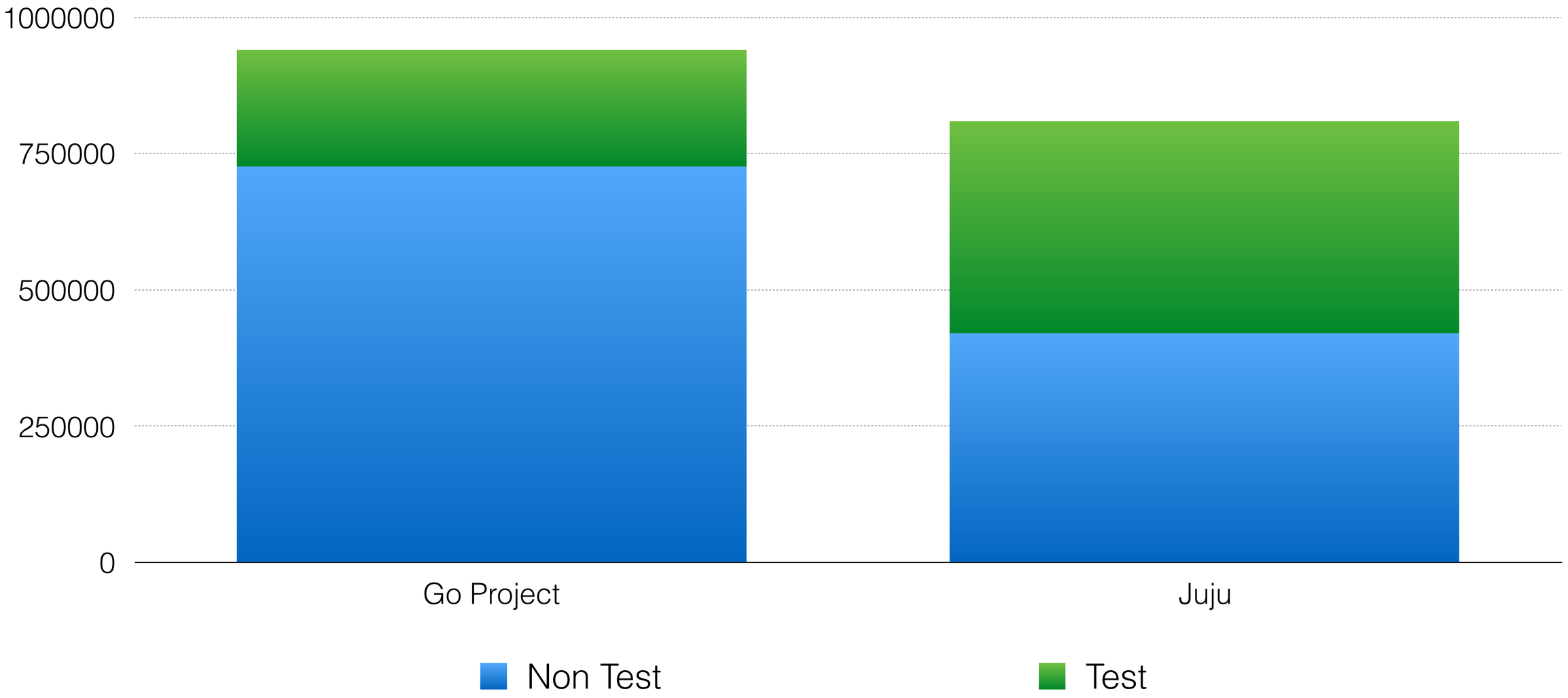
SOLID Go Design

YOW!West May 2016 – Dave Cheney


GO

Lines of code



SOLID Design

“The Go project includes the language itself, its tools and standard libraries, and last but not least, a cultural agenda of radical simplicity.”

–Alan Donovan and Brian Kernighan, The Go Programming Language

Single Responsibility Principle

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

–Robert C. Martin

“Cohesion is a measure of the strength of the association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them would only result in increased coupling and decreased reliability.”

—Tom DeMarco, Structured Analysis and System Specification

The Go package model

A well designed package starts with its name

```
// Package http provides HTTP client and server
// implementations.
package http

// Package exec runs external commands.
package exec

// Package json implements encoding and decoding of
// JSON as defined in RFC 4627.
package json
```

```
package main

import (
        "fmt"
        "log"
        "net"
        "net/http"
        "time"
)

func main() {
        l, err := net.Listen("tcp", "127.0.0.1:9001")
        if err != nil {
                log.Fatal("could not bind to port", err)
        }
        mux := http.NewServeMux()
        mux.HandleFunc("/time", func(w http.ResponseWriter, req *http.Request) {
                fmt.Fprintf(w, "At the third stroke the time will be: %v", time.Now())
        })
        http.Serve(l, mux)
}
```

Bad package names

```
package server

package private

package common

package utils
```

“This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.”

–Doug McIlroy, Quarter Century of Unix

Open / Closed principle

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

–Bertrand Meyer, Object-Oriented Software Construction

```
type A struct {
        v int
}

func (a A) Value() int { return a.v }

type B A
```

```
var a A
a.v = 99
var b = B(a)
fmt.Println(b.v) // 99
```

```
var a A
a.v = 100
fmt.Println(a.Value()) // 100

var b B
b.v = 200
fmt.Println(b.Value()) // b.Value undefined (type B
has no field or method Value)
```

```
type A struct {
        year int
}

func (a A) Greet() { fmt.Println("Hello YOW!West", a.year) }

type B struct {
        A
}

func (b B) Greet() { fmt.Println("Welcome to YOW!West", b.year) }

func main() {
        var a A
        a.year = 2016
        var b B
        b.year = 2016

        a.Greet() // Hello YOW!West 2016
        b.Greet() // Welcome to YOW!West 2016
}
```

```
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 8 }

func main() {
        var octo OctoCat
        fmt.Println(octo.Legs()) // 8
        octo.PrintLegs()         // I have 4 legs
}
```

This is not inheritance

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

Liskov Substitution Principle

Interfaces

Small interfaces

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

type Writer interface {
        // Write writes len(buf) bytes from buf to the underlying stream.
        Write(buf []byte) (n int, err error)
}

type Closer interface {
       // Close closes the underlying data stream.
       Close() error
}
```

```
package io

// MultiReader returns a Reader that's the logical
// concatenation of the provided input readers.
func MultiReader(readers ...Reader) Reader

// LimitReader returns a Reader that reads from r but
// stops with EOF after n bytes.
func LimitReader(r Reader, n int64) Reader

// TeeReader returns a Reader that writes to w what it
// reads from r.
func TeeReader(r Reader, w Writer) Reader
```

```
package strings

// NewReader returns a new Reader reading from s.
func NewReader(s string) *Reader

package bytes

// NewReader returns a new Reader reading from b.
func NewReader(b []byte) *Reader
```

```
package main

import (
        "bytes"
        "fmt"
        “io”
        "os"
)

func main() {
        var b bytes.Buffer     // bytes.Buffer needs no initialization.
        b.Write([]byte("Hello "))
        fmt.Fprintf(&b, "world!")
        io.Copy(os.Stdout, &b) // Hello world!
}
```

“Require no more, promise no less.”

–Jim Weirich

Interface Segregation Principle

“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
```

Interface abstraction

```
package io

type ReadWriteCloser interface {
        Reader
        Writer
        Closer
}
```

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

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

```
type NoCloseWriter struct {
        io.Writer
}

func (ncw NoCloseWriter) Close() error {
        return nil
}
```

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

Dependency inversion principle

“Depend on abstractions, not on concretions.”

–Robert C. Martin

“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 management

Single Responsibility Principle

Structure your functions and types into packages that exhibit natural cohesion.

Open / Close Principle

Compose types with embedding rather than extend them through inheritance.

Liskov Substitution Principle

Express the dependencies between your packages in terms of interfaces, not concrete types.

Interface Segregation Principle

Define functions and methods that depend only on the behaviour that they need.

Dependency Inversion Principle

Refactor dependencies from compile time to run time.

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

–Sandi Metz

Questions ?

@davecheney