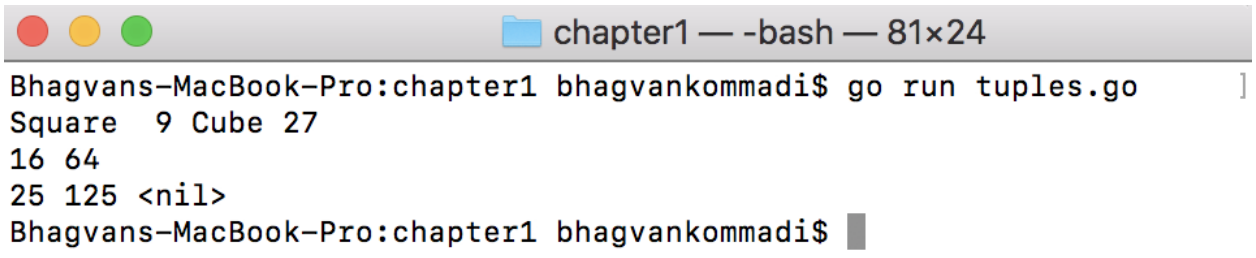
**Lists**

```
import (
"fmt"
"container/list")


var intList list.List
intList.PushBack(11)
intList.PushBack(23)
intList.PushBack(34)

for element := intList.Front(); element != nil; element=element.Next()
{
fmt.Println(element.Value.(int))
}
```

**Tuples**

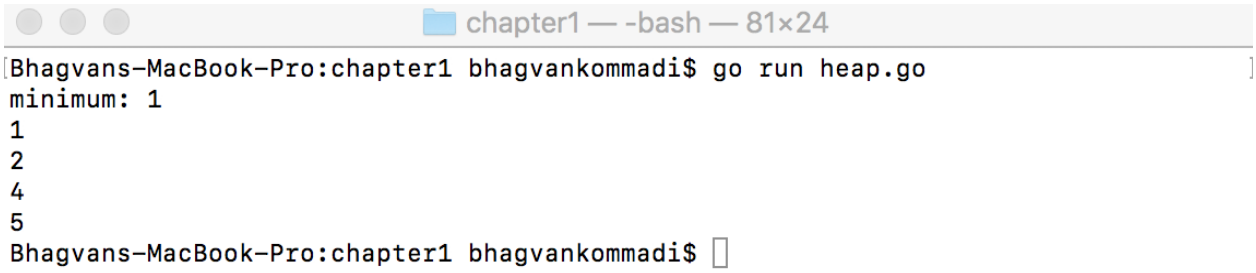
```
import (
    "fmt"
)
//gets the power series of integer a and returns tuple of square of a
// and cube of a
func powerSeries(a int)(int, int) {
    return a * a, a * a * a
}

func main() {
    var square int
    var cube int
    square, cube = powerSeries(3)
    fmt.Println("Square ", square, "Cube", cube)
}
```



**Heaps**

```
// importing fmt package and container/heap
import (
    "container/heap"
    "fmt"
)
// integerHeap a type
type IntegerHeap[] int
    // IntegerHeap method - gets the length of integerHeap
func(iheap IntegerHeap) Len() int {
        return len(iheap)
    }
    // IntegerHeap method - checks if element of i index is less than j index
func(iheap IntegerHeap) Less(i, j int) bool {
        return iheap[i] < iheap[j]
    }
    // IntegerHeap method -swaps the element of i to j index
func(iheap IntegerHeap) Swap(i, j int) {
        iheap[i], iheap[j] = iheap[j],
            iheap[i]
    }
    //IntegerHeap has a Push method that pushes the item with the interface:
    //IntegerHeap method -pushes the item
func(iheap * IntegerHeap) Push(heapintf interface {}) { * iheap = append( * iheap,
heapintf.(int))
    }
    //IntegerHeap method -pops the item from the heap
func(iheap * IntegerHeap) Pop() interface {} {
    var n int
    var x1 int
    var previous IntegerHeap = * iheap
    n = len(previous)
    x1 = previous[n - 1] * iheap = previous[0: n - 1]
    return x1
}
// main method
func main() {
    var intHeap * IntegerHeap = & IntegerHeap {
        1, 4, 5
```

```
    }
    heap.Init(intHeap)
    heap.Push(intHeap, 2)
    fmt.Printf("minimum: %d\n", ( * intHeap)[0])
    for intHeap.Len() > 0 {
        fmt.Printf("%d \n", heap.Pop(intHeap))
    }
}
```

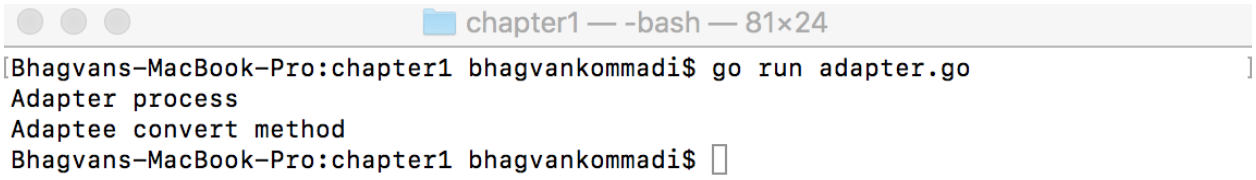



**Structural design patterns**
**Adapter**

Шаблон адаптера предоставляет оболочку с интерфейсом, который требуется клиенту API для связи.

```
// importing fmt package
import (
    "fmt"
)
//IProcess interface
type IProcess interface {
    process()
}
//Adapter struct
type Adapter struct {
    adaptee Adaptee
}
//The Adapter class has a process method that invokes the convert method on adaptee:
//Adapter class method process
func(adapter Adapter) process() {
        fmt.Println("Adapter process")
        adapter.adaptee.convert()
    }
    //Adaptee Struct
```

```
type Adaptee struct {
    adapterType int
}
// Adaptee class method convert
func(adaptee Adaptee) convert() {
        fmt.Println("Adaptee convert method")
    }
    // main method
func main() {
    var processor IProcess = Adapter {}
    processor.process()
}
```




**Bridge**

Мост отделяет реализацию от абстракции.

```
package main
// importing fmt package
import (
    "fmt"
)
//IDrawShape interface
type IDrawShape interface {
    drawShape(x[5] float32, y[5] float32)
}
//DrawShape struct
type DrawShape struct {}
```

Теперь это подставляется в разные типы:
Так
drawShape method

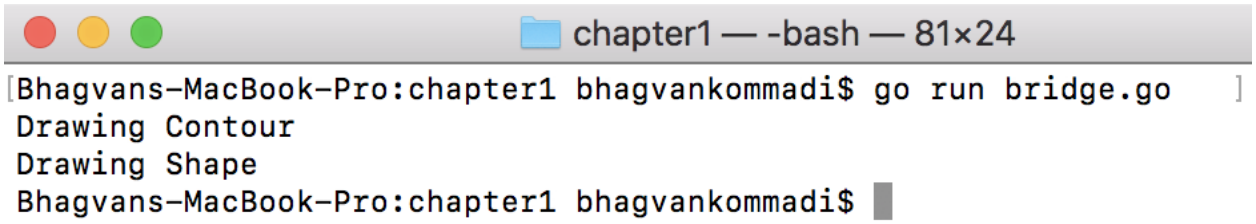
```
// DrawShape struct has method draw Shape with float x and y coordinates
func(drawShape DrawShape) drawShape(x[5] float32, y[5] float32) {
    fmt.Println("Drawing Shape")
}
```

```
//IContour interace
type IContour interface {
drawContour(x[5] float32 ,y[5] float32)
resizeByFactor(factor int)
}
//DrawContour struct
type DrawContour struct {
x[5] float32
y[5] float32
shape DrawShape
factor int
}
```

Или так

drawContour method

```
//DrawContour method drawContour given the coordinates
func(contour DrawContour) drawContour(x[5] float32, y[5] float32) {
        fmt.Println("Drawing Contour")
        contour.shape.drawShape(contour.x, contour.y)
    }
    //DrawContour method resizeByFactor given factor
func(contour DrawContour) resizeByFactor(factor int) {
        contour.factor = factor
    }
    // main method
func main() {
    var x = [5] float32 {
        1, 2, 3, 4, 5
    }
    var y = [5] float32 {
        1, 2, 3, 4, 5
    }
    var contour IContour = DrawContour {
        x, y, DrawShape {}, 2
    }
    contour.drawContour(x, y)
    contour.resizeByFactor(2)
}
```

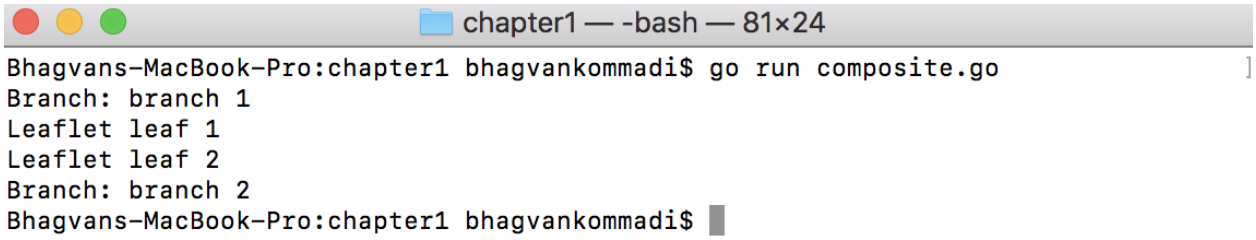



**Composite**

A composite is a group of similar objects in a single object.

```
import (
   "fmt"
)
// IComposite interface
type IComposite interface {
   perform()
}
// Leaflet struct
type Leaflet struct {
   name string
}
// Leaflet class method perform
func(leaf * Leaflet) perform() {
      fmt.Println("Leaflet " + leaf.name)
   }
   // Branch struct
type Branch struct {
   leafs[] Leaflet
   name string
   branches[] Branch
}
//The perform method of the Branch class calls the perform method on branch and
leafs, as seen in the code:
// Branch class method perform
func(branch * Branch) perform() {
      fmt.Println("Branch: " + branch.name)
      for _, leaf: = range branch.leafs {
         leaf.perform()
      }
```

```
        for _, branch: = range branch.branches {
            branch.perform()
        }
    }
    // Branch class method add leaflet
func(branch * Branch) add(leaf Leaflet) {
    branch.leafs = append(branch.leafs, leaf)
}

//As shown in the following code, the addBranch method of the Branch class adds a
new
//branch:
    //Branch class method addBranch branch
    func(branch * Branch) addBranch(newBranch Branch) {
        branch.branches = append(branch.branches, newBranch)
    }
    //Branch class method getLeaflets
func(branch * Branch) getLeaflets()[] Leaflet {
        return branch.leafs
    }
    // main method
func main() {
    var branch = & Branch {
        name: "branch 1"
    }
    var leaf1 = Leaflet {
        name: "leaf 1"
    }
    var leaf2 = Leaflet {
        name: "leaf 2"
    }
    var branch2 = Branch {
        name: "branch 2"
    }
    branch.add(leaf1)
    branch.add(leaf2)
    branch.addBranch(branch2)
    branch.perform()
}
```

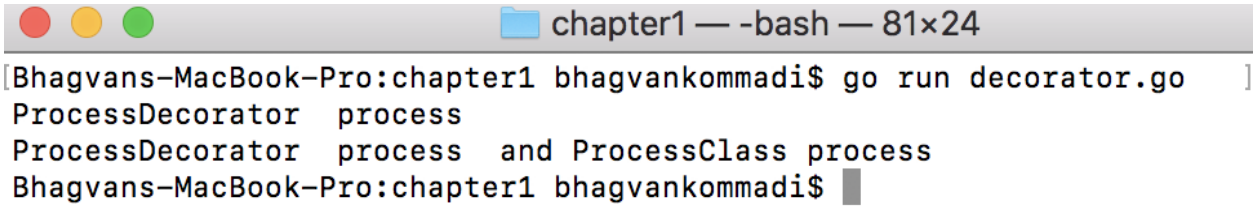



**Decorator**

В сценарии, где обязанности класса удалены или добавлены

```
package main
// importing fmt package
import (
    "fmt"
)
// IProcess Interface
type IProcess interface {
    process()
}
//ProcessClass struct
type ProcessClass struct {}
//ProcessClass method process
func(process * ProcessClass) process() {
        fmt.Println("ProcessClass process")
    }
    //ProcessDecorator struct
type ProcessDecorator struct {
    processInstance * ProcessClass
}
//In the following code, the ProcessDecorator class process method invokes the process
//method on the decorator instance of ProcessClass:
//ProcessDecorator class method process
func(decorator * ProcessDecorator) process() {
        if decorator.processInstance == nil {
            fmt.Println("ProcessDecorator process")
        } else {
            fmt.Printf("ProcessDecorator process and ")
            decorator.processInstance.process()
        }
    }
```

```
    //main method
func main() {
    var process = & ProcessClass {}
    var decorator = & ProcessDecorator {}
    decorator.process()
    decorator.processInstance = process
    decorator.process()
}
```




**Arrays**

```
var arr = [5]int {1,2,4,5,6}
```

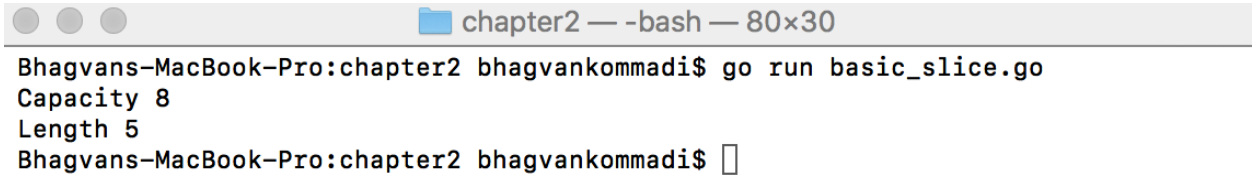
Перебор

```
var i int
for i=0; i< len(arr); i++ {
fmt.Println("printing elements ",arr[i]
}
```

**Slices**

Go Slice is an abstraction over Go Array.

```
var slice = []int{1,3,5,6}
slice = append(slice, 8)
fmt.Println(“Capacity”, cap(slice))
fmt.Println(“Length”, len(slice))
```




**Two-dimensional slices**

```
var TwoDArray [8][8]int
```

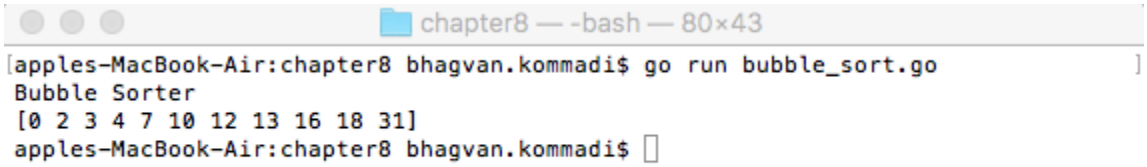
**Maps**

```
var languages = map[int]string {
3: “English”,
4: “French”,
5: “Spanish”
}
```

**Classic Algorithms**
**Sorting**
**Bubble**

```
import (
   "fmt"
)
//bubble Sorter method
func bubbleSorter(integers[11] int) {
        var num int
        num = 11
        var isSwapped bool
        isSwapped = true
        for isSwapped {
            isSwapped = false
            var i int
            for i = 1;
            i < num;
            i++{
                if integers[i - 1] > integers[i] {
                    var temp = integers[i]
                    integers[i] = integers[i - 1]
                    integers[i - 1] = temp
                    isSwapped = true
                }
            }
        }
        fmt.Println(integers)
    }
```

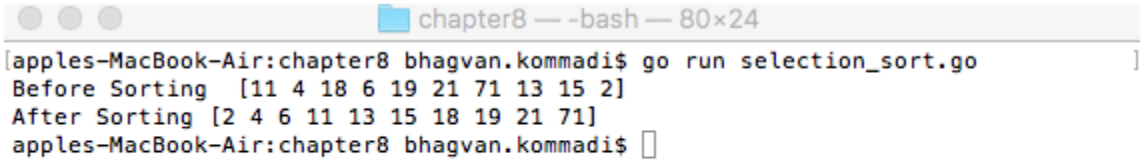
```
    // main method
func main() {
    var integers[11] int = [11] int {
        31, 13, 12, 4, 18, 16, 7, 2, 3, 0, 10
    }
    fmt.Println("Bubble Sorter")
    bubbleSorter(integers)
}
```




**Selection**

```
// importing fmt package
import (
    "fmt"
)
// Selection Sorter method
func SelectionSorter(elements[] int) {
    var i int
    for i = 0; i < len(elements) - 1; i++{
        var min int
        min = i
        var j int
        for j = i + 1;
        j <= len(elements) - 1;
        j++{
            if elements[j] < elements[min] {
                min = j
            }
        }
        swap(elements, i, min)
    }
}

//main method
func main() {
    var elements[] int
```

```
    elements = [] int {
        11, 4, 18, 6, 19, 21, 71, 13, 15, 2
    }
    fmt.Println("Before Sorting ", elements)
    SelectionSorter(elements)
    fmt.Println("After Sorting", elements)
}
```




**Insertion**

```
import (
    "fmt"
    "math/rand"
    "time"
)
// randomSequence method
func randomSequence(num int)[] int {
    var sequence[] int
    sequence = make([] int, num, num)
    rand.Seed(time.Now().UnixNano())
    var i int
    for i = 0;
    i < num;
    i++{
        sequence[i] = rand.Intn(999) - rand.Intn(999)
    }
    return sequence
}
```

**Shell**

```
// importing fmt and bytes package
import (
    "fmt"
```

```
)
// shell sorter method
func ShellSorter(elements[] int) {
    var (
        n = len(elements) intervals = [] int {
            1
        }
        k = 1
    )
    for {
        var interval int
        interval = power(2, k) + 1
        if interval > n - 1 {
            break
        }
        intervals = append([] int {
            interval
        }, intervals...)
        k++
    }
    var interval int
    for _, interval = range intervals {
        var i int
        for i = interval;
        i < n;
        i += interval {
            var j int
            j = i
            for j > 0 {
                if elements[j - interval] > elements[j] {
                    elements[j - interval], elements[j] = elements[j], elements[jinterval]
                }
                j = j - interval
            }
        }
    }
}
```

**Factory method**

```
func NewReservation(vertical, reservationDate string) Reservation {
```

```
    switch (vertical) {
        case "flight":
            return FlightReservationImpl {
                reservationDate,
            }
        case "hotel":
            return HotelReservationImpl {
                reservationDate,
            }
        default:
            return nil
    }
}
```

It can be used as follows:

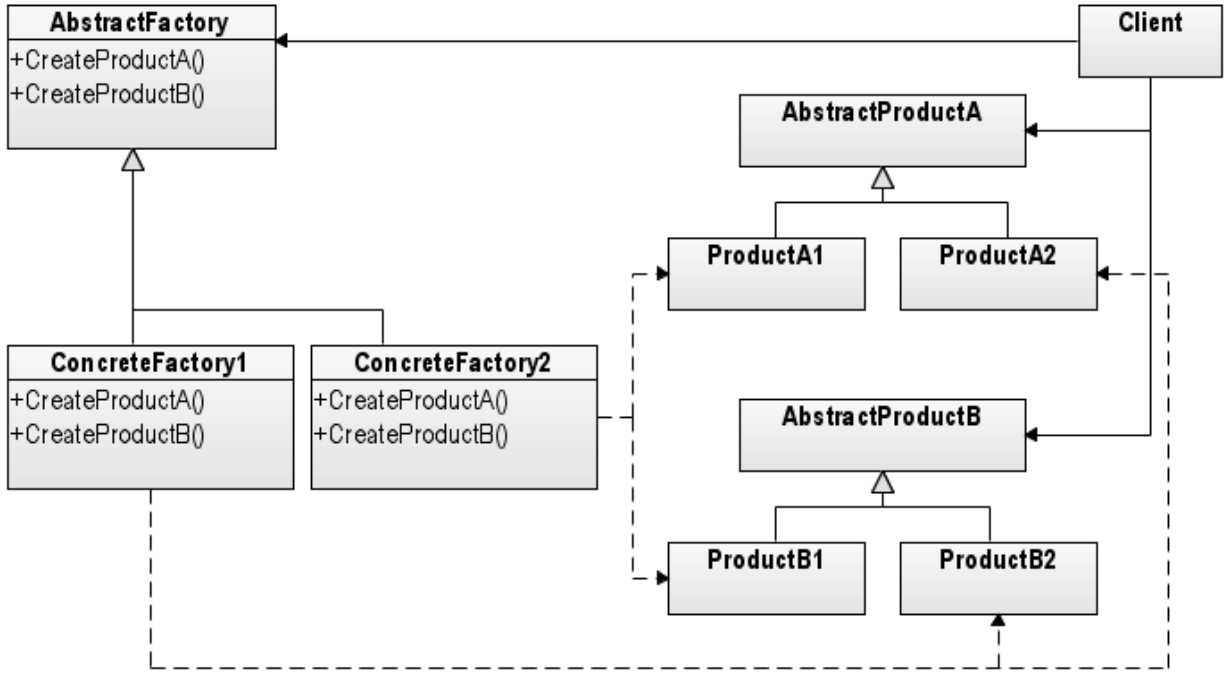
```
    hotelReservation: = NewReservation("hotel", "20180101")
```

**Builder**

```
type ReservationBuilder interface {
    Vertical(string) ReservationBuilder
    ReservationDate(string) ReservationBuilder
    Build() Reservation
}
type reservationBuilder struct {
    vertical string
    rdate string
}
func(r * reservationBuilder) Vertical(v string) ReservationBuilder {
    r.vertical = v
    return r
}
func(r * reservationBuilder) ReservationDate(date string)
ReservationBuilder {
    r.rdate = date
    return r
}
func(r * reservationBuilder) Build() Reservation {
    var builtReservation Reservation
    switch r.vertical {
        case "flight":
```

```
            builtReservation = FlightReservationImpl {
                r.rdate
            }
        case "hotel":
            builtReservation = HotelReservationImpl {
                r.rdate
            }
    }
    return builtReservation
}
func NewReservationBuilder() ReservationBuilder {
    return &reservationBuilder {}
}
```

**Abstract factory**




```
// We have Reservation and Invoice as two generic products
type Reservation interface {}
type Invoice interface {}
type AbstractFactory interface {
    CreateReservation() Reservation
    CreateInvoice() Invoice
}
type HotelFactory struct {}
func(f HotelFactory) CreateReservation() Reservation {
```

```
    return new(HotelReservation)
}
func(f HotelFactory) CreateInvoice() Invoice {
    return new(HotelInvoice)
}
type FlightFactory struct {}
func(f FlightFactory) CreateReservation() Reservation {
    return new(FlightReservation)
}
func(f FlightFactory) CreateInvoice() Invoice {
    return new(FlightReservation)
}
type HotelReservation struct {}
type HotelInvoice struct {}
type FlightReservation struct {}
type FlightInvoice struct {}
func GetFactory(vertical string) AbstractFactory {
    var factory AbstractFactory
    switch vertical {
        case "flight":
            factory = FlightFactory {}
        case "hotel":
            factory = HotelFactory {}
    }
    return factory
}


//The client can use the abstract factory as follows:

hotelFactory: = GetFactory("hotel")
reservation: = hotelFactory.CreateReservation()
invoice: = hotelFactory.CreateInvoice()
```

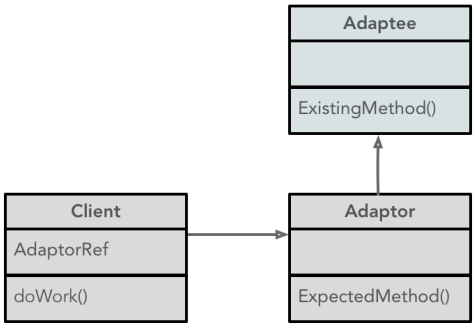
**Singleton**

```
type MyClass struct {
    attrib string
}
func(c * MyClass) SetAttrib(val string) {
```

```
    c.attrib = val
}
func(c * MyClass) GetAttrib() string {
    return c.attrib
}
var (
    once sync.Once instance * MyClass
)
func GetMyClass() * MyClass {
        once.Do(func() {
            instance = & MyClass {
                "first"
            }
        })
        return instance
    }
    //it can be used as follows:
a: = GetMyClass()
a.SetAttrib("second")
fmt.Println(a.GetAttrib()) // will print second
b: = GetMyClass()
fmt.Println(b.GetAttrib()) // will also print second
```

**Adaptor**

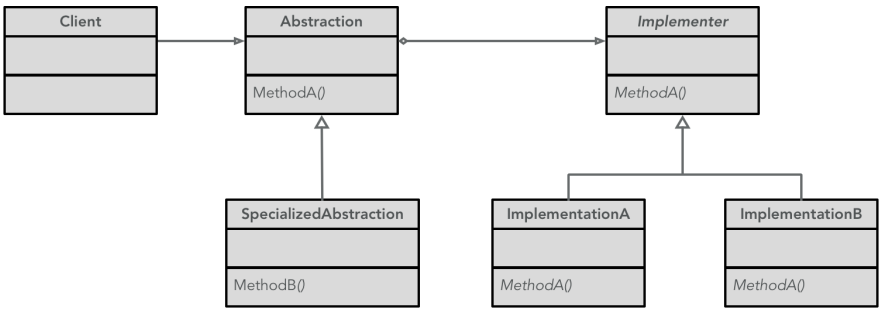
Many times when you code, you come across situations where you have a new requirement, and a component that almost meets that requirement.




```
type Adaptee struct {}
func(a * Adaptee) ExistingMethod() {
    fmt.Println("using existing method")
}
```

```
type Adapter struct {
   adaptee * Adaptee
}
func NewAdapter() * Adapter {
   return &Adapter {
      new(Adaptee)
   }
}
func(a * Adapter) ExpectedMethod() {
      fmt.Println("doing some work")
      a.adaptee.ExistingMethod()
   }
   //And the client doWork() will look as simple as this:
adaptor: = NewAdapter()
adaptor.ExpectedMethod()
```

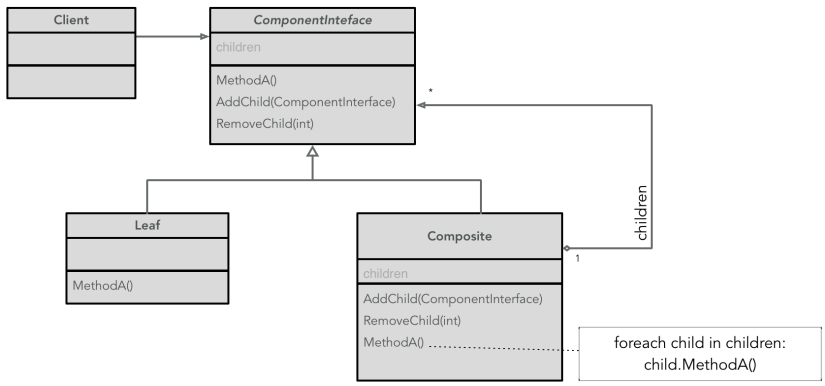
**Bridge**




```
type Reservation struct {
   sellerRef Seller // this is the implementer reference
}
func(r Reservation) Cancel() {
   r.sellerRef.CancelReservation(10) // charge $10 as cancellation feed
}
type PremiumReservation struct {
   Reservation
}
func(r PremiumReservation) Cancel() {
      r.sellerRef.CancelReservation(0) // no charges
   }
   // This is the interface for all Sellers
type Seller interface {
   CancelReservation(charge float64)
```

```
}
type InstitutionSeller struct {}
func(s InstitutionSeller) CancelReservation(charge float64) {
   fmt.Println("InstitutionSeller CancelReservation charge =", charge)
}
type SmallScaleSeller struct {}
func(s SmallScaleSeller) CancelReservation(charge float64) {
      fmt.Println("SmallScaleSeller CancelReservation charge =", charge)
   }
   //Its usage is as follows:
res: = Reservation {
   InstitutionSeller {}
}
res.Cancel() // will print 10 as the cancellation charge
premiumRes: = PremiumReservation {
   Reservation {
      SmallScaleSeller {}
   }
}
premiumRes.Cancel() // will print 0 as the cancellation charge
```

**Composite**




```
type InterfaceX interface {
   MethodA()
   AddChild(InterfaceX)
}
type Composite struct {
   children[] InterfaceX
}
```

```
func(c * Composite) MethodA() {
    if len(c.children) == 0 {
        fmt.Println("I'm a leaf ")
        return
    }
    fmt.Println("I'm a composite ")
    for _, child: = range c.children {
        child.MethodA()
    }
}
func(c * Composite) AddChild(child InterfaceX) {
        c.children = append(c.children, child)
    }
    //The usage is shown as follows:
func test() {
    var parent InterfaceX
    parent = & Composite {}
    parent.MethodA() // only a leaf and the print will confirm!
    var child Composite
    parent.AddChild( & child)
    parent.MethodA() // one composite, one leaf
}
```

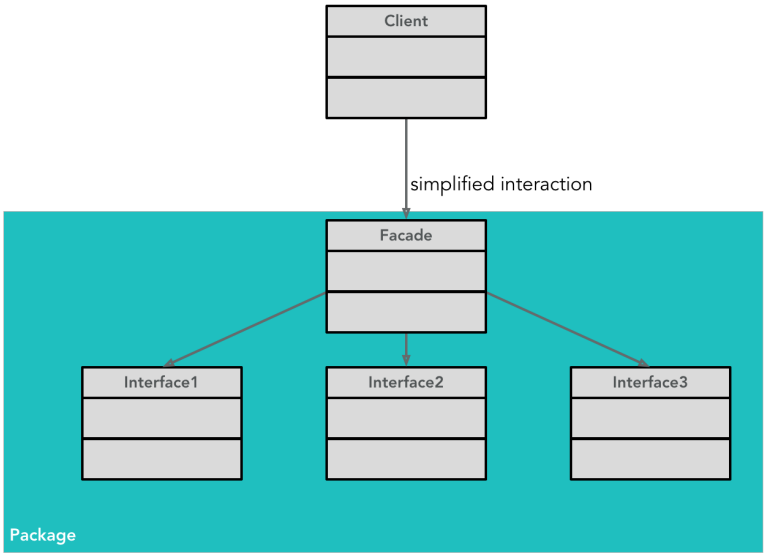
**Decorator**

The decorator pattern allows the extension of a function of an existing object dynamically without the need to alter the original object.

```
type Function func(float64) float64
    // the decorator function
func ProfileDecorator(fn Function) Function {
    return func(params float64) float64 {
        start: = time.Now()
        result: = fn(params)
        elapsed: = time.Now().Sub(start)
        fmt.Println("Function completed with time: ", elapsed)
        return result
    }
}
func client() {
```

```
    decoratedSquqreRoot: = ProfileDecorator(SquareRoot)
    fmt.Println(decoratedSquqreRoot(16))
}
```

**Facade**

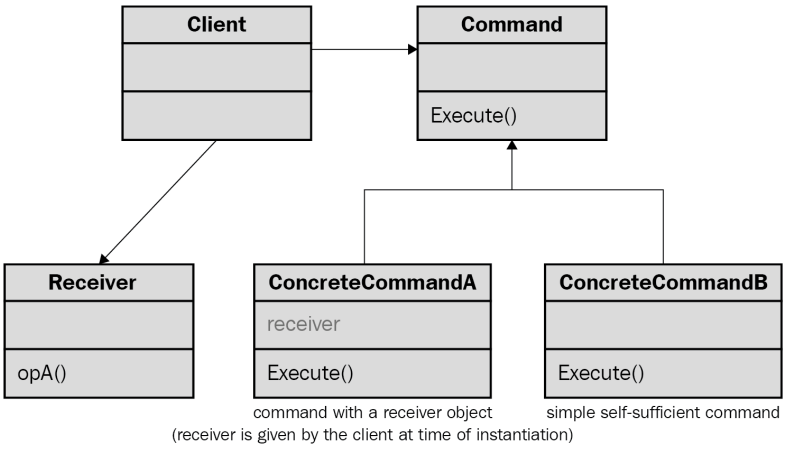



**Proxy**

```
// Proxy
type HotelBoutiqueProxy struct {
    subject * HotelBoutique
}
func(p * HotelBoutiqueProxy) Book() {
        if p.subject == nil {
            p.subject = new(HotelBoutique)
        }
        fmt.Println("Proxy Delegating Booking call")
            // The API call will happen here
            // For example sake a simple delegation is implemented
        p.subject.Book()
    }
    // Dummy Subject
type HotelBoutique struct {}
func(s * HotelBoutique) Book() {
    fmt.Println("Booking done on external site")
}
```

**Command**

The command pattern is a behavioral design pattern in which an object is used to represent a request (or actions) and encapsulate all information needed to process the same.




```
// The Command
type Report interface {
    Execute()
}
// The Concrete Commands
type ConcreteReportA struct {
    receiver * Receiver
}
func(c * ConcreteReportA) Execute() {
    c.receiver.Action("ReportA")
}
type ConcreteReportB struct {
    receiver * Receiver
}
func(c * ConcreteReportB) Execute() {
        c.receiver.Action("ReportB")
    }
    // The Receiver
type Receiver struct {}
func(r * Receiver) Action(msg string) {
        fmt.Println(msg)
    }
    // Invoker
```

```
type Invoker struct {
   repository[] Report
}
func(i * Invoker) Schedule(cmd Report) {
   i.repository = append(i.repository, cmd)
}
func(i * Invoker) Run() {
       for _, cmd: = range i.repository {
          cmd.Execute()
       }
   }
   //The client code will look as follows:
func client() {
   receiver: = new(Receiver)
   ReportA: = & ConcreteReportA {
      receiver
   }
   ReportB: = & ConcreteReportB {
      receiver
   }
   invoker: = new(Invoker)
   invoker.Schedule(ReportA)
   invoker.Run()
   invoker.Schedule(ReportB)
   invoker.Run()
}
```

**Chain of Responsibility**

Во многих случаях команды, описанные ранее, могут нуждаться в обработке, так что мы хотим приемник выполнять работу только тогда, когда он в состоянии это сделать, а затем передать команду кто-то следующий в цепочке.

```
type ChainedReceiver struct {
   canHandle string
   next * ChainedReceiver
}
func(r * ChainedReceiver) SetNext(next * ChainedReceiver) {
   r.next = next
}
```

```
func(r * ChainedReceiver) Finish() error {
    fmt.Println(r.canHandle, " Receiver Finishing")
    return nil
}
func(r * ChainedReceiver) Handle(what string) error {
    // Check if this receiver can handle the command
    if what == r.canHandle {
        // handle the command here
        return r.Finish()
    } else if r.next != nil {
        // delegate to the next guy
        return r.next.Handle(what)
    } else {
        fmt.Println("No Receiver could handle the request!")
        return errors.New("No Receiver to Handle")
    }
}
```

**Mediator**

Шаблон посредника добавляет сторонний объект (называемый посредником) для управления взаимодействием между двумя объектами (коллегами).

```
// The Mediator interface
type Mediator interface {
    AddColleague(colleague Colleague)
}
// The Colleague interface
type Colleague interface {
    setMediator(mediator Mediator)
}
// Concrete Colleague 1 - uses state as string
type Colleague1 struct {
    mediator Mediator
    state string
}
func(c * Colleague1) SetMediator(mediator Mediator) {
    c.mediator = mediator
}
func(c * Colleague1) SetState(state string) {
```

```
    fmt.Println("Colleague1: setting state: ", state)
    c.state = state
}
func(c * Colleague1) GetState() string {
        return c.state
    }
    // Concrete Colleague 2 - uses state as int
type Colleague2 struct {
    mediator Mediator
    state int
}
func(c * Colleague2) SetState(state int) {
    fmt.Println("Colleague2: setting state: ", state)
    c.state = state
}
func(c * Colleague2) GetState() int {
    return c.state
}
func(c * Colleague2) SetMediator(mediator Mediator) {
        c.mediator = mediator
    }
    // Concrete Mediator
type ConcreteMediator struct {
    c1 Colleague1
    c2 Colleague2
}
func(m * ConcreteMediator) SetColleagueC1(c1 Colleague1) {
    m.c1 = c1
}
func(m * ConcreteMediator) SetColleagueC2(c2 Colleague2) {
    m.c2 = c2
}
func(m * ConcreteMediator) SetState(s string) {
        m.c1.SetState(s)
        stateAsString, err: = strconv.Atoi(s)
        if err == nil {
            m.c2.SetState(stateAsString)
            fmt.Println("Mediator set status for both colleagues")
        }
    }
```

```
   //Client code is given here:
c1: = Colleague1 {}
c2: = Colleague2 {}
   // initialize mediator with colleagues
mediator: = ConcreteMediator {}
mediator.SetColleagueC1(c1)
mediator.SetColleagueC2(c2)
   // mediator keeps colleagues in sync
mediator.SetState("10")
```

**Memento**

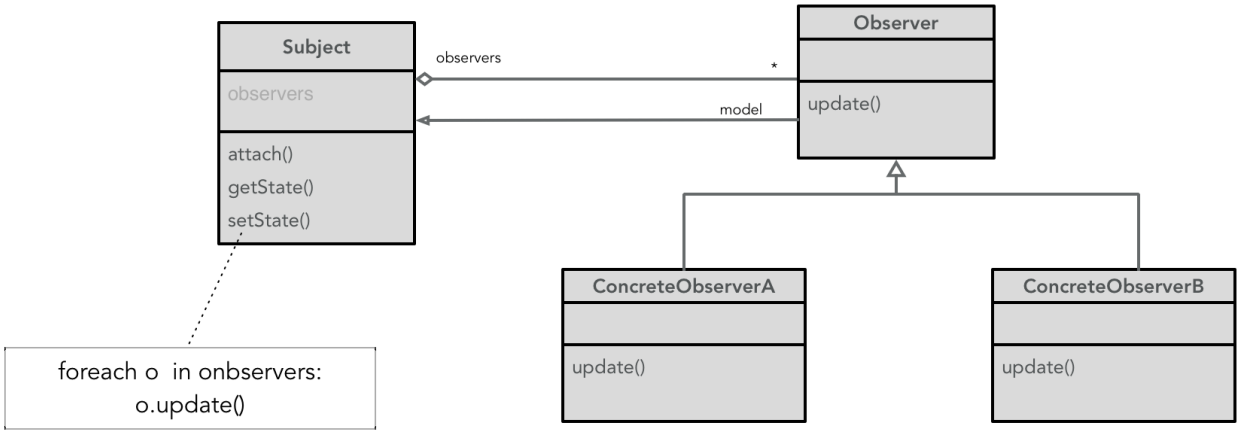
Паттерн «memento» предназначен для захвата и сохранения текущего состояния объекта таким образом, чтобы он может быть восстановлен позже в гладкой манере.

```
// Originator
type Originator struct {
   state string
}
func(o * Originator) GetState() string {
   return o.state
}
func(o * Originator) SetState(state string) {
   fmt.Println("Setting state to " + state)
   o.state = state
}
func(o * Originator) GetMemento() Memento {
   // externalize state to Momemto objct
   return Memento {
      o.state
   }
}
func(o * Originator) Restore(memento Memento) {
      // restore state
      o.state = memento.GetState()
   }
   // Momento
type Memento struct {
   serializedState string
```

```
}
func(m * Memento) GetState() string {
        return m.serializedState
    }
    // caretaker
func Caretaker() {
    // assume that A is the original state of the Orginator
    theOriginator: = Originator {
        "A"
    }
    theOriginator.SetState("A")
    fmt.Println("theOriginator state = ", theOriginator.GetState())
    // before mutating, get an momemto
    theMomemto: = theOriginator.GetMemento()
        // mutate to unclean
    theOriginator.SetState("unclean")
    fmt.Println("theOriginator state = ", theOriginator.GetState())
    // rollback
    theOriginator.Restore(theMomemto)
    fmt.Println("RESTORED: theOriginator state = ",
        theOriginator.GetState())
}
```

**Observer**

Во многих ситуациях есть один объект (субъект) с состоянием и несколько других (наблюдателей) которые заинтересованы в этом состоянии.



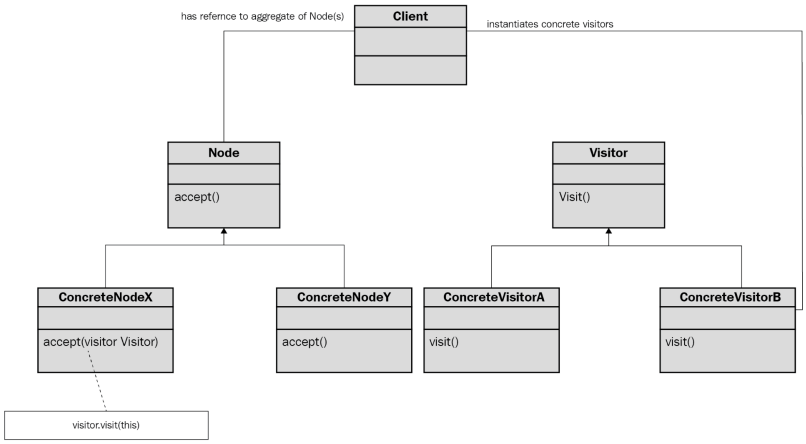

// The Subject

```
type Subject struct {
    observers[] Observer
    state string
}
func(s * Subject) Attach(observer Observer) {
    s.observers = append(s.observers, observer)
}
func(s * Subject) SetState(newState string) {
    s.state = newState
    for _, o: = range(s.observers) {
        o.Update()
    }
}
func(s * Subject) GetState() string {
        return s.state
    }
    // The Observer Inteface
type Observer interface {
    Update()
}
// Concrete Observer A
type ConcreteObserverA struct {
    model * Subject
    viewState string
}
func(ca * ConcreteObserverA) Update() {
    ca.viewState = ca.model.GetState()
    fmt.Println("ConcreteObserverA: updated view state to ", ca.viewState)
}
func(ca * ConcreteObserverA) SetModel(s * Subject) {
        ca.model = s
    }
    //And the client will call as follows:
func client() {
    // create Subject
    s: = Subject {}
        // create concrete observer
    ca: = & ConcreteObserverA {}
    ca.SetModel( & s) // set Model
    // Attach the observer
```

```
    s.Attach(ca)
    s.SetState("s1")
}
```

**Visitor**

Many times, we want to do different things with elements (nodes) of an aggregate (array, tree, list, and so on).




```
// the Node interface
type Node interface {
    Accept(Visitor)
}
type ConcreteNodeX struct {}
func(n ConcreteNodeX) Accept(visitor Visitor) {
    visitor.Visit(n)
}
type ConcreteNodeY struct {}
func(n ConcreteNodeY) Accept(visitor Visitor) {
        // do something NodeY-specific before visiting
        fmt.Println("ConcreteNodeY being visited !")
        visitor.Visit(n)
    }
    // the Vistor interface
type Visitor interface {
    Visit(Node)
}
// and an implementation
```

```
type ConcreteVisitor struct {}
func(v ConcreteVisitor) Visit(node Node) {
   fmt.Println("doing something concrete")
      // since there is no function overloading..
      // this is one way of checking the concrete node type
   switch node.(type) {
      case ConcreteNodeX:
         fmt.Println("on Node ")
      case ConcreteNodeY:
         fmt.Println("on Node Y")
   }
}
```

The client code will look as follows:

```
   func main() {
      // a simple aggregate
      aggregate: = [] Node {
            ConcreteNodeX {}, ConcreteNodeY {},
         }
         // a vistor
      visitor: = new(ConcreteVisitor)
         // iterate and visit
      for _,
      node: = range(aggregate) {
         node.Accept(visitor)
      }
   }
```

**Strategy**

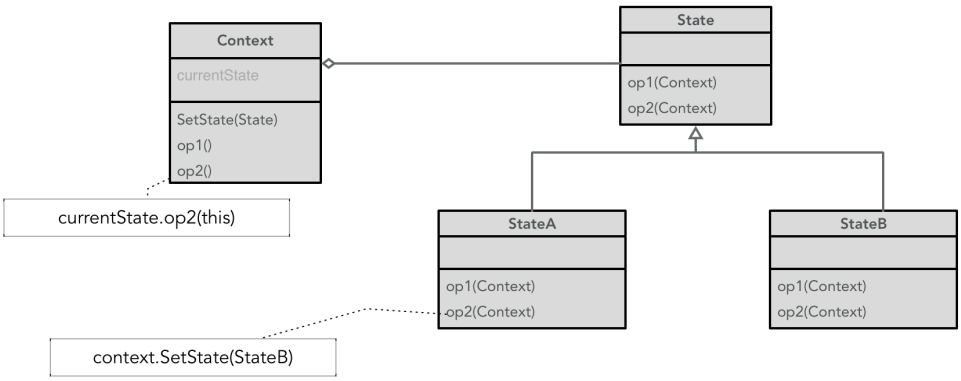
The goal of the strategy pattern is simple: allow the user to change the algorithm used without changing the rest of the code. Here, the developer focuses on the inputs and outputs of a generic algorithm and implements this as a method for an interface.

```
type Strategy interface {
   FindBreadth([] int) int // the algorithm
}
// A O(nlgn) implementation
type NaiveAlgo struct {}
func(n * NaiveAlgo) FindBreadth(set[] int) int {
      sort.Ints(set)
```

```
        return set[len(set) - 1] - set[0]
    }
    // A O(n) implementation
type FastAlgo struct {}
func(n * FastAlgo) FindBreadth(set[] int) int {
        min: = math.MaxInt32
        max: = math.MinInt64
        for _,
        x: = range(set) {
            if x < min {
                min = x
            }
            if x > max {
                max = x
            }
        }
        return max - min
    }
    // The client is ignorant of the algorithm
func client(s Strategy) int {
    a: = [] int {
        -1, 10, 3, 1
    }
    return s.FindBreadth(a)
}
```

**State**

Most real-life objects are stateful, and they change their behavior according to the state.



```
// The State Inteface with the polymorphic methods
type State interface {
    Op1( * Context)
    Op2( * Context)
}
// The Context class
type Context struct {
    state State
}
func(c * Context) Op1() {
    c.state.Op1(c)
}
func(c * Context) Op2() {
    c.state.Op2(c)
}
func(c * Context) SetState(state State) {
    c.state = state
}
func NewContext() * Context {
        c: = new(Context)
        c.SetState(new(StateA)) // Initial State
        return c
    }
    //The client is abstracted from all the state changes as shown here:
func client() {
    context: = NewContext()
        // state operations
    context.Op1()
    context.Op2() // <- This changes state to State 2
    context.Op1()
    context.Op2() // <- This changes state back to State 1
}
```