Readability in Go

Reducing Load on Our Memory

Me: Sam Nelson

Senior Engineer, stealth AI startup

Primarily a go engineer since 2013

Projects:

- https://github.com/go-gorp/gorp
- https://github.com/nelsam/hel
- https://github.com/nelsam/vidar
- https://github.com/poy/onpar

Readability

Obligatory definition slide!

read·a·bil·i·ty (/ˌrēdəˈbilədē/)

noun *the quality of being legible or decipherable*

We're using computer fonts. If your code isn't legible, you probably need to change fonts in your editor.

Deciphering Code

Why do we care?

- Code is read far more often than it's written
- Onboarding/offboarding is easier
- Locating bugs is quicker and easier
- Proving correctness is quicker and easier

Doing something complicated in software is fun

Making it look boring is an art

Deciphering Code

Intent

We need to decipher the original author's intent for the code. What problem was this _intended_ to solve?

- ruby
- perl

Reality

We need to decipher what will actually happen when we run some code. What machine code will be executed?

- C
- assembly

Clearly, there isn't just one definition of "readability" in software

Predictability

- We want to predict how code is implemented
- We want to predict how we'd use a library
- We want to predict what will happen when we execute some code

Predictable Go Code

- Clear and concise
  - Package and variable names
- Else statements
- Interface types and implementation

Refresher

- Go Proverbs
- Effective Go
- Go Code Review Comments

Refresher

- Errors are values
- A little copying is better than a little dependency
- [...] it's neither idiomatic nor necessary to put Get into the getter's name
- when an if statement doesn't flow into the next statement [...] the unnecessary else is omitted
- Interfaces with only one or two methods are common
- the further from its declaration that a name is used, the more descriptive the name must be

We know and use these every day

...but how many times have you taught someone go and they have just *been on board* with these?

Clear and Concise

```go
package user_service

type UserService struct {
        // unexported fields
}

func NewUserService(readWriteDatabaseCo
        return &UserService{
        // assignment
    }
}

func NewUserServiceWithRODatabaseConn(r
        return &UserService{
        // assignment
    }
}
```

Does the long package name add clarity?

Is it useful to have "UserService" in the constructor names?

How about those long parameter names?

```go
user_service.NewUserService(db)

// or

user_service.NewUserServiceWithR
    db,
    slavedb,
)
```

Clear and Concise

```go
package user

type Opt func(*Service) *Service

func WithRODB(*sql.DB) Opt {}

type Service struct {
    // unexported fields
}

func NewService(rwDB *sql.DB, opts ...O
    s := &Service{
        // assignment
    }
    for _, o := range opts {
        s = o(s)
    }
    return s
}
```

How much clarity did we lose here?

```go
user.NewService(db)

// or

user.NewService(db,
    user.WithRODB(slavedb),
)
```

These shorter terms also make it easier for us to predict how we use the rest of the API

Clear and Concise

Long Names

- Clarify intent

Short Names

- Reduce horizontal scrolling
- Encourage predictable patterns

the further from its declaration that a name is used, the more descriptive the name must be

Else Statements

```
 1 func Profile(auth *user.Auth) (*user.Pr
 2     var profile *user.Profile
 3     var err error
 4     if user.IsValid() {
 5         profile, err = db.LoadProfile(u
 6     } else {
 7         err = Unauthorized
 8     }
 9     return profile, err
10 }
```

Do you see the potential bug?

Clearly a contrived example ... let's try another

Else Statements

```
1  func Friends(auth *user.Auth) ([]*user.Profile, error) {
2      var (
3          profiles []*user.Profile
4          err error
5      )
6      if user.IsValid() {
7          for _, id := range auth.FriendIDs() {
8              profile, newErr := db.LoadProfile(id)
9              if err != nil && newErr != nil {
10                 err = newErr
11             } else {
12                 profiles = append(profiles, profile)
13             }
14         }
15     } else {
16         err = Unauthorized
17     }
18     return profiles, err
19 }
```

This might look a little more familiar

Else Statements

```
 1 func Friends(auth *user.Auth) ([]*user.Profil
 2     if !user.IsValid() {
 3         return nil, Unauthorized
 4     }
 5     var profiles []*user.Profile
 6     for _, id := range auth.FriendIDs() {
 7         profile, err := db.LoadProfile(id)
 8         if err != nil {
 9             return nil, fmt.Errorf("failed lo
10         }
11         profiles = append(profiles, profile)
12     }
13     return profiles, nil
14 }
```

If we encounter an error, we interrupt control flow, usually with a return

Returning for base cases early is normal

Else Statements

- Ensure that all possible cases are handled
- Avoiding early returns means no hidden returns

Clearly, we think there are downsides in go

What are they?

- Increase indentation
- Force us to merge logic flows
- Weird scoping issues (go-specific)

Interface Implementation

interfaces generally belong in the package that uses values of the interface type, not the package that implements those values

The bigger the interface, the weaker the abstraction.

There are notable exceptions!

Interface Types

Go interfaces are not like interfaces in other languages!

Frankly, they're more like duck types.

Interface Types

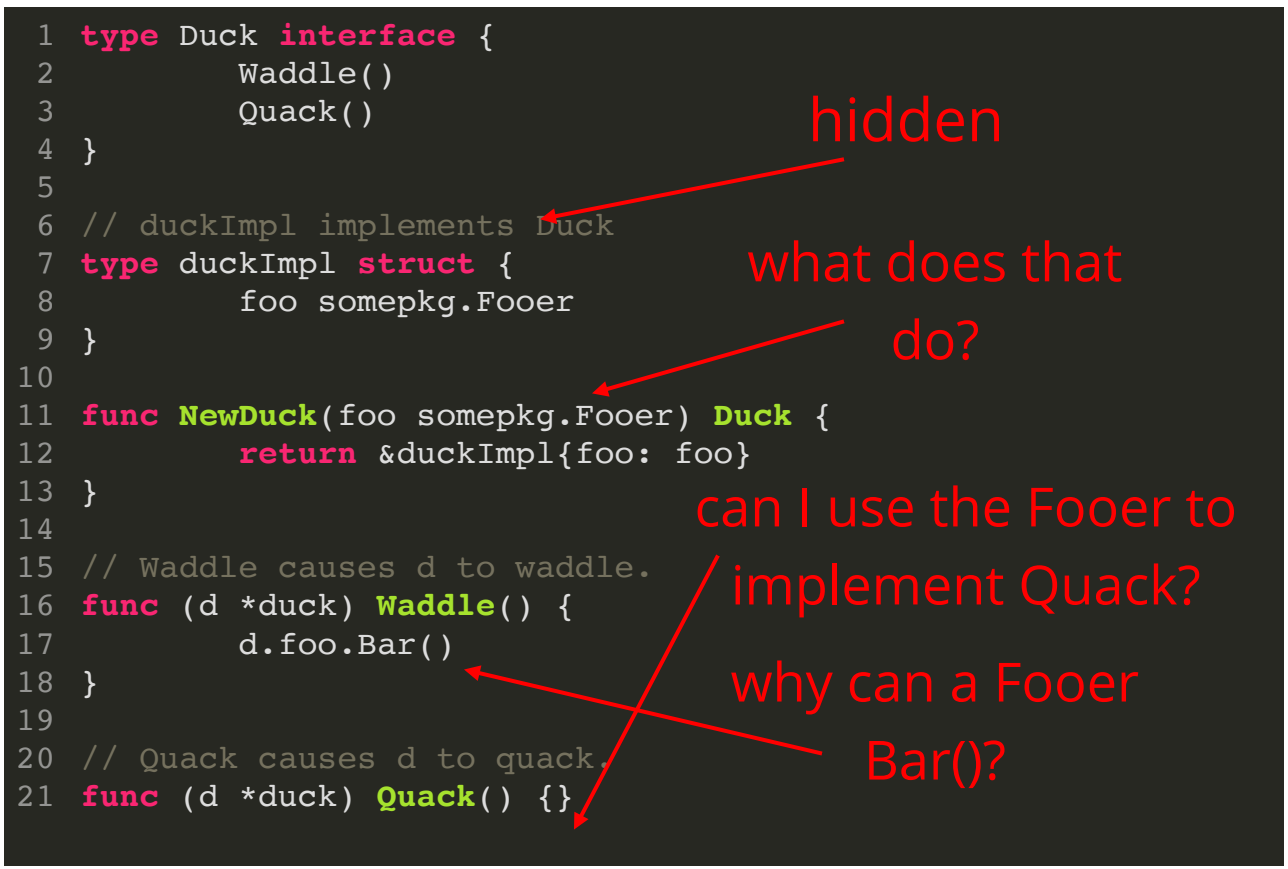
```
1 type Duck interface {
2         Waddle()
3         Quack()
4 }
```

What does the rest of this code look like?

Interface Types

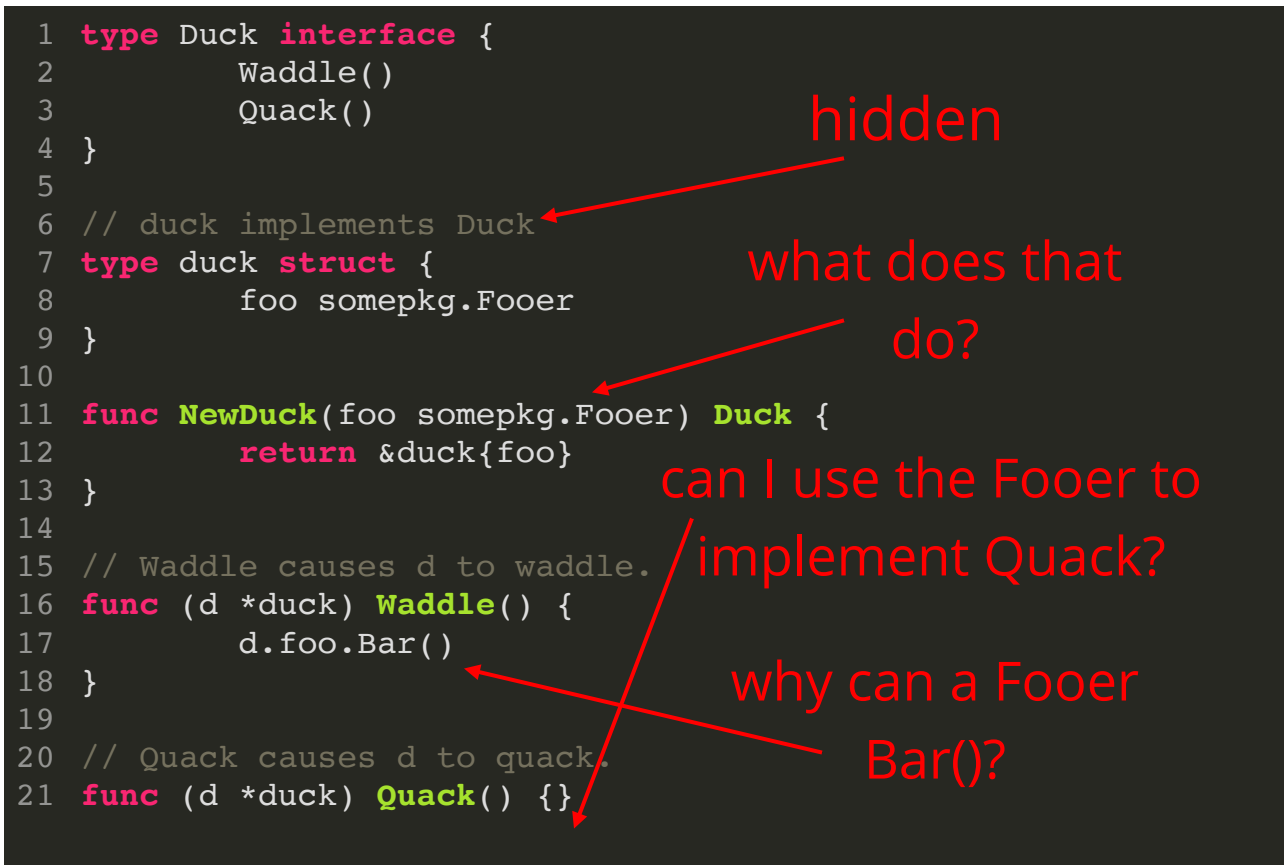
How often have you seen this? Written this? It's common coming from java.

```
 1 type Duck interface {
 2         Waddle()
 3         Quack()
 4 }
 5
 6 // duckImpl implements Duck
 7 type duckImpl struct {
 8         foo somepkg.Fooer
 9 }
10
11 func NewDuck(foo somepkg.Fooer) Duck {
12         return &duckImpl{foo: foo}
13 }
14
15 // Waddle causes d to waddle.
16 func (d *duck) Waddle() {
17         d.foo.Bar()
18 }
19
20 // Quack causes d to quack.
21 func (d *duck) Quack() {}
```



What's wrong with this code?

Interface Types



This is normal in other languages. Why not go?

Interface Types

```
type Duck interface {
        Waddle()
        Quack()
}

type Direction int

const (
    North Direction = 1 + iota
    East
    South
    West
)

type Pond struct {
        residents []Duck
}

func Migrate(d Direction, ducks ...Duck) *Pond {
        // etcetera
}
```

This is what the rest of the code *should* have looked like

But *Why?*

Let's Review

```go
package user

type Opt func(*Service) *Service

func WithRODB(*sql.DB) Opt {}

type Service struct {
    // unexported fields
}

func NewService(rwDB *sql.DB, opts ...O
    s := &Service{
        // assignment
    }
    for _, o := range opts {
        s = o(s)
    }
    return s
}
```

How well do you remember this code?

How hard is it for you track through it?

Let's Review

```
1  func Friends(auth *user.Auth) ([]*user.Profil
2      if !user.IsValid() {
3          return nil, Unauthorized
4      }
5      var profiles []*user.Profile
6      for _, id := range auth.FriendIDs() {
7          profile, err := db.LoadProfile(id)
8          if err != nil {
9              return nil, fmt.Errorf("failed lo
10         }
11         profiles = append(profiles, profile)
12     }
13     return profiles, nil
14 }
```

There is branching here ... but branching is never the problem

I'm looking at you, subversion

How often do you need to mentally *merge*?

How many branches of logic are you keeping in mind while reading it?

Let's Review

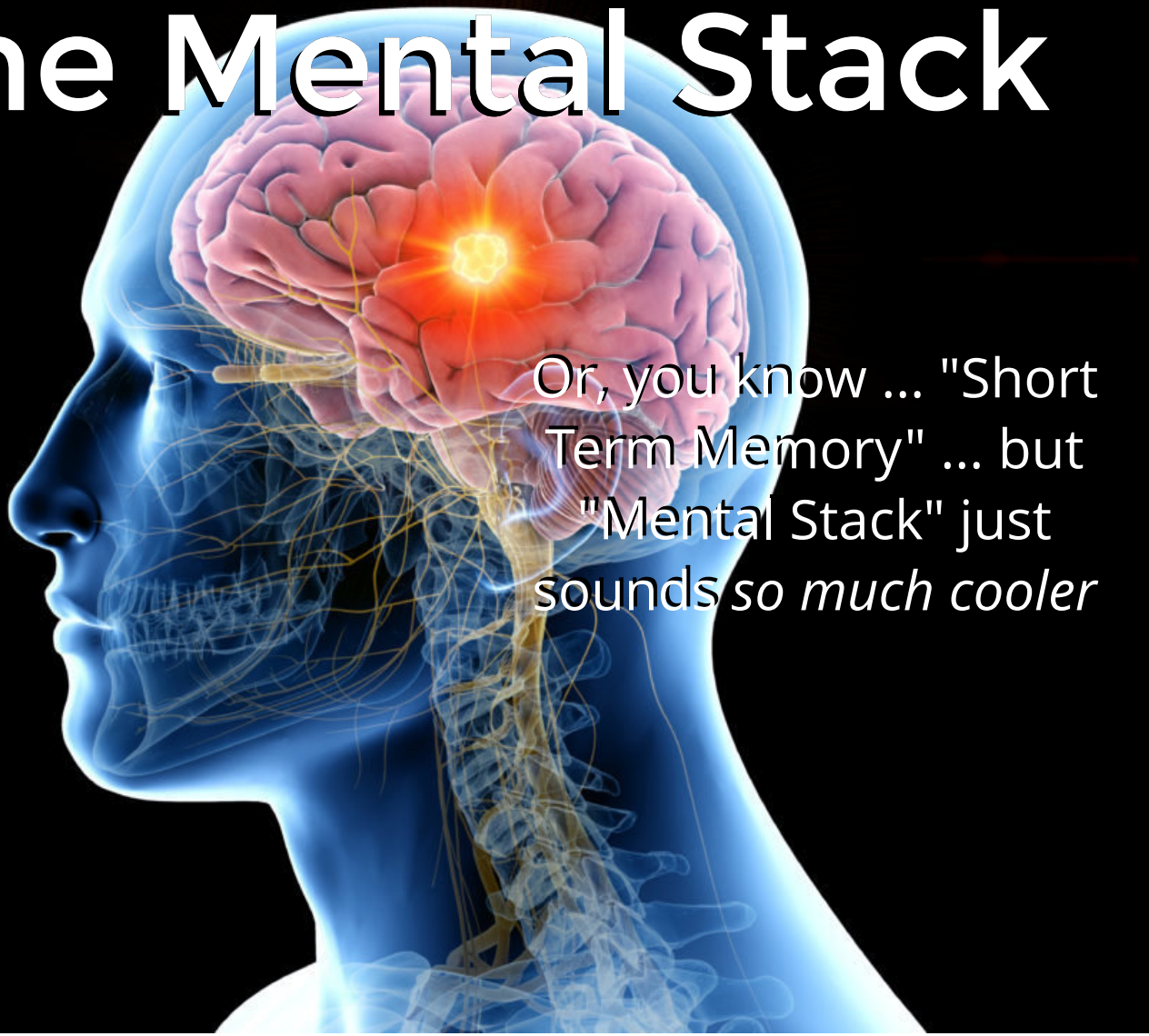
```
 1 type Duck interface {
 2         Waddle()
 3         Quack()
 4 }
 5
 6 // Direction hidden for brevity
 7
 8 type Pond struct {
 9         residents []Duck
10 }
11
12 func Migrate(d Direction, ducks ...Duck) *Pond {
13         // etcetera
14 }
```

How many imports do we need?

The larger the interface, the lower the abstraction

But one more thing... how many other packages do I need to read first before I can understand this code?

The Mental Stack



Or, you know ... "Short Term Memory" ... but "Mental Stack" just sounds *so much cooler*

Make it easier to track code in your brain

- Early returns break context off of your mental stack
  - Making room for extra information is *great*
- Removing if/elseif/else chains ensures that we only need to keep one logic branch in mind at a time
- Returning in logic branches avoids the need to merge those logic branches
- Short names take less effort to recall
- Lower indentation directly relates to less variable scope you need to track
- Small, local interfaces reduce the number of methods you need to think about in dependencies

Predictability

All of this makes it easier to predict code.
What will happen when you execute it,
what the rest of the code will look like,
what future features might look like.

Questions to Ask Yourself

- How do my eyes track when I read this code?
  - Ideal: top to bottom, left to right
- Do I ever need to check other files or scroll to other blocks of code to better understand what I'm reading?
- How hard is it to track a single code path?
  - If I choose a value, can I step through each execution and understand what will happen?
  - Do I ever have to merge logic branches in my brain?
- Do I ever have to think about logic that isn't directly represented in the code?

Exercise: Predict this code

```
 1 func ValidateUser(u User) error {
 2     var err error
 3     if u.ID <= 0 {
 4         err = fmt.Errorf("user ID %d is invalid", u.ID)
 5     } else if vErr := validatePassword(u.Password); err.HasBadCharacters {
 6         err = ErrBadCharacters
 7     } else {
 8         err = vErr
 9     }
10     return err
11 }
```

- How do your eyes track?
- How many branches of information are you tracking?
- Do you see the bug?

Exercise: Predict this code

```
1  func ValidateUser(u User) error {
2      var err error
3      if u.ID <= 0 {
4          err = fmt.Errorf("user ID %d is invalid", u.ID)
5      } else if vErr := validatePassword(u.Password); vErr != nil && vErr.HasBa
6          err = ErrBadCharacters
7      } else {
8          err = vErr
9      }
10     return err
11 }
```

that's easy enough

implicit "if u.ID > 0 &&"

where is that error coming from?

There is a bug here. It's not super obvious.

See this playground link

Exercise: Predict this code

```
1  func ValidateUser(u User) error {
2      if u.ID <= 0 {
3          return fmt.Errorf("user ID %d is invalid", u.ID)
4      }
5      err := validatePassword(u.Password)
6      if err != nil {
7          if err.HasBadCharacters {
8              return ErrBadCharacters
9          }
10         return err
11     }
12     return nil
13 }
```

- How do your eyes track?
- How many branches of information are you tracking?
- We totally got rid of that bug

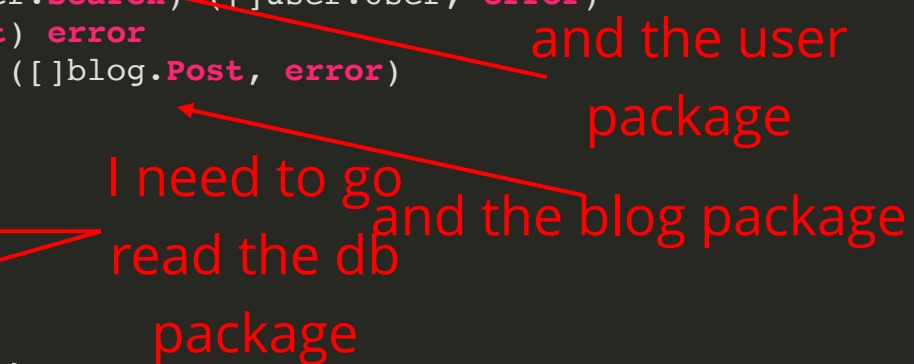
Exercise: Predict this code

```
 1 type Store interface {
 2         SaveUser(u user.User) error
 3         FindUsers(search user.Search) ([]user.User, error)
 4         SavePost(p blog.Post) error
 5         Posts(by user.User) ([]blog.Post, error)
 6 }
 7
 8 type store struct {
 9         db db.DB
10 }
11
12 func New(db db.DB) Store {
13         return &store{db: db}
14 }
15
16 func (s *store) SaveUser(u user.User) error
17 func (s *store) FindUsers(search user.Search) ([]user.User, error)
18 func (s *store) SavePost(p blog.Post) error
19 func (s *store) Posts(by user.User) ([]blog.Post, error)
```

- How do your eyes track?
- Can you tell me what the rest of the implementation looks like?
- What information are you missing?

Exercise: Predict this code

```
 1 type Store interface {
 2         SaveUser(u user.User) error
 3         FindUsers(search user.Search) ([]user.User, error)
 4         SavePost(p blog.Post) error
 5         Posts(by user.User) ([]blog.Post, error)
 6 }
 7
 8 type store struct {
 9         db db.DB
10 }
11
12 func New(db db.DB) Store {
13         return &store{db: db}
14 }
15
16 func (s *store) SaveUser(u user.User) error {}
17 func (s *store) FindUsers(search user.Search) ([]user.User, error) {}
18 func (s *store) SavePost(p blog.Post) error {}
19 func (s *store) Posts(by user.User) ([]blog.Post, error) {}
```



Exercise: Predict this code

```
type User interface {
        SetID(int)
        Email() string
        PassHash() string
}

type UserSearch interface {
        EmailPattern() *re.Regexp
}

type DB interface {
        Append(time.Time, interface{}) error
        Get(start, end time.Time) ([]interface{}, error)
}

type Store struct {
        db DB
}

func New(db DB) *Store {
        return &Store{db: db}
}

func (s *Store) SaveUser(u User) error {}
func (s *Store) FindUsers(search UserSearch) ([]User, error) {}
func (s *Store) SavePost(p BlogPost) error {}
func (s *Store) Posts(by User) ([]BlogPost, error) {}
```

- How do your eyes track?
- Can you tell me what the rest of the implementation looks like?
- What information are you missing?

Exercise: Predict this code

```
1  type User interface {
2          SetID(int)
3          Email() string
4          PassHash() string
5  }
6
7  type UserSearch interface {
8          EmailPattern() *re.Regexp
9  }
10
11 type DB interface {
12         Append(time.Time, interface{}) error
13         Get(start, end time.Time) ([]interface{}, error)
14 }
15
16 type Store struct {
17         db DB
18 }
19
20 func New(db DB) *Store {
21         return &Store{db: db}
22 }
23
24 func (s *Store) SaveUser(u User) error {}
25 func (s *Store) FindUsers(search UserSearch) ([]User, error) {}
26 func (s *Store) SavePost(p BlogPost) error {}
27 func (s *Store) Posts(by User) ([]BlogPost, error) {}
```

Okay. We can store that data.

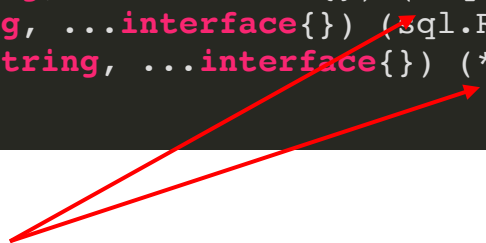
Cool, we can search that

Oh look. It's a timeseries database.

What about those exceptions?

Let's Talk About ... `database/sql`

```
1 // Clearly, you'd only include the methods you're using here
2 type DB interface {
3         QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
4         ExecContext(context.Context, string, ...interface{}) (sql.Result, error)
5         QueryRowContext(context.Context, string, ...interface{}) (*sql.Row, error)
6 }
```



These can be problems

Since they're returning concrete types, our mocks have to return those same concrete types.

Balls. All of the fields in *sql.Rows are unexported. We can't test this code.

Let's Talk About ... `database/sql`

This problem isn't *ridiculously* common,
but it definitely happens.

The older go libraries have more of these
sorts of things. It gets less common with
more modern packages.

Let's Talk About ... `database/sql`

Okay, so how do you handle it?

```
type Rows interface {
        Next() bool
        Scan(...interface{}) error
        Err() error
        Close() error
}

type DB interface {
        Query(string, ...interface{}) (Rows, error)
}

type db struct {
        *sql.DB
}

func Wrap(db *sql.DB) DB {
        return &db{DB: db}
}

func (d *db) Query(q string, a ...interface{}) (Rows, error) {
        return d.DB.Query(q, a...)
}
```

I hate this, but it's about the only thing we can do right now.

Thank You!

I was Samuel Nelson

hopefully I still am