

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

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. Вообще, 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)
- Мост (Bridge)
- Компоновщик (Composite)
- Декоратор (Decorator)
- Фасад (Facade)
- Приспособленец (Flyweight)
- Заместитель (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 классов. 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 случае 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 вместо 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 "SssuuuuZzzuuuuKkiiii"
}

// 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 := "SssuuuuZzzuuuuKiiiiii"
```

```
    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 становится 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)
    }
}
```

0}

}

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 внешнее 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"

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

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

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

// ConcreteFlyweight implements a Flyweight 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)  
    }  
}
```