

CLEANING UP YOUR GOOOP

How to break *Object Oriented Programming*
muscle memory and become better Gophers



ABOUT ME

Senior Engineer II at CrowdStrike

Full-time Go developer since Go 1.6

Software engineer since 1998 across 4 industries
and lots of languages



🎉 It's My Birthday!! 🎉



GO + OOP = GOOOOP

- Many (Most?) of us Gophers worked in another language first
- Often those "other" languages were C++, Java, or C#
- Go is a different sort of language
- Applying those C++/Java/C# patterns to our Go is less than ideal



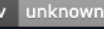
OBJECT ORIENTED PROGRAMMING

- Grew out of work at MIT in the late '50s and early '60s
- Term was first coined by Alan Kay while in grad school in the mid '60s
- Focuses on systems based on objects that capture state and methods on those objects to modify that state
 - System behaviors are defined by the interactions between objects

- IT'S
-
- AN

☰ README.md

FizzBuzzEnterpriseEdition

 Build status  codecov  unknown

Enterprise software marks a special high-grade class of software that makes careful use of relevant software architecture design principles to build particularly customizable and extensible solutions to real problems. This project is an example of how the popular FizzBuzz game might be built were it subject to the high quality standards of enterprise software.

FizzBuzz

FizzBuzz is a game that has gained in popularity as a programming assignment to weed out non-programmers during job interviews. The object of the assignment is less about solving it correctly according to the below rules and more about showing the programmer understands basic, necessary tools such as `if` -/ `else` -statements and loops. The rules of FizzBuzz are as follows:

For numbers 1 through 100,

- if the number is divisible by 3 print Fizz;
- if the number is divisible by 5 print Buzz;
- if the number is divisible by 3 and 5 (15) print FizzBuzz;
- else, print the number.

Contributing

Although this project is intended as satire, we take openness and inclusivity very seriously. To that end we have adopted the following code of conduct.

[Contributor Code of Conduct](#)

OBJECT ORIENTED PROGRAMMING

WE C++ IN MIND –

GO IS NOT *REALLY* OBJECT ORIENTED

Go is a practical language that often doesn't reward "fancy"

- Go structs are not classes
- Embedding is not Inheritance
- Packages, not types, are the most basic unit of design

STRUCTS ARE NOT CLASSES

- Go structs are closer to C structs (a named group of named/typed fields) than to Java or C++ classes
- Methods are syntax sugar
- Constructors are regular functions and are 100% optional
 - *Make the zero value useful* – The Go Proverbs
- No Destructors*
 - * `runtime.SetFinalizer()` exists but using it is often #darkarts

EMBEDDING IS NOT INHERITANCE

- Focus on what things a type can do (has-a) not which kind of thing a type represents (is-a)
- Fields and methods on the embedded type are promoted to the outer type
- Looks like inheritance, but 🐉🐉🐉

EMBEDDING IS NOT INHERITANCE

```
type BaseThing struct{}

func (b *BaseThing) doStuff() {
    fmt.Println("Everything's basic")
}

type NormalThing struct {
    BaseThing
}

type FancyThing struct {
    BaseThing
}

func (f *FancyThing) doStuff() {
    fmt.Println("It's full of stars!!")
}
```

EMBEDDING IS NOT INHERITANCE

```
func main() {  
    var (  
        t1 BaseThing  
        t2 NormalThing  
        t3 FancyThing  
    )  
  
    // all 3 things  
    t1.doStuff()  
    t2.doStuff()  
    t3.doStuff()  
}
```

```
// Output  
Everything's basic  
Everything's basic  
It's full of stars!!
```

EMBEDDING IS NOT INHERITANCE

```
func main() {  
    var (  
        t1 BaseThing  
        t2 NormalThing  
        t3 FancyThing  
    )  
    ...  
  
    // this works  
    doTheThing(&t1)  
}
```

```
// Output  
Everything's normal
```

```
func doTheThing(t *BaseThing) {  
    t.doStuff()  
}
```

EMBEDDING IS NOT INHERITANCE

```
func main() {  
    var (  
        t1 BaseThing  
        t2 NormalThing  
        t3 FancyThing  
    )  
    ...  
  
    // This does work  
    doTheThing(&t2.BaseThing)  
}  
  
...
```

```
// Output  
Everything's normal
```

```
// Output  
./main.go:42:15: cannot use &t2 (value of *NormalThing) as *BaseThing value in  
    argument to doTheThing()
```

EMBEDDING IS NOT INHERITANCE

```
func main() {  
    var (  
        t1 BaseThing  
        t2 NormalThing  
        t3 FancyThing  
    )  
    ...  
  
    // This compiles and  
    // - FancyThing.doStuff() is not called because there is no virtual dispatch  
    doTheThing(&t3.BaseThing)  
}  
  
...
```

// Output
Everything's normal

EMBEDDING IS NOT INHERITANCE

- Embedding lets you "inherit" fields and behavior by composing types
 - but things do not behave as they do in OOP languages
- Thinking in terms of base classes, derived classes, and inheritance
can lead to GOOOP

PACKAGES ARE THE SMALLEST UNIT OF CODE

- All code must live in a package
- Visibility is enforced within and outside of the package
- Importing a package brings in the entire package
- All consumer references to symbols include the package name



EXAMPLES OF GOOOP

1. Creating separate, shared components
to resolve co-dependency

SEPARATE SHARED COMPONENTS TO RESOLVE CO-DEPENDENCY

When A depends on B and B depends on A ...

- Create a third component C with the commonality
- Update A and B to depend on C and not each other

SEPARATE COMPONENTS TO RESOLVE CO-DEPENDENCY

pac

imp

FOOSERVER

BARSERVER

SEPARATE COMPONENTS TO RESOLVE CO-DEPENDENCY

```
package clients

type FooClient interface {
    ...
}

type BarClient interface {
    ...
}
```


SEPARATE COMPONENTS TO RESOLVE CO-DEPENDENCY

```
package fooserver

import "example.com/gooop/clients"

type Server struct {
    ...
    c clients.BarClient
}
```

```
package barserver

import "example.com/gooop/clients"

type Server struct {
    ...
    c clients.FooClient
}
```

SEPARATE COMPONENTS TO RESOLVE CO-DEPENDENCY

- Reduces cohesion and can create unnecessary coupling
- Often leads to "bad" module/package names
- Modifying multiple modules at once can be difficult

SEPARATE COMPONENTS TO RESOLVE CO-DEPENDENCY

Signs of GOOOP

- Packages/modules with names like "base", "clients", "interfaces", etc.
- Packages that only contain interfaces and structs with no behavior
- The new system has more moving parts



EXAMPLES OF GOOOP

2. Declaring interfaces as exported provider abstractions

INTERFACES AS EXPORTED ABSTRACTIONS

Interfaces in OOP

- Abstract base classes in Java and C++, Interfaces in C#, ...
- Enumerate the available operations on some type
- Commonly used as an insulation layer between the class and consumers
- Types often declare which interfaces they implement explicitly
- Can lead to more complicated code

INTERFACES AS EXPORTED ABSTRACTIONS

An extreme example

```
package s3iface

import "github.com/aws/aws-sdk-go-v2/service/s3"

type ClientAPI interface {

    AbortMultipartUploadRequest(
        *s3.AbortMultipartUploadInput) s3.AbortMultipartUploadRequest

    // +90 additional methods covering every S3 operation
    // :mother_of_god_meme:
}
```


INTERFACES AS EXPORTED ABSTRACTIONS

```
package server

import "example.com/gooop/foo"
...

// Service defines the operations provided by the service
type Service interface {
    // CRUD for Foo entities
    CreateFoo(name string, age int) (*foo.Foo, error)
    GetFoo(id int64) (*foo.Foo, error)
    UpdateFoo(f *foo.Foo) error
    DeleteFoo(id int64) error

    // repeat for a dozen or more entity types
}
```

INTERFACES AS EXPORTED ABSTRACTIONS

```
package server

import "example.com/gooop/foo"

func NewService() *ServiceImpl {
    ...
}

type ServiceImpl struct {}

func (s *ServiceImpl) CreateFoo(name string, age int) (*foo.Foo, error) {
    ...
}

...
```

INTERFACES AS EXPORTED ABSTRACTIONS

```
package consumer

import (
    "example.com/gooop/foo"
    "example.com/gooop/service/server"
)

func PrintFoo(id int64, s server.Service) {
    foo, err := s.GetFoo(id)
    if err != nil {
        fmt.Printf("unable to get Foo with ID=%d: %v", id, err)
        return
    }
    fmt.Printf("%#v\n", *foo)
}
```

INTERFACES AS EXPORTED ABSTRACTIONS

Interfaces in Go

- Define one or more methods that represent some set of behavior
- Satisfied implicitly by any type having the right methods
- Should be defined by the **consumer**, not the provider

INTERFACES AS EXPORTED ABSTRACTIONS

Interfaces in Go

- Insulation is "free" - consumer is not bound to *ANY* implementation
- Simplifies testing - tests only need to deal with the method(s) explicitly being called
- Eases future development
 - Implementations can satisfy many interfaces, even those that don't exist yet
 - Providers can add new functionality without breaking existing interfaces

INTERFACES AS EXPORTED ABSTRACTIONS

```
package consumer

import "example.com/gooop/foo"

// FooGetter defines a "get a Foo by ID" behavior that is satisfied by
// server.Service *without* tightly coupling this package to the server
// implementation
type FooGetter interface {
    // GetFoo accepts a 64-bit integer ID and returns the matching Foo
    // entity or an error
    GetFoo(id int64) (*foo.Foo, error)
}
```


INTERFACES AS EXPORTED ABSTRACTIONS

```
package consumer

import (
    "example.com/gooop/foo"
    "example.com/gooop/service/server"
)

func PrintFoo(id int64, g FooGetter) {
    foo, err := g.GetFoo(id)
    if err != nil {
        fmt.Printf("unable to get Foo with ID=%d: %v", id, err)
        return
    }
    fmt.Printf("%#v\n", *foo)
}
```

INTERFACES AS EXPORTED ABSTRACTIONS

Signs of GOOOP

- Used the "good" name for the interface
- The implementation has an `Impl` (or similar) suffix
- The interface re-declares every method on the implementation struct
- There are no references to the interface in the package that declares it



EXAMPLES OF GOOOP

3. Applying architectural patterns literally

ARCHITECTURAL PATTERNS

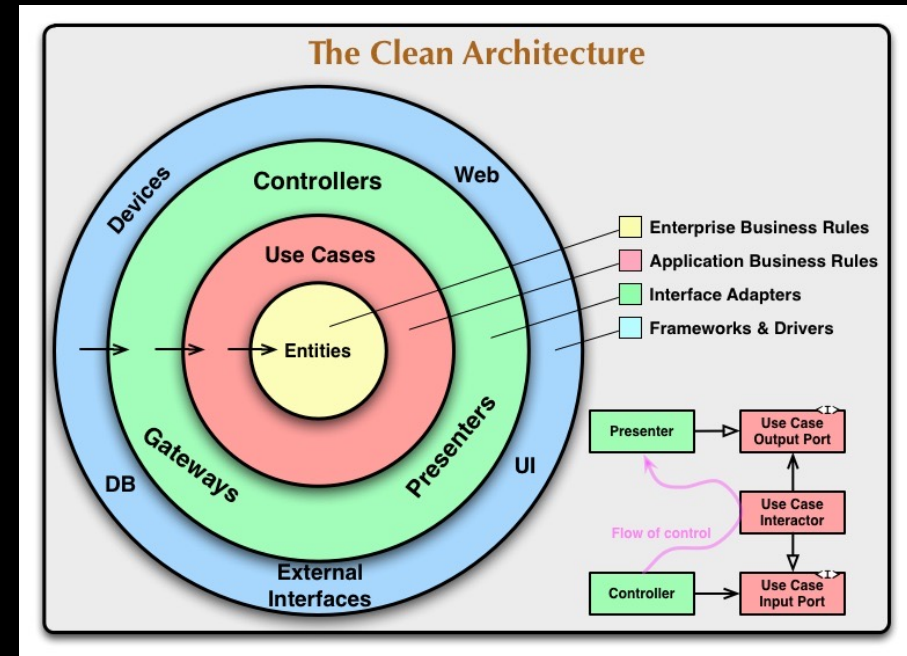
- Many new gophers ask "How should I structure my project?"
- Lots of opinions and examples
- Prior art in OOP ... but Go is not *really* OO

CLEAN ARCHITECTURE

- Defined by "Uncle Bob" Martin in 2012
 - Pattern for software with clean boundaries that make it easier to maintain
- Combines several other patterns with a focus on practical guidance

CLEAN ARCHITECTURE

- 4 layers (but allows for more)
 - Entities
 - Use Cases
 - Interfaces and Adapters
 - Frameworks and Drivers
- Dependencies can only point inward



MODEL, VIEW, *

- Focus on building user interfaces
 - Model
 - View
 - Controller/Presenter
- Common in web applications

ARCHITECTURAL PATTERNS

Signs of GOOOP

- Packages named for kinds of things that mirror the pattern layers
- Types are repeated in each package
- Any entity/model change means updating multiple packages
- Manageable in isolation, rough for cross-project integration

ARCHITECTURAL PATTERNS

- These patterns have stuck around all these years for a reason
- Go *can be* object-oriented to an extent
- Accept and apply the wisdom of the past, but do it **in Go**

```
no_more_gooop/  
  bar.go ← 1  
  foo.go  
  cmd/  
    gooopy/  
      main.go ← 4  
  internal/  
    ← 2 gooopservice/  
      datastore.go ← 3  
      grpc.go  
      service.go  
      web.go  
    ← 2 database/  
      bar.go  
      db.go ← 3  
      foo.go  
    ...  
  ...
```

ARCHITECTURAL PATTERNS

1. Model/Domain in the root package
2. Other packages are bounded contexts
3. DataStore interface in goopservice, satisfied by database
4. main.go bootstraps DB and passes it into the service

S.O.L.I.D.

- Five principles that should be applied to produce good, solid code
- Typically phrased in OOP terms but can be adjusted to apply to Go

SINGLE RESPONSIBILITY

There should never be more than one reason for a class to change

- s/class/package
- *Avoid package names like `base`, `util`, or `common`*
 - Dave Cheney, Jan 2019

OPEN/CLOSED

Software entities should be open for extension, but closed for
modification

- We should be able to add new behavior without rewriting existing code to do so
- Consumer-defined interfaces and implicit interface satisfaction

INTERFACE SEGREGATION

Clients should not be forced to depend on interfaces that they do not use

- Consumer-defined interfaces
- As narrow as possible, ideally single-method
 - *The bigger the interface, the weaker the abstraction.* – the Go Proverbs
- Compose narrower interfaces to combine behaviors

LISKOV SUBSTITUTION

Objects of a superclass should be replaceable by objects of a subclass
without affecting the correctness of the program

- Consumer-defined interfaces to the rescue once again
- Duck typing means consumers don't need to be coupled to **any** implementation, not even the "base" one

DEPENDENCY INVERSION

Depend upon abstractions, not concretions

- *A great rule of thumb for Go is accept interfaces, return structs.* – lots of Gophers in various blogs
- Define interfaces for input parameters
- Return concrete types that can satisfy consumer-defined interfaces

WRAP-UP

Single Responsibility
Open/Closed
Interface Segregation
Liskov Substitution
Dependency Inversion



WRAP-UP

- Go is not really an object-oriented language
 - OOP concepts can still be used, just not verbatim
- Understand the ideas behind those OOP patterns and practices then write idiomatic Go
- *Clear is better than clever* – the Go Proverbs