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

????????? (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)
	}
}
```

Revision #1

Created 2022-07-03 10:23:20 UTC by gasick

Updated 2022-07-03 10:24:38 UTC by gasick