Visibility and Operator Patterns

This lesson explains the possible uses of operators to gain maximum benefits without violating the visibility pattern.

WE'LL COVER THE FOLLOWING

The visibility pattern and interfaces

In Chapter 2, we saw how the simple *Visibility rule* dictates the access-mode to types, variables, and functions in Go. Chapter 8 showed how you could make the use of the `Factory` function mandatory when defining types in separate packages.

Define your interfaces in a high-level package that the other packages in your project depend on. Then they can just pass interface values around.

The operator pattern and interfaces

An operator is a *unary* or *binary* function which returns a new object and does not modify its parameters, like + and *. In C++, special infix operators (+, -, *, and so on) can be overloaded to support math-like syntax, but apart from a few special cases Go does not support operator overloading. To overcome this limitation, operators must be simulated with functions. Since Go supports a procedural as well as an object-oriented paradigm, there are two options:

Implement the operators as functions

The operator is implemented as a package-level function to operate on one or

The operator is implemented as a package-level function to operate on one or two parameters and return a new object, implemented in the package

dedicated to the objects on which they operate. For example, if we implement a matrix manipulation in a package `matrix`, this would contain the addition of matrices `Add()` and multiplication `Mult()`, which result in a matrix. These would be called on the package name itself so that we could make expressions of the form:

```
m := matrix.Add(m1, matrix.Mult(m2, m3))
```

If we would like to differentiate between the different kinds of matrices (sparse and dense) in these operations, because there is no function overloading, we would have to give them different names, as in:

```
func addSparseToDense (a *sparseMatrix, b *denseMatrix) *denseMatrix
func addDenseToDense (a *denseMatrix, b *denseMatrix) *denseMatrix
func addSparseToSparse (a *sparseMatrix, b *sparseMatrix) *sparseMatrix
```

This is not very elegant, and the best we can do is hide these as private functions and expose a single public function `Add()`. This can operate on any combination of supported parameters by type-testing them in a nested type switch:

```
func Add(a Matrix, b Matrix) Matrix {
  switch a.(type) {
  case sparseMatrix:
    switch b.(type) {
    case sparseMatrix:
      return addSparseToSparse(a.(sparseMatrix), b.(sparseMatrix))
    case denseMatrix:
      return addSparseToDense(a.(sparseMatrix), b.(denseMatrix))
    ...
  default:
    // unsupported arguments
    ...
  }
}
```

However, the more elegant and preferred way is to implement the operators as methods, as it is done everywhere in the standard library.

Implement the operators as methods

Implement the operators as methods

Methods can be differentiated according to their receiver type. Therefore, instead of having to use different function names, we can simply define an `Add()` method for each type:

```
func (a *sparseMatrix) Add(b Matrix) Matrix
func (a *denseMatrix) Add(b Matrix) Matrix
```

Each method returns a new object, which becomes the receiver of the next method call, so we can make chained

```
expressions: m := m1.Mult(m2).Add(m3)
```

This is shorter and clearer. The correct implementation can again be selected at runtime based on a type-switch:

```
func (a *sparseMatrix) Add(b Matrix) Matrix {
  switch b.(type) {
  case sparseMatrix:
    return addSparseToSparse(a.(sparseMatrix), b.(sparseMatrix))
  case denseMatrix:
    return addSparseToDense(a.(sparseMatrix), b.(denseMatrix))
  ...
  default:
    // unsupported arguments
  ...
  }
}
```

Again, this is easier than the nested type switch.

Using an interface

When operating with the same methods on different types, the concept of creating a generalizing interface to implement this polymorphism should come to mind. We could, for example, define the interface `Algebraic`:

```
type Algebraic interface {
  Add(b Algebraic) Algebraic
  Min(b Algebraic) Algebraic
  Mult(b Algebraic) Algebraic
  ...
  Elements()
}
```

and define the methods `Add()`, `Min()`, `Mult()`, ... for our matrix types.

Each type that implements the `Algebraic` interface above will allow for method chaining. Each method implementation should use a type-switch to provide optimized implementations based on the parameter type. Additionally, a default case should be specified, which relies only on the methods in the interface:

```go
func (a *denseMatrix) Add(b Algebraic) Algebraic {
  switch b.(type) {
    case sparseMatrix:
      return addDenseToSparse(a, b.(sparseMatrix))
    default:
      for x in range b.Elements() ...
  ...
}
```

If a generic implementation cannot be implemented using only the methods in the interface, you are probably dealing with classes that are not similar enough, and this operator pattern should be abandoned. For example, it does not make sense to write `a.Add(b)` if `a` is a set, and `b` is a matrix; therefore, it will be difficult to implement a generic `a.Add(b)` in terms of set and matrix operators. In this case, split your package in two and provide separate `AlgebraicSet` and `AlgebraicMatrix` interfaces.

This chapter was like a revision of the previous sections, to help you avoid the common pitfalls and follow the best practices. Meanwhile, the next chapter will bring you an overview of all the chapters you studied before with the purpose of revising the essential concepts, lesson by lesson.