Если вы видите что-то необычное, просто сообщите мне.

?????????????? ???????? (Behavioral)

Поведенческие паттерны делятся на два типа:

1. Паттерны уровня класса
2. Паттерны уровня объекта.

Паттерны уровня класса описывают взаимодействия между классами и их подклассами. Такие отношения выражаются путем наследования и реализации классов. Тут базовый класс определяет интерфейс, а подклассы - реализацию.

Паттерны уровня объекта описывают взаимодействия между объектами. Такие отношения выражаются связями - ассоциацией, агрегацией и композицией. Тут структуры строятся путем объединения объектов некоторых классов.

Ассоциация - отношение, когда объекты двух классов могут ссылаться один на другой. Например, свойство класса содержит экземпляр другого класса.

Агрегация – частная форма ассоциации. Агрегация применяется, когда один объект должен быть контейнером для других объектов и время существования этих объектов никак не зависит от времени существования объекта контейнера. Вообщем, если контейнер будет уничтожен, то входящие в него объекты не пострадают. Например, мы создали объект, а потом передали его в объект контейнер, каким-либо образом, можно в метод объекта контейнера передать или присвоить сразу свойству контейнера извне. Значит при удалении контейнера мы ни как не затронем наш созданный объект, который может взаимодействовать и с другими контейнерами.

Композиция – Тоже самое, что и агрегация, но составные объекты не могут существовать отдельно от объекта контейнера и если контейнер будет уничтожен, то всё его содержимое будет уничтожено тоже. Например, мы создали объект в методе объекта контейнера и присвоили его свойству объекта контейнера. Из вне, о нашем созданном объекте никто не знает, значит, при удалении контейнера, созданный объект убудет удален так же, т.к. на него нет ссылки извне.

К паттернам уровня класса относится только «Шаблонный метод».

Поведенческие паттерны описывают взаимодействие объектов и классов между собой и пытаются добиться наименьшей степени связанности компонентов системы друг с другом делая систему более гибкой.

* [Цепочка ответственности (Chain Of Responsibility)](ChainOfResponsibility)
* [Команда (Command)](Command)
* [Итератор (Iterator)](Iterator)
* [Посредник (Mediator)](Mediator)
* [Хранитель (Memento)](Memento)
* [Наблюдатель (Observer)](Observer)
* [Состояние (State)](State)
* [Стратегия (Strategy)](Strategy)
* [Шаблонный метод (Template Method)](TemplateMethod)
* [Посетитель (Visitor)](Visitor)

- Цепочка ответственности (Chain Of Responsibility)
- Команда (Command)
- Итератор (Iterator)
- Посредник (Mediator)
- Хранитель (Memento)
- Наблюдатель (Observer)
- Состояние (State)
- Стратегия (Strategy)
- Шаблонный метод (Template Method)
- Посетитель (Visitor)

??????? ??????????????? (Chain Of Responsibility)

Паттерн Chain Of Responsibility относится к поведенческим паттернам уровня объекта.

Паттерн Chain Of Responsibility позволяет избежать привязки объекта-отправителя запроса к объекту-получателю запроса, при этом давая шанс обработать этот запрос нескольким объектам. Получатели связываются в цепочку, и запрос передается по цепочке, пока не будет обработан каким-то объектом.

По сути это цепочка обработчиков, которые по очереди получают запрос, а затем решают, обрабатывать его или нет. Если запрос не обработан, то он передается дальше по цепочке. Если же он обработан, то паттерн сам решает передавать его дальше или нет. Если запрос не обработан ни одним обработчиком, то он просто теряется.

Требуется для реализации:

1. Базовый абстрактный класс Handler, описывающий интерфейс обработчиков в цепочки;
2. Класс ConcreteHandlerA, реализующий конкретный обработчик A;
3. Класс ConcreteHandlerB, реализующий конкретный обработчик B;
4. Класс ConcreteHandlerC, реализующий конкретный обработчик C;

Обратите внимание, что вместо хранения ссылок на всех кандидатов-получателей запроса, каждый отправитель хранит единственную ссылку на начало цепочки, а каждый получатель имеет единственную ссылку на своего преемника - последующий элемент в цепочке.

> “ В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

Код

```
// chain_of_responsibility.go
// Package chain_of_responsibility is an example of the Chain Of Responsibility Pattern.
package chain_of_responsibility

// Handler provides a handler interface.
type Handler interface {
	SendRequest(message int) string
}

// ConcreteHandlerA implements handler "A".
type ConcreteHandlerA struct {
	next Handler
}

// SendRequest implementation.
func (h *ConcreteHandlerA) SendRequest(message int) (result string) {
	if message == 1 {
		result = "Im handler 1"
	} else if h.next != nil {
		result = h.next.SendRequest(message)
	}
	return
}

// ConcreteHandlerB implements handler "B".
type ConcreteHandlerB struct {
	next Handler
}

// SendRequest implementation.
func (h *ConcreteHandlerB) SendRequest(message int) (result string) {
	if message == 2 {
		result = "Im handler 2"
	} else if h.next != nil {
		result = h.next.SendRequest(message)
	}
	return
}
```

```
// ConcreteHandlerC implements handler "C".
type ConcreteHandlerC struct {
	next Handler
}

// SendRequest implementation.
func (h *ConcreteHandlerC) SendRequest(message int) (result string) {
	if message == 3 {
		result = "Im handler 3"
	} else if h.next != nil {
		result = h.next.SendRequest(message)
	}
	return
}
```

```
//chain_of_responsibility_test.go
package chain_of_responsibility

import (
	"testing"
)

func TestChainOfResponsibility(t *testing.T) {
	expect := "Im handler 2"
	handlers := &ConcreteHandlerA{
		next: &ConcreteHandlerB{
			next: &ConcreteHandlerC{},
		},
	}
	result := handlers.SendRequest(2)
	if result != expect {
		t.Errorf("Expect result to equal %s, but %s.\n", expect, result)
	}
}
```

??????? (Command)

Паттерн Command относится к поведенческим паттернам уровня объекта.

Паттерн Command позволяет представить запрос в виде объекта. Из этого следует, что команда - это объект. Такие запросы, например, можно ставить в очередь, отменять или возобновлять.

В этом паттерне мы оперируем следующими понятиями: Command - запрос в виде объекта на выполнение; Receiver - объект-получатель запроса, который будет обрабатывать нашу команду; Invoker - объект-инициатор запроса.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду.

Требуется для реализации:

1. Базовый абстрактный класс Command описывающий интерфейс команды;
2. Класс ConcreteCommand, реализующий команду;
3. Класс Invoker, реализующий инициатора, записывающий команду и провоцирующий её выполнение;
4. Класс Receiver, реализующий получателя и имеющий набор действий, которые команда можем запрашивать;

Invoker умеет складывать команды в стопку и инициировать их выполнение по какому-то событию. Обратившись к Invoker можно отменить команду, пока та не выполнена.

ConcreteCommand содержит в себе запросы к Receiver, которые тот должен выполнять. В свою очередь Receiver содержит только набор действий (Actions), которые выполняются при обращении к ним из ConcreteCommand.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//command.go
// Package command is an example of the Command Pattern.
package command

// Command provides a command interface.
type Command interface {
	Execute() string
}

// ToggleOnCommand implements the Command interface.
type ToggleOnCommand struct {
	receiver *Receiver
}

// Execute command.
func (c *ToggleOnCommand) Execute() string {
	return c.receiver.ToggleOn()
}

// ToggleOffCommand implements the Command interface.
type ToggleOffCommand struct {
	receiver *Receiver
}

// Execute command.
func (c *ToggleOffCommand) Execute() string {
	return c.receiver.ToggleOff()
}

// Receiver implementation.
type Receiver struct {
}

// ToggleOn implementation.
func (r *Receiver) ToggleOn() string {
	return "Toggle On"
}

// ToggleOff implementation.
func (r *Receiver) ToggleOff() string {
```

```
	return "Toggle Off"
}

// Invoker implementation.
type Invoker struct {
	commands []Command
}

// StoreCommand adds command.
func (i *Invoker) StoreCommand(command Command) {
	i.commands = append(i.commands, command)
}

// UnStoreCommand removes command.
func (i *Invoker) UnStoreCommand() {
	if len(i.commands) != 0 {
		i.commands = i.commands[:len(i.commands)-1]
	}
}

// Execute all commands.
func (i *Invoker) Execute() string {
	var result string
	for _, command := range i.commands {
		result += command.Execute() + "\n"
	}
	return result
}
```

```
//command_test.go
package command

import (
	"testing"
)

func TestCommand(t *testing.T) {

	expect := "Toggle On\n" +
		"Toggle Off\n"
```

```
    invoker := &Invoker{}
    receiver := &Receiver{}

    invoker.StoreCommand(&ToggleOnCommand{receiver: receiver})
    invoker.StoreCommand(&ToggleOffCommand{receiver: receiver})

    result := invoker.Execute()

    if result != expect {
        t.Errorf("Expect result to equal %s, but %s.\n", expect, result)
    }
}
```

???????? (Iterator)

Паттерн Iterator относится к поведенческим паттернам уровня объекта.

Паттерн Iterator предоставляет механизм обхода коллекций объектов не раскрывая их внутреннего представления.

Зачастую этот паттерн используется вместо массива объектов, чтобы не только предоставить доступ к элементам, но и наделить некоторой логикой.

Iterator представляет собой общий интерфейс, позволяющий реализовать произвольную логику итераций.

Требуется для реализации:

1. Интерфейс Iterator описывающий набор методов для доступа к коллекции;
2. Класс ConcreteIterator, реализующий интерфейс Iterator. Следит за позицией текущего элемента при переборе коллекции (Aggregate).;
3. Интерфейс Aggregate описывающий набор методов коллекции объектов;
4. Класс ConcreteAggregate, реализующий интерфейс Aggregate и хранящий в себе элементы коллекции.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//iterator.go
// Package iterator is an example of the Iterator Pattern.
package iterator

// Iterator provides a iterator interface.
type Iterator interface {
	Index() int
	Value() interface{}
	Has() bool
```

```
	Next()
	Prev()
	Reset()
	End()
}

// Aggregate provides a collection interface.
type Aggregate interface {
	Iterator() Iterator
}

// BookIterator implements the Iterator interface.
type BookIterator struct {
	shelf    *BookShelf
	index    int
	internal int
}

// Index returns current index
func (i *BookIterator) Index() int {
	return i.index
}

// Value returns current value
func (i *BookIterator) Value() interface{} {
	return i.shelf.Books[i.index]
}

// Has implementation.
func (i *BookIterator) Has() bool {
	if i.internal < 0 || i.internal >= len(i.shelf.Books) {
		return false
	}
	return true
}

// Next goes to the next item.
func (i *BookIterator) Next() {
	i.internal++
	if i.Has() {
```

```
		i.index++
	}
}

// Prev goes to the previous item.
func (i *BookIterator) Prev() {
	i.internal--
	if i.Has() {
		i.index--
	}
}

// Reset resets iterator.
func (i *BookIterator) Reset() {
	i.index = 0
	i.internal = 0
}

// End goes to the last item.
func (i *BookIterator) End() {
	i.index = len(i.shelf.Books) - 1
	i.internal = i.index
}

// BookShelf implements the Aggregate interface.
type BookShelf struct {
	Books []*Book
}

// Iterator creates and returns the iterator over the collection.
func (b *BookShelf) Iterator() Iterator {
	return &BookIterator{shelf: b}
}

// Add adds an item to the collection.
func (b *BookShelf) Add(book *Book) {
	b.Books = append(b.Books, book)
}

// Book implements a item of the collection.
```

```
type Book struct {
    Name string
}
```

```
//iterator_test.go
package iterator

import (
    "testing"
)

func TestIterator(t *testing.T) {

    shelf := new(BookShelf)

    books := []string{"A", "B", "C", "D", "E", "F"}

    for _, book := range books {
        shelf.Add(&Book{Name: book})
    }

    for iterator := shelf.Iterator(); iterator.Has(); iterator.Next() {
        index, value := iterator.Index(), iterator.Value().(*Book)
        if value.Name != books[index] {
            t.Errorf("Expect Book.Name to %s, but %s", books[index], value.Name)
        }
    }
}
```

????????? (Mediator)

Паттерн Mediator относится к поведенческим паттернам уровня объекта.

Паттерн Mediator предоставляет объект-посредник, скрывающий способ взаимодействия множества других объектов-коллег. Mediator делает систему слабо связанной, избавляя объекты от необходимости ссылаться друг на друга, что позволяет изменять взаимодействие между ними независимо.

Например, у нас есть посредник между заводом производства хлебобулочных изделий, фермером и магазином сбыта. Посредник избавляет фермера от взаимодействия с заводом, который использует его сырье, а завод от взаимодействия с магазином, в который поступает продукция для сбыта.

Требуется для реализации:

1. Интерфейс Mediator - посредник описывающий организацию процесса по обмену информацией между объектами типа Colleague;
2. Класс ConcreteMediator, реализующий интерфейс Mediator;
3. Базовый абстрактный класс Colleague - коллега описывающий организацию процесса взаимодействия объектов-коллег с объектом типа Mediator;
4. Класс ConcreteColleague, реализующий интерфейс Colleague. Каждый объект-коллега знает только об объекте-медиаторе. Все объекты-коллеги обмениваются информацией только через посредника.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//mediator.go
// Package mediator is an example of the Mediator Pattern.
package mediator

// Mediator provides a mediator interface.
```

```
type Mediator interface {
	Notify(msg string)
}

// Тип ConcreteMediator, реализует посредника
type ConcreteMediator struct {
	*Farmer
	*Cannery
	*Shop
}

// Notify implementation.
func (m *ConcreteMediator) Notify(msg string) {
	if msg == "Farmer: Tomato complete..." {
		m.Cannery.AddMoney(-15000.00)
		m.Farmer.AddMoney(15000.00)
		m.Cannery.MakeKetchup(m.Farmer.GetTomato())
	} else if msg == "Cannery: Ketchup complete..." {
		m.Shop.AddMoney(-30000.00)
		m.Cannery.AddMoney(30000.00)
		m.Shop.SellKetchup(m.Cannery.GetKetchup())
	}
}

// ConnectColleagues connects all colleagues.
func ConnectColleagues(farmer *Farmer, cannery *Cannery, shop *Shop) {
	mediator := &ConcreteMediator{
		Farmer:  farmer,
		Cannery: cannery,
		Shop:    shop,
	}

	mediator.Farmer.SetMediator(mediator)
	mediator.Cannery.SetMediator(mediator)
	mediator.Shop.SetMediator(mediator)
}

// Farmer implements a Farmer colleague
type Farmer struct {
	mediator Mediator
```

```
	tomato   int
	money    float64
}

// SetMediator sets mediator.
func (f *Farmer) SetMediator(mediator Mediator) {
	f.mediator = mediator
}

// AddMoney adds money.
func (f *Farmer) AddMoney(m float64) {
	f.money += m
}

// GrowTomato implementation.
func (f *Farmer) GrowTomato(tomato int) {
	f.tomato = tomato
	f.money -= 7500.00
	f.mediator.Notify("Farmer: Tomato complete...")
}

// GetTomato returns tomatos.
func (f *Farmer) GetTomato() int {
	return f.tomato
}

// Cannery implements a Cannery colleague.
type Cannery struct {
	mediator Mediator
	ketchup  int
	money    float64
}

// SetMediator sets mediator.
func (c *Cannery) SetMediator(mediator Mediator) {
	c.mediator = mediator
}

// AddMoney adds money.
func (c *Cannery) AddMoney(m float64) {
```

```
	c.money += m
}

// MakeKetchup implementation.
func (c *Cannery) MakeKetchup(tomato int) {
	c.ketchup = tomato
	c.mediator.Notify("Cannery: Ketchup complete...")
}

// GetKetchup returns ketchup.
func (c *Cannery) GetKetchup() int {
	return c.ketchup
}

// Shop implements a Shop colleague.
type Shop struct {
	mediator Mediator
	money    float64
}

// SetMediator sets mediator.
func (s *Shop) SetMediator(mediator Mediator) {
	s.mediator = mediator
}

// AddMoney adds money.
func (s *Shop) AddMoney(m float64) {
	s.money += m
}

// SellKetchup converts ketchup to money.
func (s *Shop) SellKetchup(ketchup int) {
	s.money = float64(ketchup) * 54.75
}

// GetMoney returns money.
func (s *Shop) GetMoney() float64 {
	return s.money
}
```

```
//mediator_test.go
package mediator

import (
	"testing"
)

func TestMediator(t *testing.T) {

	farmer := new(Farmer)
	cannery := new(Cannery)
	shop := new(Shop)

	farmer.AddMoney(7500.00)
	cannery.AddMoney(15000.00)
	shop.AddMoney(30000.00)

	ConnectColleagues(farmer, cannery, shop)

	// A farmer grows a 1000kg tomato
	// and informs the mediator about the completion of his work.
	// Next, the mediator sends the tomatoes to the cannery.
	// After the cannery produces 1000 packs of ketchup,
	// he informs the mediator about his delivery to the store.
	farmer.GrowTomato(1000)

	expect := float64(54750)
	result := shop.GetMoney()

	if result != expect {
		t.Errorf("Expect result to equal %f, but %f.\n", expect, result)
	}
}
```

????????? (Memento)

Паттерн Memento относится к поведенческим паттернам уровня объекта.

Паттерн Memento получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии. Если клиенту в дальнейшем нужно "откатить" состояние исходного объекта, он передает Memento обратно в исходный объект для его восстановления.

Паттерн оперирует тремя объектами:

1. Хозяин состояния (Originator);
2. Хранитель (Memento) - Хранит в себе состояние объекта-хозяина класса Originator;
3. Смотритель (Caretaker) - Отвечает за сохранность объекта-хранителя класса Memento.

Требуется для реализации:

1. Класс Originator, у которого есть какое-то меняющиеся состояние, а так же он может создавать и принимать хранителей (Memento) своего состояния;
2. Класс Memento, реализует хранилище для состояния Originator;
3. Класс Caretaker, получает и хранит объект-хранитель (Memento), пока он не понадобится хозяину.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//memento.go
// Package memento is an example of the Memento Pattern.
package memento

// Originator implements a state master.
type Originator struct {
State string
```

```
}

// CreateMemento returns state storage.
func (o *Originator) CreateMemento() *Memento {
	return &Memento{state: o.State}
}

// SetMemento sets old state.
func (o *Originator) SetMemento(memento *Memento) {
	o.State = memento.GetState()
}

// Memento implements storage for the state of Originator
type Memento struct {
	state string
}

// GetState returns state.
func (m *Memento) GetState() string {
	return m.state
}

// Caretaker keeps Memento until it is needed by Originator.
type Caretaker struct {
	Memento *Memento
}
```

```
//memento_test.go
package memento

import (
	"testing"
)

func TestMomento(t *testing.T) {

	originator := new(Originator)
	caretaker := new(Caretaker)
```

```
originator.State = "On"

caretaker.Memento = originator.CreateMemento()

originator.State = "Off"

originator.SetMemento(caretaker.Memento)

if originator.State != "On" {
    t.Errorf("Expect State to %s, but %s", originator.State, "On")
}
}
```

?????????? (Observer)

Паттерн Observer относится к поведенческим паттернам уровня объекта.

Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются об этом и обновляются автоматически.

Основные участиники паттерна это Издатели (Subject) и Подписчики (Observer).

Имеется два способа получения уведомлений от издателя:

1. Метод вытягивания: После получения уведомления от издателя, подписчик должен пойти к издателю и забрать (вытянуть) данные самостоятельно.
2. Метод проталкивания: Издатель не уведомляет подписчика об обновлениях данных, а самостоятельно доставляет (проталкивает) данные подписчику.

Требуется для реализации:

1. Абстрактный класс Subject, определяющий интерфейс Издателя;
2. Класс ConcreteSubject, реализует интерфейс Subject;
3. Абстрактный класс Observer, определяющий общий функционал Подписчиков;
4. Класс ConcreteObserver, реализует Подписчика;

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//observer.go
// Package observer is an example of the Observer Pattern.
// Push model.
package observer

// Publisher interface.
type Publisher interface {
```

```
	Attach(observer Observer)
	SetState(state string)
	Notify()
}

// Observer provides a subscriber interface.
type Observer interface {
	Update(state string)
}

// ConcretePublisher implements the Publisher interface.
type ConcretePublisher struct {
	observers []Observer
	state     string
}

// NewPublisher is the Publisher constructor.
func NewPublisher() Publisher {
	return &ConcretePublisher{}
}

// Attach a Observer
func (s *ConcretePublisher) Attach(observer Observer) {
	s.observers = append(s.observers, observer)
}

// SetState sets new state
func (s *ConcretePublisher) SetState(state string) {
	s.state = state
}

// Notify sends notifications to subscribers.
// Push model.
func (s *ConcretePublisher) Notify() {
	for _, observer := range s.observers {
		observer.Update(s.state)
	}
}

// ConcreteObserver implements the Observer interface.
```

```
type ConcreteObserver struct {
	state string
}

// Update set new state
func (s *ConcreteObserver) Update(state string) {
	s.state = state
}
```

```
//observer_test.go
package observer

func ExampleObserver() {

	publisher := NewPublisher()

	publisher.Attach(&ConcreteObserver{})
	publisher.Attach(&ConcreteObserver{})
	publisher.Attach(&ConcreteObserver{})

	publisher.SetState("New State...")

	publisher.Notify()
}
```

????????? (State)

Паттерн State относится к поведенческим паттернам уровня объекта.

Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния и является объектно-ориентированной реализацией конечного автомата. Поведение объекта изменяется настолько, что создается впечатление, будто изменился класс объекта.

Паттерн должен применяться:

- когда поведение объекта зависит от его состояния
- поведение объекта должно изменяться во время выполнения программы
- состояний достаточно много и использовать для этого условные операторы, разбросанные по коду, достаточно затруднительно

Требуется для реализации:

1. Класс Context, представляет собой объектно-ориентированное представление конечного автомата;
2. Абстрактный класс State, определяющий интерфейс различных состояний;
3. Класс ConcreteStateA реализует одно из поведений, ассоциированное с определенным состоянием;
4. Класс ConcreteStateB реализует одно из поведений, ассоциированное с определенным состоянием.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go вместо наследования используется композиция.

```
//state.go
// Package state is an example of the State Pattern.
package state

// MobileAlertStater provides a common interface for various states.
```

```
type MobileAlertStater interface {
	Alert() string
}

// MobileAlert implements an alert depending on its state.
type MobileAlert struct {
	state MobileAlertStater
}

// Alert returns a alert string
func (a *MobileAlert) Alert() string {
	return a.state.Alert()
}

// SetState changes state
func (a *MobileAlert) SetState(state MobileAlertStater) {
	a.state = state
}

// NewMobileAlert is the MobileAlert constructor.
func NewMobileAlert() *MobileAlert {
	return &MobileAlert{state: &MobileAlertVibration{}}
}

// MobileAlertVibration implements vibration alert
type MobileAlertVibration struct {
}

// Alert returns a alert string
func (a *MobileAlertVibration) Alert() string {
	return "Vrrr... Brrr... Vrrr..."
}

// MobileAlertSong implements beep alert
type MobileAlertSong struct {
}

// Alert returns a alert string
func (a *MobileAlertSong) Alert() string {
	return "Белые розы, Белые розы. Беззащитны шипы..."
```

```
}
```

```
//state_test.go
package state

import (
	"testing"
)

func TestState(t *testing.T) {

	expect := "Vrrr... Brrr... Vrrr..." +
		"Vrrr... Brrr... Vrrr..." +
		"Белые розы, Белые розы. Беззащитны шипы..."

	mobile := NewMobileAlert()

	result := mobile.Alert()
	result += mobile.Alert()

	mobile.SetState(&MobileAlertSong{})

	result += mobile.Alert()

	if result != expect {
		t.Errorf("Expect result to equal %s, but %s.\n", expect, result)
	}
}
```

????????? (Strategy)

Паттерн Strategy относится к поведенческим паттернам уровня объекта.

Паттерн Strategy определяет набор алгоритмов схожих по роду деятельности, инкапсулирует их в отдельный класс и делает их подменяемыми. Паттерн Strategy позволяет подменять алгоритмы без участия клиентов, которые используют эти алгоритмы.

Требуется для реализации:

1. Класс Context, представляющий собой контекст выполнения той или иной стратегии;
2. Абстрактный класс Strategy, определяющий интерфейс различных стратегий;
3. Класс ConcreteStrategyA, реализует одну из стратегий представляющую собой алгоритмы, направленные на достижение определенной цели;
4. Класс ConcreteStrategyB, реализует одно из стратегий представляющую собой алгоритмы, направленные на достижение определенной цели.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//strategy.go
// Package strategy is an example of the Strategy Pattern.
package strategy

// StrategySort provides an interface for sort algorithms.
type StrategySort interface {
Sort([]int)
}

// BubbleSort implements bubble sort algorithm.
type BubbleSort struct {
}
```

```
// Sort sorts data.
func (s *BubbleSort) Sort(a []int) {
	size := len(a)
	if size < 2 {
		return
	}
	for i := 0; i < size; i++ {
		for j := size - 1; j >= i+1; j-- {
			if a[j] < a[j-1] {
				a[j], a[j-1] = a[j-1], a[j]
			}
		}
	}
}

// InsertionSort implements insertion sort algorithm.
type InsertionSort struct {
}

// Sort sorts data.
func (s *InsertionSort) Sort(a []int) {
	size := len(a)
	if size < 2 {
		return
	}
	for i := 1; i < size; i++ {
		var j int
		var buff = a[i]
		for j = i - 1; j >= 0; j-- {
			if a[j] < buff {
				break
			}
			a[j+1] = a[j]
		}
		a[j+1] = buff
	}
}

// Context provides a context for execution of a strategy.
type Context struct {
```

```
	strategy StrategySort
}

// Algorithm replaces strategies.
func (c *Context) Algorithm(a StrategySort) {
	c.strategy = a
}

// Sort sorts data according to the chosen strategy.
func (c *Context) Sort(s []int) {
	c.strategy.Sort(s)
}
```

```
//strategy_test.go
package strategy

import (
	"strconv"
	"testing"
)

func TestStrategy(t *testing.T) {

	data1 := []int{8, 2, 6, 7, 1, 3, 9, 5, 4}
	data2 := []int{8, 2, 6, 7, 1, 3, 9, 5, 4}

	ctx := new(Context)

	ctx.Algorithm(&BubbleSort{})

	ctx.Sort(data1)

	ctx.Algorithm(&InsertionSort{})

	ctx.Sort(data2)

	expect := "1,2,3,4,5,6,7,8,9,"

	var result1 string
```

```
    for _, val := range data1 {
        result1 += strconv.Itoa(val) + ","
    }

    if result1 != expect {
        t.Errorf("Expect result1 to equal %s, but %s.\n", expect, result1)
    }

    var result2 string
    for _, val := range data2 {
        result2 += strconv.Itoa(val) + ","
    }

    if result2 != expect {
        t.Errorf("Expect result2 to equal %s, but %s.\n", expect, result2)
    }
}
```

????????? ????? (Template Method)

Паттерн Template Method относится к поведенческим паттернам уровня класса.

Паттерн Template Method формирует структуру алгоритма и позволяет в производных классах реализовать, перекрыть или переопределить определенные шаги алгоритма, не изменяя структуру алгоритма в целом.

Проектировщик решает, какие шаги алгоритма являются неизменными, а какие изменяемыми. Абстрактный базовый класс реализует стандартные неизменяемые шаги алгоритма и может предоставлять реализацию по умолчанию для изменяемых шагов. Изменяемые шаги могут предоставляться клиентом компонента в конкретных производных классах.

Требуется для реализации:

1. Абстрактный класс AbstractClass, реализующий Template Method, который описывает порядок действий;
2. Класс ConcreteClass, реализующий изменяемые действия.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

Т.к. в Go нет понятия "Абстрактный Класс" и знакомого нам полиморфизма на наследовании, следует использовать встравивания общего для ConcreteClass типа с реализацией Template Method.

```
//template_method.go
// Package template_method is an example of the Template Method Pattern.
// In fact, this pattern is based on Abstract Class and Polymorphism.
// But there’s nothing like that in Go, so the composition will be applied.
```

```
package template_method

// QuotesInterface provides an interface for setting different quotes.
type QuotesInterface interface {
	Open() string
	Close() string
}

// Quotes implements a Template Method.
type Quotes struct {
	QuotesInterface
}

// Quotes is the Template Method.
func (q *Quotes) Quotes(str string) string {
	return q.Open() + str + q.Close()
}

// NewQuotes is the Quotes constructor.
func NewQuotes(qt QuotesInterface) *Quotes {
	return &Quotes{qt}
}

// FrenchQuotes implements wrapping the string in French quotes.
type FrenchQuotes struct {
}

// Open sets opening quotes.
func (q *FrenchQuotes) Open() string {
	return "«"
}

// Close sets closing quotes.
func (q *FrenchQuotes) Close() string {
	return "»"
}

// GermanQuotes implements wrapping the string in German quotes.
type GermanQuotes struct {
}
```

```
// Open sets opening quotes.
func (q *GermanQuotes) Open() string {
	return "„"
}

// Close sets closing quotes.
func (q *GermanQuotes) Close() string {
	return "“"
}
```

```
//template_method_test.go
package template_method

import (
	"testing"
)

func TestTemplateMethod(t *testing.T) {

	expect := "«Test String»"

	qt := NewQuotes(&FrenchQuotes{})

	result := qt.Quotes("Test String")

	if result != expect {
		t.Errorf("Expect result to equal %s, but %s.\n", expect, result)
	}
}
```

?????????? (Visitor)

Паттерн Visitor относится к поведенческим паттернам уровня объекта.

Паттерн Visitor позволяет обойти набор элементов (объектов) с разнородными интерфейсами, а также позволяет добавить новый метод в класс объекта, при этом, не изменяя сам класс этого объекта.

Требуется для реализации:

1. Абстрактный класс Visitor, описывающий интерфейс визитера;
2. Класс ConcreteVisitor, реализующий конкретного визитера. Реализует методы для обхода конкретного элемента;
3. Класс ObjectStructure, реализующий структуру(коллекцию), в которой хранятся элементы для обхода;
4. Абстрактный класс Element, реализующий интерфейс элементов структуры;
5. Класс ElementA, реализующий элемент структуры;
6. Класс ElementB, реализующий элемент структуры.

[!] В описании паттерна применяются общие понятия, такие как Класс, Объект, Абстрактный класс. Применимо к языку Go, это Пользовательский Тип, Значение этого Типа и Интерфейс. Также в языке Go за место общепринятого наследования используется агрегирование и встраивание.

```
//visitor.go
// Package visitor is an example of the Visitor Pattern.
package visitor

// Visitor provides a visitor interface.
type Visitor interface {
	VisitSushiBar(p *SushiBar) string
	VisitPizzeria(p *Pizzeria) string
	VisitBurgerBar(p *BurgerBar) string
}

// Place provides an interface for place that the visitor should visit.
```

```
type Place interface {
	Accept(v Visitor) string
}

// People implements the Visitor interface.
type People struct {
}

// VisitSushiBar implements visit to SushiBar.
func (v *People) VisitSushiBar(p *SushiBar) string {
	return p.BuySushi()
}

// VisitPizzeria implements visit to Pizzeria.
func (v *People) VisitPizzeria(p *Pizzeria) string {
	return p.BuyPizza()
}

// VisitBurgerBar implements visit to BurgerBar.
func (v *People) VisitBurgerBar(p *BurgerBar) string {
	return p.BuyBurger()
}

// City implements a collection of places to visit.
type City struct {
	places []Place
}

// Add appends Place to the collection.
func (c *City) Add(p Place) {
	c.places = append(c.places, p)
}

// Accept implements a visit to all places in the city.
func (c *City) Accept(v Visitor) string {
	var result string
	for _, p := range c.places {
		result += p.Accept(v)
	}
	return result
```

```
}

// SushiBar implements the Place interface.
type SushiBar struct {
}

// Accept implementation.
func (s *SushiBar) Accept(v Visitor) string {
	return v.VisitSushiBar(s)
}

// BuySushi implementation.
func (s *SushiBar) BuySushi() string {
	return "Buy sushi..."
}

// Pizzeria implements the Place interface.
type Pizzeria struct {
}

// Accept implementation.
func (p *Pizzeria) Accept(v Visitor) string {
	return v.VisitPizzeria(p)
}

// BuyPizza implementation.
func (p *Pizzeria) BuyPizza() string {
	return "Buy pizza..."
}

// BurgerBar implements the Place interface.
type BurgerBar struct {
}

// Accept implementation.
func (b *BurgerBar) Accept(v Visitor) string {
	return v.VisitBurgerBar(b)
}

// BuyBurger implementation.
```

```
func (b *BurgerBar) BuyBurger() string {
    return "Buy burger..."
}
```

```
//visitor_test.go
package visitor

import (
    "testing"
)

func TestVisitor(t *testing.T) {

    expect := "Buy sushi...Buy pizza...Buy burger..."

    city := new(City)

    city.Add(&SushiBar{})
    city.Add(&Pizzeria{})
    city.Add(&BurgerBar{})

    result := city.Accept(&People{})

    if result != expect {
        t.Errorf("Expect result to equal %s, but %s.\n", expect, result)
    }
}
```