go-design-patterns

steven

Published
with GitBook

Table of Contents

- Introduction 0
- Creational Patterns 1
  - Abstract Factory 1.1
  - Builder 1.2
  - Factory Method 1.3
  - Prototype 1.4
  - Singleton 1.5
- Structural Patterns 2
  - Adapter 2.1
  - Bridge 2.2
  - Composite 2.3
  - Decorator 2.4
  - Facade 2.5
  - Flyweight 2.6
  - Proxy 2.7
- Behavioral Patterns 3
  - Chain of Responsibility 3.1
  - Command 3.2
  - Interpreter 3.3
  - Iterator 3.4
  - Mediator 3.5
  - Observer 3.6
  - State 3.7
  - Strategy 3.8
  - Template Method 3.9
  - Visitor 3.10

My Awesome Book

This file file serves as your book's preface, a great place to describe your book's content and ideas.

Creational Patterns

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

Abstract Factory

Builder

Intent The intention is to abstract steps of construction of objects so that different implementations of these steps can construct different representations of objects.

Example Now we have three type messages, different message has it own build procedure

- command message `command|message body`
- message with headers `header1:value1 header2:value2|message body`
- message with multiparts `message body1|message body2|message body3`

```
type MessageBuilder interface {
    BuildMessage(s String) Message
}
```

> Message base methods, we want process those messages

```
type Message interface {
    Process()
}

type CommandMessage struct {
    command String
    body String
}
func (c *CommandMessage)Process(){
}
func (c *CommandMessage)BuildMessage(s String) Message {
    c.command = //decode command from s
    c.body = //decode body from s

    return c
}

type HeaderMessage struct {
    headers map[String]String
```

```
    body String
}
func (c *HeaderMessage)Process(){
}
func (c *HeaderMessage)BuildMessage(s String) Message {
    //decode headers from s
    c.body = //decode body from s

    return c
}

type MultipartMessage struct {
    body []String
}
func (c *MultipartMessage)Process(){
}
func (c *MultipartMessage)BuildMessage(s String) Message {
    //decode bodys from s
    return c
}
```

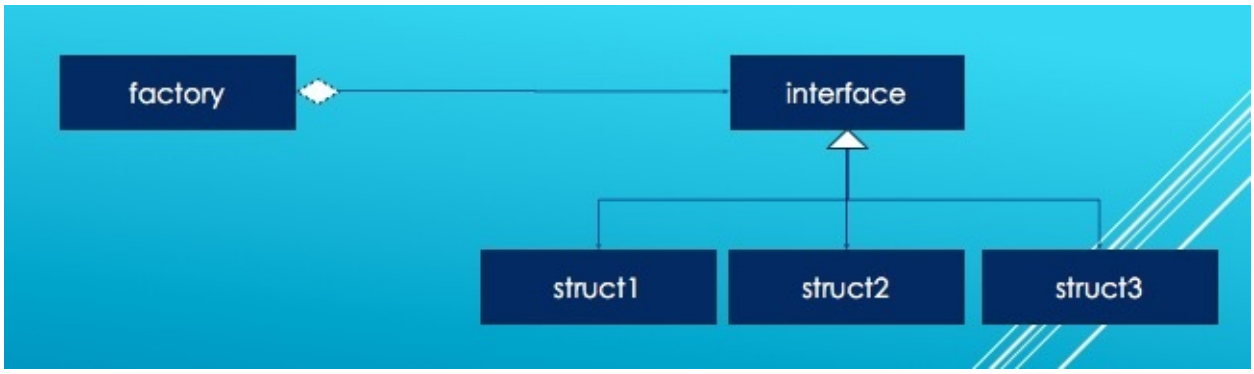
```
func BuildMessage(m MessageBuilder, s String) Message {
    return m.BuildMessage(s)
}
```

Client

```
func main() {
    m := BuildMessage(&MultipartMessage{}, "body1|body2|body3")
    m.Process()
    //...
}
```

Factory Method

Intent Define an interface for creating an object, but let sub classes decide which type to instantiate.



Example We often need to handle different type messages, in this case we use factory method to process three type messages.

type 1: text message, we need save text message to database.

type 2: image massage, we need save image massage to file system.

type 3: audio message, we need discard those messages.

```
type Message interface {
    Process()
}

type TextMessage struct {
    type String
    body String
}
func (t *TextMessage) Process() {
    //save to database
}

type ImageMessage struct {
    type String
    body String
}
func (t *TextMessage) Process() {
    //save to file system
}

type AudioMessage struct {
    type String
    body String
}
func (t *TextMessage) Process() {
    //discard
}
```

Message factory return the real instance.

```
func MessageFactory(message) *Message {
    var m *Message
    type := GetType(message)
    switch type {
    case "text":
        m = TextMessage{type:type, body:message}
    case "image":
        m = ImageMessage{type:type, body:message}
    case "audio":
        m = AudioMessage{type:type, body:message}
    }

    return m
}
```

Client:

```
func main() {
    messages = ["text:hello world", "image:this is image", "audio:t

    for m in messages {
        message := MessageFactory(m)
        message.Process()
    }
}
```

Prototype

Intent Create objects by cloning a prototypical instance.

```
type Room struct {
    length int
    wigth int
}

func(r *Room) Clone() Room {
    room := Room{}
    room.length = r.length
    room.wigth = r.wigth

    return room
}
```

Singleton

Intent Ensure that only a single instance of a type exists in a program, and provides a global access point to that object.

Example We want to count how many times user requests a page.

```
package count

type count struct {
    times int
}

func (c *count)Add() {
    c.times++
}

var instance *count

func Get() *count{
    if instance == nil {
        instance = &count{times:0}
    }
    return instance
}
```

```
import (
    "count"
    "fmt"
)

func main() {
    c := count.Get()
    c.Add()

    d := count.Get()
    d.Add()

    e := count.Get()
    e.Add()

    fmt.Println(e.times)
}
```

Structural Patterns

Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

Behavioral Patterns

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Observer

State

Strategy

Template Method

Visitor