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

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)
	}
}
```

Revision #3

Created 2022-07-03 10:10:56 UTC by gasick

Updated 2023-09-28 19:19:04 UTC by gasick