LESSONS FROM GO

# 5 RULES FOR WRITING EFFECTIVE CODE

# ABOUT ME

- Eric Gravert
- Principal Engineer, Stitch Fix, Inc.
- 7 years of Go experience

> Go is an open source programming language that enables the production of simple, efficient, and reliable software at scale.

Go Mission

# 1. ERRORS ARE VALUES

# WHAT PROBLEM ARE WE TRYING TO SOLVE?

“We want to signal to the caller of our function that something went wrong.”
– Dave Cheney

```
func GetUsername(id int) string {
  var name string
  db.QueryRow(“SELECT * FROM user WHERE ID = $1”, id).Scan(&name)
  return name
}
```

# EXCEPTIONS DON'T SOLVE SIGNALING WELL

- They introduce an additional code path.
- They are often caught far from where they occur.
- Stack traces are hard to read and parse.

# WHAT IS AN ERROR IN GO?

## DEFINITION

```
type error interface {
    Error() string
}
```

## IMPLEMENTING

```
type ErrBufferFull struct {

  e string

}


func (e *ErrBufferFull) Error() string {
    return e
}
```

## USING

```
// creating an error
err := &ErrBufferFull{“out of memory”}

// checking for an error
buf, err := buffer.New()
if err != nil {
  // do something interesting
}
```

# WHAT IS AN ERROR IN GO?

## DEFINITION

```
type error interface {
    Error() string
}
```

## IMPLEMENTING

```
type ErrBufferFull struct {

  e string

}


func (e *ErrBufferFull) Error() string {
    return e
}
```

## USING

```
// creating an error
err := &ErrBufferFull{“out of memory”}

// checking for an error
buf, err := buffer.New()
if err != nil {
  // do something interesting
}
```

# WHAT IS AN ERROR IN GO?

## DEFINITION

```
type error interface {
    Error() string
}
```

## IMPLEMENTING

```
type ErrBufferFull struct {

  e string

}


func (e *ErrBufferFull) Error() string {
    return e
}
```

## USING

```
// creating an error
err := &ErrBufferFull{“out of memory”}

// checking for an error
buf, err := buffer.New()
if err != nil {
  // do something interesting
}
```

# HOW GO SOLVES SIGNALING

```
func GetUsername(id int) (string, error) {
  var name string
  err := db.QueryRow("SELECT * FROM user WHERE ID = $1", id).Scan(&name)
  return name, err
}


func DoImportantThing(userID int) {

  un, err := GetUserName(userID)
  if err != nil {
    // decide how to act upon the error
  }

  // boring important logic
}
```

# WHAT CAN WE DO WITH ERRORS?

- We can add attributes to make them more meaningful.
- We can act on them where they occur.
- We can format them to make them clear and meaningful.

# DECORATING ERRORS

```go
type ErrConfig struct {
    Setting string // The setting being read from env
    Value   string // The value received
    Err     error  // The error encountered
}

func (ec *ErrConfig) Error() string {
    return ec.Error()
}


func loadConfig() (*Config, *ErrConfig) {
    pe := os.Getenv("port")

    port, err := strconv.Atoi(pe)
    if err != nil {
        return nil, &ErrConfig{
            Setting: "port",
            Value:   pe,
            Err:     err,
        }

    }
    // more config
    return &Config{}, nil
}
```

# DECORATING ERRORS

```
type ErrConfig struct {
    Setting string // The setting being read from env
    Value   string // The value received
    Err     error  // The error encountered
}

func (ec *ErrConfig) Error() string {
    return ec.Error()
}


func loadConfig() (*Config, *ErrConfig) {
    pe := os.Getenv("port")

    port, err := strconv.Atoi(pe)
    if err != nil {
        return nil, &ErrConfig{
            Setting: "port",
            Value:   pe,
            Err:     err,
        }

    }
    // more config
    return &Config{}, nil
}
```

# DECORATING ERRORS

```
func main() {
  cfg, err := loadConfig()
  if err != nil {
    fmt.Println("invalid setting %s: %s. %v\n",
                 err.Setting,
                 err.Value,
                 err.Err)
    os.Exit(1)
  }
}
```

# ACTING ON ERRORS

```
func Estimate(pkg Dimensions, from, to Address) (Cost, error) {
    cost, err := usps.Estimate(pkg, from, to)
    return cost, err
}
```

# ACTING ON ERRORS

```
package usps

var ErrUSPSUnavailable = errors.New("usps service is unavailable")

func Estimate(dim Dimensions, from, to Address) (Cost, error) {

  // do work to get estimate

  if timeoutOccurred {
    return Cost{}, ErrUSPSUnavailable
  }

  return cost, nil
}
```

# ACTING ON ERRORS

```
func Estimate(pkg Dimensions, from, to Address) (Cost, error) {
  var err error

  cost, err := usps.Estimate(pkg, from, to)
  if err == usps.ErrUSPSUnavailable {
      cost, err = fedex.Estimate(pkg, from, to)
  }

  return cost, err
}
```

# BENEFITS

- Allow us to signal when something goes wrong and act accordingly.
- Forces us to be intentional with how we treat errors.
- Testing and code coverage are more straightforward.

# 2. INITIALIZE APPLICATION STATE AT STARTUP

# WHAT IS APPLICATION STATE?

- Configuration settings
  - Database urls
  - Service endpoints
- The application's object dependency graph
- Globals
  - Loggers
  - Metrics

# RUNTIME CONFIGURATION

- Results in a system with non-deterministic behavior.
- Testing may require complex integration tests.
- Production systems may fail silently, or not at all.
- Subtle concurrency bugs can be introduced.

# BEST PRACTICES

- Avoid globals.
- Panic during startup. Crash early, and crash hard.
- Explicitly define your dependencies in main().

# AVOID GLOBALS

```
import “global/logger”

// don’t do this
func (s *service)Create(cor CreateOrderRequest)  {
  // important logic
  logger.Infof("settling direct order %s", on)
  // more important logic
}
```

# AVOID GLOBALS

```
// do this
type service struct {
  logger InfoLogger // an interface defined in our package
}


func New(l InfoLogger) *service {
  return &service{logger: l}
}
func (s *service)Create(cor CreateOrderRequest)  {
  // important logic
  s.logger.Infof("settling direct order %s", on)
  // more important logic
}
```

# PANIC DURING STARTUP. CRASH EARLY; CRASH HARD.

```go
package db

func Connect(driver, dbURL string) *sql.DB {
  db, err := sql.Open(driver, dbURL)
  if err != nil {
    panic(err)
  }
  if db.Ping() != nil {
    panic(err)
  }
}
```

# EXPLICITLY DEFINE YOUR DEPENDENCY GRAPH

```
func main() {

  // create a logger instance
  lg := logger.New(os.StdOut)

  // establish a connection pool
  conn := db.Connect()

  // inject the connection pool into our data store
  repo := order.NewPostgresStore(conn, lg)

  // pass the store to our services, and other deps to our services
  orderService := order.NewService(repo, lg, metric.Metrics{})
  .
  .
  .
}
```

# BENEFITS

- The application behavior is predictable.
- We can monitor logs during a deployment, and immediately see if we have configuration issues.
- The dependency graph is clear.
- Concurrency bugs are less likely due to initialization timing.

# 3. ACCEPT INTERFACES, RETURN STRUCTS

# THE ROBUSTNESS PRINCIPLE

- Accept interfaces
  - Reduces coupling and interface  size
  - Facilitates easier testing
  - Leverage interface-segregation principle
- Returning structs
  - A struct can implement many small interfaces

# FAT INTERFACES

```
package metric

type Metrics interface {
  IncOrderSettled()
  IncShipped()
  IncPicked()
}

type StatsdMetrics struct { }

func New() Metrics {
  return StatsdMetrics{}
}

func (m StatsdMetrics)IncOrderSettled() {
  …
}

func (m StatsdMetrics)IncShipped() {
  …
}

func (m StatsdMetrics)IncPicked() {
  …
}
```

# FAT INTERFACES

```
package orders

type Service struct {
  m metrics.Metric
}

func NewService(m metric.Metrics) *Service {
  return &Service{metrics: m}
}

func (s *Service)ShipOrder(on OrderNumber) {
 s.metric.IncShipped()
}
```

# WRITING A TEST

```
package orders

func TestShippingOrder(t *testing.T) {

  m := &mockMetrics{}

  s := NewService(m)
  s.ShipOrder()

  if m.shipCount != 1 {
    t.Errorf("shipping metrics were not updated")
  }

}

type mockMetrics struct {
  shipCount int
}

func (m mockMetrics)IncOrderSettled() {
  // no-op
}

func (m mockMetrics)IncShipped() {
  m.shipCount++
}

func (m mockMetrics)IncPicked() {
  // no-op
}
```

# WRITING A TEST

```
package orders

func TestShippingOrder(t *testing.T) {

  m := &mockMetrics{}

  s := NewService(m)
  s.ShipOrder()

  if m.shipCount != 1 {
    t.Errorf("shipping metrics were not updated")
  }

}

type mockMetrics struct {
  shipCount int
}

func (m mockMetrics)IncOrderSettled() {
  // no-op
}

func (m mockMetrics)IncShipped() {
  m.shipCount++
}

func (m mockMetrics)IncPicked() {
  // no-op
}
```

# WRITING A TEST

```
package orders

func TestShippingOrder(t *testing.T) {

  m := &mockMetrics{}

  s := NewService(m)
  s.ShipOrder()

  if m.shipCount != 1 {
    t.Errorf("shipping metrics were not updated")
  }

}

type mockMetrics struct {
  shipCount int
}

func (m mockMetrics)IncOrderSettled() {
  // no-op
}

func (m mockMetrics)IncShipped() {
  m.shipCount++
}

func (m mockMetrics)IncPicked() {
  // no-op
}
```

# ACCEPT INTERFACES

```
package orders


type ShippingMetrics interface {
  IncShipped()
}

type Service struct {
  metrics metrics.Metric
  sm ShippingMetrics
}

func NewService(m metric.Metrics) *Service {
func NewService(sm ShippingMetrics) *Service {
  return &Service{sm: sm}
}

func (s *Service)ShipOrder(on OrderNumber) {
 s.sm.IncShipped()
}
```

# ACCEPT INTERFACES

```
package orders


type ShippingMetrics interface {
  IncShipped()
}

type Service struct {
  sm ShippingMetrics
}

func NewService(sm ShippingMetrics) *Service {
  return &Service{sm: sm}
}

func (s *Service)ShipOrder(on OrderNumber) {
 s.sm.IncShipped()
}
```

# RETURN STRUCTS

```
package metric

type Metrics interface {
  IncOrderSettled()
  IncShipped()
  IncPicked()
}

type StatsdMetrics struct { }

func New() Metrics {
  return StatsdMetrics{}
}

func New() StatsdMetrics {
  return StatsdMetrics{}
}

func (m StatsdMetrics)IncOrderSettled() {
  …
}

func (m StatsdMetrics)IncShipped() {
  …
}

func (m StatsdMetrics)IncPicked() {
  …
}
```

# RETURN STRUCTS

```
package metric

type StatsdMetrics struct { }


func New() StatsdMetrics {
  return StatsdMetrics{}
}

func (m StatsdMetrics)IncOrderSettled() {
  …
}

func (m StatsdMetrics)IncShipped() {
  …
}

func (m StatsdMetrics)IncPicked() {
  …
}
```

# WRITING A TEST

```
package orders

func TestShippingOrder(t *testing.T) {

  m := &mockMetrics{}

  s := NewService(m)
  s.ShipOrder()

  if m.shipCount != 1 {
    t.Errorf(“shipping metrics were not updated”)
  }

}

type mockMetrics struct {
  shipCount int
}

func (m mockMetrics)IncShipped() {
  m.shipCount++
}
```

# BENEFITS

- Testing is much easier.
- The intent of the code is much clearer.
- Our interfaces provide a stronger abstraction.
- Composition is improved.

# 4. MAKE BUSINESS LOGIC DRY, NOT YOUR CODE

# DON'T REPEAT YOURSELF

- "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."
  – The Pragmatic Programmer

# DRY LEADS TO CLEVER CODE

- DRY code is often brittle.
- Testing becomes complex.
- DRY code often violate the Single Responsibility Principle.
- An application's dependencies grow, increasing brittleness, and increasing build times.

# SINGLE RESPONSIBILITY VIOLATIONS

```
type Return struct {
  Weight int
}

type Shipment struct {
  Weight int
}

func (r *Return)ShippingCost() int {
  return r.Weight * 50
}

func (sh *Shipment)ShippingCost() int {
  return sh.Weight * 50
}
```

# SINGLE RESPONSIBILITY VIOLATIONS

```
type Return struct {
  Weight int
}

type Shipment struct {
  Weight int
}

type Package interface {
  Weight() int
}

func PackageShippingCost(pkg Package) int {
  return pkg.Weight() * 50
}
```

# SINGLE RESPONSIBILITY VIOLATIONS

```
type Return struct {
  Weight int
}

type Shipment struct {
  Weight int
}

type Package interface {
  Weight() int
  IsMailer() int
}

func PackageShippingCost(pkg Package) int {
  if pkg.IsMailer() {
    return pkg.Weight() * 42
  }
  return pkg.Weight() * 50
}
```

# SINGLE RESPONSIBILITY VIOLATIONS

```
func ShippingCost(pkg Package) int {
    if pkg.IsMailer() {
        return pkg.Weight() * MailerRate
    }
    if sh, ok := pkg.(*Shipment); ok {
        if sh.Length > Oversized {
            return pkg.Weight() * OversizeRate
        }
    }
    return pkg.Weight() * NormalRate
}
```

# FOCUS ON DRY BUSINESS LOGIC

- Take the time to understand the difference between code that looks similar, but belongs to different domains, and code that actually is the same.
- It's better to copy a little code, than take on a little dependency.
- Use a a strategy pattern to break out responsibilities.
- You can still use tests to ensure implementations are compatible.

# BENEFITS

- You reduce the brittleness of your system.
- The intent of the code is more clear.
- You have less risk of single responsibility violations due to unintended edge cases.

# 5. CLEAR IS BETTER THAN CLEVER

# WHY CLEAR CODE IS IMPORTANT?

- Software Engineering is what happens to programming when you add time and other programmers.
  – Russ Cox

# LANGUAGE FEATURES

- Less is more. Go has only 24 keywords.
- Go lacks features that permit highly compact code you may see in other languages.
  - Only one looping construct, “for”.
  - No ternary operator.
  - Only boolean expressions allowed in control structures.
  - Very few syntactic shortcuts allowed.

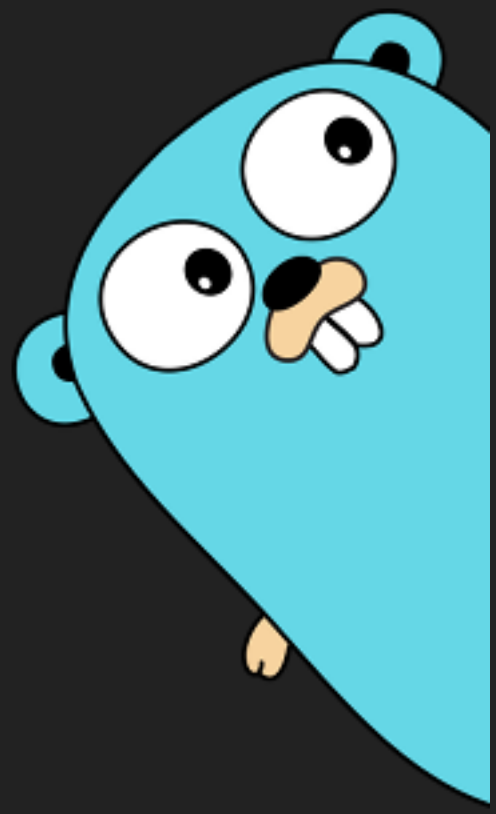
# COMMUNITY VALUES

- Leverages guard clauses, and “Keep to the Left”.
- Prefers kits and packages over frameworks.
- Promotes strong tooling support.

# BENEFITS

- Engineers can ramp up on an unfamiliar code base more quickly.
- Troubleshooting is more straightforward. The mental model required to effectively solve a problem is much smaller.
- Systems are better designed. Clarity requires intentional design, and thoughtful consideration of your business domain.
- Clear code tends to be less brittle, and more testable.

# SUMMARY



- Be intentional in your design, and treat errors as normal part of your application's logic.
- Reduce unexpected runtime errors by handling initialization up front, and treating misconfiguration as catastrophic.
- Create stronger abstractions in your code, and improve its clarity be leveraging interfaces.
- Optimize the clarity of your domain logic, not the number of lines of code.
- Write maintainable code. Cleverness hides intent, and reduces maintainability. Strive for clarity instead.