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

Паттерны проектирования с примерами на Golang

Это коллекция из 22 популярных паттернов проектирования с примерами кода на языке Go и кратким описанием паттерна.

В кратких описаниях будут употребляться классические термины, такие как Класс, Объект, Абстрактный Класс. Применимо к языку Go, это Тип, Значение этого типа и Интерфейс (где это возможно).

Умение правильно использовать шаблоны проектирования, так сказать, в нужном месте и в нужное время, помогут сохранить ваши нервные клетки.

Установка
Вы можете скачать этот репозиторий и запустить тесты

$ go get github.com/alexandergrom/go-patterns

- Поведенческие паттерны (Behavioral)
  - 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)

- Порождающие паттерны (Creational)
  - Абстрактная фабрика (Abstract Factory)
  - Строитель (Builder)
  - Фабричный метод (FactoryMethod)
  - Прототип (Prototype)
  - Одиночка (Singleton)

- Структурные паттерны (Structural)
  - Адаптер (Adapter)
  - Мост (Bridge)
  - Компоновщик (Composite)
  - Декоратор (Decorator)
  - Фасад (Facade)
  - Приспособленец (Flyweight)
  - Заместитель (Proxy)

- Спецификация (Specification)

Поведенческие паттерны (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)

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)
	}
}
```

Команда (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)
	}
}
```

Итератор (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)
		}
	}
}
```

Посредник (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)
	}
}
```

Хранитель (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")
}
}
```

Наблюдатель (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()
}
```

Состояние (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)
	}
}
```

Стратегия (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)
    }
}
```

Шаблонный метод (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)
	}
}
```

Посетитель (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)
	}
}
```

Порождающие паттерны (Creational)

Порождающие паттерны делятся на два типа:

1. Паттерны уровня класса
2. Паттерны уровня объекта.

Паттерны уровня класса изменяют класс создаваемого объекта с помощью наследования.

Паттерны уровня объекта создают новые объекты с помощью других объектов.

К паттернам уровня класса относится только «Фабричный метод».

Порождающие паттерны отвечают за создание классов и объектов. Другими словами порождают классы и порождают объекты.

* [Абстрактная фабрика (Abstract Factory)](AbstractFactory)
* [Строитель (Builder)](Builder)
* [Фабричный метод (Factory Method)](FactoryMethod)
* [Прототип (Prototype)](Prototype)
* [Одиночка (Singleton)](Singleton)

Абстрактная фабрика (Abstract Factory)

Паттерн Abstract Factory относится к порождающим паттернам уровня объекта.

Паттерн Abstract Factory предоставляет общий интерфейс для создания семейства взаимосвязанных объектов. Это позволяет отделить функциональность системы от внутренней реализации каждого класса, а обращение к этим классам становится возможным через абстрактные интерфейсы.

В общем виде абстрактная фабрика выглядит следующим образом. Для каждого из семейств объектов, создается конкретная фабрика (наследник абстрактной), посредством которой создаются продукты этого семейства.

Пример: Есть две фабрики по производству газировки, Кока-Кола и Пепси. Эти фабрики выпускают семейство продуктов (объектов) - бутылка, крышка, этикетка, жидкость. Каждая из этих фабрик выпускает продукты, которые взаимодействуют между собой и не могут жить отдельно друг от друга. Фабрика Кока-Кола не может поставлять клиентам пустые бутылки.

Что бы реализовать простое создание семейства объектов, должен быть интерфейс, по которому работает фабрика, так же фабрика должна выпускать продукты с определенным интерфейсом. Например, бутылки обеих компаний обладают одним интерфейсом - у них есть горлышко через которое они наполняются жидкостью, так же мы можем узнать объем бутылок. Дальше бутылки могут отличаться по форме, объему или материалу, нас это не касается, нам нужно только знать, куда наливать жидкость, а так же, сколько этой жидкости нужно.

Требуется для реализации:

1. Класс абстрактной фабрики AbstractFactory, описывающий общий интерфейс фабрики, от которой будет наследоваться каждая конкретная фабрика;
2. Класс абстрактного продукта AbstractProduct, описывающий общий интерфейс продукта, от которого будет наследоваться каждый конкретный продукт;
3. Класс конкретной фабрики Factory;
4. Класс конкретного продукта ProductA.
5. Класс конкретного продукта ProductB.

Подведем итог.

Абстрактная фабрика представляет собой базовый класс, описывающий интерфейс конкретных фабрик, создающих продукты. Производные от него классы конкретных фабрик, должны реализовать этот интерфейс.

Также абстрактная фабрика должна описывать абстрактные продукты, которые она производит, что бы конкретные фабрики производили продукты с нужными интерфейсами.

[!] 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.

```
//abstract_factory.go
// Package abstract_factory is an example of the Abstract Factory Pattern.
package abstract_factory

// AbstractFactory provides an interface for creating families of related objects.
type AbstractFactory interface {
	CreateWater(volume float64) AbstractWater
	CreateBottle(volume float64) AbstractBottle
}

// AbstractWater provides a water interface.
type AbstractWater interface {
	GetVolume() float64
}

// AbstractBottle provides a bottle interface.
```

```
type AbstractBottle interface {
	PourWater(water AbstractWater) // Bottle interacts with a water.
	GetBottleVolume() float64
	GetWaterVolume() float64
}

// CocaColaFactory implements AbstractFactory interface.
type CocaColaFactory struct {
}

// NewCocaColaFactory is the CocaColaFactory constructor.
func NewCocaColaFactory() AbstractFactory {
	return &CocaColaFactory{}
}

// CreateWater implementation.
func (f *CocaColaFactory) CreateWater(volume float64) AbstractWater {
	return &CocaColaWater{volume: volume}
}

// CreateBottle implementation.
func (f *CocaColaFactory) CreateBottle(volume float64) AbstractBottle {
	return &CocaColaBottle{volume: volume}
}

// CocaColaWater implements AbstractWater.
type CocaColaWater struct {
	volume float64 // Volume of drink.
}

// GetVolume returns volume of drink.
func (w *CocaColaWater) GetVolume() float64 {
	return w.volume
}

// CocaColaBottle implements AbstractBottle.
type CocaColaBottle struct {
	water  AbstractWater // Bottle must contain a drink.
	volume float64       // Volume of bottle.
}
```

```
// PourWater pours water into a bottle.
func (b *CocaColaBottle) PourWater(water AbstractWater) {
	b.water = water
}

// GetBottleVolume returns volume of bottle.
func (b *CocaColaBottle) GetBottleVolume() float64 {
	return b.volume
}

// GetWaterVolume returns volume of water.
func (b *CocaColaBottle) GetWaterVolume() float64 {
	return b.water.GetVolume()
}
```

```
//abstract_factory_test.go
package abstract_factory

import (
	"testing"
)

func TestAbstractFactory(t *testing.T) {

	cocacolaFactory := NewCocaColaFactory()

	cocacolaWater := cocacolaFactory.CreateWater(2.5)
	cocacolaBottle := cocacolaFactory.CreateBottle(2.5)

	cocacolaBottle.PourWater(cocacolaWater)

	if cocacolaBottle.GetWaterVolume() != cocacolaBottle.GetBottleVolume() {
		t.Errorf("Expect volume to %.1fL, but %.1fL", cocacolaBottle.GetWaterVolume(),
cocacolaBottle.GetBottleVolume())
	}
}
```

Строитель (Builder)

Паттерн Builder относится к порождающим паттернам уровня объекта.

Паттерн Builder определяет процесс поэтапного построения сложного продукта. После того как будет построена последняя его часть, продукт можно использовать.

В примере паттерна Abstract Factory приводился пример двух фабрик Кока-Кола и Перси. Возьмем одну фабрику, она производит сложный продукт, состоящий из 4 частей (крышка, бутылка, этикетка, напиток), которые должны быть применены в нужном порядке. Нельзя вначале взять крышку, бутылку, завинтить крышку, а потом пытаться налить туда напиток. Для реализации объекта, бутылки Кока-Колы, которая поставляется клиенту, нам нужен паттерн Builder.

Важно понимать, что сложный объект это не обязательно объект оперирующий несколькими другими объектами в смысле ООП. Например, нам нужно получить документ состоящий из заголовка, введения, содержания и заключения. Наш документ, это сложный объект. Что бы был какой-то единый порядок составления документа, мы будем использовать паттерн Builder.

Требуется для реализации:

1. Класс Director, который будет распоряжаться строителем и отдавать ему команды в нужном порядке, а строитель будет их выполнять;
2. Базовый абстрактный класс Builder, который описывает интерфейс строителя, те команды, которые он обязан выполнять;
3. Класс ConcreteBuilder, который реализует интерфейс строителя и взаимодействует со сложным объектом;
4. Класс сложного объекта Product.

[!] 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 заместо общепринятого наследования используется агрегирование и

встраивание.

```
//builder.go
// Package builder is an example of the Builder Pattern.
package builder

// Builder provides a builder interface.
type Builder interface {
	MakeHeader(str string)
	MakeBody(str string)
	MakeFooter(str string)
}

// Director implements a manager
type Director struct {
	builder Builder
}

// Construct tells the builder what to do and in what order.
func (d *Director) Construct() {
	d.builder.MakeHeader("Header")
	d.builder.MakeBody("Body")
	d.builder.MakeFooter("Footer")
}

// ConcreteBuilder implements Builder interface.
type ConcreteBuilder struct {
	product *Product
}

// MakeHeader builds a header of document..
func (b *ConcreteBuilder) MakeHeader(str string) {
	b.product.Content += "<header>" + str + "</header>"
}

// MakeBody builds a body of document.
func (b *ConcreteBuilder) MakeBody(str string) {
	b.product.Content += "<article>" + str + "</article>"
}
```

```
// MakeFooter builds a footer of document.
func (b *ConcreteBuilder) MakeFooter(str string) {
	b.product.Content += "<footer>" + str + "</footer>"
}

// Product implementation.
type Product struct {
	Content string
}

// Show returns product.
func (p *Product) Show() string {
	return p.Content
}
```

```
//builder_test.go
package builder

import (
	"testing"
)

func TestBuilder(t *testing.T) {

	expect := "<header>Header</header>" +
		"<article>Body</article>" +
		"<footer>Footer</footer>"

	product := new(Product)

	director := Director{&ConcreteBuilder{product}}
	director.Construct()

	result := product.Show()

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

Фабричный метод (FactoryMethod)

Паттерн Factory Method относится к порождающим паттернам уровня класса и сфокусирован только на отношениях между классами.

Паттерн Factory Method полезен, когда система должна оставаться легко расширяемой путем добавления объектов новых типов. Этот паттерн является основой для всех порождающих паттернов и может легко трансформироваться под нужды системы. По этому, если перед разработчиком стоят не четкие требования для продукта или не ясен способ организации взаимодействия между продуктами, то для начала можно воспользоваться паттерном Factory Method, пока полностью не сформируются все требования.

Паттерн Factory Method применяется для создания объектов с определенным интерфейсом, реализации которого предоставляются потомками. Другими словами, есть базовый абстрактный класс фабрики, который говорит, что каждая его наследующая фабрика должна реализовать такой-то метод для создания своих продуктов.

Реализация фабричного метода может быть разной, в большинстве случаем это зависит от языка реализации. Это может быть полиморфизм или параметризированный метод.

Пример: К нам приходят файлы трех расширений .txt, .png, .doc. В зависимости от расширения файла мы должны сохранять его в одном из каталогов /file/txt/, /file/png/ и /file/doc/. Значит, у нас будет файловая фабрика с параметризированным фабричным методом, принимающим путь к файлу, который нам нужно сохранить в одном из каталогов. Этот фабричный метод возвращает нам объект, используя который мы можем манипулировать с нашим файлом (сохранить, посмотреть тип и каталог для сохранения). Заметьте, мы никак не указываем какой экземпляр объекта-продукта нам нужно получить, это делает фабричный метод путем определения расширения файла и на его основе выбора подходящего класса продукта. Тем самым, если наша система будет расширяться и

доступных расширений файлов станет, например 25, то нам всего лишь нужно будет изменить фабричный метод и реализовать классы продуктов.

Требуется для реализации:

1. Базовый абстрактный класс Creator, описывающий интерфейс, который должна реализовать конкретная фабрика для производства продуктов. Этот базовый класс описывает фабричный метод.
2. Базовый класс Product, описывающий интерфейс продукта, который возвращает фабрика. Все продукты возвращаемые фабрикой должны придерживаться единого интерфейса.
3. Класс конкретной фабрики по производству продуктов ConcreteCreator. Этот класс должен реализовать фабричный метод;
4. Класс реального продукта ConcreteProductA;
5. Класс реального продукта ConcreteProductB;
6. Класс реального продукта ConcreteProductC.

Factory Method отличается от Abstract Factory, тем, что Abstract Factory производит семейство объектов, эти объекты разные, обладают разными интерфейсами, но взаимодействуют между собой. В то время как Factory Method производит продукты придерживающиеся одного интерфейса и эти продукты не связаны между собой, не вступают во взаимодействие.

[!] 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.

```
//factory_method.go
// Package factory_method is an example of the Factory Method pattern.
package factory_method

import (
"log"
)

// action helps clients to find out available actions.
```

```
type action string

const (
	A action = "A"
	B action = "B"
	C action = "C"
)

// Creator provides a factory interface.
type Creator interface {
	CreateProduct(action action) Product // Factory Method
}

// Product provides a product interface.
// All products returned by factory must provide a single interface.
type Product interface {
	Use() string // Every product should be usable
}

// ConcreteCreator implements Creator interface.
type ConcreteCreator struct{}

// NewCreator is the ConcreteCreator constructor.
func NewCreator() Creator {
	return &ConcreteCreator{}
}

// CreateProduct is a Factory Method.
func (p *ConcreteCreator) CreateProduct(action action) Product {
	var product Product

	switch action {
	case A:
		product = &ConcreteProductA{string(action)}
	case B:
		product = &ConcreteProductB{string(action)}
	case C:
		product = &ConcreteProductC{string(action)}
	default:
		log.Fatalln("Unknown Action")
```

```
	}

	return product
}

// ConcreteProductA implements product "A".
type ConcreteProductA struct {
	action string
}

// Use returns product action.
func (p *ConcreteProductA) Use() string {
	return p.action
}

// ConcreteProductB implements product "B".
type ConcreteProductB struct {
	action string
}

// Use returns product action.
func (p *ConcreteProductB) Use() string {
	return p.action
}

// ConcreteProductC implements product "C".
type ConcreteProductC struct {
	action string
}

// Use returns product action.
func (p *ConcreteProductC) Use() string {
	return p.action
}
```

```
//factory_method_test.go
package factory_method

import (
```

```
    "testing"
)

func TestFactoryMethod(t *testing.T) {

    assert := []string{"A", "B", "C"}

    factory := NewCreator()
    products := []Product{
        factory.CreateProduct(A),
        factory.CreateProduct(B),
        factory.CreateProduct(C),
    }

    for i, product := range products {
        if action := product.Use(); action != assert[i] {
            t.Errorf("Expect action to %s, but %s.\n", assert[i], action)
        }
    }
}
```

Прототип (Prototype)

Паттерн Prototype относится к порождающим паттернам уровня объекта.

Паттерн Prototype позволяет создавать новые объекты, путем копирования (клонирования) созданного ранее объекта-оригинала-продукта (прототипа).

Паттерн описывает процесс создания объектов-клонов на основе имеющегося объекта-прототипа, другими словами, паттерн Prototype описывает способ организации процесса клонирования.

Требуется для реализации:

1. Базовый класс Prototype, объявляющий интерфейс клонирования. Все классы его наследующие должны реализовывать этот механизм клонирования;
2. Класс продукта ConcretePrototypeA, который должен реализовывать этот прототип;
3. Класс продукта ConcretePrototypeB, который должен реализовывать этот прототип.

Обычно операция клонирования происходит через метод clone(), который описан в базовом классе и его должен реализовать каждый продукт.

[!] 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.

```
//prototype.go

// Package prototype is an example of the Singleton Pattern.
package prototype

// Prototyper provides a cloning interface.
type Prototyper interface {
	Clone() Prototyper
```

```
	GetName() string
}

// ConcreteProduct implements product "A"
type ConcreteProduct struct {
	name string // Имя продукта
}

// NewConcreteProduct is the Prototyper constructor.
func NewConcreteProduct(name string) Prototyper {
	return &ConcreteProduct{
		name: name,
	}
}

// GetName returns product name
func (p *ConcreteProduct) GetName() string {
	return p.name
}

// Clone returns a cloned object.
func (p *ConcreteProduct) Clone() Prototyper {
	return &ConcreteProduct{p.name}
}
```

```
//prototype_test.go
package prototype

import (
	"testing"
)

func TestPrototype(t *testing.T) {

	product := NewConcreteProduct("A")
	cloneProduct := product.Clone()

	if cloneProduct.GetName() != product.GetName() {
		t.Error("Expect name \"A\" to equal, but not equal.")
```

```
□}
}
```

Одиночка (Singleton)

Паттерн Singleton относится к порождающим паттернам уровня объекта. Паттерн контролирует создание единственного экземпляра некоторого класса и предоставляет доступ к нему. Другими словами, Singleton гарантирует, что у класса будет только один экземпляр и предоставляет к нему точку доступа, через фабричный метод.

Требуется для реализации:

1. Функция GetInstance, создающая экземпляр класса Singleton только один раз. Если до этого экземпляр уже был создан, то просто возвращает этот экземпляр.

[!] 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.

```
//singleton.go
// Package singleton is an example of the Singleton Pattern.
package singleton

import (
	"sync"
)

// Singleton implementation.
type Singleton struct {
}

var (
	instance *Singleton
	once     sync.Once
)
```

```
// GetInstance returns singleton
func GetInstance() *Singleton {
    once.Do(func() {
        instance = &Singleton{}
    })
    return instance
}
```

```
//singleton_test.go
package singleton

import (
    "testing"
)

func TestSingleton(t *testing.T) {

    instance1 := GetInstance()
    instance2 := GetInstance()

    if instance1 != instance2 {
        t.Error("Objects are not equal!\n")
    }
}
```

Структурные паттерны (Structural)

Структурные 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 будет удален так же, т.к. на него нет ссылки извне.

К паттернам уровня класса относится только «Адаптер». Смысл его работы в том, что если у вас есть класс и его интерфейс не совместим с библиотеками вашей системы, то что бы разрешить этот конфликт, мы не изменяем код этого класса, а пишем для него адаптер.

Все структурные паттерны отвечают за создание правильной структуры системы, в которой без труда смогут взаимодействовать между собой уже имеющиеся классы и объекты.

* [Адаптер (Adapter)](Adapter)
* [Мост (Bridge)](Bridge)
* [Компоновщик (Composite)](Composite)
* [Декоратор (Decorator)](Decorator)
* [Фасад (Facade)](Facade)
* [Приспособленец (Flyweight)](Flyweight)
* [Заместитель (Proxy)](Proxy)

Адаптер (Adapter)

Паттерн Adapter относится к структурным паттернам уровня класса.

Часто в новом проекте разработчики хотят повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью и иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн Adapter.

Смысл работы этого паттерна в том, что если у вас есть класс и его интерфейс не совместим с кодом вашей системы, то что бы разрешить этот конфликт, мы не изменяем код этого класса, а пишем для него адаптер. Другими словами Adapter адаптирует существующий код к требуемому интерфейсу (является переходником).

Требуется для реализации:

1. Интерфейс Target, описывающий целевой интерфейс (тот интерфейс с которым наша система хотела бы работать);
2. Класс Adaptee, который наша система должна адаптировать под себя;
3. Класс Adapter, адаптер реализующий целевой интерфейс.

[!] 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.

```
//adapter.go
// Package adapter is an example of the Adapter Pattern.
package adapter

// Target provides an interface with which the system should work.
type Target interface {
	Request() string
}
```

```
// Adaptee implements system to be adapted.
type Adaptee struct {
}

// NewAdapter is the Adapter constructor.
func NewAdapter(adaptee *Adaptee) Target {
	return &Adapter{adaptee}
}

// SpecificRequest implementation.
func (a *Adaptee) SpecificRequest() string {
	return "Request"
}

// Adapter implements Target interface and is an adapter.
type Adapter struct {
	*Adaptee
}

// Request is an adaptive method.
func (a *Adapter) Request() string {
	return a.SpecificRequest()
}
```

```
//adapter_test.go
package adapter

import (
	"testing"
)

func TestAdapter(t *testing.T) {

	adapter := NewAdapter(&Adaptee{})

	req := adapter.Request()

	if req != "Request" {
		t.Errorf("Expect volume to %s, but %s", "Request", req)
```

```
□}
}
```

Мост (Bridge)

Паттерн Bridge относится к структурным паттернам уровня объекта.

Паттерн Bridge позволяет разделить объект на абстракцию и реализацию так, чтобы они могли изменяться независимо друг от друга.

Если для одной абстракции возможно несколько реализаций, то обычно используют наследование. Однако такой подход не всегда удобен, так как наследование жестко привязывает реализацию к абстракции, что затрудняет независимую модификацию и усложняет их повторное использование.

Паттерн следует применять, когда у нас имеется абстракция и несколько её реализаций. Разумеется, нет смысла отделять абстракцию от реализации, если реализация может быть только одна.

Я не нашел не одного адекватного описания паттерна "Мост". Все что мне встречалось, либо не соответствует действительности и примеры высосаны из пальца или очень размыты. Из того, что я понял и могу объяснить на пальцах - Мост это хитрая агрегация. Класс реализующий изделие, реализует интерфейс агрегируемого класса, который подсовывается на этапе создания экземпляра класса изделия.

Как я понял... у нас есть 3 машины и 3 разных двигателя. Каждый двигатель подходит к каждой машине, т.е. она реализует его интерфейс. Если делать это наследованием, мы получим 9 разных классом. Получается у каждой машины 3 модификации. Это неудобно, поэтому мы будем подсовывать двигатель на этапе создания машины. Так же каждый двигатель, может работать на разном топливе, дизель или бензин, что бы не плодить 6 разных реализаций, при создании двигателя мы будем подсовывать в него тип топлива.

Для реализации паттерна в этом примере необходимо в базовом классе автомобилей добавить поле для хранения указателя на тип реализации, значение которого класс будет получать в своём конструкторе, и вызывать по необходимости методы вложенного объекта.

Требуется для реализации:

1. Базовый абстрактный класс (в нашем случаем описывающий автомобиль);
2. Класс реализующий базовый класс. В нем есть свойство в которое мы будем подсовывать указатель на используемый двигатель (машина может работать с любым из представленных двигателей);
3. Абстракция двигателя;
4. Реализация двигателя.

В общем свойство хранящее указатель на используемый объект и есть мост. Мы в него можем подсовывать разные объекты, главное, что бы они имели одинаковый интерфейс.

[!] 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 заместо общепринятого наследования используется агрегирование и встраивание.

```
//bridge.go
// Package bridge is an example of the Bridge Pattern.
package bridge

// Carer provides car interface.
type Carer interface {
Rase() string
}

// Enginer provides engine interface.
type Enginer interface {
GetSound() string
}

// Car implementation.
type Car struct {
engine Enginer
}

// NewCar is the Car constructor.
func NewCar(engine Enginer) Carer {
```

```
	return &Car{
		engine: engine,
	}
}

// Rase implementation.
func (c *Car) Rase() string {
	return c.engine.GetSound()
}

// EngineSuzuki implements Suzuki engine.
type EngineSuzuki struct {
}

// GetSound returns sound of the engine.
func (e *EngineSuzuki) GetSound() string {
	return "SssuuuuZzzuuuuKkiiiii"
}

// EngineHonda implements Honda engine.
type EngineHonda struct {
}

// GetSound returns sound of the engine.
func (e *EngineHonda) GetSound() string {
	return "HhoooNnnnnnnnnDddaaaaaaa"
}

// EngineLada implements Lada engine.
type EngineLada struct {
}

// GetSound returns sound of the engine.
func (e *EngineLada) GetSound() string {
	return "PhhhhPhhhhPhPhPhPhPh"
}
```

```
//bridge_test.go
package bridge
```

```
import (
	"testing"
)

func TestBridge(t *testing.T) {

	expect := "SssuuuuZzzuuuuKkiiiii"

	car := NewCar(&EngineSuzuki{})

	sound := car.Rase()

	if sound != expect {
		t.Errorf("Expect sound to %s, but %s", expect, sound)
	}
}
```

Компоновщик (Composite)

Паттерн Composite относится к структурным паттернам уровня объекта.

Паттерн Composite группирует схожие объекты в древовидные структуры.

Для построения дерева будут использоваться массивы, представляющие ветви дерева.

Требуется для реализации:

1. Базовый абстрактный класс Component который предоставляет интерфейс, как для ветвей, так и для листьев дерева;
2. Класс Composite, реализующий интерфейс Component и являющийся ветвью дерева;
3. Класс Leaf, реализующий интерфейс Component и являющийся листом дерева.

Обратите внимание, что лист дерева является классом листовых узлов и не может иметь потомков (из листа не может вырасти ветвь или другой лист).

Ветви дерева задают поведение объектов, входящих в структуру дерева, у которых есть потомки, а также сами хранит в себе компоненты дерева. Другим словами ветви могут содержать другие ветви и листья.

Основным назначением паттерна, является обеспечение единого интерфейса как к составному (ветви) так и конечному (листу) объекту, что бы клиент не задумывался над тем, с каким объектом он работает.

[!] 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.

```
//composite.go
// Package composite is an example of the Composite Pattern.
package composite
```

```
// Component provides an interface for branches and leaves of a tree.
type Component interface {
	Add(child Component)
	Name() string
	Child() []Component
	Print(prefix string) string
}

// Directory implements branches of a tree
type Directory struct {
	name   string
	childs []Component
}

// Add appends an element to the tree branch.
func (d *Directory) Add(child Component) {
	d.childs = append(d.childs, child)
}

// Name returns name of the Component.
func (d *Directory) Name() string {
	return d.name
}

// Child returns child elements.
func (d *Directory) Child() []Component {
	return d.childs
}

// Print returns the branche in string representation.
func (d *Directory) Print(prefix string) string {
	result := prefix + "/" + d.Name() + "\n"
	for _, val := range d.Child() {
		result += val.Print(prefix + "/" + d.Name())
	}
	return result
}

// File implements a leaves of a tree
```

```
type File struct {
	name string
}

// Add implementation.
func (f *File) Add(child Component) {
}

// Name returns name of the Component.
func (f *File) Name() string {
	return f.name
}

// Child implementation.
func (f *File) Child() []Component {
	return []Component{}
}

// Print returns the leave in string representation.
func (f *File) Print(prefix string) string {
	return prefix + "/" + f.Name() + "\n"
}

// NewDirectory is constructor.
func NewDirectory(name string) *Directory {
	return &Directory{
		name: name,
	}
}

// NewFile is constructor.
func NewFile(name string) *File {
	return &File{
		name: name,
	}
}
```

```
//composite_test.go
package composite
```

```
import (
    "testing"
)

func TestComposite(t *testing.T) {

    expect := "/root\n/root/usr\n/root/usr/B\n/root/A\n"

    rootDir := NewDirectory("root")

    usrDir := NewDirectory("usr")
    fileA := NewFile("A")

    rootDir.Add(usrDir)
    rootDir.Add(fileA)

    fileB := NewFile("B")

    usrDir.Add(fileB)

    result := rootDir.Print("")

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

Декоратор (Decorator)

Паттерн Decorator относится к структурным паттернам уровня объекта.

Паттерн Decorator используется для расширения функциональности объектов путем динамического добавления объекту новых возможностей. При реализации паттерна используется отношение композиции.

Сущность работы декоратора заключается в обёртывании готового объекта новым функционалом, при этом весь оригинальный интерфейс объекта остается доступным, путем передачи декоратором всех запросов обернутому объекту.

Требуется для реализации:

1. Базовый абстрактный класс Component который предоставляет интерфейс для класса декоратора и компонента;
2. Класс ConcreteDecorator, реализующий интерфейс Component и перезагружающий все методы компонента, по необходимости к ним добавляется функционал;
3. Класс ConcreteComponent реализующий интерфейс Component и который будет обернут декоратором.

При такой структуре нам не важно является ли компонент декоратором или конкретной реализацией, так как интерфейс у них совпадает, и мы можем делать цепочки декораторов. Тем самым динамически менять состояние и поведение объекта.

Я слышал пример с Калсоном и мне он очень понравился. У нас есть Карлсон, мы на него одеваем комбинезон тем самым меняя его состояние, потом на штаны одеваем пропеллер тем самым меняем поведение. Пропеллер в зависимости от ситуации можно снять, изменив поведение на обратное или можно одеть другой комбинезон с другими свойствами.

[!] 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.

```
//decorator.go
// Package decorator is an example of the Decorator Pattern.
package decorator

// Component provides an interface for a decorator and component.
type Component interface {
	Operation() string
}

// ConcreteComponent implements a component.
type ConcreteComponent struct {
}

// Operation implementation.
func (c *ConcreteComponent) Operation() string {
	return "I am component!"
}

// ConcreteDecorator implements a decorator.
type ConcreteDecorator struct {
	component Component
}

// Operation wraps operation of component
func (d *ConcreteDecorator) Operation() string {
	return "<strong>" + d.component.Operation() + "</strong>"
}
```

```
//decorator_test.go
package decorator

import (
	"testing"
)

func TestDecorator(t *testing.T) {
```

```
    expect := "<strong>I am component!</strong>"

    decorator := &ConcreteDecorator{&ConcreteComponent{}}

    result := decorator.Operation()

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

Фасад (Facade)

Паттерн Facade относится к структурным паттернам уровня объекта.

Паттерн Facade предоставляет высокоуровневый унифицированный интерфейс в виде набора имен методов к набору взаимосвязанных классов или объектов некоторой подсистемы, что облегчает ее использование.

Разбиение сложной системы на подсистемы позволяет упростить процесс разработки, а также помогает максимально снизить зависимости одной подсистемы от другой. Однако использовать такие подсистемы становиться довольно сложно. Один из способов решения этой проблемы является паттерн Facade. Наша задача, сделать простой, единый интерфейс, через который можно было бы взаимодействовать с подсистемами.

В качестве примера можно привести интерфейс автомобиля. Современные автомобили имеют унифицированный интерфейс для водителя, под которым скрывается сложная подсистема. Благодаря применению навороченной электроники, делающей большую часть работы за водителя, тот может с лёгкостью управлять автомобилем, не задумываясь, как там все работает.

Требуется для реализации:

1. Класс Facade предоставляющий унифицированный доступ для классов подсистемы;
2. Класс подсистемы SubSystemA;
3. Класс подсистемы SubSystemB;
4. Класс подсистемы SubSystemC.

Заметьте, что фасад не является единственной точкой доступа к подсистеме, он не ограничивает возможности, которые могут понадобиться "продвинутым" пользователям, желающим работать с подсистемой напрямую.

[!] 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.

```
//facade.go
// Package facade is an example of the Facade Pattern.
package facade

import (
	"strings"
)

// NewMan creates man.
func NewMan() *Man {
	return &Man{
		house: &House{},
		tree:  &Tree{},
		child: &Child{},
	}
}

// Man implements man and facade.
type Man struct {
	house *House
	tree  *Tree
	child *Child
}

// Todo returns that man must do.
func (m *Man) Todo() string {
	result := []string{
		m.house.Build(),
		m.tree.Grow(),
		m.child.Born(),
	}
	return strings.Join(result, "\n")
}

// House implements a subsystem "House"
type House struct {
}
```

```
// Build implementation.
func (h *House) Build() string {
	return "Build house"
}

// Tree implements a subsystem "Tree"
type Tree struct {
}

// Grow implementation.
func (t *Tree) Grow() string {
	return "Tree grow"
}

// Child implements a subsystem "Child"
type Child struct {
}

// Born implementation.
func (c *Child) Born() string {
	return "Child born"
}
```

```
//facade_test.go
package facade

import (
	"testing"
)

func TestFacade(t *testing.T) {

	expect := "Build house\nTree grow\nChild born"

	man := NewMan()

	result := man.Todo()

	if result != expect {
```

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

Приспособленец (Flyweight)

Паттерн Flyweight относится к структурным паттернам уровня объекта.

Паттерн Flyweight используется для эффективной поддержки большого числа мелких объектов, он позволяет повторно использовать мелкие объекты в различном контексте.

Требуется для реализации:

1. Класс FlyweightFactory, являющейся модифицированным паттерном фабрики, для создания приспособленцев;
2. Базовый абстрактный класс Flyweight, для описания общего интерфейса приспособленцев;
3. Класс ConcreteFlyweight реализующий приспособленца, который будет замещать собой одинаковые мелкие объекты.

Суть в том, что мы можем запрашивать приспособленцев у фабрики по запросу, в свою очередь она будет отдавать те объекты, которые уже были созданы, или создавать новые. Это означает, что мы будем использовать уже созданные объекты, а не создавать ещё больше, если объекты под наши нужны уже имеются. Также стоит обратить внимание, что приспособленцы имеют внутреннее и внешние состояние. Фабрика находит приспособленцев по внутреннему состоянию, а внешнее состояние передается в его методы.

[!] 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.

```
//flyweight.go
// Package flyweight is an example of the Flyweight Pattern.
package flyweight

import "fmt"

// Flyweighter interface
type Flyweighter interface {
	Draw(width, height int, opacity float64) string
}

// FlyweightFactory implements a factory.
// If a suitable flyweighter is in pool, then returns it.
type FlyweightFactory struct {
	pool map[string]Flyweighter
}

// GetFlyweight creates or returns a suitable Flyweighter by state.
func (f *FlyweightFactory) GetFlyweight(filename string) Flyweighter {
	if f.pool == nil {
		f.pool = make(map[string]Flyweighter)
	}
	if _, ok := f.pool[filename]; !ok {
		f.pool[filename] = &ConcreteFlyweight{filename: filename}
	}
	return f.pool[filename]
}

// ConcreteFlyweight implements a Flyweighter interface.
type ConcreteFlyweight struct {
	filename string // internal state
}

// Draw draws image. Args width, height and opacity is external state.
func (f *ConcreteFlyweight) Draw(width, height int, opacity float64) string {
	return fmt.Sprintf("draw image: %s, width: %d, height: %d, opacity: %.2f", f.filename, width,
height, opacity)
}
```

```
//flyweight_test.go
package flyweight

import (
	"strconv"
	"testing"
)

func TestFlyweight(t *testing.T) {
	var testCases = []struct {
		filename string
		width    int
		height   int
		opacity  float64
		expect   string
	}{
		{"cat.jpg", 100, 100, 0.95, "draw image: cat.jpg, width: 100, height: 100, opacity: 0.95"},
		{"cat.jpg", 200, 200, 0.75, "draw image: cat.jpg, width: 200, height: 200, opacity: 0.75"},
		{"dog.jpg", 300, 300, 0.50, "draw image: dog.jpg, width: 300, height: 300, opacity: 0.50"},
	}

	var factory = new(FlyweightFactory)

	for i, tt := range testCases {
		t.Run("case "+strconv.Itoa(i), func(t *testing.T) {
			var flyweight = factory.GetFlyweight(tt.filename)
			var result = flyweight.Draw(tt.width, tt.height, tt.opacity)
			if result != tt.expect {
				t.Errorf("Expect result to equal %s, but %s.\n", tt.expect, result)
			}
		})
	}
}
```

Заместитель (Proxy)

Паттерн Proxy относится к структурным паттернам уровня объекта.

Паттерн Proxy предоставляет объект для контроля доступа к другому объекту.

Другое название паттерна - "Суррогат". В этом понимании, это предмет или продукт, заменяющий собой какой-либо другой предмет или продукт, с которым суррогат имеет лишь некоторые общие свойства, но он не обладает всеми качествами оригинального предмета или продукта.

Паттерна Proxy выдвигается ряд важных требований, а именно то, что оригинальный объект и его суррогат должны взаимодействовать друг с другом, а также должна быть возможность, замещения оригинальным объектом, суррогата в месте его использования, соответственно интерфейсы взаимодействия оригинального объекта и его суррогата должны совпадать.

Вам будет легче понять паттерн, если вы смотрели фильм "Суррогаты".

Требуется для реализации:

1. Интерфейс Subject, являющейся общим интерфейсом для реального объекта и его суррогата;
2. Класс RealSubject, реализующий реальный объект;
3. Класс Proxy, реализующий объект суррогата. Хранит в себе ссылку на реальный объект, что позволяет заместителю обращаться к реальному объект напрямую;

Например, паттерн Proxy можно использовать, если нам нужно управлять ресурсоемкими объектами, но мы не хотим создавать экземпляры таких объектов до момента их реального использования.

Вы можете подумать, что это тоже самое, что и Adapter или Decorator. Но...

Proxy предоставляет своему объекту тот же интерфейс. Adapter предоставляет другой интерфейс. Decorator предоставляет расширенный интерфейс.

[!] 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.

```
//proxy.go
// Package proxy is an example of the Adapter Pattern.
package proxy

// Subject provides an interface for a real subject and its surrogate.
type Subject interface {
	Send() string
}

// Proxy implements a surrogate.
type Proxy struct {
	realSubject Subject
}

// Send sends a message
func (p *Proxy) Send() string {
	if p.realSubject == nil {
		p.realSubject = &RealSubject{}
	}
	return "<strong>" + p.realSubject.Send() + "</strong>"
}

// RealSubject implements a real subject
type RealSubject struct {
}

// Send sends a message
func (s *RealSubject) Send() string {
	return "I’ll be back!"
}
```

```
//proxy_test.go
package proxy

import (
    "testing"
)

func TestProxy(t *testing.T) {

    expect := "<strong>I'll be back!</strong>"

    proxy := new(Proxy)

    result := proxy.Send()

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

Спецификация (Specification)

Спецификация — шаблон проектирования, посредством которого представление правил бизнес логики может быть преобразовано в виде цепочки объектов, связанных операциями булевой логики.

Больше информации в Wikipedia https://en.wikipedia.org/wiki/Specification_pattern

```
//specification.go
// Pattern Specification
//
// In the following example, we are retrieving invoices and sending them to a collection
agency if
//  1. they are overdue,
//  2. notices have been sent, and
//  3. they are not already with the collection agency.
// This example is meant to show the end result of how the logic is 'chained' together.
//
// This usage example assumes a previously defined OverdueSpecification class
// that is satisfied when an invoice's due date is 30 days or older,
// a NoticeSentSpecification class that is satisfied when three notices
// have been sent to the customer, and an InCollectionSpecification class
// that is satisfied when an invoice has already been sent to the collection
// agency. The implementation of these classes isn't important here.

package specification

// Data for analysis
type Invoice struct {
Day    int
Notice int
IsSent bool
}
```

```
/////

// Invoice Specification Interface
type Specification interface {
	IsSatisfiedBy(Invoice) bool
	And(Specification) Specification
	Or(Specification) Specification
	Not() Specification
	Relate(Specification)
}

/////

// Invoice BaseSpecification
type BaseSpecification struct {
	Specification
}

// Check specification
func (self *BaseSpecification) IsSatisfiedBy(elm Invoice) bool {
	return false
}

// Condition AND
func (self *BaseSpecification) And(spec Specification) Specification {
	a := &AndSpecification{
		self.Specification, spec,
	}
	a.Relate(a)
	return a
}

// Condition OR
func (self *BaseSpecification) Or(spec Specification) Specification {
	a := &OrSpecification{
		self.Specification, spec,
	}
	a.Relate(a)
	return a
```

```
}

// Condition NOT
func (self *BaseSpecification) Not() Specification {
	a := &NotSpecification{
		self.Specification,
	}
	a.Relate(a)
	return a
}

// Relate to specification
func (self *BaseSpecification) Relate(spec Specification) {
	self.Specification = spec
}

/////

// AndSpecification
type AndSpecification struct {
	Specification
	compare Specification
}

// Check specification
func (self *AndSpecification) IsSatisfiedBy(elm Invoice) bool {
	return self.Specification.IsSatisfiedBy(elm) && self.compare.IsSatisfiedBy(elm)
}

/////

// OrSpecification
type OrSpecification struct {
	Specification
	compare Specification
}

// Check specification
func (self *OrSpecification) IsSatisfiedBy(elm Invoice) bool {
	return self.Specification.IsSatisfiedBy(elm) || self.compare.IsSatisfiedBy(elm)
```

```
}

/////

// NotSpecification
type NotSpecification struct {
	Specification
}

// Check specification
func (self *NotSpecification) IsSatisfiedBy(elm Invoice) bool {
	return !self.Specification.IsSatisfiedBy(elm)
}

/////

// Invoice's due date is 30 days or older
type OverDueSpecification struct {
	Specification
}

// Check specification
func (self *OverDueSpecification) IsSatisfiedBy(elm Invoice) bool {
	return elm.Day >= 30
}

// Constructor
func NewOverDueSpecification() Specification {
	a := &OverDueSpecification{&BaseSpecification{}}
	a.Relate(a)
	return a
}

// Three notices have been sent to the customer
type NoticeSentSpecification struct {
	Specification
}

// Check specification
func (self *NoticeSentSpecification) IsSatisfiedBy(elm Invoice) bool {
```

```
	return elm.Notice >= 3
}

// Constructor
func NewNoticeSentSpecification() Specification {
	a := &NoticeSentSpecification{&BaseSpecification{}}
	a.Relate(a)
	return a
}

// Invoice has already been sent to the collection agency.
type InCollectionSpecification struct {
	Specification
}

// Check specification
func (self *InCollectionSpecification) IsSatisfiedBy(elm Invoice) bool {
	return !elm.IsSent
}

// Constructor
func NewInCollectionSpecification() Specification {
	a := &InCollectionSpecification{&BaseSpecification{}}
	a.Relate(a)
	return a
}
```

```
//specification_test.go
package specification

import (
	"testing"
)

func TestSpecification(t *testing.T) {
	overDue := NewOverDueSpecification()
	noticeSent := NewNoticeSentSpecification()
	inCollection := NewInCollectionSpecification()

	sendToCollection := overDue.And(noticeSent).And(inCollection.Not())
```

```
	invoice := Invoice{
		Day:    31,    // >= 30
		Notice: 4,     // >= 3
		IsSent: false, // false
	}

	// true!
	result := sendToCollection.IsSatisfiedBy(invoice)

	if !result {
		t.Errorf("Expect result to equal %v, but %v.\n", false, true)
	}
}
```