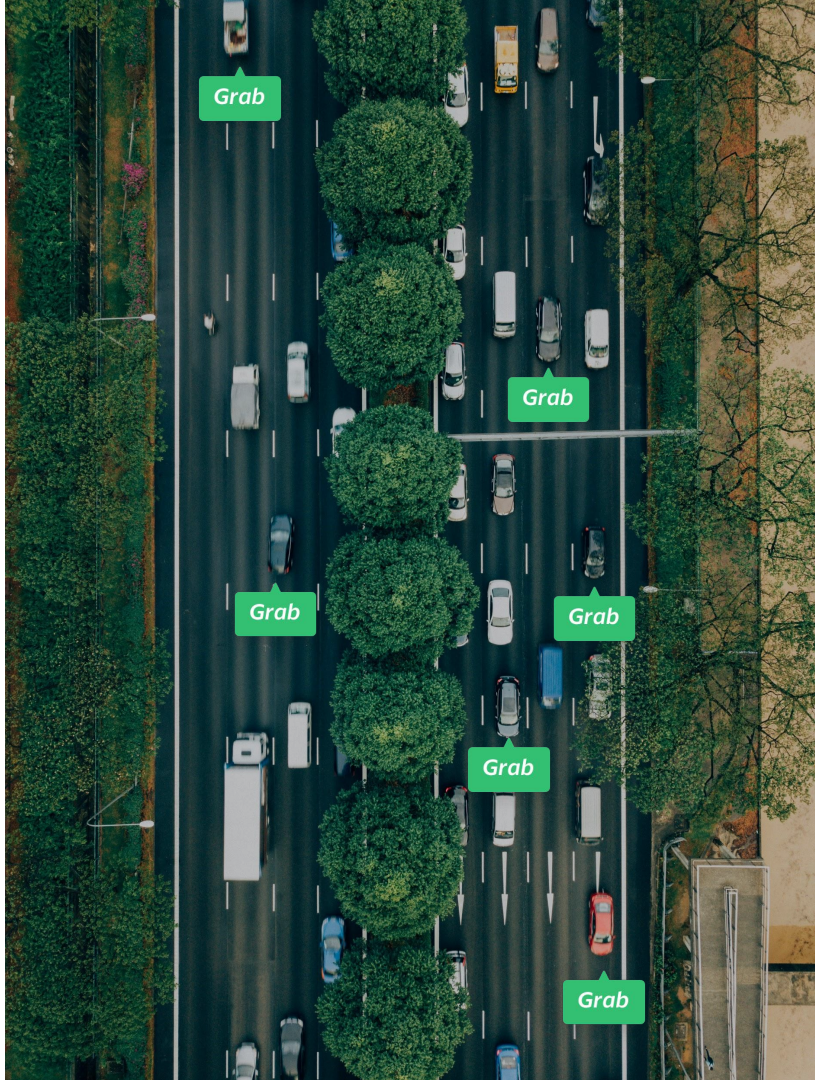
Grab

Week 5

Dependency Injection

Dat Ha
An Nguyen
Ivan Le



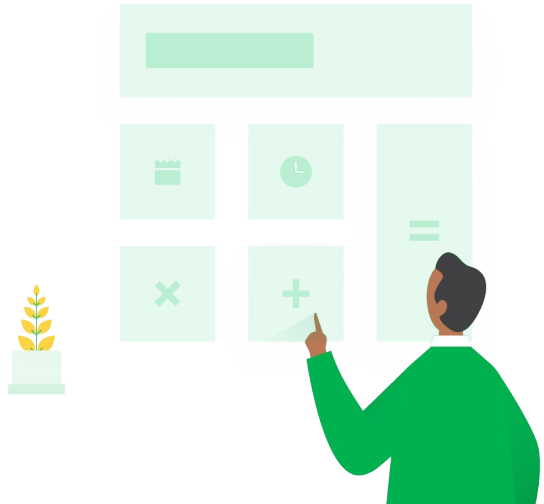
Grab

Agenda

- What is Dependency Injection?
- Why do we need Dependency Injection?
- Dependency Injection in example.
- Types of Dependency Injection.
- Dependency Inversion.
- Fix tightly coupled code.
- Assignment.

Week 5

What is Dependency Injection?



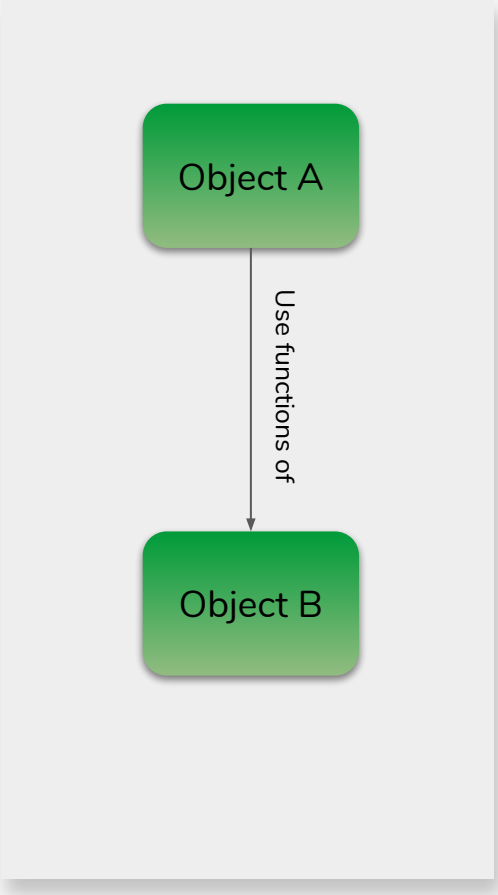
Grab

Dependency Injection (DI) is a style of software construction that focuses on **defining the relationships** between the different parts of the software **in an abstract, generalized manner**.

Grab

Software as a collection of different objects; in Go this takes the form of **structs** and **functions**.

Relationship between the objects can be defined using the term dependency.



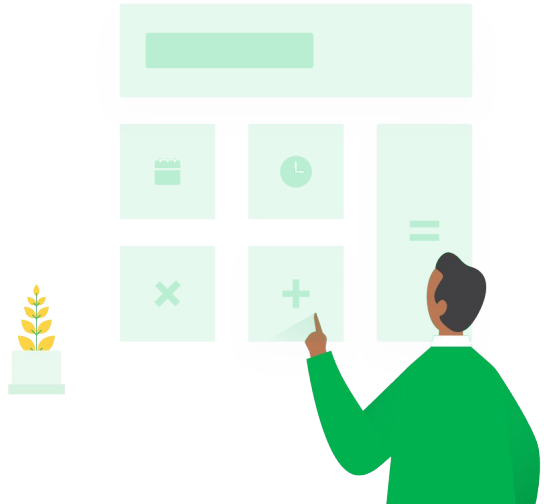
Dependency

Still confused?

When object A uses some functionality of object B, then it’s said that class A has a dependency of class B.

Week 5

Why do we need Dependency Injection?



Grab

Consider this example

```
type Jimi struct {
       guitar *ElectricGuitar
}

func (j *Jimi) Play(){
       j.guitar.Pluck()
       j.guitar.Strum()
       j.guitar.Thrash()
}

type ElectricGuitar struct {
       // attributes omitted
}
```

```
type Jimi struct {
       guitar *ElectricGuitar
}

func (j *Jimi) Play(){
       j.guitar.Pluck()
       j.guitar.Strum()
       j.guitar.Thrash()
}

type ElectricGuitar struct {
       // attributes omitted
}
```

Consider this example

`Jimi` is considered to depend on `ElectricGuitar`. We therefore consider `ElectricGuitar` as a dependency of `Jimi`.

1. Unable (Hard) to test them independently of each other.
2. Unable to replace the `ElectricGuitar` with a different one.
3. Unable to easily determine (e.g, without reading all the code) exactly what parts of `ElectricGuitar` that `Jimi` needs.

```
type Guitar interface {
      Strum()
      Pluck()
      Thrash()
}
```

```
type Jimi struct {
      guitar Guitar
}

func (j *Jimi) Play(){
      j.guitar.Pluck()
      j.guitar.Strum()
      j.guitar.Thrash()
}
```

What have we achieved?

Decoupled (separated) our objects

Tested separately, easier

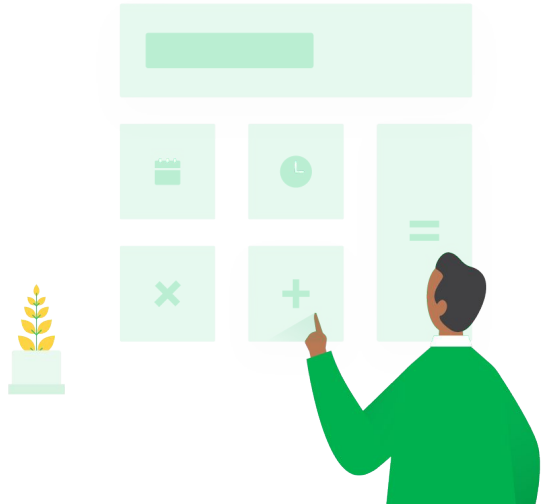
Maintained (extended) separately

Easier to swap out guitar without breaking Jimi

Less risky, faster

Week 5

Types of Dependency Injection



Grab

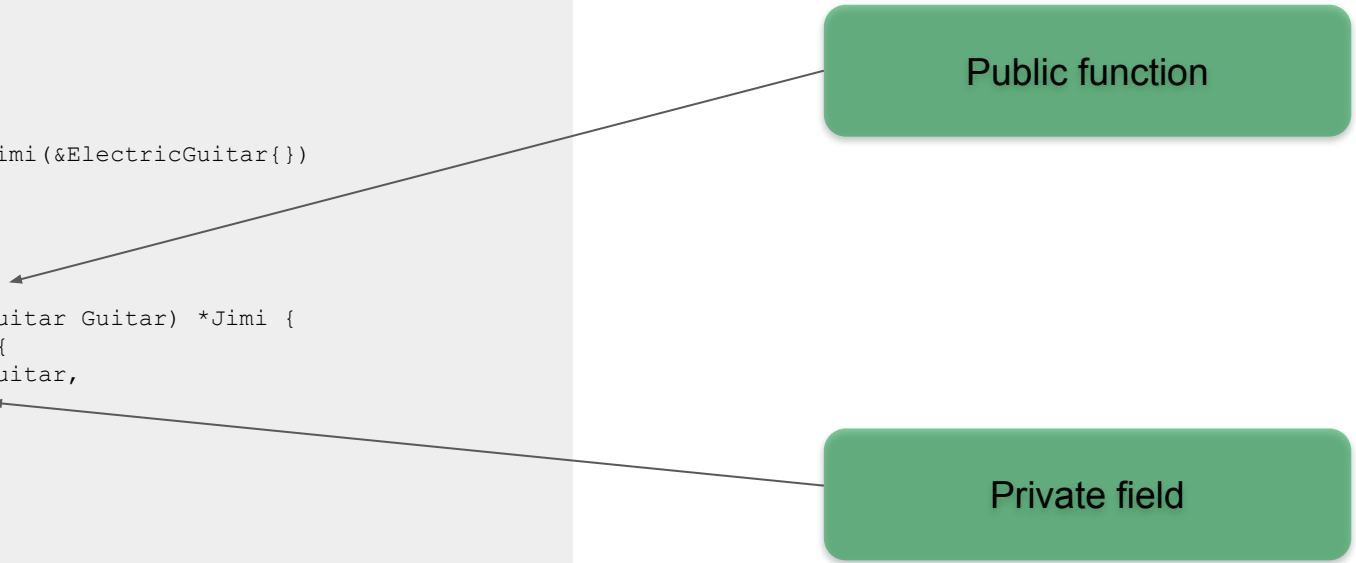
Constructor Injection

```
type Jimi struct {
  guitar Guitar
}

func (j *Jimi) Play() {
  j.guitar.Pluck()
  j.guitar.Strum()
  j.guitar.Thrash()
}

type ElectricGuitar struct {
  // attributes omitted
}

func (e *ElectricGuitar) Pluck()  {}
func (e *ElectricGuitar) Strum()  {}
func (e *ElectricGuitar) Thrash() {}


type Guitar interface {
  Pluck()
  Strum()
  Thrash()
}

func main() {
  jimi := &Jimi{
     guitar: &ElectricGuitar{},
  }
  jimi := &Jimi{
     guitar: &AcousticGuitar{},
  }
}
```

Constructor Injection

```
func main() {
  jimi := NewJimi(&ElectricGuitar{})
}
```

```
func NewJimi(guitar Guitar) *Jimi {
  return &jimi{
     guitar: guitar,
  }
}
```

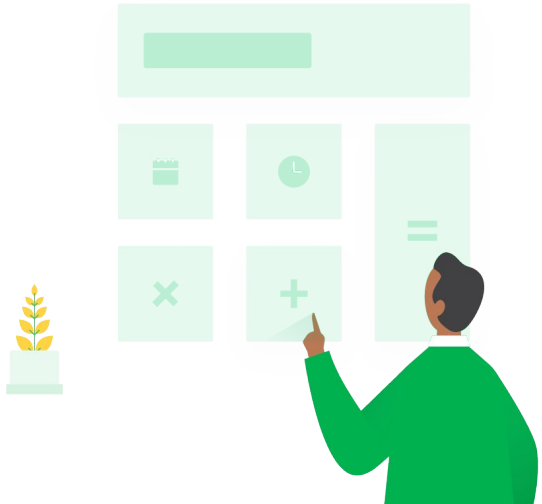


Method Injection

```
type Guitar interface {
  Pluck()
  Strum()
  Thrash()
}

type Jimi struct {}

func (j *Jimi) Play(guitar Guitar) {
  guitar.Pluck()
  guitar.Strum()
  guitar.Thrash()
}
```

Week 5

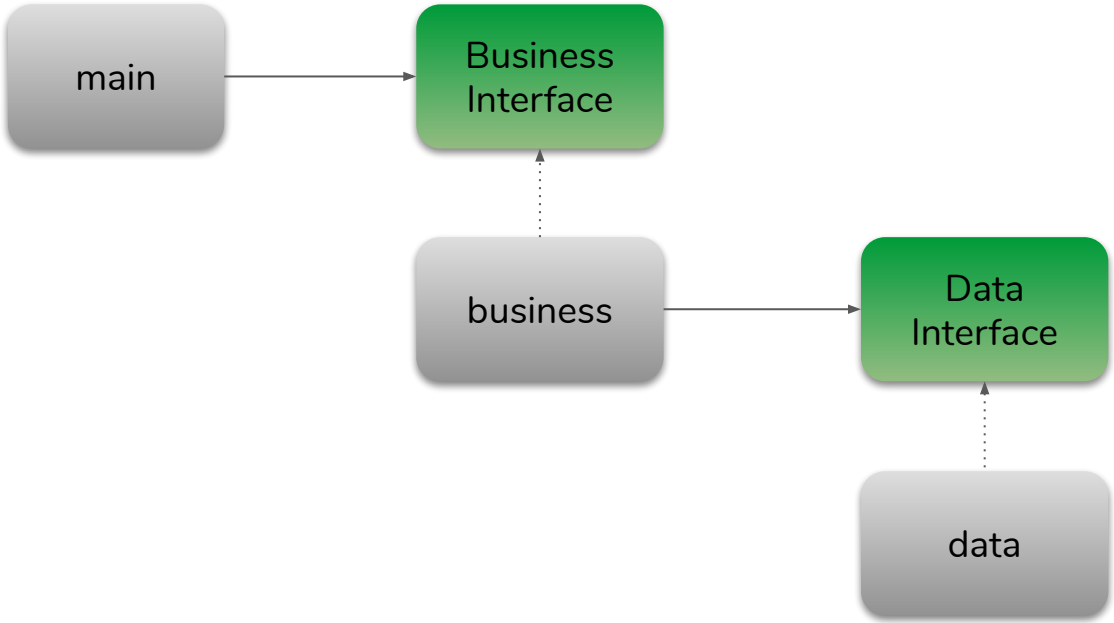
Dependency Inversion



"High level modules should not depend on low level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend on abstractions"
–Robert C. Martin

Grab

High-level packages should not depend on low-level packages



Grab

Structs should not depend on Structs

```
type PizzaMaker struct{}

func (p *PizzaMaker) MakePizza( oven *SuperPizaOven5000) {
      pizza := p.buildPizza()
      oven.Bake(pizza)
}
```

```
type PizzaMaker struct{}

func (p *PizzaMaker) MakePizza( oven Oven) {
      pizza := p.buildPizza()
      oven.Bake(pizza)
}

type Oven interface {
      Bake(pizza Pizza)
}
```

Grab

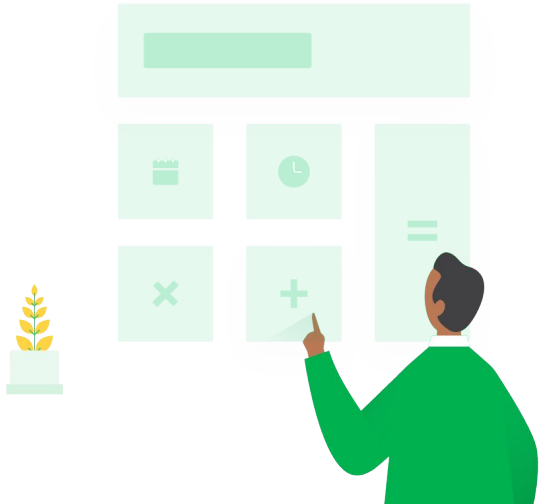
Interfaces should not depend on Structs

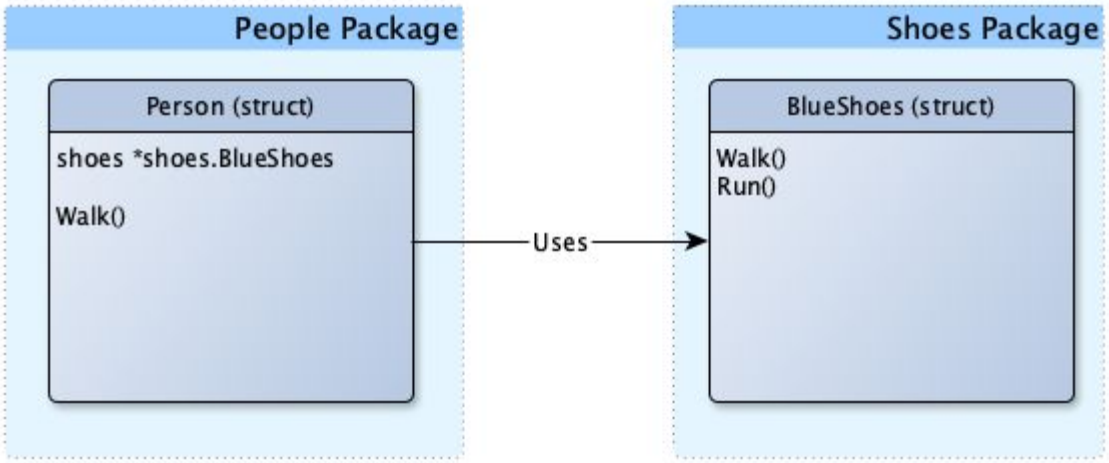
```
type Config struct {
      DSN             string
      MaxConnections  int
      Timeout         time.Duration
}

type PersonLoader interface {
      Load(cfg *Config, ID int) *Person
}
```

```
type PersonLoaderConfig interface {
      DSN() string
      MaxConnections() int
      Timeout() time.Duration
}

type PersonLoader interface {
      Load(cfg PersonLoaderConfig, ID int) *Person
}
```

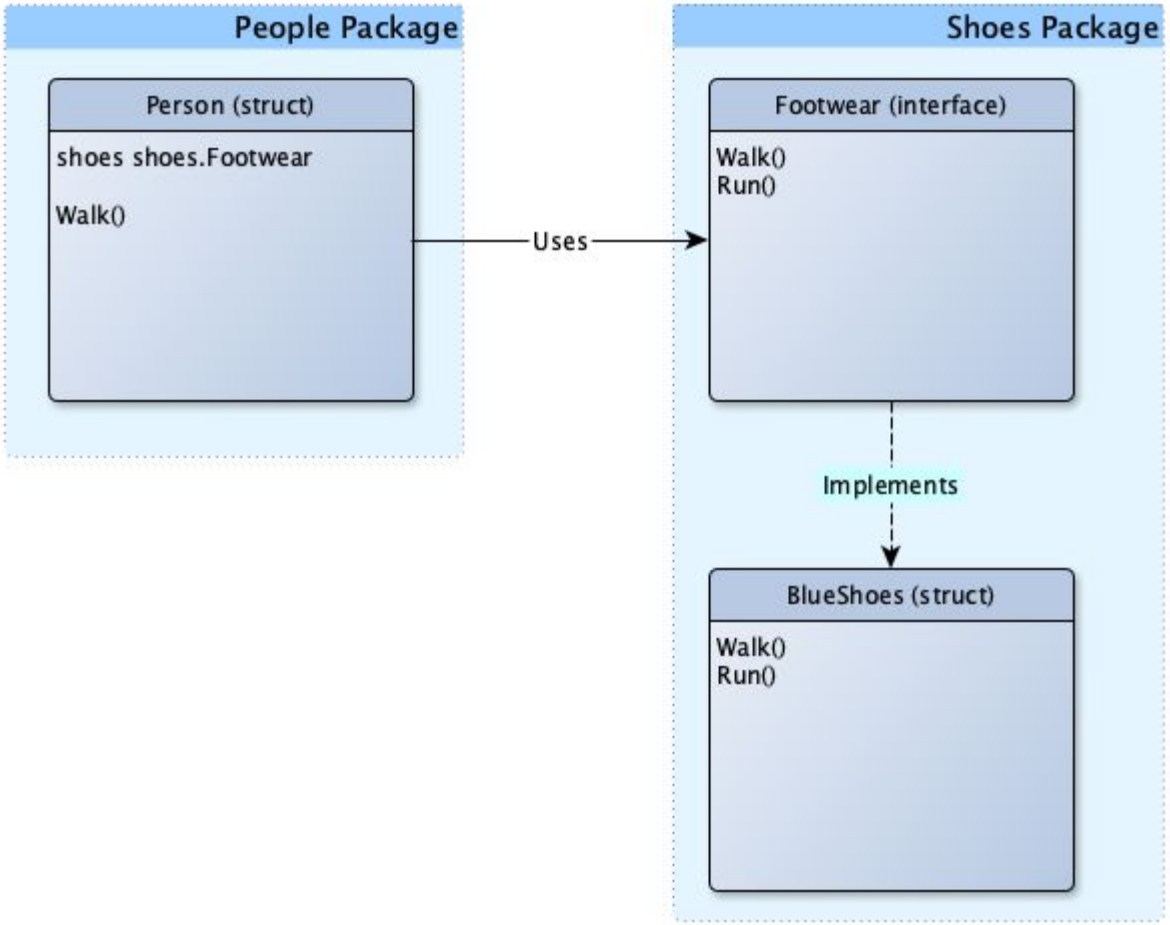
Week 5

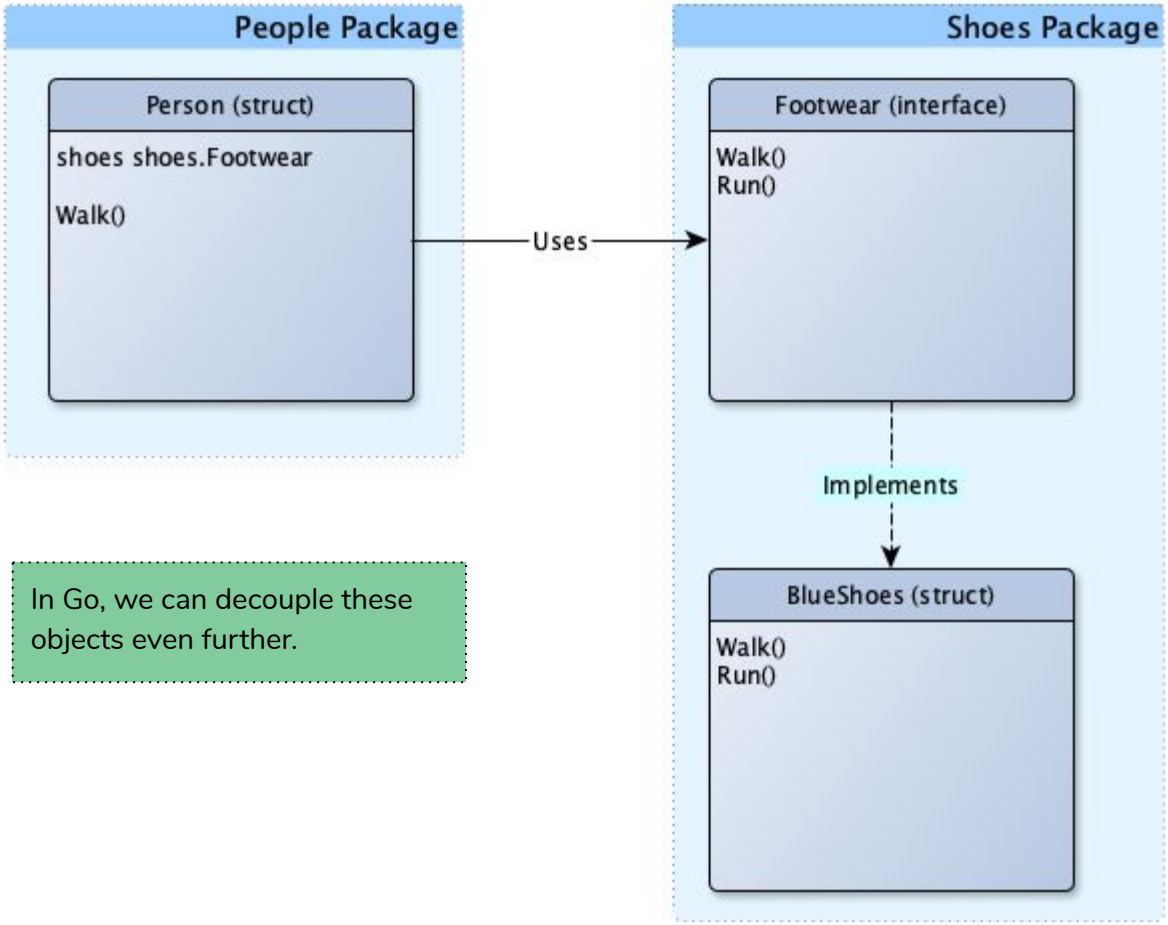
Fix tightly coupled code



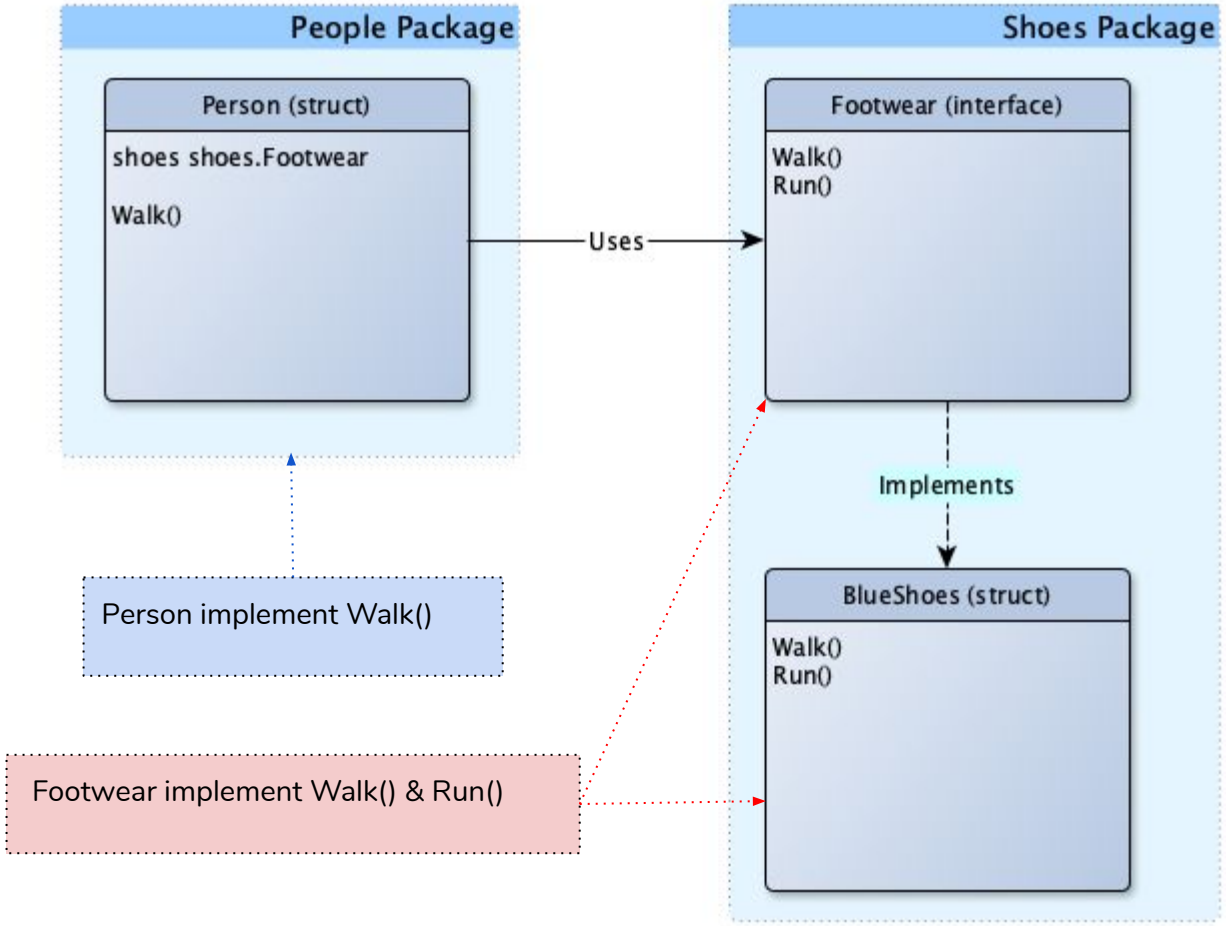


Person is tightly coupled with **BlueShoes**

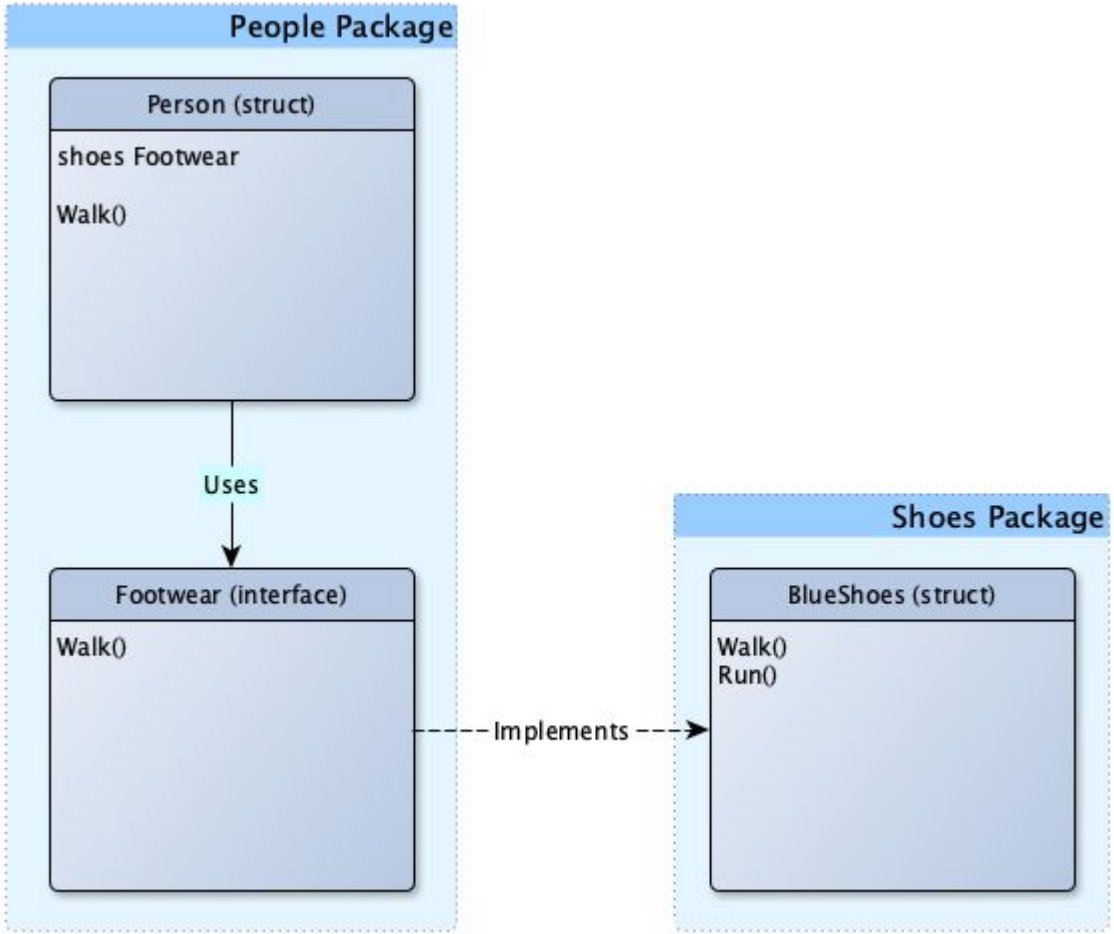

People Package
Person (struct)
shoes shoes.Footwear
Walk()
Uses
Shoes Package
Footwear (interface)
Walk()
Run()
Implements
BlueShoes (struct)
Walk()
Run()



In Go, we can decouple these objects even further.

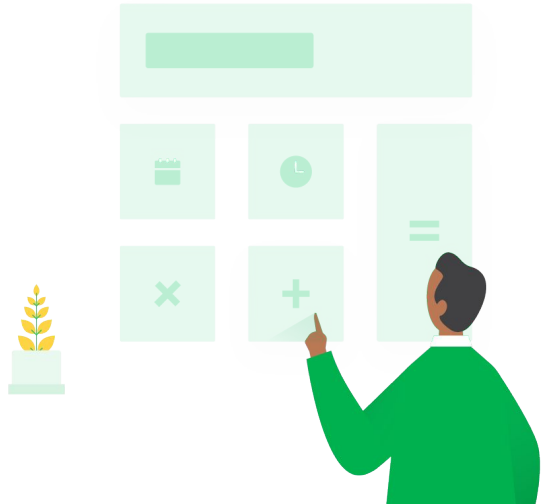

People Package
Person (struct)
shoes shoes.Footwear
Walk()
Uses
Shoes Package
Footwear (interface)
Walk()
Run()
Implements
BlueShoes (struct)
Walk()
Run()
Person implement Walk()
Footwear implement Walk() & Run()

“Clients should not be forced to depend on methods they do not use.”
–Robert C. Martin


People Package
Person (struct)
shoes Footwear
Walk()
Uses
Footwear (interface)
Walk()
Implements
Shoes Package
BlueShoes (struct)
Walk()
Run()

Week 5

Assignment



Grab

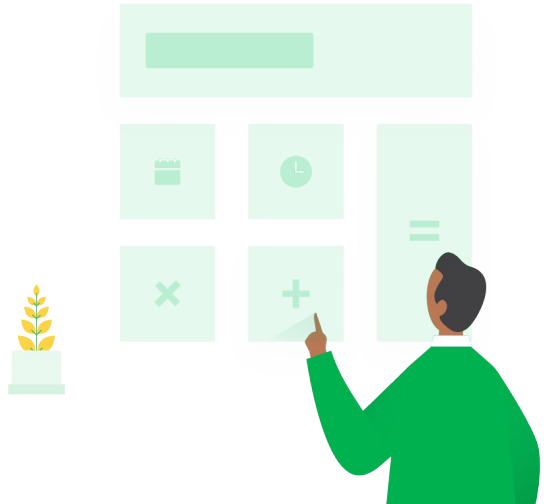
Assignment

1. Go to GitHub repo to see the requirement: https://github.com/havinhdat/GrabGoTrainingWeek5Assignment
2. Create a pull request.

That's it!

Appendix

Read more: Hands-On Dependency Injection in Go - Corey Scott



We're hiring! https://grab.careers

Email: dat.havinh@grab.com


THANK
you