Refactoring with Go

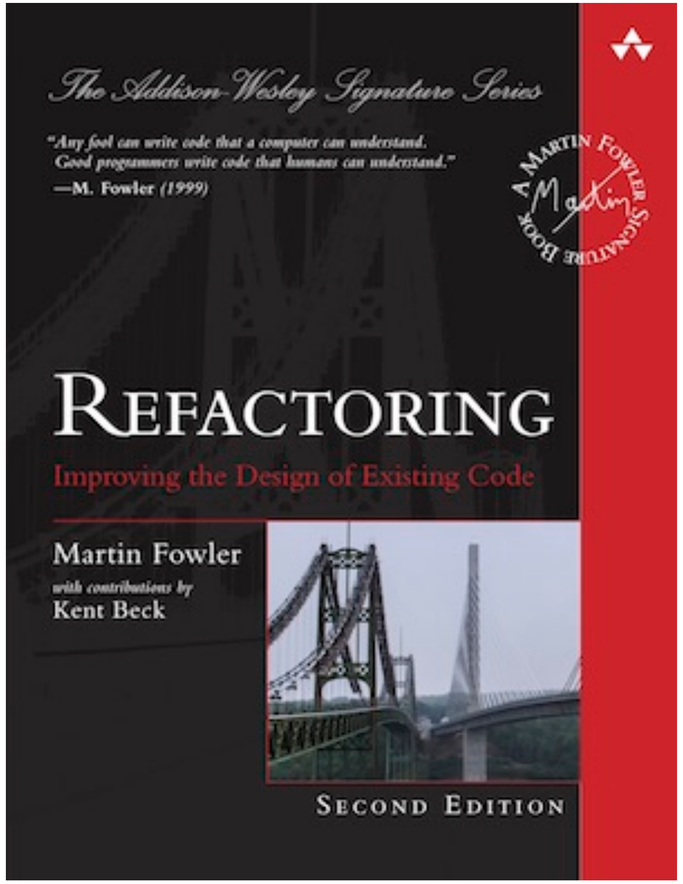
Improving the design without alter the external behavior

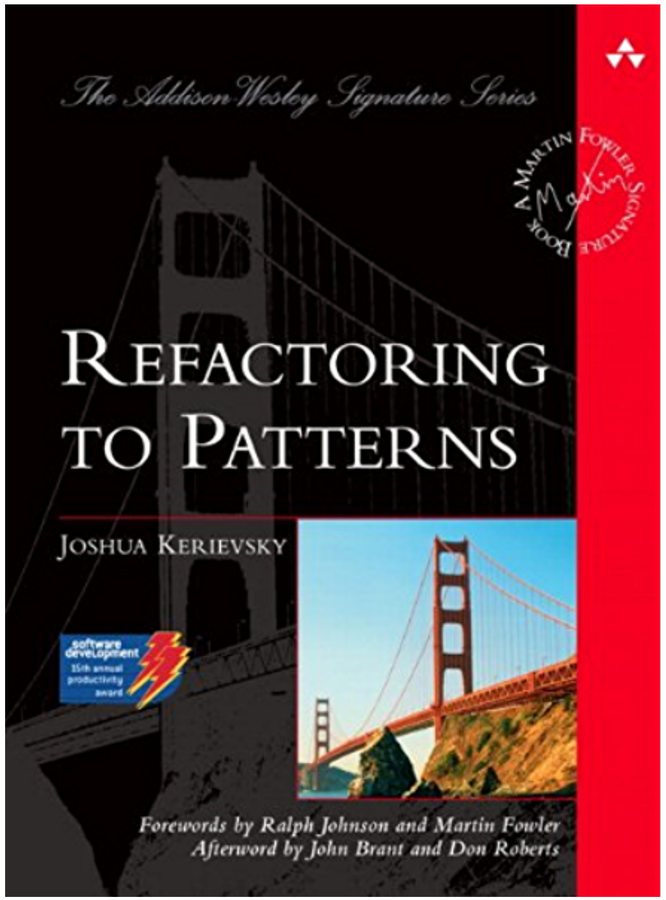
AnuchitO

Intro

- แนะนำตัว
  - จับคู่ แนะนำเพื่อน (งานอดิเรก, เป็นคนที่ไหน, ชอบภาษาอะไรเพราะอะไร)
- เขียนโกมานานแค่ไหน ปี เดือน
- เขียนภาษาอะไรมาบ้าง (1โพสอิท ต่อ 1 ภาษา)
- ความคาดหวัง

Reference books





How do you feel about this code?

Code Smells?

Code Smells

- by definition something that's **quick to spot**
- they are often an **indicator** of a problem rather than the problem themselves.
- Signs of your code **needs improving**

Classic Code Smells

Alternative Classes with Different Interfaces

Comments

Data Class

Data Clumps

Dead Code

Divergent Change

Duplicate Code

Feature Envy

Inappropriate Intimacy

Incomplete Library Class

Large Class

Lazy Class

Long Method

Long Parameter List

Message Chains

Middle Man

Parallel Inheritance Hierarchies

Primitive Obsession

Refused Bequest

Shotgun Surgery

Speculative Generality

Switch Statements

Temporary Field

Workshop - code smell presentation

1. Name of code smell? (three smells per group)
2. How can you spot it?
3. why is it bad?
4. Show some examples of the code smell.

Research: 45 minutes

Presentation: 10 minutes per group

Resource : https://refactoring.guru/refactoring/smells

Classic Code Smells

Alternative Classes with Different Interfaces

Comments

Data Class

Data Clumps

Dead Code

Divergent Change

Duplicate Code

Feature Envy

Inappropriate Intimacy

Incomplete Library Class

Large Class

Lazy Class

Long Method

Long Parameter List

Message Chains

Middle Man

Parallel Inheritance Hierarchies

Primitive Obsession

Refused Bequest

Shotgun Surgery

Speculative Generality

Switch Statements

Temporary Field

Classic Code Smells

Alternative Classes with Different Interfaces

Comments

Data Class

Data Clumps

Dead Code

Divergent Change

Duplicate Code

Feature Envy

Inappropriate Intimacy

Incomplete Library Class

Large Class

Lazy Class

Long Method

Long Parameter List

Message Chains

Middle Man

Parallel Inheritance Hierarchies

Primitive Obsession

Refused Bequest

Shotgun Surgery

Speculative Generality

Switch Statements

Temporary Field

Comments

Data Class

Lazy Class

Dead Code

Speculative Generality

Duplicate Code

Message Chains

Middle Man

Feature Envy

Inappropriate Intimacy

Incomplete Library Class

Parallel Inheritance Hierarchies

Shotgun Surgery

Divergent Change

Large Class

Data Clumps

Long Method

Long Parameter List

Primitive Obsession

Alternative Classes with Different Interfaces

Switch Statements

Temporary Field

Refused Bequest

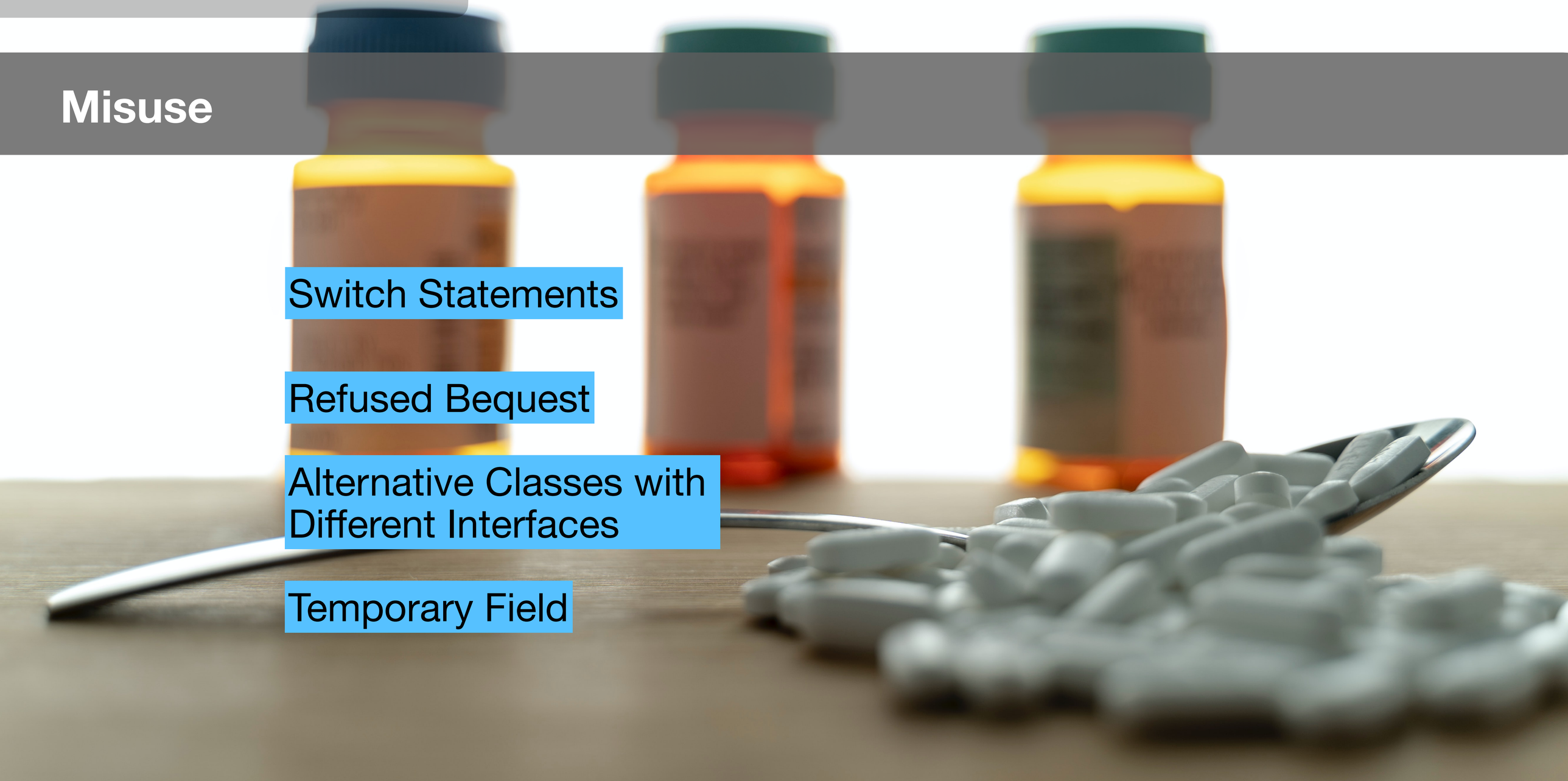
Bloaters

Do not need to be that BIG

- Large Class
- Long Method
- Data Clumps
- Long Parameter List
- Primitive Obsession

Tool Abusers

Misuse



Switch Statements

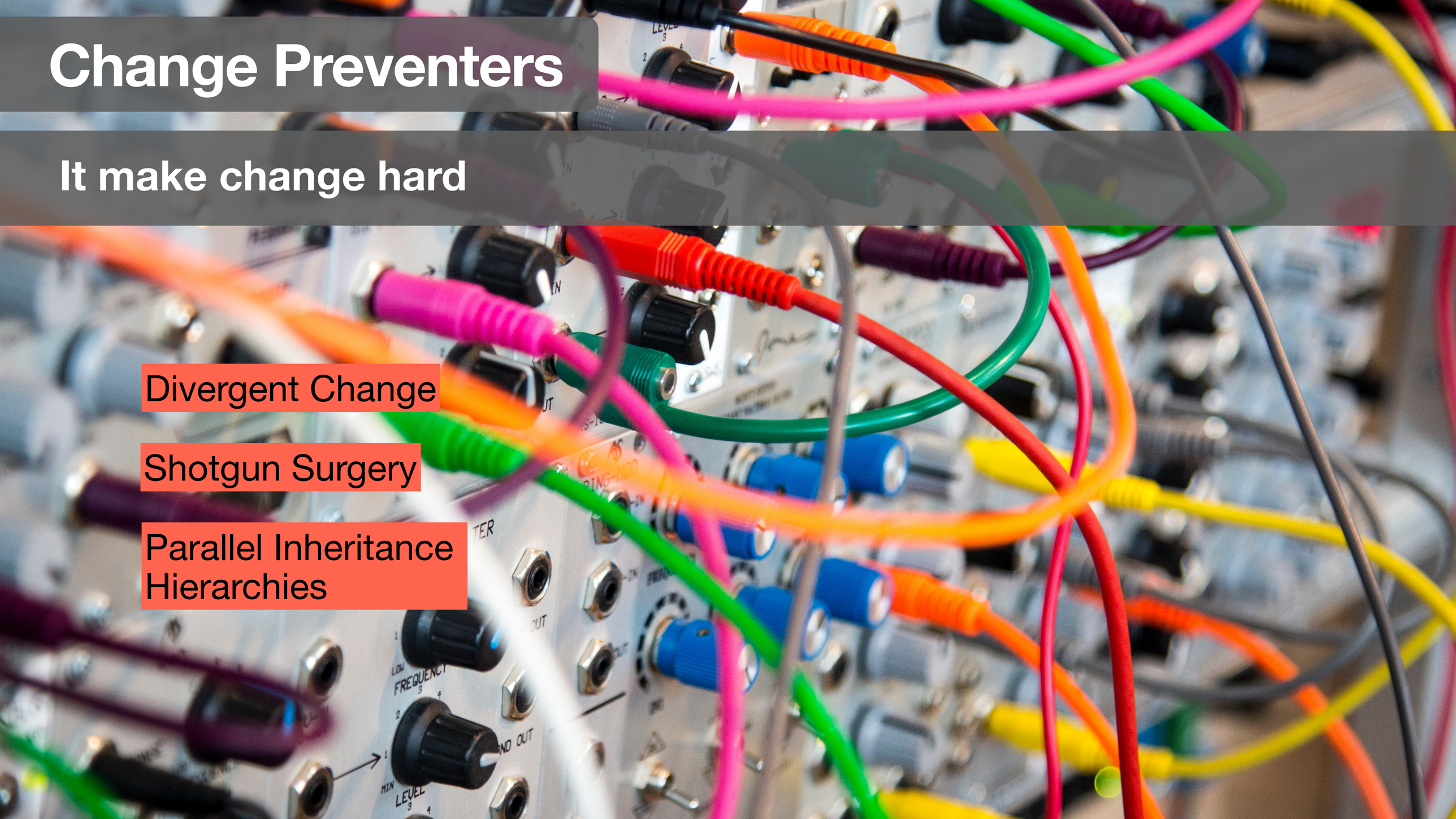
Refused Bequest

Alternative Classes with Different Interfaces

Temporary Field

Change Preventers

It make change hard



Dispensables

something pointless and unneeded

- Comments
- Data Class
- Lazy Class
- Dead Code
- Speculative Generality
- Duplicate Code

Couplers

Too much depend on each other

- Feature Envy
- Inappropriate Intimacy
- Middle Man
- Message Chains

How many code smells?

Refactoring

Defining Refactoring

(noun): a **change** made to the internal **structure** of software to make it easier to understand and cheaper to modify **without changing** its observable **behavior**.

(verb): to **restructure** software by **applying** a series of **refactorings** without changing its observable behavior.

— Martin Fowler

“Behavior-preserving transformation”

— Refactoring to Patterns

When to Refactor?

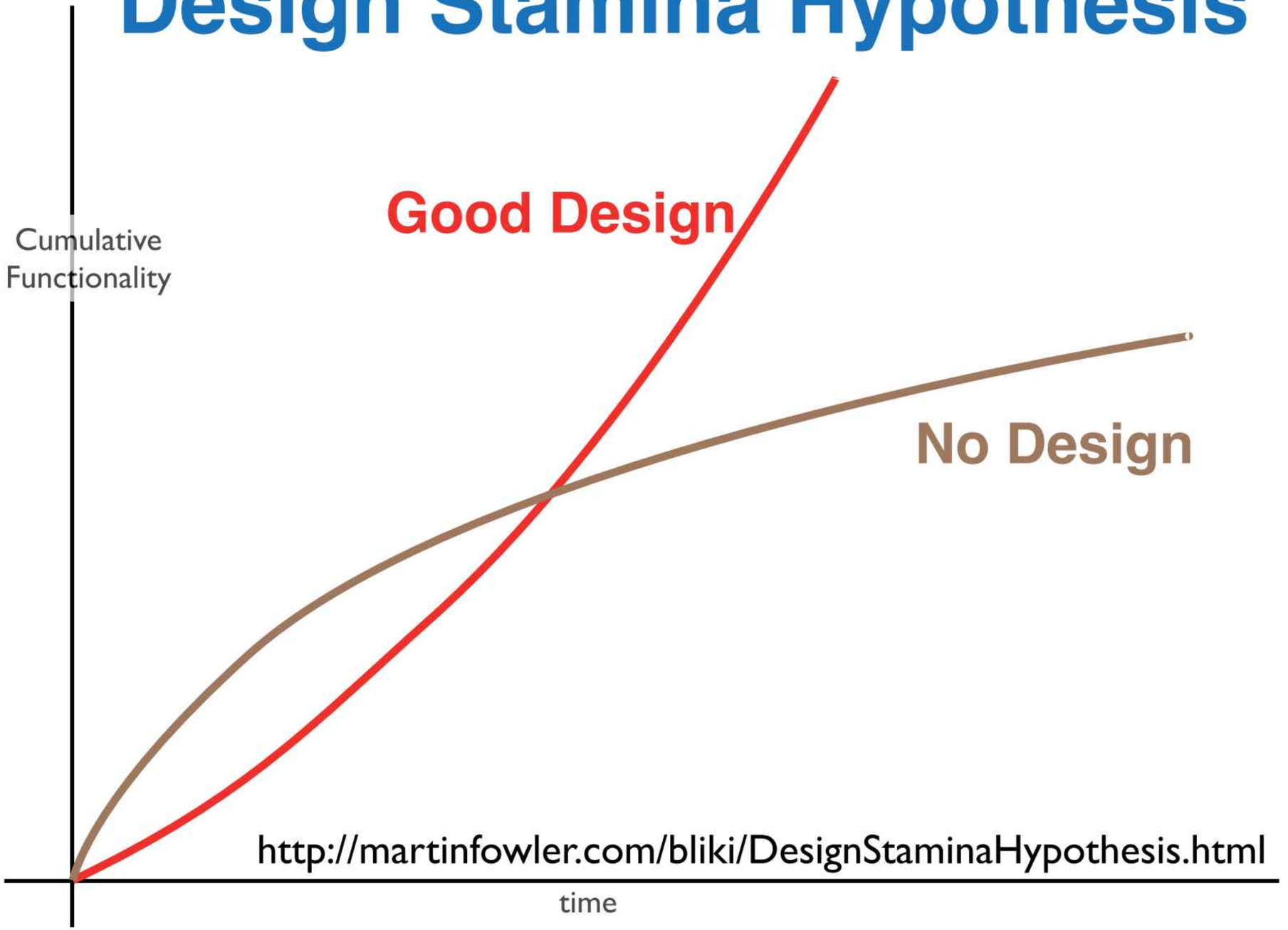
When code have have tests
and they are green!

Refactoring is done during development!!

Why Refactor?

- Clean Code!
- Quality!
- Professionalism!
- Right Thing!
- improve the code!

Design Stamina Hypothesis



Clear is better than clever

—Go Proberbs

Inline Function

Mechanics

1. Check that this isn’t a polymorphic method
2. Find all the callers of the function
3. Replace each call with the function’s body
4. Test after each replacement
5. Remove the function definition

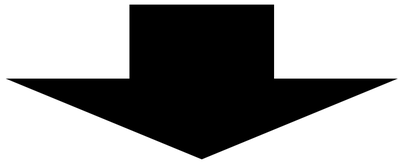
```
func getRating(driver Driver) int {
  if moreThanFiveLateDeliveries(driver) {
    return 1
  }
  return 2
}

func moreThanFiveLateDeliveries(driver Driver) bool {
  return driver.numberOfLateDeliverires > 5
}
```

```
func getRating(driver Driver) int {
  if driver.numberOfLateDeliverires > 5 {
    return 1
  }
  return 2
}
```

Extract Function (106)

1. Create new function
2. Copy the extracted code from the source function into the new function
3. Scan the extracted code for references to any variables that are local in scope
4. Compile after all variables are dealt with
5. Replace the extracted code in the source function
6. Test
7. Look for other code that’s the same or similar to the code just extracted
   and consider using “Replace Inline Code with Function Call”

```go
func printOwing(invoice Invoice) {
  outstanding := calculateOutstanding()

  //print details
  fmt.Println("name:", invoice.Customer)
  fmt.Println("oustanding:", outstanding)
}
```



```go
func printOwing(invoice Invoice) {
  outstanding := calculateOutstanding()

  printDetails(invoice, outstanding)
}

func printDetails(invoice Invoice, outstanding int) {
  fmt.Println("name:", invoice.Customer)
  fmt.Println("oustanding:", outstanding)
}
```

Any fool can write code that a computer can understand.

Good Programmers write code that **humans** can **understand.**

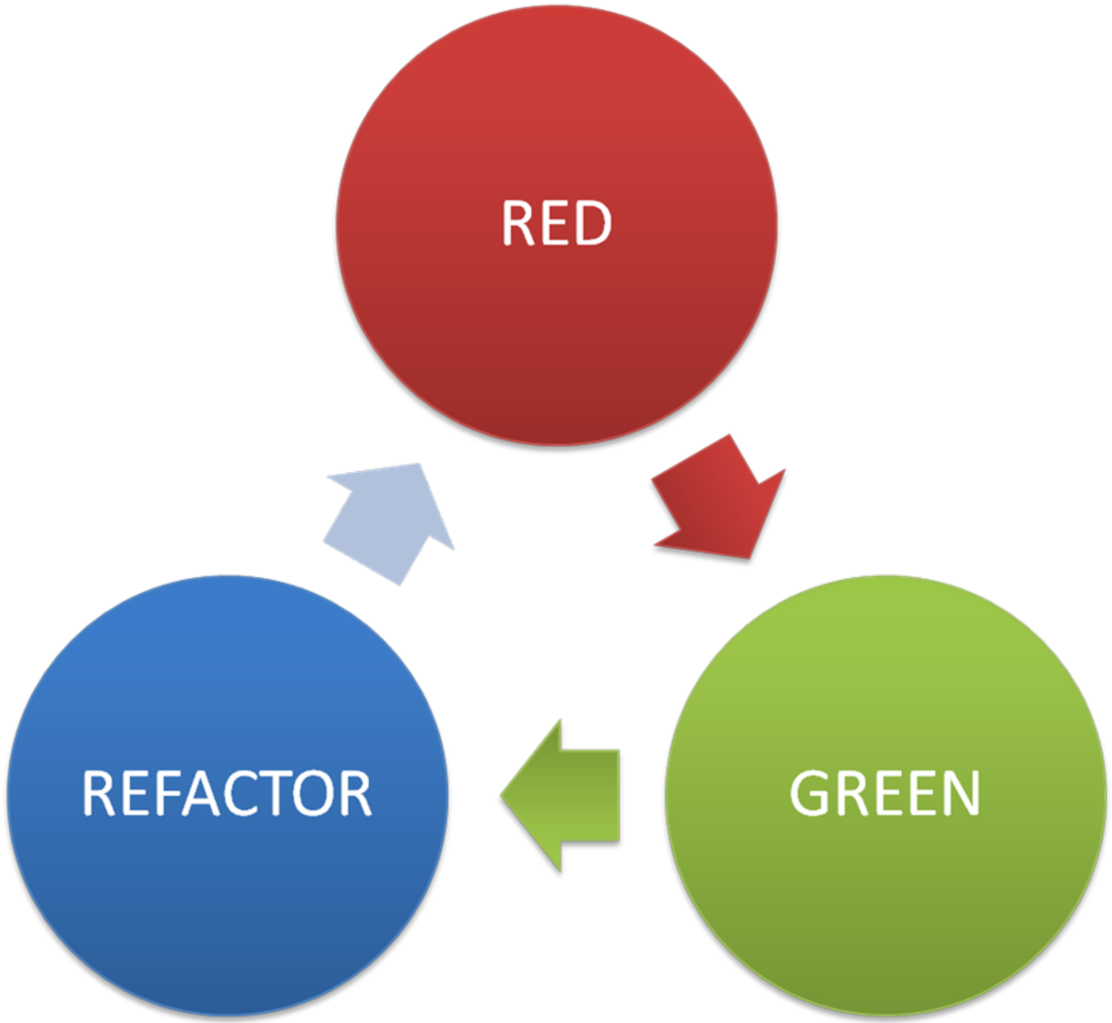
Refactoring 4C

- Create - create new definition
- Comment - comment old code
- Call - call a new definition
- Clean - remove old code

How to Refactor

1. Find some code smells
2. Make sure you have solid set of tests for that section of code and passing
3. Make a code change
4. Do Refactor step and do NOT rewrite
5. Make sure the tests pass
6. Repeat the TDD cycle until the smell is gone

Test-driven development (TDD)



To be Continue

Let’s Refactoring

Replace Nested Conditional with Guard Clauses

```
func getPayAmount() int {
  result := 0
  if isDead {
    result = deadAmount()
  } else {
    if isSeparated {
      result = separatedAmount()
    } else {
      if isRetired {
        result = retiredAmount()
      } else {
        result = normalPayAmount()
      }
    }
  }
  return result
}
```

```
func getPayAmount() int {
  if isDead {
    return deadAmount()
  }
  if isSeparated {
    return separatedAmount()
  }
  if isRetired {
    return retiredAmount()
  }
  return normalPayAmount()
}
```

Early return

```go
func totalAmount(amounts []int) int {
  if amounts != nil {
    total := 0
    for _, a := range amounts {
      total += a
    }
    return  total
  }
  return 0
}
```

```go
func totalAmount(amounts []int) int {
  if amounts == nil {
    return 0
  }

  total := 0
  for _, a := range amounts {
    total += a
  }
  return  total
}
```

Gilded rose

Safety net

Sulfuras

Aged Brie

Backstage passes

Dexterity Vest

Conjured

Let’s Negate if

First class function

```go
func WithBasicSetup(c *Config) {
  c.httpClient = &http.Client{}
}

type Config struct {
  endpoint   string
  httpClient *http.Client
}

func NewConn(options ...func(*Config)) *Conn {
  var c Conn
  for _, option := range options {
    option(&c.config)
  }
  return &c
}
```

```go
func main() {
  c := NewConn(WithBasicSetup)
  // ...
}
```

```go
func WithBasicSetup(c *Config) {
  c.httpClient = &http.Client{}
}

type Config struct {
  endpoint   string
  httpClient *http.Client
}

func NewConn(options ...func(*Config)) *Conn {
  var c Conn
  for _, option := range options {
    option(&c.config)
  }
  return &c
}
```

```go
func main() {
  c := NewConn(WithBasicSetup)
  // ...
}
```

```go
func WithEndpoint(url string) func(c *Config) {
  return func(c *Config) {
    c.endpoint = url
  }
}
```

```go
func main() {
  c := NewConn(
    WithBasicSetup,
    WithEndpoint("/resources"),
  )
  // ...
}
```

```go
func WithEndpoint(url string) func(c *Config) {
  return func(c *Config) {
    c.endpoint = url
  }
}
```

```go
func main() {
  c := NewConn(
    WithBasicSetup,
    WithEndpoint("/resources"),
  )
  // ...
}
```

Function as first class values

```
package math
func Min(a, b float64) float64
```

```
package bytes
func NewReader(b []byte) *Reader
```

```
func WithEndpoint(url string) func(*Config)
```

```go
package conn

type Option interface {
  Apply(*Config)
}

func NewConn(options ...Option) *Conn {
  var config Config
  for _, option := range options {
    option.Apply(&config)
  }
  return &Conn{config}
}
```

```go
type withToken struct {
  token string
}

func (t *withToken) Apply(c *conn.Config) {
  c.token = t.token
}

func WithToken(token string) conn.Option {
  return &withToken{
    token: token,
  }
}
```

```go
func main() {
  c := NewConn(
    WithToken("secret token"),
  )
  // ...
}
```

Function as first class values

```
type Calculator struct {
  acc float64
}
const (
  OP_ADD = 1 << iota
  OP_SUB
  OP_MUL
)
func (c *Calculator) Do(op int, v float64) float64 {
  switch op {
  case OP_ADD:
    c.acc += v
  case OP_SUB:
    c.acc -= v
  case OP_MUL:
    c.acc *= v
  default:
    panic("unhandled operation")
  }
  return c.acc
}
```

```
func main() {
  var c Calculator
  fmt.Println(c.Do(OP_ADD, 100))      // 100
  fmt.Println(c.Do(OP_SUB, 50))       // 50
  fmt.Println(c.Do(OP_MUL, 2))        // 100
}
```

rewrite our calculator a little

```
type Calculator struct {
  acc float64
}

type opfunc func(float64, float64) float64

func (c *Calculator) Do(op opfunc, v float64) float64 {
  c.acc = op(c.acc, v)
  return c.acc
}

func Add(a, b float64) float64 { return a + b }

func Sub(a, b float64) float64 { return a - b }
func Mul(a, b float64) float64 { return a * b }

func main() {
  var c Calculator
  fmt.Println(c.Do(Add, 5))        // 5
  fmt.Println(c.Do(Sub, 3))        // 2
  fmt.Println(c.Do(Mul, 8))        // 16
  c.Do(Sqrt, 0) // operand ignored
}

func Sqrt(n, _ float64) float64 {
  return math.Sqrt(n)
}
```

Functional : Only Functions with 0 or 1 Arguments

```
func main() {
  var c Calculator
  c.Do(Add(1))     // 1
  c.Do(Add(1))     // 2
  c.Do(Sqrt())     // 1.41421356237
  c.Do(math.Sqrt) // 1.41421356237
  c.Do(math.Cos)   // 0.99969539804
}
```

```
type Calculator struct {
  acc float64
}

func (c *Calculator) Do(op func(float64) float64) float64 {
  c.acc = op(c.acc)
  return c.acc
}

func Add(n float64) func(float64) float64 {
  return func(acc float64) float64 {
    return acc + n
  }
}

func Sub(n float64) func(float64) float64 {
  return func(acc float64) float64 {
    return acc - n
  }
}

func Mul(n float64) func(float64) float64 {
  return func(acc float64) float64 {
    return acc * n
  }
}

func Sqrt() func(float64) float64 {
  return func(n float64) float64 {
    return math.Sqrt(n)
  }
}
```

Write in Go


SOLID Principle

SOLID Principle

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle (SRP)

A class should have one, and only one, reason to change.

–Robert C Martin

Coupling & Cohesion

Coupling — describes two things changing together

Cohesion — related, but separate, notion, a force of mutual attraction.

Package name in Go

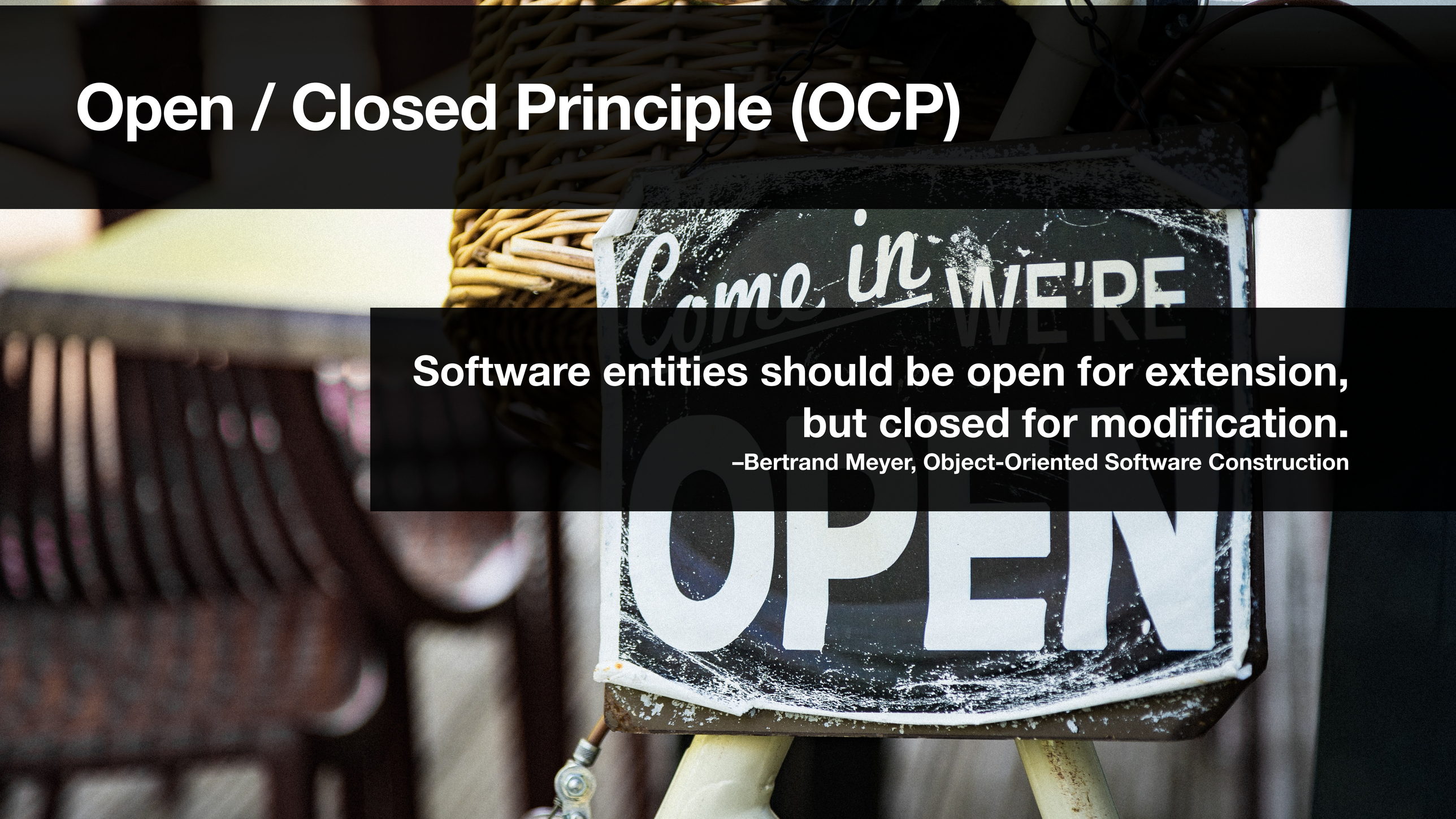
- **net/http** — provides http clients and servers.
- **os/exec** — runs external commands.
- **encoding/json** — implements encoding/decoding of JSON.

Bad package names

- package server
- package private
- package common
- package utils

Go package is itself a small Go program,
a single unit of change, with a single responsibility.

Open / Closed Principle (OCP)



Software entities should be open for extension, but closed for modification.

–Bertrand Meyer, Object-Oriented Software Construction

Open for extension

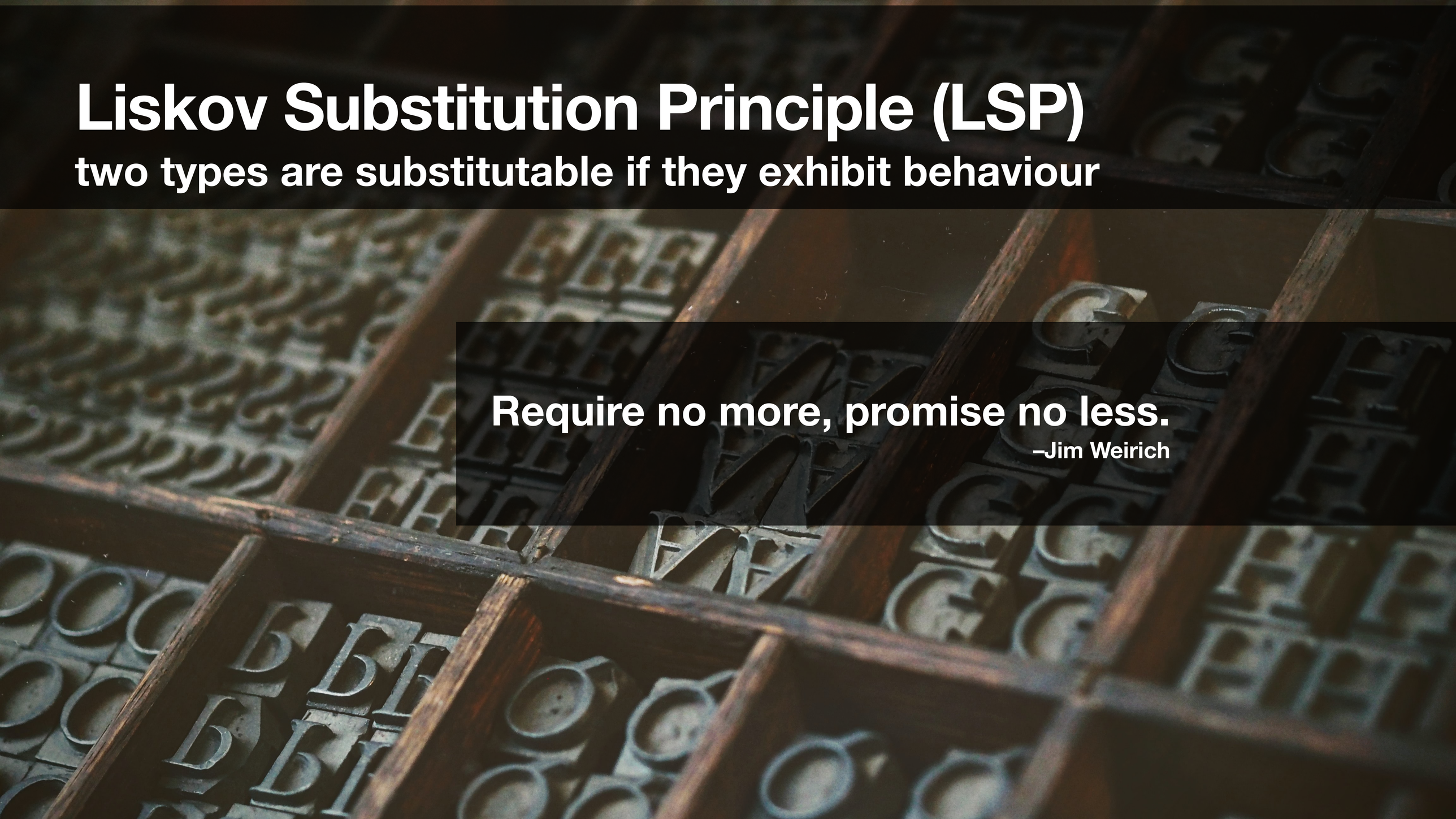
```go
package main

import "fmt"

type A struct {
        member int
}

func (a A) Sawadee() { fmt.Println("Sawadee!!", a.member) }

type B struct {
        A
}

func (b B) Sawadee() { fmt.Println("ซ่ำบายดี!!", b.member) }

func main() {
        var a A
        a.member = 2021
        var b B
        b.member = 2021
        a.Sawadee() // Sawadee!! 2021
        b.Sawadee() // ซ่ำบายดี!! 2021
}
```

Cat

Octocat

Closed for modification

```go
package main

import "fmt"

type Cat struct {
        Name string
}

func (c Cat) Legs() int { return 4 }

func (c Cat) PrintLegs() {
        fmt.Printf("I have %d legs\n", c.Legs())
}

type OctoCat struct {
        Cat
}

func (o OctoCat) Legs() int { return 5 }

func main() {
        var octo OctoCat
        fmt.Println(octo.Legs()) // 5
        octo.PrintLegs()         // I have 4 legs
}
```



Liskov Substitution Principle (LSP)

two types are substitutable if they exhibit behaviour

Require no more, promise no less.

–Jim Weirich

Interface

Small Interface

Simple Implementations

Common Behavior

Reader Interface

```
type Reader interface {
        // Read reads up to len(buf) bytes into buf.
        Read(buf []byte) (n int, err error)
}
```

Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use.

–Robert C Martin

```
// Save writes the contents of doc to the file f.
func Save(f *os.File, doc *Document) error
```

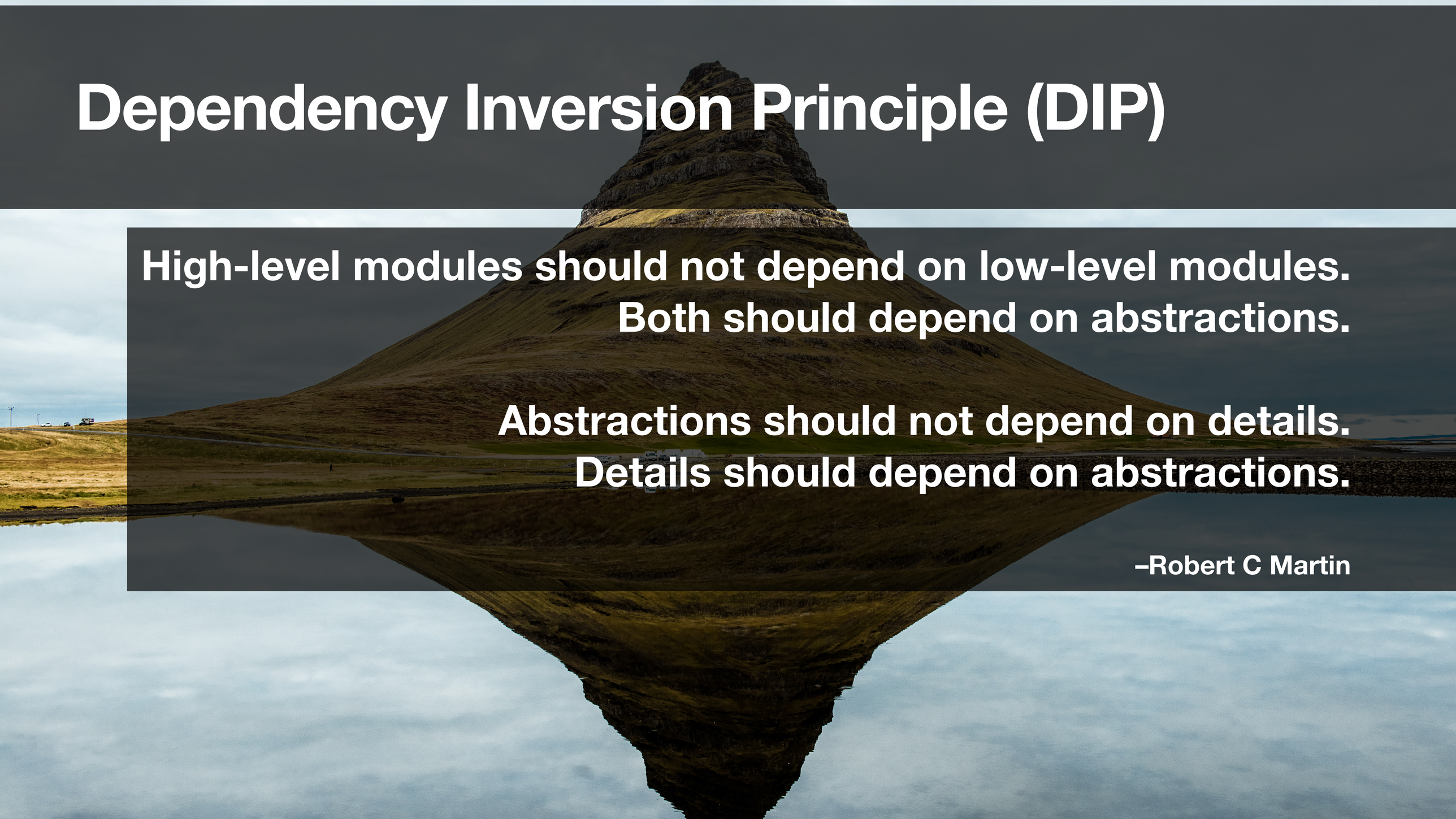
```go
// Save writes the contents of doc to the supplied ReadWriterCloser.
func Save(rwc io.ReadWriteCloser, doc *Document) error
```

```go
// Save writes the contents of doc to the supplied WriteCloser.
func Save(wc io.WriteCloser, doc *Document) error
```

```
type NopCloser struct {
        io.Writer
}

// Close has no effect on the underlying writer.
func (c *NopCloser) Close() error { return nil }
```

```go
// Save writes the contents of doc to the supplied Writer.
func Save(w io.Writer, doc *Document) error
```

Dependency Inversion Principle (DIP)

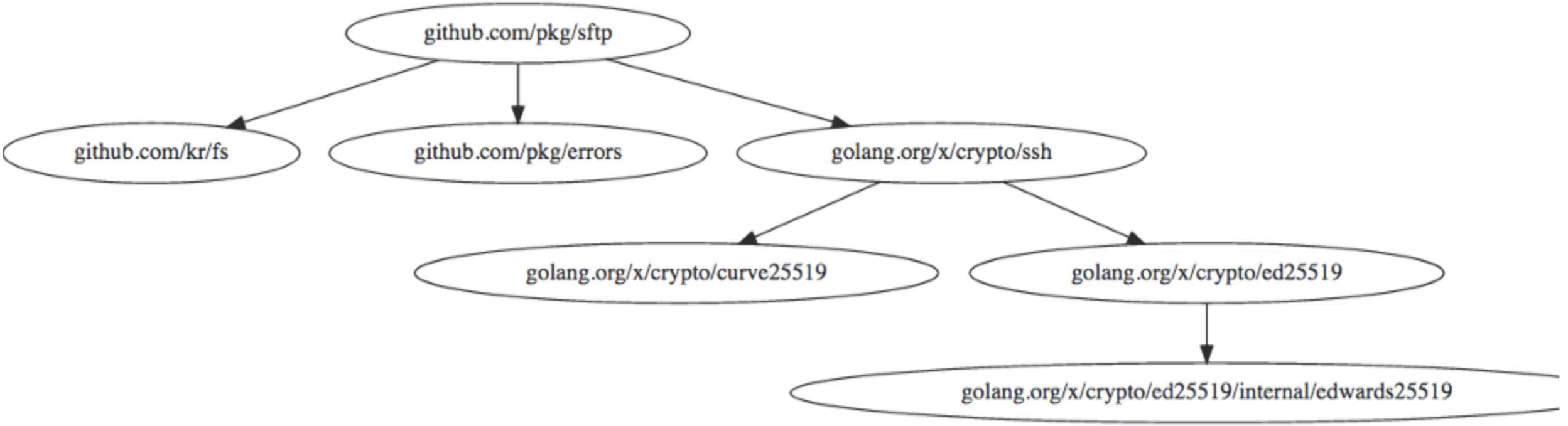


High-level modules should not depend on low-level modules.
Both should depend on abstractions.

Abstractions should not depend on details.
Details should depend on abstractions.

–Robert C Martin

Dependency graph



SOLID in GO

Single Responsibility Principle

Encourage to **structure** function, types, methods
into packages that exhibit natural **cohesion** the **types** belong together
serve a **single purpose** they want to be together

SOLID in GO

Open / Closed Principle

Encourages to **compose** simple **types** into more complex ones
using embedding

SOLID in GO

Liskov Substitution Principle

Encourages **express** the **dependencies** between packages
in terms of interfaces not concrete types by defining **small interfaces**
we can be more **confident** the **implementations** will faithfully **satisfy** that **contract**

SOLID in GO

Interface Segregation Principle

Take LSP further and encourages to **define functions** and **methods**
that **depend only on** the **behavior** that they **need** and
if a function requires a parameter of an interface type with a single method
then it is more likely that function has only on responsibility

SOLID in GO

Dependency Inversion Principle

encourages to **push** the **responsibility** for the specifics,
as high as possible **up** the import graph,
leaving the lower level code to deal with abstractions–interfaces

Interface allow us to apply the SOLID principles to Go programs

Design is the art of arranging code that needs to work today, and to be easy to change forever.

–Sandi Metz

Go programmers need to start **talking** less about frameworks,
and start talking **more** about **design**.
We need to stop focusing on performance at all cost,
and **focus** instead on **reuse** at all cost.

—Dave Cheney


Design Patterns in Go

Factory

Decorator

Builder

Strategy

Observer

Adapter

Package Design

- Who is the user of the package?
- What are they trying to do?
- Why are they doing ti?
- Why are they using your package?

Package names

Good package names are short and clear. They are lower case, with no under_scores or mixedCaps. They are

- package **time** (provides functionality for measuring and displaying time)
- package **list** (implements a doubly linked list)
- package **http** (provides HTTP client and server implementations)

ย่ออย่างรอบคอบ

- package **strconv** (string conversion)
- package **syscall** (system call)
- package **fmt** (formatted I/O)

อย่าให้ package แย่งชื่อ

buffered I/O ชื่อ package **bufio** ไม่ชื่อ buf, เพราะ buf เอาไว้ตั้งชื่อตัวแปรของ buffer

Naming package contents

- **Avoid stutter** เช่น http.Server ไม่ใช่ http.HTTPServer
- **Simplify function names (pkg.Pkg หรือ *pkg.Pkg)**

```
start := time.Now()                                  // start is a time.Time
t, err := time.Parse(time.Kitchen, "6:06PM")         // t is a time.Time
ctx = context.WithTimeout(ctx, 10*time.Millisecond)  // ctx is a context.Context
ip, ok := userip.FromContext(ctx)                    // ip is a net.IP
q := list.New()                                      // q is a *list.Lis
```

Naming package contents

- **Simplify function names (pkg.T หรือ *pkg.T)**

```
d, err := time.ParseDuration("10s")    // d is a time.Duration
elapsed := time.Since(start)           // elapsed is a time.Duration
ticker := time.NewTicker(d)            // ticker is a *time.Ticker
timer := time.NewTimer(d)              // timer is a *time.Timer
```

Package paths

```
import (
    "context"                       // package context
    "fmt"                           // package fmt
    "golang.org/x/time/rate"        // package rate
    "os/exec"                       // package exec
)
```

```
package  runtime/pprof

package  net/http/pprof
```

Bad package

- package **computeServiceClient**
- package **priority_queue**

Bad package

- Avoid meaningless package names.
  Packages named **util**, **common**, or **misc**

Bad package

```
package util
func NewStringSet(...string) map[string]bool {...}
func SortStringSet(map[string]bool) []string {...}
```

client code

```
set := util.NewStringSet("c", "a", "b")
fmt.Println(util.SortStringSet(set))
```

Bad package

```
package util
func NewStringSet(...string) map[string]bool {...}
func SortStringSet(map[string]bool) []string {...}
```

ควรจะเป็นอย่างไร?

client code

```
set := util.NewStringSet("c", "a", "b")
fmt.Println(util.SortStringSet(set))
```

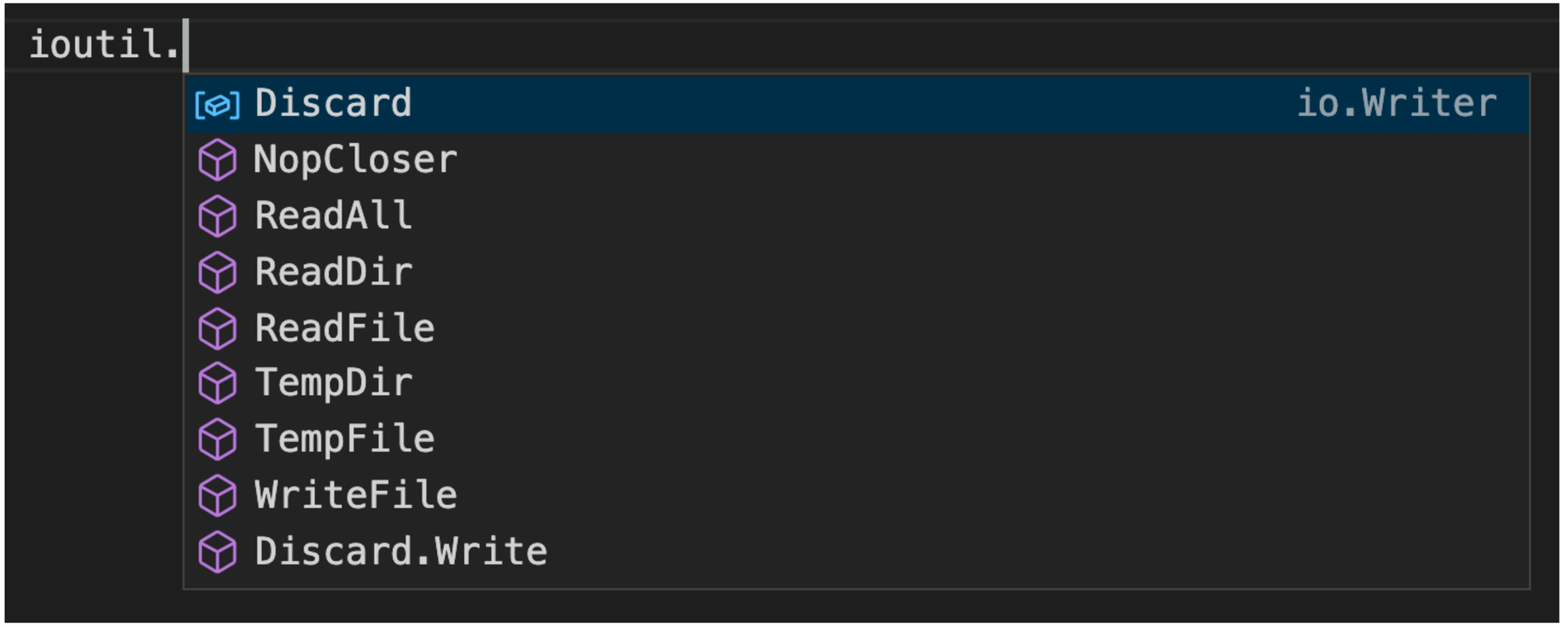
Rearrange

```
package stringset
func New(...string) map[string]bool {...}
func Sort(map[string]bool) []string {...}
```

client code becomes

```
set := stringset.New("c", "a", "b")
fmt.Println(stringset.Sort(set))
```

Smaller are better



```go
// Save writes the contents of doc to the file f.
func Save(f *os.File, doc *Document) error
```

```go
// Save writes the contents of doc to the supplied Writer.
func Save(w io.Writer, doc *Document) error
```

io.Writer

os.File

bytes.Buffer

http.ResponseWriter

encoding/zip

Single method interfaces

Very easy to implement

It will be used more

- Echo project
- https://github.com/labstack/echo

Testing in Go

```
$go test .                  //รันทดสอบทั้งหมด จาก package ปัจจุบันลงไป
$go test -p 1  ./…          //รันทดสอบทีละ package
$go test -run TestName      //รันทดสอบตามชื่อ test case
$go test -v ./...           //รันทดสอบแสดงผลการรันทั้งหมด
```

Tags your test

```go
// +build integration

package handler

import "testing"

func TestIntegrationGetProductHandler(t *testing.T) {
  t.Skip("TODO: implement integration test.")
}
```

```
$go test -tags=integration
```

Test usage

```
package sample_test
```

Test usage

```
package car_test

import (
  "github.com/anuchito/testing/car"
  "testing"
)

func TestBuildCar(t *testing.T) {
  assembly := car.CarBuilder().Paint(car.RedColor)

  familyCar := assembly.Wheels(car.SportsWheels).TopSpeed(50 * car.MPH).Build()
  familyCar.Drive()

  sportsCar := assembly.Wheels(car.SteelWheels).TopSpeed(150 * car.MPH).Build()
  sportsCar.Drive()
}
```

Test usage : improve

```go
package car_test

import (
  "github.com/anuchito/testing/car"
  "testing"
)

func TestBuildCar(t *testing.T) {
  assembly := car.Builder().RedColor()

  familyCar := assembly.SportsWheels().SpeedKPH(50).Build()
  familyCar.Drive()

  sportsCar := assembly.SteelWheels().SpeedMPH(150).Build()
  sportsCar.Drive()
}
```

How can we test this?

```
package sql

import "database/sql"

func execQuery(db *sql.DB, query string, args ...interface{}) (int64, error) {
  res, err := db.Exec(query, args...)
  if err != nil {
    return 0, err
  }

  ra, err := res.RowsAffected()
  if err != nil {
    return 0, err
  }

  return ra, nil
}
```

Define your own small interface

```
package sql

import "database/sql"

type DB interface {
  Exec(query string, args ...interface{}) (sql.Result, error)
}

func execQuery(db DB, query string, args ...interface{}) (int64, error) {
    // ….
}
```

Create mock

```
type mockDB struct {
  query string
  lastInsertID int64
  rowsAffected int64
}

func (m *mockDB) LastInsertId() (int64, error){
  return m.lastInsertID, nil
}
func (m *mockDB) RowsAffected() (int64, error){
  return m.rowsAffected, nil
}

func (m *mockDB) Exec(query string, args ...interface{}) (sql.Result, error){
  m.query = query
  return m, nil
}
```

Use mock for assertion

```
func TestExecQuery(t *testing.T) {
  mock := &mockDB{
    rowsAffected: 32,
  }

  r, _ := execQuery(mock, "SELECT * FROM sql")

  if mock.query != "SELECT * FROM sql" {
    t.Error("should have been call db.Exec with query but it not.")
  }

  if r != 32 {
    t.Errorf("should return row effect %d but it got %d.", 32, r)
  }
}
```

Too many interface to implement

```
package database

type Database interface {
   SetDBName(dbName string)
   DBName() string
   Insert(collection string, data interface{}) error
   Count(collection string, query interface{}) (int, error)
   FindOne(collection string, query interface{}, data interface{}) error
   FindOnePrimary(collection string, query interface{}, data interface{}) error
   Find(collection string, query interface{}, data interface{}) error
   FindAndSelect(collection string, query interface{}, selector interface{}, data interface{}) error
   FindLimit(collection string, query interface{}, limit int, data interface{}) error
   FindLimitAndOrderBy(collection string, query interface{}, limit int, orderBy string, data interface{}) error
   FindLimitAndOrderByM(collection string, query interface{}, limit int, orderBy []string, data interface{}) error
   FindOrderByOne(collection string, query interface{}, orderby string, data interface{}) error
   FindByOrder(collection string, query interface{}, orderBy []string, data interface{}) error
   FindOrderAndUpdate(collection string, query interface{}, orderBy []string, changeQuery interface{}, afterItem interface{}) error
   Upsert(collection string, query interface{}, data interface{}) error
   Update(collection string, query interface{}, data interface{}) error
   UpdateAll(collection string, query interface{}, data interface{}) error
   Aggregate(collection string, query interface{}, data interface{}) error
   RemoveAll(collection string, query interface{}) error
}

type Mongo struct {
   db string
}

func Insert(db Database, collection string, data interface{}) error {
  return db.Insert(collection, data)
}
```

Embedding interface

```
type mockDb struct {
  Database
}

func (*mockDb) Insert(collection string, data interface{}) error {
  return nil
}

func TestInsert(t *testing.T) {
  mock := &mockDb{}

  err := Insert(mock, "product", `{}`)

  if err != nil {
    t.Error(err.Error())
  }
}
```

Test table

```
type Cases struct {
  in  []*Item
  out []*Item
}

func TestTableExample(t *testing.T) {
  t.Run("test table", func(t *testing.T) {
    cases := []Cases{
      {in: []*Item{{"Dexterity", 10}}, out: []*Item{{"Dexterity", 9}}},
      {in: []*Item{{"Aged Brie", 2}}, out: []*Item{{"Aged Brie", 1}}},
      {in: []*Item{{"Elixir", 5}}, out: []*Item{{"Elixir", 4}}},
      {in: []*Item{{"Sulfuras", 0}}, out: []*Item{{"Sulfuras", 0}}},
      {in: []*Item{{"Backstage", 15}}, out: []*Item{{"Backstage", 14}}},
    }

    assert(t, cases)
  })
}
```

ServeHTTP

```go
func TestNewServeMux(t *testing.T) {
  is := is.New(t)
  w := httptest.NewRecorder()
  req, err := http.NewRequest("GET", "/path", nil)
  is.NoErr(err)

  srv := NewServeMux()

  srv.ServeHTTP(w, req)

  is.Equal(w.Code, http.StatusOK)
  is.Equal(w.Body.String(), `{"name": "anuchit"}`)
}
```

setup/teardown

```
func setup(filename string) func() {
  teardown := func() {
    err := os.Remove(filename)
    if err != nil {
      // panic("could not delete file")
    }
  }
  // create file ...
  return teardown
}

func TestSomething(t *testing.T) {
  teardown := setup("filename")
  defer teardown()

  t.Skip("TODO: implement.")
}
```

Where test data belong to?

- https://golang.org/src/cmd/gofmt/
- testdata

Refactor API : Thaiwin