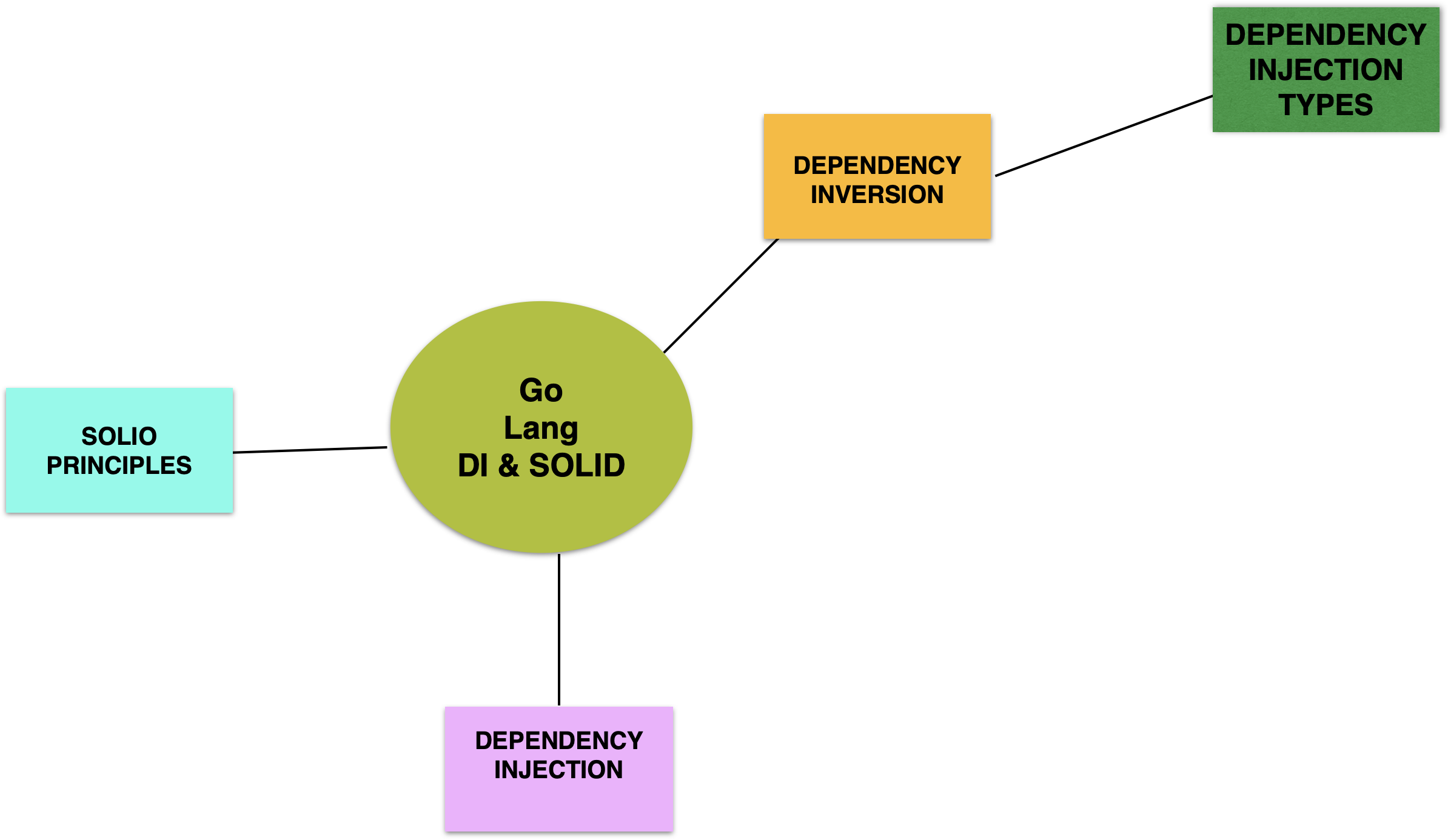
**C H A P T E R 9**

CHAPTER 9

# *Go Dependency Injection and SOLID* Introduction

In this Chapter, Reader will understand dependency injection using Go Language. Reader will understand SOLID principles which include dependency inversion. Reader will know different types of dependency injection like Constructor, Property and Method.



## Structure

The chapter covers the following topics:

* SOLID Principles
* Dependency Inversion
* Dependency Injection
* Dependency Injection Types

## Objectives

In this chapter, we are going to look at design principles like SOLID, Dependency inversion, dependency injection, and dependency injection types.

# SOLID Principles

SOLID principles were created by Robert C.Martin. They are design tenets for creating and developing software. These tenets help in making the software extensible, scalable, and enhance-able.

The SOLID principles are listed below:

* Single Responsibility Principle (SRP)
* Open/Closed Principle (OCP)
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)



**SOLID Principles**

Let us start look at Single Responsibility Principle first.

## Single Responsibility Principle

*A class should have one, and only one, reason to change. –Robert C Martin*

A class or module needs have a single responsibility. This helps in making the software modular. New enhancements can be added to the software easily. Defects can be analyzed, and root cause can be detected easily.

In life, we pay for online services and for ordering food and groceries. Every time we pay online using google Pay or any other payment service, we send the order information and account information for processing the order payments. Based on Single Responsibility principle, a payment service needs to have only responsibility of processing the payments and sharing a response that payment is processed with transaction information. Invoice service will process the invoices and share the invoice link to you. Order management service will process the orders after communicating the success of payment to the vendor.

As the payment service matures, new enhancements are required to process the payment asynchronously or when the connectivity is available. Based on the success, the order is processed. As SRP states, this feature can be added to the payment service very easily. Based on the success of the payment, invoice and order services receive the payment success event to process the workflow.

Customer service maintains the customer information. We can have customer loyalty service and customer address service for handling loyalty programs and addresses of the customer. Typically, loyalty program will consists of different levels like Platinium, Gold, Silver, and Bronze. These levels are defined at different range of points. Loyalty points are gathered by the customer during ordering groceries and other items. Offers are targeted at different loyalty membership levels to attract the customer to leap from one membership level to another.

Customer address service manages the address of the customer. This service has the contact information like mobile, email, and landline contacts also. This helps in sending marketing offers and targeting campaigns to the customer.

First, let us look at the code for SRP principle. We will start off with a LandVehicle struct which implements the interface Vehicle.

**srp\_principle.go**

package main

import "fmt"

type Vehicle interface {

drive()

}

type LandVehicle struct {

Vehicle

name string

}

func (dr \*LandVehicle) init() {

dr.checkBattery()

dr.checkEngine()

}

func (dr \*LandVehicle) drive() {

fmt.Printf(">>> driving Car[%s] ... \n", dr.name)

dr.init()

dr.start()

dr.monitorHealth()

}

func (dr \*LandVehicle) checkBattery() {

fmt.Println(" checking battery's status ... ")

}

func (dr \*LandVehicle) checkEngine() {

fmt.Println(" checking engine's status ... ")

}

func (dr \*LandVehicle) start() {

fmt.Println("starting now ... ")

}

func (dr \*LandVehicle) monitorHealth() {

fmt.Println("[ everything is ok ... ")

}

func main() {

dr := &LandVehicle{name: "Car"}

dr.drive()

}

You can now compile and run the srp\_example.go.

go run srp\_example.go

The output will be as shown below:

(base) apples-MacBook-Air:code bhagvan.kommadi$ go run srp\_example.go

>>> driving Car[Car] ...

checking battery's status ...

checking engine's status ...

starting now ...

[ everything is ok ...

(base) apples-MacBook-Air:code bhagvan.kommadi$

In this example, Land vehicle has a single responsibility of handling the driving part.

## Open/Closed Principle (OCP)

*“A software artifact should be open for extension but closed for modifications.”*

Software is built such a way that it is open for extending functionality and it is closed for modifying the existing functionality. Adding new behaviors in the software can be done easily and new features can be added to the software. This principle helps in making the software agile and enhance-able with much effort. Using Object oriented abstraction and Interfaces, it is very easy to follow these principles while designing and developing the software.

In the above example of payment service, there are new payment methods coming up in the market as EMI, Buy Now Pay Later, and others. Old methods continue to exist like credit card, debit card, net banking, bill pay, wallets, and others. Payment service needs to have capability to add a new payment method. This can be achieved by having a Payment Method type as an interface and pass the interface typed instance for payment processing function in Payment Service.

Similarly, Customer Loyalty service needs to have a capability to transfer or accept the loyalty points from partner programs. This service needs to have a Loyalty Point Typed interface as the parameter for addPoints function.

Recap from chapter 1, we had an example for interface types.

*In Go Language, interface is an abstract type which consists of method signatures. You cannot have instances of the interface. You can create an interface with type (keyword) interface name interface (keyword). More than one interface can be implemented by a data type.*

*Vehicle is an interface and car is a struct. When car is instantiated, it returns a vehicle*.

Modifying the example with another struct, we will demonstrate the open closed principle.

**ocp\_principle.go**

package main

import "fmt"

type vehicle interface {

getSpeed() float64

getDistanceTravelled() float64

}

type Car struct {

brand string

manufacturing\_year int

}

func (Car car) getSpeed() float64 {

return 120

}

func (Car car) getDistanceTravelled() float64 {

return 4000

}

type Truck struct {

brand string

manufacturing\_year int

size string

}

func (Truck truck) getSpeed() float64 {

return 160

}

func (Truck truck) getDistanceTravelled() float64 {

return 40000

}

func main() {

var veh vehicle

veh = car{"Toyota Tercel", 1997}

fmt.Println("Speed of the Vehicle :", veh.getSpeed())

fmt.Println("Distance Travelled by the Vehicle:", veh.getDistanceTravelled())

}

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

go run instance\_type.go

The output will be as shown below:

(base) apples-Air:code bhagvan.kommadi$ go run interface\_type.go

Speed of the Vehicle : 120

Distance Travelled by the Vehicle: 4000

(base) apples-Air:code bhagvan.kommadi$

The output shows how the interface abstracts out the method implementation. Vehicle is an interface and car is a struct. When car is instantiated, it returns a vehicle.

## Liskov Substitution Principle (LSP)

*What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T”*

In the software, we have structs, interfaces or classes which can be substitute for other equivalent types. In the payment service example, payment method typed interface will help in substituting different struct which implement this interface.

Modifying the previous example with enhanced interface of the vehicle, we will demonstrate the Liskov substitution principle.

**lsp\_principle.go**

package main

import "fmt"

type vehicle interface {

getSpeed() float64

getDistanceTravelled() float64

start()

}

type car struct {

brand string

manufacturing\_year int

}

func (Car car) getSpeed() float64 {

return 120

}

func (Car car) getDistanceTravelled() float64 {

return 4000

}

func (Car car) start() {

fmt.Printf(">>> starting the [%s] car ... \n", Car.brand)

}

type truck struct {

brand string

manufacturing\_year int

}

func (Truck truck) getSpeed() float64 {

return 180

}

func (Truck truck) getDistanceTravelled() float64 {

return 40000

}

func (Truck truck) start() {

fmt.Printf(">>> starting the [%s] Truck ... \n", Truck.brand)

}

func getVehicles() []vehicle {

return []vehicle{&car{}, &truck{}}

}

func main() {

var veh vehicle

veh = car{"Toyota Tercel", 1997}

fmt.Println("Car's Speed of the Vehicle :", veh.getSpeed())

fmt.Println("Car's Distance Travelled by the Vehicle:", veh.getDistanceTravelled())

veh = truck{"Land Rover", 2001}

fmt.Println("Truck's Speed of the Vehicle :", veh.getSpeed())

fmt.Println("Truck's Distance Travelled by the Vehicle:", veh.getDistanceTravelled())

for \_, vehicle := range getVehicles() {

vehicle.start()

}

}

You can now compile and run the lsp\_principle.go

The command is shown as below:

go run lsp\_principle.go

The output will be as shown below:

(base) apples-MacBook-Air:code bhagvan.kommadi$ go run lsp\_example.go

Car's Speed of the Vehicle : 120

Car's Distance Travelled by the Vehicle: 4000

Truck's Speed of the Vehicle : 180

Truck's Distance Travelled by the Vehicle: 40000

>>> starting the [] car ...

>>> starting the [] Truck ...

(base) apples-MacBook-Air:code bhagvan.kommadi$

The output shows how the interface abstracts out the method implementation of starting a vehicle whether it is a truck or a car. Vehicle is an interface and car and truck are structs implementing the interface. You can have get vehicles which can handle both car and truck.

## Interface Segregation Principle (ISP)

Software Clients need not be relying on interfaces. Typically, interfaces are designed with minimal functionality. This helps in cutting down the dependencies between the modules.

In the previous example, we add one more interface to group land vehicles. Let us see this in the code below

**isp\_principle.go**

package main

import "fmt"

type vehicle interface {

getSpeed() float64

getDistanceTravelled() float64

start()

}

type group interface {

addToTheGroup()

}

type car struct {

brand string

manufacturing\_year int

}

func (Car car) getSpeed() float64 {

return 120

}

func (Car car) getDistanceTravelled() float64 {

return 4000

}

func (Car car) start() {

fmt.Printf(">>> starting the [%s] car ... \n", Car.brand)

}

type truck struct {

brand string

manufacturing\_year int

}

func (Truck truck) getSpeed() float64 {

return 180

}

func (Truck truck) getDistanceTravelled() float64 {

return 40000

}

func (Truck truck) start() {

fmt.Printf(">>> starting the [%s] Truck ... \n", Truck.brand)

}

func (Truck truck) addToTheGroup(member truck) {

fmt.Printf(">>> adding another [%s] Truck to the group ... \n", Truck.brand)

}

func getVehicles() []vehicle {

return []vehicle{&car{}, &truck{}}

}

func main() {

var veh vehicle

veh = car{"Toyota Tercel", 1997}

fmt.Println("Car's Speed of the Vehicle :", veh.getSpeed())

fmt.Println("Car's Distance Travelled by the Vehicle:", veh.getDistanceTravelled())

var truck1 truck

truck1 = truck{"Land Rover", 2001}

fmt.Println("Truck's Speed of the Vehicle :", truck1.getSpeed())

fmt.Println("Truck's Distance Travelled by the Vehicle:", truck1.getDistanceTravelled())

var truck2 truck

truck2 = truck{"Hyundai Trunk", 2011}

fmt.Println("Truck's Speed of the Vehicle :", truck2.getSpeed())

fmt.Println("Truck's Distance Travelled by the Vehicle:", truck2.getDistanceTravelled())

truck2.addToTheGroup(truck1)

}

You can now compile and run the isp\_principle.go

The command is shown as below:

go run isp\_principle.go

The output will be as shown below:

(base) apples-MacBook-Air:code bhagvan.kommadi$ go run isp\_example.go

Car's Speed of the Vehicle : 120

Car's Distance Travelled by the Vehicle: 4000

Truck's Speed of the Vehicle : 180

Truck's Distance Travelled by the Vehicle: 40000

Truck's Speed of the Vehicle : 180

Truck's Distance Travelled by the Vehicle: 40000

>>> adding another [Hyundai Trunk] Truck to the group ...

(base) apples-MacBook-Air:code bhagvan.kommadi$

The output shows how the interface abstracts out the method implementation of adding a vehicle to the group. vehicle and group are interfaces and truck is a struct. Creating a group of cars is achieved by having a new interface implemented. Vehicle interface provides the functionality of speeding and maintaining the distance travelled in the odometer.

## Dependency Inversion Principle (DIP)

Top layer software modules need not have to rely on bottom layer modules. Top and bottom layers need to rely on software interfaces and abstractions. Detailed features are abstracted out by having abstract classes. This helps in making the software modular and decoupled. The lesser decoupled is the software, it helps in enhancing and testing the features quickly.

Now let us look at the previous example. In the vehicle, we need an engine to be added. A new car can be created with an engine for a specific brand. Let us look at the code now.

**dip\_principle.go**

package main

import "fmt"

type vehicle interface {

getSpeed() float64

getDistanceTravelled() float64

start()

}

type Engine struct {

engineType string

}

type car struct {

brand string

manufacturing\_year int

engine \*Engine

}

func NewCar(brand string, year int, engine \*Engine) \*car {

return &car{

brand: brand,

manufacturing\_year: year,

engine: engine,

}

}

func (Car car) getSpeed() float64 {

return 120

}

func (Car car) getDistanceTravelled() float64 {

return 4000

}

func (Car car) start() {

fmt.Printf(">>> starting the [%s] car ... \n", Car.brand)

}

func main() {

var veh vehicle

var engine = &Engine{engineType: "V3"}

veh = NewCar("Toyota Tercel", 1997, engine)

fmt.Println("Car's Speed of the Vehicle :", veh.getSpeed())

fmt.Println("Car's Distance Travelled by the Vehicle:", veh.getDistanceTravelled())

veh.start()

} You can now compile and run the dip\_principle.go

The command is shown as below:

go run dip\_principle.go

The output will be as shown below:

(base) apples-MacBook-Air:code bhagvan.kommadi$ go run dip\_example.go

Car's Speed of the Vehicle : 120

Car's Distance Travelled by the Vehicle: 4000

>>> starting the [Toyota Tercel] car ...

(base) apples-MacBook-Air:code bhagvan.kommadi$

The output shows how an engine is added to create a car. Vehicle is an interface and car & engine are structs. When car is instantiated, brand and engine are passed in the constructor of the car

# Dependency Injection

Design principles and patterns are followed by the architects and leads to build software. Better reusability and improvement in productivity is observed when design patterns and principles are applied during software engineering. Dependency injection is a design pattern applicable in many tech stacks.

Let us start off looking at how a computer is assembled for the order. Most of the customers have in mind the computer hardware requirements like ram memory, cpu frequency, and disk space. They are also interested to have accessories which go with the computer like keyboard, mouse, monitor, and power backup. Many vendors work on these requirements to assemble the computer based on these requirements.

While assembling the hardware, vendors are aware of the mother boards and dependent components which go with them.

Based on the above example, we can extend the principles of dependency injection and build software. Some of the mother boards are loosely coupled and work with components of different brands and interfaces. Similarly, software can be built loosely coupled. The components in the software scenario needs to have services and interfaces which are easily pluggable to other services or a container. There are many popular containers in Go lang like Wire, Inject, Dig, Dingo , and goioc/di. These containers help in designing and building components which are easily pluggable into the containers.

Let us first look at an example of dependency injection through a constructor. In this case, there is no container. A media storage system will have services to store the media like video, audio, and images.

**di\_constructor\_example.go**

package main

import "fmt"

type Video struct {

content []byte

storage VideoStorage

publisher string

}

func NewVideo(ps VideoStorage, publisher string) \*Video {

return &Video{storage: ps, publisher: publisher}

}

func (video \*Video) Load(content string) {

video.content = video.storage.Load(content)

}

func (video \*Video) Save(title string) {

video.storage.Save(title, video.content)

}

type VideoStorage interface {

Load(string) []byte

Save(string, []byte)

}

type MediaStorage struct {

video []byte

}

func NewMediaStorage() \*MediaStorage {

return &MediaStorage{

video: []byte{},

}

}

func (mStore \*MediaStorage) Save(name string, contents []byte) {

mStore.video = contents

}

func (mStore \*MediaStorage) Load(name string) []byte {

return mStore.video

}

func (mStore \*MediaStorage) Type() string {

return "MediaStorage"

}

func main() {

storage := NewMediaStorage()

video := NewVideo(storage, "STAR MUSIC")

video.Load("Cricket Match NZ vs AUS 1987. Cricket match happened in Perth and Australia won the match")

video.Save("Cricket Match NZ vs AUS 1987 ")

fmt.Printf("video : %s \n", video.publisher)

}

You can now compile and run the di\_constructor\_example.go. The command to execute the code is shown as below:

go run di\_constructor\_example.go

The output will be as shown below:

(base) apples-MacBook-Air:code bhagvan.kommadi$ go run di\_constructor\_example.go

video : STAR MUSIC

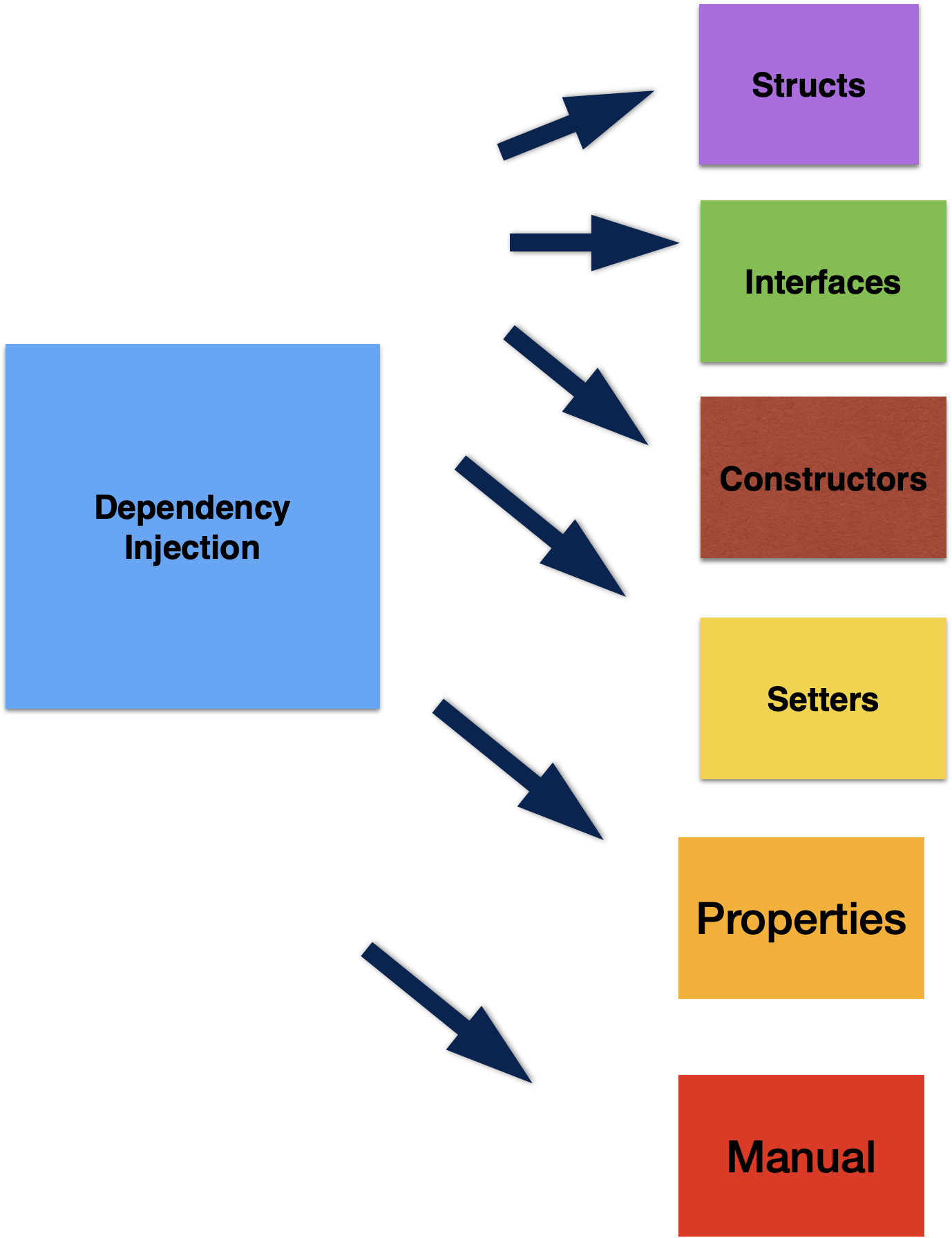
(base) apples-MacBook-Air:code bhagvan.kommadi$

In the above example, we have video storage service with Media Storage and Video Storage interfaces. Video is a struct. The constructor of the video takes the media storage interface typed instance. Video struct implements video storage for saving and loading the video information.

# Dependency Injection Types

Dependency injection method helps in ensuring the dependencies are loaded and software is enhance-able and scalable. The dependent components can be replaced with any component having same interface or features. In Go Lang tech stack, dependency injection can be done through structs & interfaces, through constructors, setters, methods, and manual injection. Go lang dependency injection has evolved and many frameworks like Wire, Inject, Dig, Dingo, goioc/di, and others support dependency injection. These frameworks have different features and appropriate framework can be selected based on user support, popularity, scalability, usability, and performance. Using these containers, you can build software which can be maintained very easily, and new features can be added within time to market.

There are different types of Dependency injection. They are based on structs/interfaces, constructors, setters, properties, and manual injection types.



**Dependency Injection Types**

First let us look at different injection methods.

## Creating Structs and Interfaces

A class or an object is called in another class for a specific feature, then these two classes are dependent on each other. While creating the class through constructor, new class can be instantiated and passed as a parameter. This makes these two classes coupled and changing a class to another class becomes tough. Structs and interfaces come for rescue. Structs can be used for creating objects and interfaces can be used for describing the object behavior.

If two classes have the same interface defined for a feature, you can interchange them in the parent class easily. In Go Lang, we use structs to implement the interfaces. Objects are instantiated and passed on as parameters to the parent class functions or methods.

## Manual Dependency Injection

Manually creating the dependencies and plugging in the right one in a class is another option. In this case, it looks simple and nice, but the coupling becomes higher and managing the software becomes really tough.

## Constructor Injection

Dependency injection through constructor is another option. This helps in making the injection unmodifiable. Dependencies can be instantiated and kept ready for plugging into right class constructor. You can see the example below:

func main() {

storage := NewMediaStorage()

video := NewVideo(storage, "STAR MUSIC")

}

## Setter/Method Injection

Setter injection is another injection type where the required dependency is can be set on a method. This is better in terms of coupling factor. Service class will have different functions based on its behaviors to be exposed. Client invokes the service for a specific function. Service needs to have the dependent component initialized.

func (video \*Video) setVideoStorage(vStorage VideoStorage) {

video.storage = vStorage

}

func main() {

storage := NewMediaStorage()

video := NewVideo(storage, "STAR MUSIC")

vStorage := new(VideoStorage)

video.setVideoStorage(vStorage)

}

Now let us look at the dependency injection frameworks in Go Lang.

# Dependency Injection Frameworks

There are many dependency injection frameworks. In Go Lang, Wire, inject, Dig, Dingo, and goioc/di are the popular DI frameworks. These frameworks help in creating software which is loosely coupled and agile in nature.

## Google’s Wire

Most of the time real life software has many dependencies. Dependency graph or tree can be created to identify the level of the dependencies and the root. Changing the dependent components in a complex software is challenging because of the complex build process and compilation. Build tools need to have capability to identify the set of dependent components and building the packages. Cleaning the old builds and restarting new builds is a required feature for build management tools.

Google’s Wire is a popular dependency injection tool which can handle the dependency tree management easily. The initialization of these dependencies in the parent classes is done by the framework. Wire has two important aspects which are providers and injectors. Providers are Go Language function which process the parameters to return the values.

func NewMediaStorage() \*MediaStorage {

return &MediaStorage{

video: []byte{},

}

}

The above code shows the provider for Media Storage.

## Facebook’s Inject

Facebook’s inject framework uses injection method based on reflection. Go Lang reflection package is used for reflecting the required types and constructors to create dependent objects in the dependency tree. Inject has struct tags and these have three different types like inject.””, inject.”private”, and inject.”dev logger”.

## Uber’s Dig

Uber’s Dig uses reflection method for dependency injection. It has features to build a dependency tree and bootstrap the required objects during the startup. It can handle singletons in the dependencies. The parameters to the function need to implement an interface. Dig has support for method or property injection.

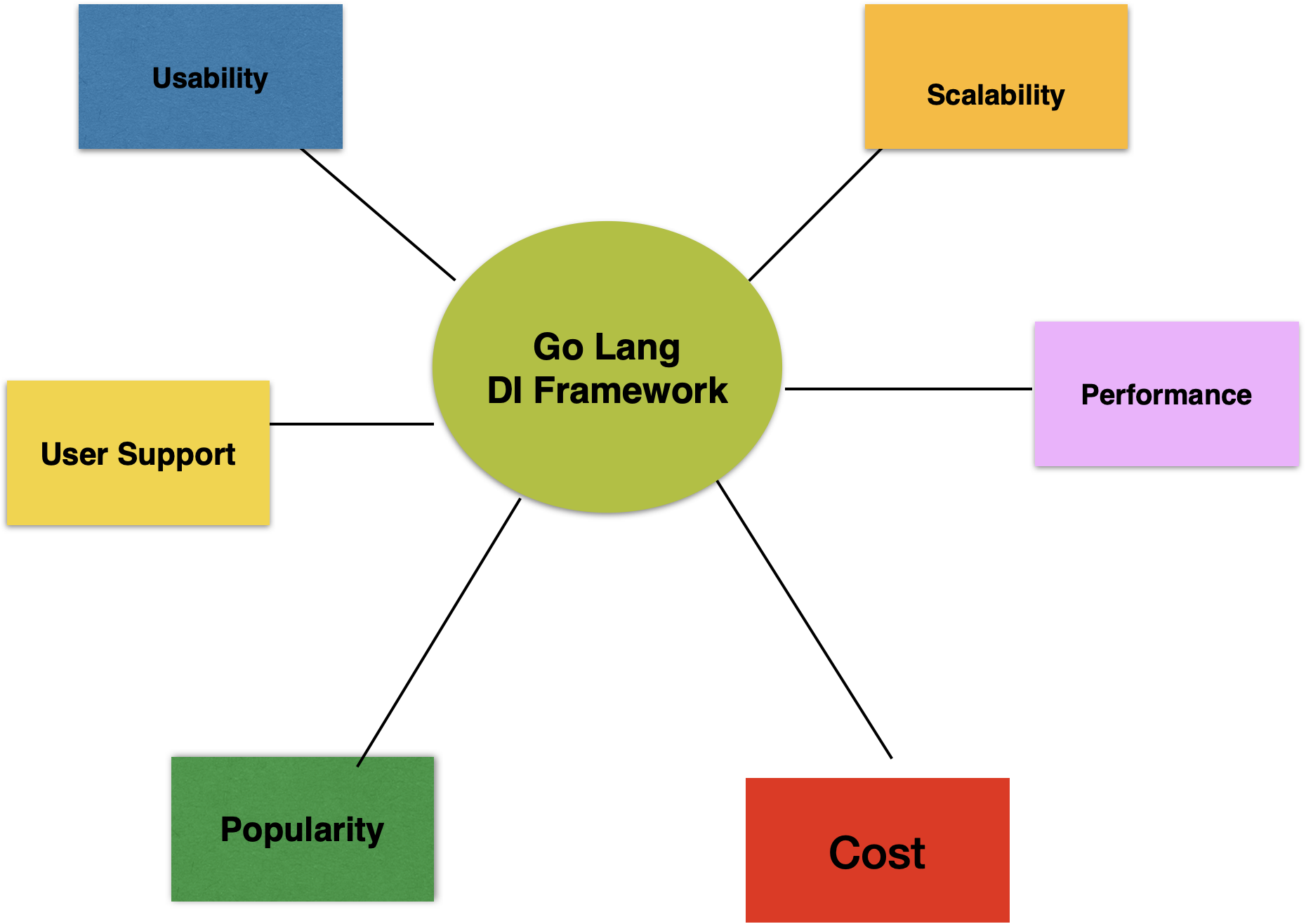
## Dingo

Dingo is a similar dependency framework like Guice. Struct tags can be used in this framework. It can also support through request injection type.

## Which is the Framework Suitable for You?

There are various factors of identifying an appropriate framework for your software product. The different factors are listed below:

* Usability
* Scalability
* Performance
* Popularity
* User Support
* Commercial/Opensource
* Cost
* Future Proofed



**Dependency Injection Framework Selection : Factors**

# Conclusion

In this chapter we have covered topics related to SOLID principles and dependency injection. Different dependency injection types were covered and frameworks supporting dependency injection were discussed.

* SOLID principles are Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion priniciples.
* Dependency injection can be used to inject dependencies into the software. Dependencies can be managed in the software with different types of dependency injection.
* Dependency injection can be done through structs & interfaces, manual, constructor based, setter, and property based injection types.
* There are popular dependency injection frameworks like Wire, Inject, Dig, Dingo, and others.