Composition with Go

Sep 13, 2015

https://www.goinggo.net/2015/09/composition-with-go.html

Composition goes beyond the mechanics of type embedding. It's a paradigm we can leverage to design better APIs and to build larger programs from smaller parts. It all starts from the declaration and implementation of types that have a single purpose. Programs that are architected with composition in mind have a better chance to grow and adapt to changing needs. They are also much easier to read and reason about.

To demonstrate these concepts, we will be reviewing the following program:

Sample Code

This code sample explores the mechanics behind embedding and provides us with an opportunity to discuss how, with composition, we can design for both flexibility and readability. Every identifier that is exported from a package makes up the package's API. This includes all the constants, variables, types, methods and functions that are exported. Comments are a frequently-overlooked aspect of every package's API, so be very clear and concise when communicating information to the user of the package.

The example is long so let's break it down in pieces and walk our way through it.

The idea behind this program is we have a contractor that we hired to renovate our house. In particular, there are some boards in the house that have rotted and need to be yanked out, as well as new boards that need to be nailed in. The contractor will be given a supply of nails, boards to work with and tools to perform the work.

Listing 1

```
13 // Board represents a surface we can work on.
14 type Board struct {
15     NailsNeeded int
16     NailsDriven int
17 }
```

In listing 1, we have the declaration of the `Board` type. A `Board` has two fields, the number of nails the board needs and the current number of nails driven into the board. Now, let's look at the interfaces that are declared:

Listing 2

```
21 // NailDriver represents behavior to drive nails into a board.
22 type NailDriver interface {
23     DriveNail(nailSupply *int, b *Board)
24 }
25
26 // NailPuller represents behavior to remove nails into a board.
27 type NailPuller interface {
28     PullNail(nailSupply *int, b *Board)
29 }
30
31 // NailDrivePuller represents behavior to drive/remove nails into
a board.
32 type NailDrivePuller interface {
33     NailDriver
34     NailPuller
35 }
```

Listing 2 shows the interfaces that declare the behavior we need from the tools that may be used by the contractor. The `NailDriver` interface on line 22 declares behavior to drive a single nail into a board. The method is given a supply of nails and the board to drive the nail into. The `NailPuller` interface on line 27 declares the opposite behavior. The method is given a supply of nails and the board, but it will pull a nail from the board to place back into the supply.

These two interfaces, `NailDriver` and `NailPuller`, each implement a single, well defined behavior. This is what we want. Being able to break down behavior into individual, simple acts, lends itself to composability, flexibility and readability as you will see.

The final interface declared on line 32 is named `NailDrivePuller`:

Listing 3

```
32     type NailDrivePuller interface {
33         NailDriver
34         NailPuller
35     }
```

This interface is composed from both the `NailDriver` and `NailPuller`interfaces. This is a very common pattern you will find in Go, taking existing interfaces and grouping them into composed behaviors. You will see how this plays into the code later on. For now, any concrete type value that implements both the driver and puller behaviors will also implement the `NailDrivePuller` interface.

With the behaviors defined, it is time to declare and implement some tools:

Listing 4

```
39 // Mallet is a tool that pounds in nails.
40 type Mallet struct{}
41
42 // DriveNail pounds a nail into the specified board.
43 func (Mallet) DriveNail(nailSupply *int, b *Board) {
44     // Take a nail out of the supply.
45     *nailSupply-
46
47     // Pound a nail into the board.
48     b.NailsDriven++
49
50     fmt.Println("Mallet: pounded nail into the board.")
51 }
```

In listing 4 on line 40, we have the declaration of a struct type named `Mallet`. This type is declared as an empty struct since there is no state that needs to be maintained. We only need to implement behavior.

A mallet is a tool that is used to hammer in nails. So on line 43, the `NailDriver`interface is implemented via the declaration of the `DriveNail` method. The implementation of the method is irrelevant, it subtracts a value from the nail supply and adds a value to the board.

Let's look at the declaration and implementation of our second tool:

Listing 5

```
53 // Crowbar is a tool that removes nails.
54 type Crowbar struct{}
55
56 // PullNail yanks a nail out of the specified board.
57 func (Crowbar) PullNail(nailSupply *int, b *Board) {
58     // Yank a nail out of the board.
59     b.NailsDriven-
60
61     // Put that nail back into the supply.
62     *nailSupply++
63
64     fmt.Println("Crowbar: yanked nail out of the board.")
65 }
```

In listing 5, we have the declaration of our second tool on line 54. This type represents a crowbar which can be used to pull nails from a board. On line 57, we have the implementation of the `NailPuller` interface via the declaration of the `PullNail` method. The implementation once again is irrelevant but you can see how a nail is subtracted from the board and added to the nail supply.

At this point in the code, we have declared tooling behavior via our interfaces and implemented that behavior via a set of struct types that represent two distinct tools our contractor can use. Now, let's create a type that represents a contractor who performs the work using these tools:

Listing 6

```
69 // Contractor carries out the task of securing boards.
70 type Contractor struct{}
71
72 // Fasten will drive nails into a board.
73 func (Contractor) Fasten(d NailDriver, nailSupply *int, b *Board)
{
74     for b.NailsDriven < b.NailsNeeded {
75         d.DriveNail(nailSupply, b)
76     }
77 }
```

In listing 6, we have the declaration of the `Contractor` type on line 70. Again, we don't need any state so we are using an empty struct. Then on line 73, we see the `Fasten` method, one of the three methods declared against the `Contractor` type.

The method `Fasten` is declared to provide a contractor the behavior to drive the number of nails that are needed into a specified board. The method requires the user to pass as the first parameter a value that implements the `NailDriver`interface. This value represents the tool the contractor will use to execute this behavior. Using an interface type for the this parameter allows the user of the API to later create and use different tools without the need for the API to change. The user is providing the behavior of the tooling and the `Fasten` method is providing the workflow for when and how the tool is used.

Notice we are using the `NailDriver` interface and not the `NailDrivePuller`interface for the parameter's type. This is very important since the only behavior that is needed by the `Fasten` method is the ability to drive nails. By declaring only the behaviors we need, our code is easier to comprehend while also being easier to use due to minimal coupling. Now, let's look at the `Unfasten` method:

Listing 7

```
79 // Unfasten will remove nails from a board.
80 func (Contractor) Unfasten(p NailPuller, nailSupply *int, b
*Board) {
81     for b.NailsDriven > b.NailsNeeded {
82         p.PullNail(nailSupply, b)
83     }
84 }
```

The `Unfasten` method declared in listing 7 provides the contractor the opposite behavior of the `Fasten` method. This method will remove as many nails as necessary from the specified board and add those back into the supply. This method accepts only tools that implement the `NailPuller` interface. This is exactly what we want since this is the only behavior required in the implementation of this method.

The final behavior for a contractor is a method called `ProcessBoards` which allows the contractor to work on a set of boards at one time:

Listing 8

```
86 // ProcessBoards works against boards.
87 func (c Contractor) ProcessBoards(dp NailDrivePuller, nailSupply
*int, boards []Board) {
88     for i := range boards {
89         b := &boards[i]
90
91         fmt.Printf("contractor: examining board #%d: %+v\n", i+1,
b)
92
93         switch {
94         case b.NailsDriven < b.NailsNeeded:
95             c.Fasten(dp, nailSupply, b)
96
97         case b.NailsDriven > b.NailsNeeded:
98             c.Unfasten(dp, nailSupply, b)
99         }
100     }
101 }
```

In listing 8, we see the declaration and implementation of the `ProcessBoards`method. This method accepts as its first parameter only values that implement both the `NailDriver` and `NailPuller` interface:

Listing 9

```
87 func (c Contractor) ProcessBoards(dp NailDrivePuller, nailSupply
*int, boards []Board) {
```

Listing 9 shows how the method is declaring the exact behavior it needs via the `NailDrivePuller` interface type. We want our API's to specify only the behaviors that are required or are being used. The `Fasten` and `Unfasten` methods require a value that implement a single act of behavior but `ProcessBoards`requires a value that implements two behaviors at the same time:

Listing 10

```
93         switch {
94         case b.NailsDriven < b.NailsNeeded:
95             c.Fasten(dp, nailSupply, b)
96
97         case b.NailsDriven > b.NailsNeeded:
98             c.Unfasten(dp, nailSupply, b)
99         }
```

Listing 10 shows a piece of the implementation of `ProcessBoards` and how the value of interface type `NailDrivePuller` is used in the method calls to `Fasten`and `Unfasten`.

When the call to `Fasten` is made on line 95, the `NailDrivePuller` interface value is passed into the method as an interface value of type `NailDriver`. Let's look at the declaration of the `Fasten` method once again:

Listing 11

```
73 func (Contractor) Fasten(d NailDriver, nailSupply *int, b *Board)
{
```

Notice the `Fasten` method requires a value of interface type `NailDriver` and we are passing a value of interface type `NailDrivePuller`. This is possible because the compiler knows that any concrete type value that can be stored inside a `NailDrivePuller` interface value must also implement the `NailDriver`interface. Therefore, the compiler accepts the method call and the assignment between these two interface type values.

The same is true for the method call to `Unfasten`:

Listing 12

```
80 func (Contractor) Unfasten(p NailPuller, nailSupply *int, b
*Board) 81 {
```

The compiler knows that any concrete type value that is stored inside an interface value of type `NailDriverPuller` also implements the `NailPuller` interface. Therefore, passing an interface value of type `NailDriverPuller` is assignable to an interface value of type `NailPuller`. Because there is a static relationship between these two interfaces, the compiler is able to forego generating a runtime type assertion and instead generate an interface conversion.

With the contractor in place, we can now declare a new type that declares a toolbox for use by a contractor:

Listing 13

```
105 // Toolbox can contains any number of tools.
106 type Toolbox struct {
107     NailDriver
108     NailPuller
109
110     nails int
111 }
```

Every good contractor has a toolbox and in listing 13 we have the declaration of a toolbox. The `Toolbox` is a struct type that embeds on line 107 an interface value of type `NailDriver` and on line 108 an interface value of type `NailPuller`. Then on line 110, a supply of nails is declared via the `nails` field.

When embedding a type inside of another type, it is good to think of the new type as the outer type and the type being embedded as the inner type. This is important because then you can see the relationship that embedding a type creates.

Any type that is embeded always exists as an inner value, inside the outer type value. It never loses its identity and is always there. However, thanks to inner type promotion, everything that is declared within the inner type is promoted to the outer type. This means through a value of the outer type, we can access any field or method associated with the inner type value directly based on the rules of exporting.

Let's look at an example of this:

Listing 14

```
01 package main
02
03 import "fmt"
04
05 // user defines a user in the program.
06 type user struct {
07     name  string
08     email string
09 }
10
11 // notify implements a method that can be called via
12 // a pointer of type user.
13 func (u *user) notify() {
14     fmt.Printf("Sending user email To %s<%s>\n",
15         u.name,
16         u.email)
17 }
```

In listing 14, we have the declaration of a type named `user`. This type contains two `string` fields and a method named `notify`. Now, let's embed this type into another type:

Listing 15

```
19 // admin represents an admin user with privileges.
20 type admin struct {
21     user  // Embedded Type
22     level string
23 }
```

Now we can see the relationship between the inner type and outer type in listing 15. The `user` type is now an inner type of `admin`, the outer type. This means that we can call the `notify` method directly from a value of type `admin` thanks to inner type promotion. But since the inner type value also exists, in and of itself, we can also call the `notify` method directly from the inner type value:

Listing 16

```
25 // main is the entry point for the application.
26 func main() {
27     // Create an admin user.
28     ad := admin{
29         user: user{
30             name:  "john smith",
31             email: "john@yahoo.com",
32         },
33         level: "super",
34     }
35
36     // We can access the inner type's method directly.
37     ad.user.notify()
38
39     // The inner type's method is promoted to the outer type.
40     ad.notify()
41 }
```

Listing 16 shows how a value of the outer type `admin` is created using a struct literal on line 28. Inside the struct literal a second struct literal is used to create and initialize a value for the inner type `user`. With a value of type `admin` in place, we can call the `notify` method directly from the inner type value on line 37 or through the outer type value on line 40.

One last thing, this is neither subtyping nor subclassing. Values of type `admin` can't be used as values of type `user`. Values of type `admin` are only values of type `admin` and values of type `user` are only values of type `user`. Thanks to inner type promotion, the fields and methods of the inner type can be accessed directly through a value of the outer type.

Now back to our toolbox:

Listing 17

```
105 // Toolbox can contains any number of tools.
106 type Toolbox struct {
107     NailDriver
108     NailPuller
109
110     nails int
111 }
```

We have not embedded a struct type into our `Toolbox` but two interface types. This means any concrete type value that implements the `NailDriver` interface can be assigned as the inner type value for the `NailDriver` embedded interface type. The same holds true for the embedded `NailPuller` interface type.

Once a concrete type is assigned, the `Toolbox` is then guaranteed to implement this behavior. Even more, since the toolbox embeds both a `NailDriver` and `NailPuller` interface type, this means a `Toolbox` also implements the `NailDrivePuller` interface as well:

Listing 18

```
31      NailDrivePuller interface {
32          NailDriver
33          NailPuller
34      }
```

In listing 18 we see the declaration of the `NailDrivePuller` interface again. Embedding interface types takes the idea of inner type promotion and interface compliance to the next level.

Now we are ready to put everything together by implementing the `main` function:

Listing 19

```
115 // main is the entry point for the application.
116 func main() {
117     // Inventory of old boards to remove, and the new boards
118     // that will replace them.
119     boards := []Board{
120         // Rotted boards to be removed.
121         {NailsDriven: 3},
122         {NailsDriven: 1},
123         {NailsDriven: 6},
124
125         // Fresh boards to be fastened.
126         {NailsNeeded: 6},
127         {NailsNeeded: 9},
128         {NailsNeeded: 4},
129     }
```

The `main` function starts out on line 116 in listing 19. Here we create a slice of boards and specify what each board needs in terms of nails. Next, we create our toolbox:

Listing 20

```
131     // Fill a toolbox.
132     tb := Toolbox{
133         NailDriver: Mallet{},
134         NailPuller: Crowbar{},
135         nails:      10,
136     }
137
138     // Display the current state of our toolbox and boards.
139     displayState(&tb, boards)
```

In listing 20, we create our `Toolbox` value and create a value of the struct type `Mallet` and assign it to the inner interface type `NailDriver`. Then we create a value of struct type `Crowbar` and assign it to the inner interface type `NailPuller`. Finally, we add 10 nails to the toolbox.

Now let's create the contractor and process some boards:

Listing 21

```
141     // Hire a contractor and put our contractor to work.
142     var c Contractor
143     c.ProcessBoards(&tb, &tb.nails, boards)
144
145     // Display the new state of our toolbox and boards.
146     displayState(&tb, boards)
147 }
```

On line 142 in listing 21 we create a value of type `Contractor` and call the `ProcessBoard` method against it on line 143. Let's look at the declaration of the `ProcessBoards` method once more:

Listing 22

```
87 func (c Contractor) ProcessBoards(dp NailDrivePuller, nailSupply
*int, boards []Board) {
```

Once again, we see how the `ProcessBoard` method takes as its first parameter any concrete type value that implements the `NailDrivePuller` interface. Thanks to the type embedding of the `NailDriver` and `NailPuller` interface types inside the `Toolbox` struct type, a pointer of type `Toolbox` implements the `NailDrivePuller` interface and can be passed through.

***Note**: The `Mallet` and `Crowbar` types implement their respective interfaces with value receivers. Based on the rules for Method Sets, both values and pointers then satisfy the interface. This is why we can create and assign values of our concrete type values to the embedded interface type values. The call to `ProcessBoards` in listing 21 on line 143 is using the address of the `Toolbox` value because we want to share a single `Toolbox`. However, a value of type `Toolbox` also satisfies the interface. If the concrete types used pointer receivers to implement the interfaces, then only pointers would have satisfied the interface.*

To be complete, here is the implementation of the `displayState` function:

Listing 23

```
149 // displayState provide information about all the boards.
150 func displayState(tb *Toolbox, boards []Board) {
151     fmt.Printf("Box: %#v\n", tb)
152     fmt.Println("Boards:")
153 
154     for _, b := range boards {
155         fmt.Printf("\t%+v\n", b)
156     }
157 
158     fmt.Println()
159 }
```

In conclusion, this example has tried to show how composition can be applied and the things to think about when writing your code. How you declare your types and how they can work together is very important. Here is a summary of the things we touch upon in the post:

* Declare the set of behaviors as discrete interface types first. Then think about how they can be composed into a larger set of behaviors.

* Make sure each function or method is very specific about the interface types they accept. Only accept interface types for the behavior you are using in that function or method. This will help dictate the larger interface types that are required.

* Think about embedding as an inner and outer type relationship. Remember that through inner type promotion, everything that is declared in the inner type is promoted to the outer type. However, the inner type value exists in and of itself as is always accessible based on the rules for exporting.

* Type embedding is not subtyping nor subclassing. Concrete type values represent a single type and can't be assigned based on any embedded relationship.

* The compiler can arrange interface conversions between related interface values. Interface conversion, at compile time, doesn't care about concrete types - it knows what to do merely based on the interface types themselves, not the implementing concrete values they could contain.