is no factory with id %d\n", c)

}

}

func main() {

civilT, \_ := CreateAbstractFactory(CivilType)

truckM, \_ := civilT.Create(StandardModel)

truck, ok := truckM.(Truck)

if !ok {

fmt.Errorf("Invalid model")

}

fmt.Printf("%v Truck has %d tyres\n", truck.GetModelName(), truck.NumOfTyres())

}

CreateAbstractFacotry method takes the type of the truck and returns the Truck Factory class.

You can now compile and run the abstract\_factory.go. The command is shown as below:

go run abstract\_factory.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air code % go run abstract\_factory.go

Standard car has 6 wheels bhagvanarch@Bhagvans-MacBook-Air code %

Now let us look at the Builder Pattern.

## Builder

You can use builder pattern to create a composite object which has multiple objects. Let us look at an example like Truck having multiple Tyres and an engine with a capacity for driving.

package main

import (

"fmt"

"strconv"

)

type Truck struct {

Tyres int

Capacity int

}

type TruckBuildProcess interface {

SetTyreNumber() TruckBuildProcess

SetCapacity() TruckBuildProcess

GetVehicle() Truck

}

type TruckShopFloor struct {

tbuilder TruckBuildProcess

}

func (f \*TruckShopFloor) SetBuilder(tb TruckBuildProcess) {

f.tbuilder = tb

}

func (f \*TruckShopFloor) Construct() {

f.tbuilder.SetCapacity().SetTyreNumber()

}

TruckBuilder is used to set the tyres and the engine capacity.

type TruckBuilder struct {

t Truck

}

func (c \*TruckBuilder) SetTyreNumber() TruckBuildProcess {

c.t.Tyres = 6

return c

}

func (c \*TruckBuilder) SetCapacity() TruckBuildProcess {

c.t.Capacity = 4

return c

}

func (c \*TruckBuilder) GetVehicle() Truck {

return c.t

}

func (c \*TruckBuilder) Build() Truck {

return c.t

}

func main() {

shopFloor := TruckShopFloor{}

truckBuilder := &TruckBuilder{}

shopFloor.SetBuilder(truckBuilder)

shopFloor.Construct()

truck := truckBuilder.Build()

if truck.Tyres != 6 {

fmt.Errorf("It is wrong that : " + strconv.Itoa(truck.Tyres) + " tyres found")

} else {

fmt.Printf("Truck has " + strconv.Itoa(truck.Tyres) + " tyres\n")

}

}

TruckBuilder is used on the shop floor to create a truck with multiple tyres and engine with capacity to drive.

You can now compile and run the builder\_example.go. The command is shown as below:

go run builder\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air code % go run builder\_example.go

Truck has 6 tyres

bhagvanarch@Bhagvans-MacBook-Air code %

Now let us now look at Factory Method pattern.

## Factory Method

You can use Factory Method pattern to create objects agnostic of the class that it belongs to. Interface is created for object creation. Let us look at an example where we can create truck of different types such as Rugged and Standard models.

package main

import (

"fmt"

)

const (

RuggedModel = 1

StandardModel = 2

)

type Truck interface {

NumOfTyres() int

GetModelName() string

}

Truck interface has NumOfTyres and GetModelName methods.

func GetTruck(truck int) (Truck, error) {

switch truck {

case RuggedModel:

return new(RuggedModelType), nil

case StandardModel:

return new(StandardModelType), nil

default:

return nil, fmt.Errorf("Not a Known Truck Model")

}

return nil, fmt.Errorf("Not implemented yet")

}

type RuggedModelType struct{}

func (f \*RuggedModelType) NumOfTyres() int {

return 6

}

func (f \*RuggedModelType) GetModelName() string {

return "Rugged"

}

type StandardModelType struct {

}

func (f \*StandardModelType) NumOfTyres() int {

return 6

}

func (f \*StandardModelType) GetModelName() string {

return "Standard"

}

func main() {

truck, err := GetTruck(RuggedModel)

if err != nil {

fmt.Errorf("This model does not exist")

}

fmt.Printf("%v Truck has %d tyres\n", truck.GetModelName(), truck.NumOfTyres())

}

GetTruck method takes the model type and creates the truck of different types.

You can now compile and run the factory\_example.go. The command is shown as below:

go run factory\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air code % go run factory\_example.go

Rugged Truck has 6 tyres

bhagvanarch@Bhagvans-MacBook-Air code %

Now let us look at the Prototype pattern.

## Prototype

You can use prototype pattern to clone objects. This pattern is useful to create multiple cloned objects like a pool of objects. Let us look at an example for cloning Flight objects.

FlightCloner is the interface which has GetClone method which takes the number of objects to be cloned.

package main

import (

"fmt"

)

type FlightCloner interface {

GetClone(s int) (ItemInfoGetter, error)

}

const (

Boeing = 1

Bombardier = 2

Embraer = 3

)

func GetFlightsCloner() FlightCloner {

return new(FlightsCache)

}

GetFlightsCloner method returns the FlightCloner. FlightsCache struct has a method GetClone of flight type. Flight can be of multiple types like Boeing, Bombardier, and Embraer.

type FlightsCache struct{}

func (s \*FlightsCache) GetClone(flight int) (ItemInfoGetter, error) {

switch flight {

case Boeing:

newItem := \*flightPrototype

return &newItem, nil

}

return nil, fmt.Errorf("Not implemented yet")

}

type ItemInfoGetter interface {

GetInfo() string

}

type FlightEngine byte

type Flight struct {

Price float32

Model string

Engine FlightEngine

}

func (s \*Flight) GetInfo() string {

return "" + s.Model

}

var flightPrototype \*Flight = &Flight{

Price: 15.00,

Model: "777",

Engine: Boeing,

}

func (i \*Flight) GetPrice() float32 {

return i.Price

}

func main() {

flightCache := GetFlightsCloner()

if flightCache == nil {

fmt.Errorf("The current cache is not valid")

}

firstFlight, err := flightCache.GetClone(Boeing)

fmt.Printf("Flight cloned is of Model type : %s\n", firstFlight.GetInfo())

if err != nil {

fmt.Println(err)

}

if firstFlight == flightPrototype {

fmt.Errorf("firstitem cannot be equal to the white prototype")

}

}

You can now compile and run the prototype\_example.go. The command is shown as below:

go run prototype\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air code % go run prototype\_example.go

Flight cloned is of Model type : 777

bhagvanarch@Bhagvans-MacBook-Air code %

Now let us look at Singleton Pattern.

## Singleton

You can use singleton pattern to have a single instance of an object. This single instance is accessible at the global level. It can also be used in the case of a pool having a specified number of objects. Let us look at the singleton pattern in code.

package main

import (

"fmt"

"sync"

)

var singlock = &sync.Mutex{}

type singleton struct {

}

var singletonInstance \*singleton

func getSingleton() \*singleton {

if singletonInstance == nil {

singlock.Lock()

defer singlock.Unlock()

if singletonInstance == nil {

fmt.Println("Creating singleton first time")

singletonInstance = &singleton{}

} else {

fmt.Println("Singleton instance is already created.")

}

} else {

fmt.Println("Single instance already created.")

}

return singletonInstance

}

func main() {

for i := 0; i < 30; i++ {

go getSingleton()

}

fmt.Scanln()

}

getSingleton method returns the singleton instance. The multiple calls will return the single instance after first time creation.

You can now compile and run the singleton\_example.go. The command is shown as below:

go run singleton\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air code % go run singleton\_example.go

Creating singleton first time

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

Singleton instance is already created.

……

bhagvanarch@Bhagvans-MacBook-Air code %

Now let us look at Structural Design Patterns.

# Structural Design Patterns

Developers use structural design patterns to create objects which belong to large, structured class. Structural Design Patterns are mentioned below:

* Adapter
* Bridge
* Composite
* Decorator
* Façade

Let us first look at Adapter Design pattern.

## Adapter

You can use adapter when you have multiple interfaces which are different and there is a plan to integrate them to create an object. The other scenario is when you want change the current interface to a prescribed specification.

Let us look at the example where we have multiple databases and have a single client interfafce to interact and query the databases. Database Interface has the GetDatabaseInfo method which gives the name of the current database connected.

package main

import "fmt"

type MySQL struct{}

func (mysql \*MySQL) GetDBInfo() {

fmt.Println("MySQL DB")

}

type MySQLAdapter struct {

mysqlServer \*MySQL

}

func (mysql \*MySQLAdapter) GetDatabaseInfo() {

fmt.Println("MySQL adapter gets DB Info")

mysql.mysqlServer.GetDBInfo()

}

MySQL Database above has implemented the Database interface. Now let us look at PostGres Database which implements the Database Interface

type PostGres struct {

}

func (pg \*PostGres) GetDatabaseInfo() {

fmt.Println("PostGres")

}

DBClient implements the Database Interface.

type DBClient struct {

}

func (client \*DBClient) GetDatabaseInfo(db Database) {

fmt.Println("Client gets Database Info from the specific DB")

db.GetDatabaseInfo()

}

type Database interface {

GetDatabaseInfo()

}

func main() {

client := &DBClient{}

pg := &PostGres{}

client.GetDatabaseInfo(pg)

mysqlServer := &MySQL{}

mysqlAdapter := &MySQLAdapter{

mysqlServer: mysqlServer,

}

client.GetDatabaseInfo(mysqlAdapter)

}

DBClient is instantiated and multiple databases are created to be queried by the client.

You can now compile and run the adapter\_example.go. The command is shown as below:

go run adapter\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run adapter\_example.go

Client gets Database Info from the specific DB

PostGres

Client gets Database Info from the specific DB

MySQL adapter gets DB Info

MySQL DB

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Bridge Design Pattern.

## Bridge

You can use bridge design pattern to segregate the interface of an object from the actual implementation. The goal here is to have loose coupling between the implementation and the abstraction. Let us look at an example where a SQL Compiler is implemented for Postgres. SQLCompiler is an interface which execute SQL. Each Database might have different constructs and support for special query syntax. SQLCompiler is implemented by the PGSQLCompiler.

package main

import "fmt"

type SQLCompiler interface {

ExecuteSQL(string)

}

type PGSQLCompiler struct {

}

func (pgc \*PGSQLCompiler) ExecuteSQL(sql string) {

fmt.Println("PostGres executing sql- ", sql)

}

type PostGres struct {

sqlCompiler SQLCompiler

}

func (pg \*PostGres) GetDatabaseInfo() {

fmt.Println("PostGres")

}

func (pg \*PostGres) SetSQLCompiler(sql SQLCompiler) {

fmt.Println("PostGres setting SQLCompiler")

pg.sqlCompiler = sql

}

func (pg \*PostGres) ExecuteSQL(sql string) {

fmt.Println("PostGres executing sql")

pg.sqlCompiler.ExecuteSQL(sql)

}

type Database interface {

GetDatabaseInfo()

SetSQLCompiler(SQLCompiler)

ExecuteSQL(string)

}

func main() {

pg := &PostGres{}

pgsql := &PGSQLCompiler{}

pg.SetSQLCompiler(pgsql)

sql := "Select \* FROM Customer"

pg.ExecuteSQL(sql)

}

PostGres Database is instantiated and PGSQLCompiler is created to set on the PostGres Database.

You can now compile and run the bridge\_example.go. The command is shown as below:

go run bridge\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run bridge\_example.go

PostGres setting SQLCompiler

PostGres executing sql

PostGres executing sql- Select \* FROM Customer

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Composite Design Pattern.

## Composite

Developers use composite pattern to create a class which has complicated structure. Car has multiple wheels, engine, and other parts. We look at an example where composite class is created with different classes for Wheel and Engine.

package main

import "fmt"

type Car struct {

wheels []Wheel

name string

engine string

}

func (car \*Car) AddWheel(wheel Wheel) {

car.wheels = append(car.wheels, wheel)

fmt.Print("adding :")

wheel.GetType()

}

func (car \*Car) GetModel() {

fmt.Println("The Car model is", car.name)

}

type Wheel interface {

GetType()

}

type CarWheel struct {

wheelType string

}

func (wheel \*CarWheel) GetType() {

fmt.Println("The wheel type is", wheel.wheelType)

}

func main() {

car := new(Car)

car.name = "Toyota Tercel"

wheel1 := &CarWheel{wheelType: "Good year"}

car.AddWheel(wheel1)

wheel2 := &CarWheel{wheelType: "Good year"}

car.AddWheel(wheel2)

wheel3 := &CarWheel{wheelType: "Good year"}

car.AddWheel(wheel3)

wheel4 := &CarWheel{wheelType: "Good year"}

car.AddWheel(wheel4)

car.GetModel()

fmt.Println("The number of wheels in the car are :", len(car.wheels))

}

Wheel objects are created first and then the Car is created with model name and engine name. Wheels are added on the Car to create a Car Object.

You can now compile and run the composite\_example.go. The command is shown as below:

go run composite\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run composite\_example.go

adding :The wheel type is Good year

adding :The wheel type is Good year

adding :The wheel type is Good year

adding :The wheel type is Good year

The Car model is Toyota Tercel

The number of wheels in the car are : 4

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now Let us see the Decorator Pattern.

## Decorator

Decorator Pattern is used to add functionality to an object which belong to a class which does not have that functionality. The interface of the class will not be modified because of the decorator pattern. Let us look at an example where IPhoto is an interface with method get Title. Painting implements the interface IPhoto. We want to add an YellowFrame to the Painting. Yellow Frame struct is created with the photo object which implements the IPhoto interface.

package main

import "fmt"

type IPhoto interface {

getTitle() string

}

type Painting struct {

title string

}

func (paint \*Painting) getTitle() string {

return paint.title

}

type YellowFrame struct {

photo IPhoto

}

func (yframe \*YellowFrame) getTitle() string {

return yframe.photo.getTitle()

}

func main() {

painting := new(Painting)

painting.title = "Mona Lisa"

yframe := &YellowFrame{photo: painting}

fmt.Println("The tile of the yellowFramed painting is ", yframe.getTitle())

}

Paintigng is created and title is set on the painting. Yellow Frame object is created with painting object which implements the IPhoto interface.

You can now compile and run the decorator\_example.go. The command is shown as below:

go run decorator\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run decorator\_example.go

The tile of the yellowFramed painting is Mona Lisa

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Façade Design Pattern.

## Facade

Developers can use Façade design pattern to design an interface which consists of multiple interface methods. Let us now look at Employee Façade which has the instances of Employee, Address, and Profile. These objects are created using the Façade. They are persisted, modified and deleted through this Employee Business Façade.

package main

import "fmt"

type EmployeeFacade struct {

employee \*Employee

address \*Address

profile \*Profile

}

func newEmployeeFacade(empName string, line1 string, line2 string, ssn string) \*EmployeeFacade {

fmt.Println("Creating the employee Facade components")

employeeFacade := &EmployeeFacade{

employee: newEmployee(empName),

address: newAddress(line1, line2),

profile: newProfile(ssn),

}

fmt.Println("Employee Facade created")

return employeeFacade

}

func (empFacade \*EmployeeFacade) saveEmployeeBO() {

empFacade.employee.saveEmployee()

empFacade.address.saveAddress()

empFacade.profile.saveProfile()

}

type Employee struct {

name string

}

func newEmployee(empName string) \*Employee {

return &Employee{

name: empName,

}

}

func (emp \*Employee) saveEmployee() {

fmt.Println("Persisting the employee in the datastore")

}

type Address struct {

line1 string

line2 string

}

func newAddress(line1 string, line2 string) \*Address {

return &Address{

line1: line1,

line2: line2,

}

}

func (address \*Address) saveAddress() {

fmt.Println("Persisting the address in the datastore")

}

type Profile struct {

ssn string

}

func newProfile(ssn string) \*Profile {

return &Profile{ssn: ssn}

}

func (profile \*Profile) saveProfile() {

fmt.Println("Persisting theprofile in the datastore")

}

func main() {

empFacade := newEmployeeFacade("Jay Smith", "200 South Blvd", "Boston, MA 01234", "23211111")

empFacade.saveEmployeeBO()

}

Employee, Address and Profile objects are created through the EmployeeFacade. The objects are persisted in the datastore through the EmployeeFacade.

You can now compile and run the facade\_example.go. The command is shown as below:

go run facade\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run facade\_example.go

Creating the employee Facade components

Employee Facade created

Persisting the employee in the datastore

Persisting the address in the datastore

Persisting theprofile in the datastore

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Behavioural Design Patterns.

# Behavioural Design Patterns

Developers use behavioural design patterns to create objects of classes which perform tasks. Behavioural Design patterns are as follows:

* Chain of Responsibility
* Command
* Interpreter
* Iterator
* Mediator
* Memento
* Observer
* State
* Strategy
* Template Method
* Visitor

First let us look at the Chain of Responsibility Design Pattern.

## Chain of Responsibility

You can use the chain of responsibility design pattern to create a chain of objects to pass the messages and each object processes the request. Let us look at an example where Sections in the shopping store are modeled. Customer goes through different Sections. Section is an interface which has the process method and goto methods.

package main

import "fmt"

type Customer struct {

name string

parkingCompleted bool

securityCompleted bool

trialCompleted bool

paymentCompleted bool

welcomeCompleted bool

}

type PaymentCounter struct {

next Section

}

func (c \*PaymentCounter) process(customer \*Customer) {

if customer.paymentCompleted {

fmt.Println("Customer has paid the bill")

}

fmt.Println("PaymentCounter getting money from customer")

}

func (pc \*PaymentCounter) goTo(next Section) {

pc.next = next

}

type Section interface {

process(\*Customer)

goTo(Section)

}

type SectionBase struct {

nextSection Section

}

Customer goes through the welcome, parking, trial, and payment sections to complete the shopping journey.

type Trial struct {

next Section

}

func (tr \*Trial) process(customer \*Customer) {

if customer.trialCompleted {

fmt.Println("Customer has completed the trial")

tr.next.process(customer)

return

}

fmt.Println("Trial performed by the customer")

customer.trialCompleted = true

tr.next.process(customer)

}

func (tr \*Trial) goTo(next Section) {

tr.next = next

}

type Security struct {

next Section

}

func (sec \*Security) process(customer \*Customer) {

if customer.securityCompleted {

fmt.Println("Customer has completed the parking")

sec.next.process(customer)

return

}

fmt.Println("Security check performed to the customer")

customer.securityCompleted = true

sec.next.process(customer)

}

func (sec \*Security) goTo(next Section) {

sec.next = next

}

type Parking struct {

next Section

}

func (pr \*Parking) process(customer \*Customer) {

if customer.parkingCompleted {

fmt.Println("Customer has completed the parking")

pr.next.process(customer)

return

}

fmt.Println("Parking completed by the customer")

customer.parkingCompleted = true

pr.next.process(customer)

}

func (pr \*Parking) goTo(next Section) {

pr.next = next

}

type Welcome struct {

next Section

}

func (wel \*Welcome) process(customer \*Customer) {

if customer.welcomeCompleted {

fmt.Println("Customer was welcomed")

wel.next.process(customer)

return

}

fmt.Println("Customer is getting welcomed")

customer.welcomeCompleted = true

wel.next.process(customer)

}

func (wel \*Welcome) goTo(next Section) {

wel.next = next

}

func main() {

pc := &PaymentCounter{}

trial := &Trial{}

trial.goTo(pc)

security := &Security{}

security.goTo(trial)

parking := &Parking{}

parking.goTo(security)

customer := &Customer{name: "John Smith"}

welcome := &Welcome{}

welcome.goTo(parking)

welcome.process(customer)

}

A chain of sections is created for processing customers coming into the shopping store. Customer goes through different sections and each section processes the customer request for parking, security check, product trial, and payments.

You can now compile and run the chain\_of\_responsibility\_example.go. The command is shown as below:

go run chain\_of\_responsibility \_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run chain\_of\_responsibility.go

Customer is getting welcomed

Parking completed by the customer

Security check performed to the customer

Trial performed by the customer

PaymentCounter getting money from customer

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Command Design Pattern.

## Command

Developers can use command design pattern to model operations which needs to be processed. The goal is to have loose coupling between the operation and the actual implementation. Let us look at the implementation of this pattern in the case of the Mobile Phone. Command interface has process method. Multiple keys on phone exist like On, Off, Dial, and the characters/numbers. Each key is a command to be executed on the mobile phone. The how part depends on the mobile phone vendor. Command Pattern helps in decoupling the process part from the mobile phone interface.

package main

import "fmt"

type Command interface {

process()

}

type MobilePhone struct {

isOn bool

}

func (p \*MobilePhone) switchOn() {

p.isOn = true

fmt.Println("The mobile is on")

}

func (p \*MobilePhone) switchOff() {

p.isOn = false

fmt.Println("The mobile is off")

}

func (p \*MobilePhone) dial() {

if p.isOn {

fmt.Println("The mobile is dialing the number")

}

}

Phone interface has methods switchOn, switchOff, and dial methods.

type Phone interface {

switchOn()

switchOff()

dial()

}

type Key struct {

command Command

}

func (b \*Key) press() {

b.command.process()

}

type SwitchOnCommand struct {

phone Phone

}

func (sw \*SwitchOnCommand) process() {

sw.phone.switchOn()

}

type SwitchOffCommand struct {

phone Phone

}

func (sw \*SwitchOffCommand) process() {

sw.phone.switchOff()

}

type DialCommand struct {

phone Phone

number string

}

func (dial \*DialCommand) process() {

dial.phone.dial()

}

func main() {

mobile := &MobilePhone{}

onCommand := &SwitchOnCommand{

phone: mobile,

}

offCommand := &SwitchOffCommand{

phone: mobile,

}

dialCommand := &DialCommand{

phone: mobile,

number: "321191911",

}

onKey := &Key{

command: onCommand,

}

onKey.press()

dialKey := &Key{

command: dialCommand,

}

dialKey.press()

offKey := &Key{

command: offCommand,

}

offKey.press()

}

MobilePhone object is created and Commands with keys are created for the mobile phone interface. Each key is pressed and the commands gets executed to respond to the key.

You can now compile and run the command\_example.go. The command is shown as below:

go run command\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run command\_example.go

The mobile is on

The mobile is dialing the number

The mobile is off

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

## Interpreter

You can use the interpreter design pattern to have a class to interpret an language.

Let us look at an example where lexer struct has methods getNumTokens, process, and the other methods to analyze the sentences.

package main

import (

"errors"

"fmt"

"strconv"

"strings"

)

type lexer struct {

sentence string

}

func (i \*lexer) getNumTokens() []string {

return strings.Split(i.sentence, " ")

}

func (i \*lexer) process() int {

sum := 0

tokens := i.getNumTokens()

for k, item := range tokens {

if item == "\*" {

fmt.Println(i.getNumTokens())

a, \_ := strconv.Atoi(string(tokens[k-1]))

b, \_ := strconv.Atoi(string(tokens[k+1]))

return a \* b

}

if item != "+" {

number, \_ := strconv.Atoi(item)

sum += number

}

}

return sum

}

func keyedStrMap() map[string]string {

var m map[string]string

m = make(map[string]string)

m["+"] = "plus"

return m

}

func (i \*lexer) has(s string) bool {

m := keyedStrMap()

if \_, ok := m[s]; ok {

return true

}

return false

}

func (i \*lexer) checkNormal(s string) error {

if s == "normal" {

return errors.New("non va")

}

i.sentence = s

return nil

}

func (i \*lexer) getCountWords() int {

s := i.getNumTokens()

return len(s)

}

Lexer has methods to count the number of words in the sentences, and process the sentence for natural language grammar

func main() {

sentence := "This is a sentence"

lex := lexer{}

lex.checkNormal(sentence)

fmt.Println("No of words in a sentence", lex.getCountWords())

if lex.getCountWords() != 2 {

fmt.Println("sentence has more than 2 words")

}

sentence = "6 \* 8"

lex.checkNormal(sentence)

val := lex.process()

fmt.Println("Product of 6 and 8 is", val)

if val != 48 {

fmt.Println([]string{

"Multiplication between 6 and 8",

"shouldnt be",

strconv.Itoa(val),

})

}

}

In the example above, multiplication is expressed as in English – the product of two numbers. This expression is processed by the lexer to give the product.

You can now compile and run the intrepreter\_example.go. The command is shown as below:

go run intrepreter \_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run interpreter\_example.go

No of words in a sentence 4

sentence has more than 2 words

[6 \* 8]

Product of 6 and 8 is 48

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Interpreter design pattern.

## Iterator

You can use the iterator design pattern to create process a group of objects in order. This can be done agnostic of the class for which the object belongs to. Let us see an example where Iterator interface is created to traverse the group of objects in order.

package main

import "fmt"

type Iterator interface {

hasNext() bool

getNext() \*Customer

}

type Collection interface {

newIterator() Iterator

}

Customer struct and Customer Collection are defined. Customer Collection has newIterator method to create an Iterator.

type Customer struct {

name string

ssn string

}

type CustomerCollection struct {

customers []\*Customer

}

func (cc \*CustomerCollection) newIterator() Iterator {

return &CustomerIterator{

customers: cc.customers,

}

}

type CustomerIterator struct {

index int

customers []\*Customer

}

func (ci \*CustomerIterator) hasNext() bool {

if ci.index < len(ci.customers) {

return true

}

return false

}

func (ci \*CustomerIterator) getNext() \*Customer {

if ci.hasNext() {

customer := ci.customers[ci.index]

ci.index++

return customer

}

return nil

}

CustomerIterator has methods to traverse an array of Customers.

func main() {

customer1 := &Customer{

name: "Andy Gerberg",

ssn: "321324234",

}

customer2 := &Customer{

name: "Bill Lalelle",

ssn: "3213223538",

}

customerCollection := &CustomerCollection{

customers: []\*Customer{customer1, customer2},

}

citerator := customerCollection.newIterator()

for citerator.hasNext() {

customer := citerator.getNext()

fmt.Println("Customer is ", customer.name)

}

}

Customer objects are created to form a customerCollection. CustomerCollection has method newIterator to create an Iterator. The iterator has methods hasNext, and getNext.

You can now compile and run the interator\_example.go. The command is shown as below:

go run iterator\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run iterator\_example.go

Customer is Andy Gerberg

Customer is Bill Lalelle

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Mediator Design Pattern.

## Mediator

You can use the mediator design pattern to have loose coupling between the mediator object and the objects which interact and communicate with the mediator.

Let us see an example where Mediator interface has methods canLand and notifyAboutTakeOff.

.

package main

import "fmt"

type Mediator interface {

canLand(Plane) bool

notifyAboutTakeOff()

}

type Plane interface {

land()

takeOff()

allowLanding()

}

Plane interface has methods land, takeOff, and allowLanding.

type AirTrafficController struct {

isRunwayFree bool

planeQueue []Plane

}

func newAirTrafficController() \*AirTrafficController {

return &AirTrafficController{

isRunwayFree: true,

}

}

func (s \*AirTrafficController) canLand(t Plane) bool {

if s.isRunwayFree {

s.isRunwayFree = false

return true

}

s.planeQueue = append(s.planeQueue, t)

return false

}

func (s \*AirTrafficController) notifyAboutTakeOff() {

if !s.isRunwayFree {

s.isRunwayFree = true

}

if len(s.planeQueue) > 0 {

firstPlaneInQueue := s.planeQueue[0]

s.planeQueue = s.planeQueue[1:]

firstPlaneInQueue.allowLanding()

}

}

AirTrafficController struct implements the mediator pattern and has the planeQueue.

type PassengerPlane struct {

mediator Mediator

}

func (g \*PassengerPlane) land() {

if !g.mediator.canLand(g) {

fmt.Println("PassengerPlane: Landing stopped, waiting")

return

}

fmt.Println("PassengerPlane: Landed")

}

func (g \*PassengerPlane) takeOff() {

fmt.Println("PassengerPlane: Leaving")

g.mediator.notifyAboutTakeOff()

}

func (g \*PassengerPlane) allowLanding() {

fmt.Println("PassengerPlane: Landing permitted, Landing")

g.land()

}

PassengerPlane has the mediator object and messages were notified to the mediator object to process.

func main() {

airTrafficController := newAirTrafficController()

passengerPlane1 := &PassengerPlane{

mediator: airTrafficController,

}

passengerPlane2 := &PassengerPlane{

mediator: airTrafficController,

}

passengerPlane1.land()

passengerPlane2.land()

passengerPlane1.takeOff()

}

You can now compile and run the mediator\_example.go. The command is shown as below:

go run mediator\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run mediator\_example.go

PassengerPlane: Landed

PassengerPlane: Landing stopped, waiting

PassengerPlane: Leaving

PassengerPlane: Landing permitted, Landing

PassengerPlane: Landed

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Memento Design Pattern.

## Memento

Developers can use the Memento Design Pattern to persist and retrieve the state of the object. Memento example mentioned below has the state information as the struct property.

package main

import "fmt"

type Memento struct {

state string

}

func (m \*Memento) retrieveState() string {

return m.state

}

type AssociateManager struct {

mementoArray []\*Memento

}

func (c \*AssociateManager) append(m \*Memento) {

c.mementoArray = append(c.mementoArray, m)

}

func (c \*AssociateManager) find(index int) \*Memento {

return c.mementoArray[index]

}

type Manager struct {

state string

}

func (e \*Manager) newMemento() \*Memento {

return &Memento{state: e.state}

}

func (e \*Manager) retrieveMemento(m \*Memento) {

e.state = m.retrieveState()

}

func (e \*Manager) update(state string) {

e.state = state

}

func (e \*Manager) retrieveState() string {

return e.state

}

Manager and Associate Manager objects are created. The state of the application is passed on from the branch manager to assocManager. The state is retrieved from the associate Manager through memento pattern.

func main() {

assocManager := &AssociateManager{

mementoArray: make([]\*Memento, 0),

}

branchManager := &Manager{

state: "state1",

}

fmt.Println("Manager Current State is", branchManager.retrieveState())

assocManager.append(branchManager.newMemento())

branchManager.update("state2")

fmt.Println("Manager Current State is", branchManager.retrieveState())

assocManager.append(branchManager.newMemento())

branchManager.update("state3")

fmt.Println("Manager Current State is", branchManager.retrieveState())

assocManager.append(branchManager.newMemento())

branchManager.retrieveMemento(assocManager.find(1))

fmt.Println("Restored to State is", branchManager.retrieveState())

branchManager.retrieveMemento(assocManager.find(0))

}

The above example shows how the account application is processed in a bank from the branch manager to associate manager. Branch manager can retrieve the state of the application from the associate manager.

You can now compile and run the memento\_example.go. The command is shown as below:

go run memento\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run memento\_example.go

Manager Current State is state1

Manager Current State is state2

Manager Current State is state3

Restored to State is state2

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Observer Design pattern.

## Observer

You can use observer design pattern to design a class for observing the objects of the other class and how the object’s state changes. Let us look at an example where Observer and Subject interfaces are defined.

package main

import "fmt"

type Observer interface {

refresh(string)

retrieveIdentifier() string

}

type Subject interface {

addObserver(observer Observer)

removeObserver(observer Observer)

updateAll()

}

Student type is defined with ObserverList as a property. Student struct implements the Subject interface.

type Student struct {

observerList []Observer

name string

allocated bool

}

func newStudent(name string) \*Student {

return &Student{

name: name,

}

}

func (s \*Student) updateAvailability() {

fmt.Println("Teacher is not allocated to Student", s.name)

s.allocated = true

s.updateAll()

}

func (s \*Student) addObserver(o Observer) {

s.observerList = append(s.observerList, o)

}

func (s \*Student) removeObserver(o Observer) {

s.observerList = removeFromlist(s.observerList, o)

}

func (s \*Student) updateAll() {

for \_, observer := range s.observerList {

observer.refresh(s.name)

}

}

func removeFromlist(observerList []Observer, observerToRemove Observer) []Observer {

observerListLength := len(observerList)

for i, observer := range observerList {

if observerToRemove.retrieveIdentifier() == observer.retrieveIdentifier() {

observerList[observerListLength-1], observerList[i] = observerList[i], observerList[observerListLength-1]

return observerList[:observerListLength-1]

}

}

return observerList

}

Teacher struct implements the Observer interface.

type Teacher struct {

id string

}

func (t \*Teacher) refresh(itemName string) {

fmt.Println("Sending email to teacher", t.id)

}

func (t \*Teacher) retrieveIdentifier() string {

return t.id

}

func main() {

student := newStudent("Nick Jordan")

teacher1 := &Teacher{id: "teacher1@gmail.com"}

teacher2 := &Teacher{id: "teacher2@gmail.com"}

student.addObserver(teacher1)

student.addObserver(teacher2)

student.updateAvailability()

}

Student object and Teacher objects are created. Teachers are added to the student as the observers. Student can notify and update his availability by sending email.

You can now compile and run the observer\_example.go. The command is shown as below:

go run observer\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run observer\_example.go

Teacher is not allocated to Student Nick Jordan

Sending email to teacher teacher1@gmail.com

Sending email to teacher teacher2@gmail.com

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the State Design Pattern.

## State

You can use state design pattern to create an object and modify the object’s state. This pattern helps in changing the object’s behavior. Code below shows the State interface which has requestMoney, insertMoney, and dispenseMoney methods. ATM struct has multiple states and money as the property.

package main

import "fmt"

type State interface {

requestMoney(int)

insertMoney(int)

dispenseMoney(int)

}

type ATM struct {

hasMoney State

noMoney State

MoneyRequested State

presentState State

money int

}

func newATM(money int) \*ATM {

atm := &ATM{

money: money,

}

hasMoneyState := &HasMoneyState{

atm: atm,

}

moneyRequestedState := &RequestedMoneyState{

atm: atm,

}

noMoneyState := &NoMoneyState{

atm: atm,

}

atm.setState(hasMoneyState)

atm.MoneyRequested = moneyRequestedState

atm.hasMoney = hasMoneyState

atm.noMoney = noMoneyState

return atm

}

ATM implements the State interface by having methods requestMoney, insertMoney, and dispenseMoney.

func (atm \*ATM) requestMoney(money int) {

atm.presentState.requestMoney(money)

}

func (atm \*ATM) insertMoney(money int) {

atm.presentState.insertMoney(money)

}

func (atm \*ATM) dispenseMoney(money int) {

atm.presentState.dispenseMoney(money)

}

func (atm \*ATM) setState(s State) {

atm.presentState = s

}

func (atm \*ATM) incrementMoney(money int) {

fmt.Println("Adding money", money)

atm.money = atm.money + money

if atm.money > 0 {

atm.setState(atm.hasMoney)

}

}

func (atm \*ATM) decrementMoney(money int) {

fmt.Println("Decreasing money", money)

if money <= atm.money {

atm.money = atm.money - money

if atm.money == 0 {

atm.setState(atm.noMoney)

}

} else {

fmt.Println("Not enough money to decrement")

}

}

HasMoneyState, NoMoneyState, and RequestedMoneyState structs are defined below. Each of these structs implement State interface.

type HasMoneyState struct {

atm \*ATM

}

func (i \*HasMoneyState) requestMoney(money int) {

if money <= i.atm.money {

fmt.Println("Money in ATM", i.atm.money)

} else {

fmt.Println("Not Enough money in ATM", i.atm.money)

}

}

func (i \*HasMoneyState) dispenseMoney(money int) {

fmt.Println("Money dispense in progress", money)

i.atm.decrementMoney(money)

}

func (i \*HasMoneyState) insertMoney(money int) {

fmt.Println("Money is there- not added", i.atm.money)

}

type NoMoneyState struct {

atm \*ATM

}

func (i \*NoMoneyState) requestMoney(money int) {

if money <= i.atm.money {

fmt.Println("Money in ATM")

} else {

fmt.Println("Not Enough money in ATM")

}

}

func (i \*NoMoneyState) dispenseMoney(money int) {

fmt.Println("No money")

}

func (i \*NoMoneyState) insertMoney(money int) {

fmt.Println("Money is added now", money)

i.atm.incrementMoney(money)

}

type RequestedMoneyState struct {

atm \*ATM

}

func (i \*RequestedMoneyState) requestMoney(money int) {

fmt.Println("Money requested", money)

}

func (i \*RequestedMoneyState) dispenseMoney(money int) {

fmt.Println("dispense will start soon")

}

func (i \*RequestedMoneyState) insertMoney(money int) {

fmt.Println("Money is requested and it cannot be added")

}

ATM is created with newATM method by specifying the initial money as 10 units.

func main() {

atm := newATM(10)

atm.requestMoney(10)

atm.dispenseMoney(10)

atm.insertMoney(500)

}

In the above example, request for 10 units of money is sent. ATM dispenses 10 units of money. 500 units of Money are inserted in the ATM.

You can now compile and run the state\_example.go. The command is shown as below:

go run state\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run state\_example.go

Money in ATM 10

Money dispense in progress 10

Decreasing money 10

Money is added now 500

Adding money 500

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Strategy Design Pattern.

## Strategy

You can use the strategy design pattern to create a group of techniques, approaches, and methods and implement any of them when required. Let us see an example where DeductionStrategy interface is defined with getDeduction methods. Product struct has deductionstrategy has the property.

package main

import "fmt"

type DeductionStrategy interface {

getDeduction(int) int

}

type Product struct {

name string

deductionStrategy DeductionStrategy

cost int

}

func (prod \*Product) getCost() {

prod.cost = prod.cost - prod.deductionStrategy.getDeduction(prod.cost)

fmt.Println("Cost of the Product after deduction", prod.cost)

}

FixedDeduction and PropDeduction strategies are created with fixed cost deduction and proportionate deduction respectively.

type FixedDeduction struct {

}

func (deduction \*FixedDeduction) getDeduction(cost int) int {

return 10

}

type PropDeduction struct {

}

func (deduction \*PropDeduction) getDeduction(cost int) int {

return int(10 \* cost / 100)

}

func main() {

fixedDeduction := &FixedDeduction{}

propDeduction := &PropDeduction{}

product1 := &Product{name: "IPad", deductionStrategy: propDeduction, cost: 30000}

product1.getCost()

product2 := &Product{name: "Diary", deductionStrategy: fixedDeduction, cost: 30}

product2.getCost()

}

FixedDeduction and PropDeduction objects are created. Product object is created with name and proportionate & fixed deduction. In both cases, getCost method returns the calculated cost after deduction.

You can now compile and run the strategy\_example.go. The command is shown as below:

go run strategy\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run strategy\_example.go

Cost of the Product after deduction 27000

Cost of the Product after deduction 20

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the Template Method design Pattern.

## Template Method

You can use the template design pattern to create a class to define the technique. The class which subclasses from the template class can have different implementations. Content Manager interface has getContent method. Document implements the interface.

package main

import (

"fmt"

"strings"

)

type ContentManager interface {

getContent() string

}

type Document struct {

content string

}

func (doc \*Document) getContent() string {

return doc.content

}

Template interface has methods intro,desc, and placeholder.

type Template interface {

intro() string

desc() string

placeHolder(ContentManager) string

}

FixedTemplate and DynamicTemplate implement Template interface but have different implementations of the methods – intro, desc, and placeholder.

type FixedTemplate struct{}

func (t \*FixedTemplate) intro() string {

return "This is the introduction section"

}

func (t \*FixedTemplate) desc() string {

return "This is the description section"

}

func (t \*FixedTemplate) placeholder(cm ContentManager) string {

return strings.Join(

[]string{

t.intro(),

t.desc(),

cm.getContent(),

},

" ",

)

}

type DynamicTemplate struct {

intro\_text string

desc\_text string

}

func (a \*DynamicTemplate) intro() string {

return a.intro\_text

}

func (a \*DynamicTemplate) desc() string {

return a.desc\_text

}

func (a \*DynamicTemplate) placeholder(cm ContentManager) string {

return strings.Join(

[]string{

a.intro(),

a.desc(),

cm.getContent(),

},

" ",

)

}

func main() {

doc := &Document{content: "This is a detailed section"}

fixed := &FixedTemplate{}

document := fixed.placeholder(doc)

fmt.Println("Fixed Template based document is", document)

dynamic := &DynamicTemplate{intro\_text: "Dynamic introduction", desc\_text: "Dynamic Description"}

dynamic\_text := dynamic.placeholder(doc)

fmt.Println("Dynamic Template based document is", dynamic\_text)

}

In the example above, a document is created with different templates – fixed and dynamic. The document object has the initial text which is passed on to the template to change the document to the specified template.

You can now compile and run the template\_example.go. The command is shown as below:

go run template\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run template\_example.go

Fixed Template based document is This is the introduction section This is the description section This is a detailed section

Dynamic Template based document is Dynamic introduction Dynamic Description This is a detailed section

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now let us look at the visitor design pattern.

## Visitor

Developers use visitor pattern to segregate the technique from the object by defining a new object from a class which defines the technique. This also helps to incorporate multiple techniques for the same object through visitor. Let us look at the example below where Vistior interface is defined with getTruckVelocity and get CarVelocity methodsl

package main

import (

"fmt"

)

type Visitor interface {

getTruckVelocity(\*Truck) int

getCarVelocity(\*Car) int

}

Truck and Car structs are defined with acceptVisitor method

type Truck struct {

highestSpeed int

}

func (t \*Truck) acceptVisitor(v Visitor) {

fmt.Println("Velocity of the truck is", v.getTruckVelocity(t))

}

func (t \*Truck) getVehicleType() string {

return "Truck"

}

func (t \*Truck) getTruckVelociy() int {

return t.highestSpeed

}

type Car struct {

highestSpeed int

}

func (c \*Car) acceptVisitor(v Visitor) {

fmt.Println("Velocity of the Car is:", v.getCarVelocity(c))

}

func (c \*Car) getVehicleType() string {

return "Car"

}

VehicleVelocity is the visitor class which has the truck and car velocity methods.

type VehicleVelocity struct {

}

func (v \*VehicleVelocity) getTruckVelocity(t \*Truck) int {

return t.highestSpeed

}

func (v \*VehicleVelocity) getCarVelocity(c \*Car) int {

return c.highestSpeed

}

func main() {

truck := &Truck{highestSpeed: 160}

car := &Car{highestSpeed: 120}

velocity := &VehicleVelocity{}

truck.acceptVisitor(velocity)

car.acceptVisitor(velocity)

}

In the example above, VehicleVelocity visitor separates the speed calculation like an odometer in a truck and car.

You can now compile and run the visitor\_example.go. The command is shown as below:

go run visitor\_example.go

The output will be as shown below:

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns % go run visitor\_example.go

Velocity of the truck is 160

Velocity of the Car is: 120

bhagvanarch@Bhagvans-MacBook-Air Behavioural Design Patterns %

Now look at the Object-Oriented Design Patterns.

# Object Oriented Design Patterns

While developing software, different methodologies like OOD, DDD and other methodologies are used. Object Oriented Design (OOD) focuses on the classes, their responsibilities, and the collaboration between the classes. General Responsibility Assignment Software Patterns (GRASP) are the principles which help in Object oriented design.

GRASP Patterns are based on the principles:

* Creation of Objects
* Information Expertise
* Low Coupling
* High Cohesion
* Controller
* Polymorphism
* Abstraction
* Pure Fabrication

Now let us look at these principles in detail.

## GRASP Patterns

GRASP principles are used while designing using object-oriented design methodology. These principles help in designing the systems which are flexible, adaptable, resilient, fail proof, maintainable, and extensible. Future requirements can be added to the upcoming releases of the software easily.

### Creator

This principle is about the class which creates objects. The creator class needs to be decoupled from the objects of the classes which are created. The creator class can be an aggregate or a composite. This principle helps in determining which class takes the responsibility of object creation.

### Information Expertise

This principle emphasizes the importance of the class having the information related to the task to be processed. The goal is to have cohesiveness and lessen the dependencies in the classes. The class need to responsible for a single task and have all the information related to the task.

### Low Coupling

Low coupling between the classes is basis of this principle. Classes need to have lesser interconnections. The goal is to have a design which is modular and flexible. This makes the software easy to maintain. Change requests can be handled easily as new features can be added easily. This improves the reusability.

### High Cohesion

High cohesion principle states the class design needs to factor in encapsulation and single responsibility for a class. This will help in improving the reuse and handle changes in future.

Each class needs to have well defined purpose and focus.

### Controller

Developers use controller principle to model classes which are mediators between the UI and the business logic layer. These controller classes help in processing the client requests from the UI to invoke business layer classes and respond to the User interface with response. This helps in decoupling the different aspects of the design. Controller class is responsible for system operations encapsulation.

### Polymorphism

Polymorphism principle is an object-oriented design methodology principle. Multiple classes can have a single interface. The goal is to have design adaptable and extensible. This principle helps in reducing the coupling and improving the reuse. The subclasses can be easily referred to the super class because of the polymorphism principles.

### Abstraction

Single Layer of Abstraction (SLAP) principle encourages the abstraction in the design and to have layered design to handle presentation, business, external interface integration, data storage, and data persistence features.

### Pure Fabrication

Pure Fabrication principle is used for creating classes which are not related to the context. These classes are helper classes and help in decoupling the functionality from the domain objects. They help in improving cohesion and reducing the coupling with the external systems. Some of the helper classes can be related to input and output handling, data format transformation, and data persistence.

# Conclusion

In this chapter we have covered topics related to Gang of Four Design Patterns and Object-oriented design patterns.

* We looked at Creational Design Patterns with examples and where they can be applied.
* Behavioural and Structural design patterns were discussed in detail with context and examples.
* Object oriented design principles and GRASP patterns were presented in detail.