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

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)
- Строитель (Builder)
- Фабричный метод (FactoryMethod)
- Прототип (Prototype)
- Одиночка (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")
    }
}
```