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

# 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)
		}
	}
}
```

---

Revision #1

Created 2022-07-03 10:24:44 UTC by gasick

Updated 2022-07-03 10:25:24 UTC by gasick