Object-Oriented Programming - Composing Objects

nprenet@bsu.edu

CS431, Spring 2022

Contents

Composing objects

Extending a type through composition

Goal: extending an existing type with additional functionalities.

turtlecomposites.go

```
type Turtle struct {
        Name string
        Position
}

func (t *Turtle) Walk(d Direction) {
        if d%2 == 0 { // limit to N,E,S,W
                offset := OffsetFromDirection(d)
                t.X += offset.X
                t.Y += offset.Y
        }
}

type FlyingTurtle struct {
        Fuel float32
        Turtle                  // anonymous embedded field
}

func (t *FlyingTurtle) Fly(d Direction) {
```

```
        offset := OffsetFromDirection(d)
        t.X += offset.X
        t.Y += offset.Y
        t.Fuel -= 0.5
}
```

Comments:

- Both `Turtle` and `FlyingTurtle` are concrete types.
- A `Turtle` object is *embedded* into the `FlyingTurtle` and provides it with the `Walk` functionality.
- In the OOP literature, this pattern is called **delegation**: an object (`FlyingTurtle`) forwards the handling of its requests to another object (`Turtle`).
- In Go, the exported fields of an *anonymous* embedded `struct` type are automatically promoted to the level their enclosing type, so that the following reference: `myTurtle.Turtle.Name` is equivalent to: `myTurtle.Name`.[1]

In use:

```
var t1 = Turtle{Name: "test1"}
t1.Walk(East)
var t2 = FlyingTurtle{Turtle: Turtle{Name: "test2"} , fuel: 2}
t2.Walk(South)          // Walk--the embedded Turtle's own property-- can be accessed the same
t2.Fly(Southwest)       // way as Fly
```

Considerations:

- Use when extension by subclassing is impractical or impossible (as in Go) and typically produces an explosion of subclasses to support every combination.
- Because every `FlyingTurtle` has its own copy of a `Turtle` object, responsibilities are easy to identify and the code easy to understand.
- However, because it mimicks inheritance, coupling is relatively tight, especially if the embedded object shares some implementation details with the enclosing structure. In the code above, both `Walk` and `Fly` modify the same fields: in some way, `FlyingTurtle` **inherits some of its implementation** from `Turtle`.

Decorating a component

Goal: add additional responsibilities to an object dynamically.

turtledecorator.go

```
type Turtle struct {
    Name string
    Position
}

func (t *Turtle) Walk(d Direction) { ... }

type FlyingTurtle struct {
    Fuel float32
    *Turtle          // embedded field is a pointer
}

func (t *FlyingTurtle) Fly(d Direction) { ... }
```

Comments:

[1] Usual rules apply: only capitalized names are exported to the client packages.

- The code is the same as in the previous section, but the embedded type is a *pointer* to a `Turtle`.
- What is made possible: two `FlyingTurtle` objects may share a `Turtle`; a `Turtle` component may used in more than one capacity, depending on the client's needs: as a plain `Turtle` component, or as a `FlyingTurtle`.
- The pattern is similar to the Gang of Four's **Decorator**: note, however, that a full-fledged Decorator uses interfaces and/or abstract classes, in order to document and enforce the `FlyingTurtle`'s conformance to the `Turtle` type; you may think of this one as a "lightweight" decorator.

In use:

```
var t1 = Turtle{Name: "test1"}
t1.Walk(South)
fmt.Println(t1.Position) // {0,1}
var t2 = FlyingTurtle{ Turtle: &t1, Fuel: 2 } // t2 decorates t1
fmt.Println(t2.Position) // {0,1}
t2.Fly(Northeast)
fmt.Println(t1.Position) // {1,0}  <--> Flying t2 moves t1
```

Considerations:

- Use when extension by subclassing is impractical or impossible (as in Go).
- To attach responsibilities to individual objects, not to a class.
- Easy to add or withdraw responsibilities to a component.
- Interactions between lots of little objects may be hard to track.

Subtype polymorphism through interfaces

A diversity of implementations for a common, abstract type

Goal: for a single declared type, provide different concrete implementations.

`turtlesinterfacesubtypes.go`

```
type Turtle interface {
        Move(d Direction)
        ToString() string
}

type Walker struct {
        Name string
        Position
}

type FlyingTurtle struct {
        Name string
        fuel float32
        Position
}

func GetFlyingTurtle( name string) *Flyer {            // Factory function
        return &Flyer{Name: name, fuel: 2.0}
}

func (t *WalkingTurtle) Move(d Direction) Turtle {
        if d%2 == 0 { // limit to N,E,S,W
                ...
```

```go
	}
	return t
}
func (t WalkingTurtle) ToString() string {
	return fmt.Sprintf("Name: %s, position: %d", t.Name, t.Position)
}

func (t *FlyingTurtle) Move(d Direction) Turtle {
	...
	return t
}
func (t FlyingTurtle) ToString() string {
	return fmt.Sprintf("Name: %s, position: %d, fuel: %f", t.Name, t.Position, t.fuel)
}
```

Comments:

- The `Turtle` interface defines which methods the concrete types are expected to implement (`Move` and `ToString`).
- Each concrete type has its own implementation of the interface methods.
- A concrete type may have additional data fields and methods, but they are not exposed by the interface. Ex. the `FlyingTurtle` concrete type has a hidden `fuel` field, that cannot be initialized from the client package: however, our package exposes the factory function `GetFlyingTurtle`, which may be used for that purpose.
- This pattern does not make use of composition.
- The `Turtle` interface type may be assigned a concrete type variable, or a pointer to it, if the `Move` method is to be called (because it mutates the structure, this method expects a pointer receiver).
- Because each structure has its own implementation, the interface method `Move` can return a reference to the object, thus making it possible chain method calls.

In use:

```go
var t1 Turtle                              // declaring an interface type Turtle
t1 = &WalkingTurtle{Name: "test1"}         // address of concrete type: Walker
t1.Move(South)
t1.ToString()                              // Name: test1, position: {0 1}

var t2 Turtle = GetFlyingTurtle("test2")   // hidden field initialized by factory function
t2.Move(South).Move(Southwest)             // chaining
t2.ToString()                              // Name: test2, position: {-1 2}, fuel: 1.0
```

Considerations:

- Use when implementation details and intrisic operations of types are to be hidden into a single, abstract type, that exposes a common interface to the calling code: can be used to make pluggable, replaceable subcomponents, that configure the runtime behaviour of a composite type (see next section). **Polymorphism** depends on this ability to define objects that inherit identical interfaces (instead of inheriting implementation through subclassing). OOP subtypes inherit one or several interfaces, not implementations, so that any `Flyer` may be substituted for a `Walker` in a function that accepts a `Turtle`.

- Use when different concrete types need to be **aggregated** in a container. For instance:

  ```go
  turtleArray := [...]Turtle{
  		&WalkingTurtle{Name: "Robert"}},
  		GetFlyingTurtle("Amelia")
  }
  ```

Equipping a concrete type with different implementations

Goal: through composition and the use of interfaces, configure a concrete type with one of several possible behaviors.

turtlestrategies.go

```
type Turtle struct {            // Concrete, composite type
        Name string
        Mover
}
func (t Turtle) ToString() string {
        return fmt.Sprintf("Name: %s, %s", t.Name, t.Mover.toString())
}

type Mover interface {          // Mover interface
        Move(d Direction)
        toString() string
}
type Walker struct {            // Concrete Mover (of the Walker kind)
        Position
}
func (t *Walker) Move(d Direction) { ... }
func (t Walker) toString() string {
        return fmt.Sprintf("position: %d", t.Position)
}

type Flyer struct {             // Concrete Mover (of the Flyer kind)
        fuel float32
        Position
}

func (t *Flyer) Move(d Direction) { ... }
func (t Flyer) toString() string {
        return fmt.Sprintf("position: %d, fuel: %.1f", t.Position, t.fuel)
}
func GetFlyer() *Flyer {
        return &Flyer{fuel: 2}
}
```

Comments:

- the `Turtle` concrete type has only one data field of its own (`Name`); its delegates its `Move` functionality to an *embedded* object that satisfies the `Mover` interface.
- Since both `Walker` and `Flyer` implement the `Mover` interface, they can both be used to provide new `Turtle` object with a specific behaviour: changing the guts of an object, while keeping its interface the same, is called the **strategy** pattern by the Gang of Four.
- Note the difference in case between the `Turtle` type's `ToString` method and the `Mover` type's `toString` method: the first one is exported to the client package, while the latter is not.
- The embedded Mover's `Move` function name is promoted to the containing structure's level, so that the client code can access it through a straightforward dot notation.

In use:

```
var t1 Turtle = Turtle{Name: "test1", Mover: &Walker{}}
t1.Move(West)
fmt.Printf("%s\n", t1.ToString()) // Name: test1, position: {-1 0}
```

```go
var t2 Turtle = Turtle{Name: "test2", Mover: GetFlyer()}
t2.Move(Southwest)
fmt.Printf("%s\n", t2.ToString()) // Name: test2, position: {-1 1}, fuel: 1.5
```

Considerations:

- Used to configure runtime behavior of an object, by plugging a component into it. This is a key aspect of **polymorphism**: it lets a client (`Turtle`) make few assumptions about a component (`Mover`) beyond supporting the interface.

Ad-hoc polymorphism: type assertions

Querying dynamic type of an interface value

When we are not sure of the dynamic (concrete) type of an interface value, and we'd like to test whether it is some particular type:

```go
var t Turtle = GetFlyingTurtle()            // Where Turtle is an interface that may
if t, ok := t.(*WalkingTurtle); ok  {       // be either a walking or flying turtle
        t.Walk()
} else if t, ok := t.(*FlyingTurtle); ok {
        t.Fly()                             // t's declared type changed to concrete type
}
```

For a type value `x`, a type assertion of the kind `xc, ok := x.(T)` checks whether `x`'s concrete type is identical to `T`.

- If successful, the second return value `ok` is assigned `true`, and `xc` is assigned the concrete value.
- Otherwise, `ok` is assigned `nil`, and `xc` is assigned `nil`.

Querying behavior of a concrete or interface type

An interface type *hides* every method of its concrete (dynamic) value that is not defined in it. For instance, in the code below, variable `t` has interface type that has specifies only a `Walk` function. Accordingly, `t` cannot fly, even though its dynamic value (`FlyingTurtle`) implements such a function.

```go
var t interface{                                    // non-flying declared type
    Walk(d Direction)
} = &FlyingTurtle{Turtle{ Name: "Clelia"}, Fuel: 2}

t.Fly(Southwest) // panic: t.Fly undefined (type Turtler has no field or method Fly)
```

To check that `t`'s dynamic type does have a `Fly` function, we define an interface `Flyer` that has just this method. Then we use a type assertion to change `t`'s declared type to the `Flyer` interface type, thus making the `Fly` function accessible:

```go
type Flyer interface {
        Fly(d Direction)
}
if t, ok := t.( Flyer ){
        t.Fly(Southwest)
}
```

When used in this manner, you may think of interfaces as on/off switches for an object's features.

Expressing unions of concrete types through empty interfaces

Goal: make a generic function able to deal with a variety of types.

Imagine an arithmetic parser that feeds on a sequence of tokens, where

- each token has a concrete type, e.g. `Id`, `LeftPar`, `RightPar`, `Multiply`... These types have no methods, and expose their state.
- the `Token` interface is an *empty interface* defined as follows:

  ```
  type Token interface{}
  ```

  Such an interface defines a *union* of concrete types, where the goal is not hide and abstract implementations details out of view (there are none), but to make us able to handle a variety of types, and to handle them accordingly.

Then, we can then define a function `Parse( []Token )`, whose logic discriminates between the concrete types:

```
func Parse( []Token ){
        prs := arithm.Parser(os.Stdin)
        for {
                tok, err := prs.Token()

                switch tok := tok.( type ){
                case arithm.Id:
                        ...  // do something
                case arithm.LeftPar:
                        ... // do something else
                }
        }
        ...
}
```

In the code above[2] the `tok.( type )` syntax returns the type value of a given `Token` object. In contrast with subtype polymorphism, where an interface is defined first and may then be satisfy with a (possibly infinite) variety of concrete types, the set of tokens is fixed, exposed to the client code and the `Token` type is their *union*.

[2]Inspired by an XML parser in Donovan, Kernighan, *The Go Programming Language*, p. 214.