Software Architecture

Jean-Guillaume Louis

References

Clean Code: A Handbook of Agile Software Craftsmanship (Anglais) (Robert C. Martin)

Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin)

Patterns of Enterprise Application Architecture (Addison-Wesley Signature Series (Fowler))

Refactoring: Improving the Design of Existing Code (Addison-Wesley Signature Series (Fowler))

Interesting talks - Youtube playlist

What is Software Architecture?

Why? What? How?

What we build *is* the **architecture**.

It's the *system* itself.

It's also the process that *builds*, *test* and *deliver* the software.

Why? What? How?

... But we first need to know **why** we build it.

... For **whom**?

... For **what**?

... Under what **constraints**?

... Optimized toward what **characteristics**?

Think “analysis”, “requirements gathering”, “planning” , ‘scope definition”

Why? What? How?

Some call it “Design”.

It’s the nitty gritty of the system implementation. All the (ugly) details holding the system together.

Think “implementation choices”, “design pattern”, “tooling”

Why? What? How?

Where is the frontier?

Answer is: **you shouldn’t care**.

There is not point at which, during a the building of a system, you should stop asking yourself these questions.

In practice, architecture is an ongoing activity.

A much better definition

“A shared understanding of the system”

- *Ralph E. Johnson*

The challenge being in how you communicate that knowledge.

Architecturing is about communicating the intent of your system.

Addressing some BIG words

Quality

Quality of a system is typically on a multi-criteria scale.

To improve quality of a system, you need to improve the characteristics that matters.

What matters for one system won’t matter the same for another.

Maintainability

Maintainability is a hidden quality metrics for the end user.

But it's arguably the only one that matters to the developer!

Always build with maintainability in mind.

Performance

Performance metrics comes in many flavours: processing time, memory usage, network latency…

Performance is measurable.

When you improve for performance, you need to start measuring.

… Remember: you cannot have it all!

Scalability

“The ability to scale the load of the system”

Whatever *load* means: network traffic, user count, concurrent requests, storage space, computing power.

Architecture of the course

Spoiler alert: there is a twist at the end!

The intent of this course

It’s not:

- a recipe book for architectural pattern
- a theoretical course

The intent of this course

It’s not:

- a recipe book for architectural pattern
- a theoretical course

… But it is:

- a time to train your critical sense
- a practical survival guide
- a moment to talk about real world software
- a bottom-up approach to software architecture

The plan

- Coding warm up - OO basics
- *Survival guide* - Git & how to use it in a team
- SOLID principles
- (Architectural dependencies)
- *Survival guide* - testing
- Concurrent programming
- Client-Server architecture
- Layered architecture
- *Survival guide* - continuous integration
- Architectural patterns

Evaluation

- Exercises (30% pts)
  - Each course section will contain exercices
  - All exercices have to be done for next session
- Group work (70 % pts)
  - There are 5 topics - class will be split accordingly
  - Group should be homogenous in skills
  - Objective: present an architectural pattern
    - By *any* (legal) means: ex cathedra presentation, exercice, real world software analysis, usage of specific technology.
    - It’s not because you are giving the course, that the audience should be inactive. Makes us participate.
    - Expect questions.

Evaluation - List of topics

- Event-driven (Event sourcing) architecture
- Microkernel
- Data driven architecture (or to an extent, alternatives to OO)
- MicroServices
- Map Reduce

Evaluation grid

You’ll be evaluated both by me and your peers. At the end of the presentation, we will concert ourselves and try answer these questions together.

Was the presentation clear?

Was the subject treated with enough depth?

Did the audience learn something?

A word on the tools

We'll use Golang as a main programming language for course exercices.

We'll try to reduce dependency on frameworks, for simplicity sake. However, I do NOT discourage the usage of such tools!

We'll use Git and github and maybe Travis CI.

You are responsible for what other tools you bring to this course, like the IDE you choose. VS Code and Goland are popular. The later is proprietary.

Why Golang

Golang is, like this course, very opinionated.

As a software, it’s a textbook example of architecture as a shared understanding.

It’s very close to C.

It is designed for web oriented concurrent application. Also it’s very open-source friendly.

It has native unit testing.

OO basics

Variables…

```python
a = 42 #int
b = 3.2 #float
c = "hello" #str
```

```go
var a int = 2

var b float64 = 3.2

var c string = “hello”
```

or , within a function:

```go
a := 2

b := 3.2

c := “hello”
```

… and Functions

```python
def hello():
    return “hello golang”
```

```go
func hello() string{
    return "hello python. What’s up?"
}
```

What is an object?

An object is an abstracted type

Variables define its state.

Methods define a communication protocol and associated behaviours.

An object defines both its interface and an implementation for that interface.

The definition of an object is language dependant.

It is possible to use OO paradigm in non-OO language.

What is an object?

```python
# shape.py
class Rectangle():
    def __init__(self, length, width):
        self.width  = width
        self.length = length

    def get_area(self):
        return self.width * self.length
```

```go
// shape.go
package shape

type Rectangle struct {
    Width  float64
    Length float64
}

func (r Rectangle) Area() float64{
    return r.Width  * r.Length
 }
```

Golang Struct & methods

```
type Rectangle struct {
    Width  float64
    Length float64
}
```

A struct is a collection of fields.

```
func (r Rectangle) Area() float64{
    return r.Width  * r.Length
 }
```

A method is a function with a receiver.

Namespace

```python
# shape.py
class Rectangle():
    def __init__(self, length, width):
        self.width  = width
        self.length = length

    def get_area(self):
        return self.width * self.length
```

```go
// shape.go
package shape

type Rectangle struct {
    Width  float64
    Length float64
}

func (r Rectangle) Area() float64{
    return r.Width  * r.Length
 }
```

Namespace

```python
import shape

if __name__ == "__main__":
    rec = shape.Rectangle(2, 3)
     print("area: " + rec.get_area())
```

```go
package main

import (
    "fmt"
    "shape"
)

func main() {
    r := shape.Rectangle{2, 3}
    fmt.Printf("area: %d", r.Area())
 }
```

Polymorphism

```
package main

import (
    "fmt"
    "shape"
)

type Shape interface {
    Area() float64
}

func main() {
    var totalArea float64
    shapes := []Shape{shape.Rectangle{2, 3}, shape.Circle{5}}
    for _, shape := range shapes {
        totalArea += shape.Area()
    }
    fmt.Printf("total area: %d", totalArea)
}
```

Inheritance

Golang does not feature type inheritance.

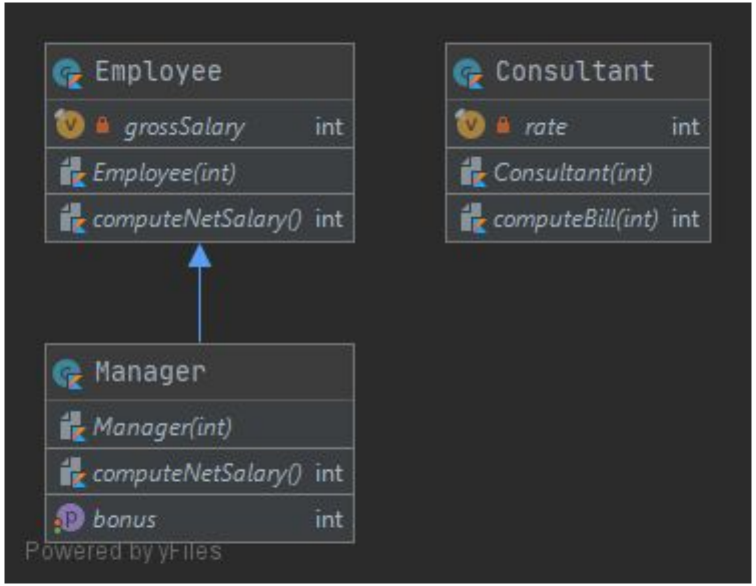
Inheritance is problematic with deep hierarchy.

While its purpose is to avoid code repetition, it often ends up with logic that is split between child and parent, breaking encapsulation and thus featuring relationships that are hard to break.

Golang favors polymorphic interfaces and composition instead. There are other ways to avoid code duplication.

Golang offers something that might resemble inheritance (type embedding) but is actually composition.

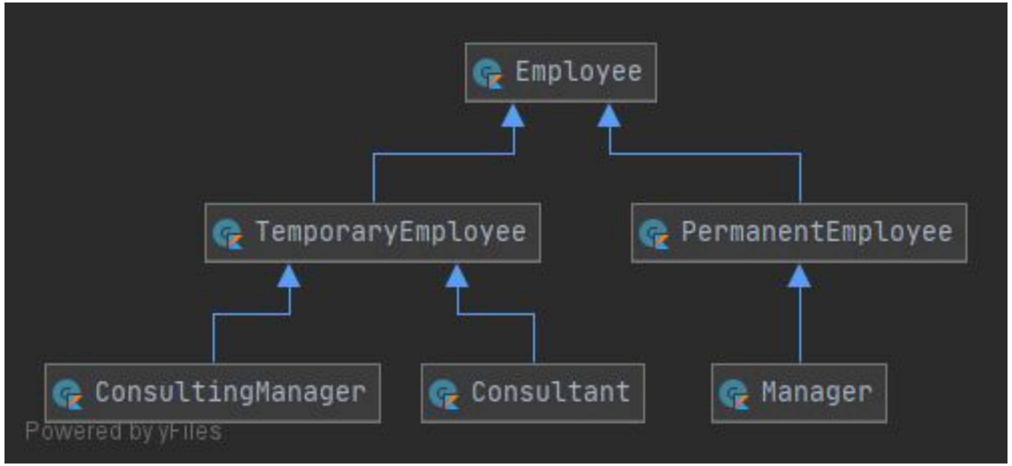
Inheritance (example)



Here is current version of a payroll system

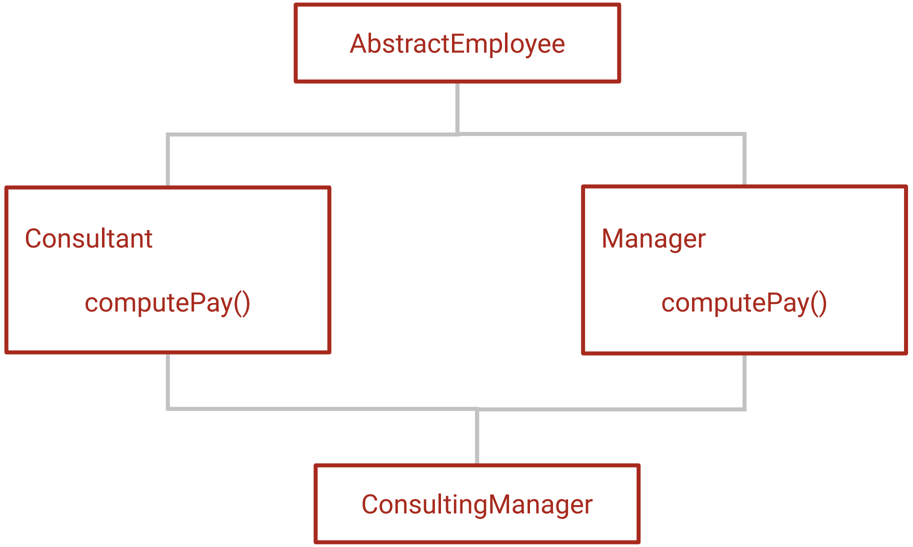
What about a managing consultant?

Inheritance (example continued)



A pure inheritance solution causes strong coupling in your code.

The diamond problem



What implementation of computePay will ConsultingManager inherit?

Interface

An interface defines a contract by defining method signatures without implementation*.

This is your main tool in OO for decoupling.

When your problems are caused by strong coupling, using interface is the tool you need.

We will learn specific techniques later on.

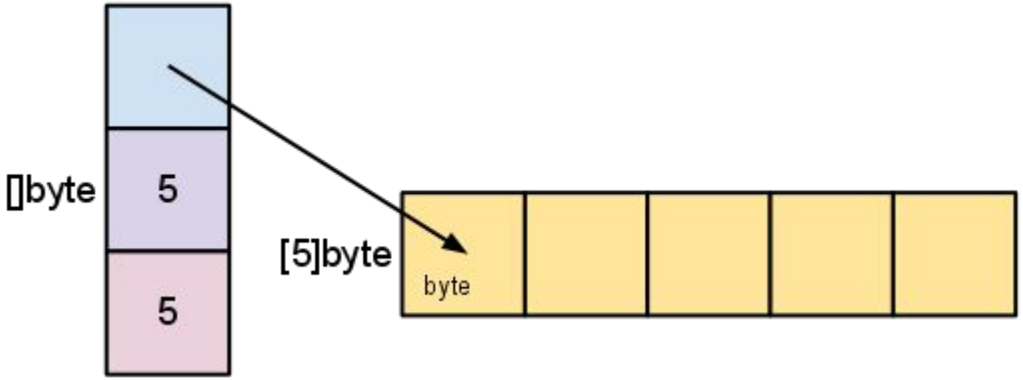
Important “low” level details

What is a pointer?

```
var p *int    // pointer declaration : “p is of type pointer to an int”


i := 42

p = &i        // operator p generate a pointer to its operand


*p = 21         // set i through the pointer p (dereferencing)
```

What is a pointer?

a Golang slice contains a pointer to an array. It’s an elegant way to provide dynamic sized lists.



Passing by value / Passing by reference

Pass by value:

- Data type is small enough
- You don’t care about state duplication of your passed type

Pass by reference when:

- Data type is large or unbound
- You want to modify the state of the passed type
- You are passing slices, channels or maps because they contain pointer to the underlying data

Stack and Heap

A Go process spawns as many stacks as goroutines. The heap is common.

Contrary to some other languages, the developer does not choose freely where to allocate memory. The compiler uses a process called *escape analysis* to determine what goes on stack (static data with known lifecycle) and on the heap (dynamic data).

Remember this: Stack is small. Heap is large. Large data should go on the heap.

1. Avoid passing large data by value. Value is copied to the stack.
2. Avoid unbound recursion.

Garbage collector

Garbage collector do the deallocation of objects on heap memory for us.

It hides complexity but it may also hides problems.

Memory leaks are still possible if you don’t realize you hold a reference to an object.

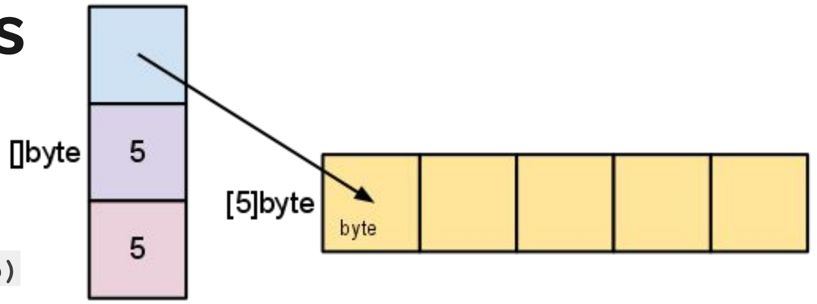
On the other hand, usage of weak references techniques are common and may cause unwanted behaviour.

As a rule of thumb, pay attention that each resource held is released in a **symmetrical** way.

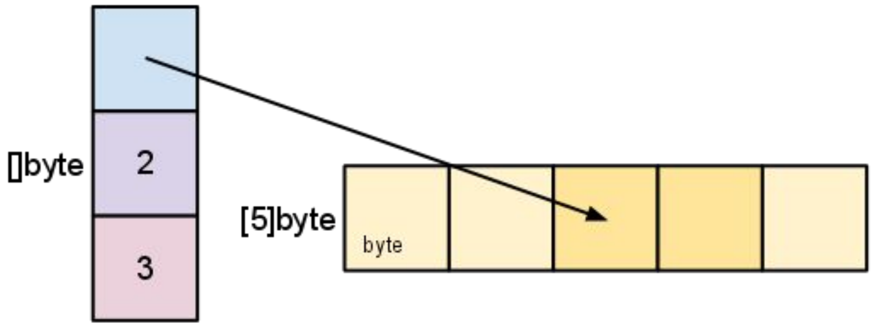
Memory leaks are insidious

The second line of code does not recreate a new backing array.

```
s := make([]byte, 5)
```



```
s = s[2:4]
```



Imagine if we had allocated a few MB!

The array will only be garbage collected once all pointing reference are themselves out of scope or garbage collected.

The most important advice

Code is meant to be read

Code is read more than written. So the emphasis should be put on the ease of reading it, rather than writing it. Don’t be too “clever” with your code.

As a rule of thumb, adapt your code style to (in order of priority):

1. The style of the existing codebase
2. What is idiomatic in the language

Introduction to Go

https://tour.golang.org/

Your objective before going further: get a feel for the language.

You should be able to go through the "Basics" section.

We'll go together deeper with" Methods and interfaces". Go as far as you can. Don't hesitate to skip what seems trivial.

Designing by interface

Interface in Go

A type is said to implement an interface if it implements all its methods.

Stringer

-- /src/fmt/print.go:62 --

```go
type Stringer interface {
        String() string
}
```

```go
type Book struct {
    Title  string
    Author string
}

func (b Book) String() string {
    return fmt.Sprintf("Book: %s - %s", b.Title, b.Author)
}
```

Stringer - exercice

1. Define a Rectangle type that implements fmt.Stringer
2. Define a Circle type that implements fmt.Stringer
3. Define a Shape interface, implemented by Circle and Rectangle such as this code:

```go
func main() {
    r := &Rectangle{2, 3}
    c := &Circle{5}

    shapes := []Shape{r, c}

    for _, s := range shapes {
        fmt.Println(s)
    }
}
```

outputs:

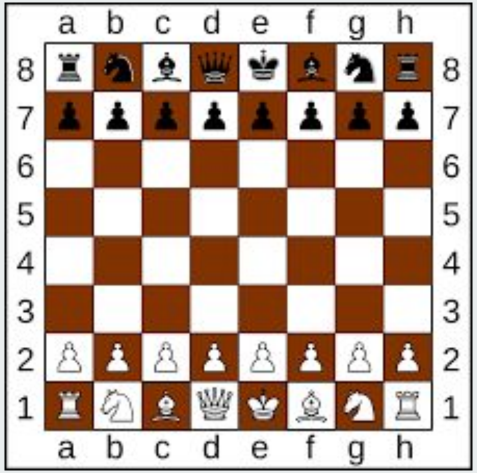
```
Square of width 2 and length 3
Circle of radius 5
```

Reader/Writer - exercice

1. Read the documentation for the io.Reader type
2. Implement a type that satisfies the io.Reader interface and reads from another io.Reader, modifying the stream by removing spaces.

```
type spaceEraser struct {
    r io.Reader
}

func main() {
    s := strings.NewReader("H e l l o w o r l d!")
    r := spaceEraser{s}
    io.Copy(os.Stdout, &r)
 }
```

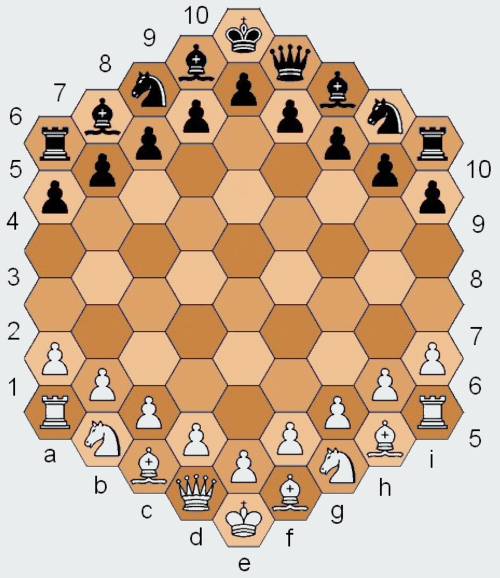
Modelize this



Your code should at least have domain model logic for:

- Describing the board and its coordinates
- Describing each piece movement
- Moving a piece on the board
- The game state (which turn is it?)

CEO got an idea: Hexagonal chess



What changes in our model?

How would you handle coordinates on a hex grid?

Functionality - save Game

How would you handle saving the game?

… on a File?

… on the network?

Command line interface

Create a CLI for our game of chess.