Contents

Aprenda Go com Testes

Arte por Denise

- Formatos: Gitbook, EPUB ou PDF
- Versão original: English
- Traduções: Chinese

Motivação

- Explore a linguagem Go escrevendo testes
- **Tenha uma base com TDD**. O Go é uma boa linguagem para aprender TDD por ser simples de aprender e ter testes nativamente
- Tenha confiança de que você será capaz de escrever sistemas robustos e bem testados em Go
- Assista a um vídeo ou leia sobre o motivo pelo qual testes unitários e TDD são importantes

Explicação

Tenho experiência em apresentar Go a equipes de desenvolvimento e tenho testado abordagens diferentes sobre como evoluir um grupo de pessoas que têm curiosidade sobre Go para criadores extremamente eficazes de sistemas em Go.

O que não funcionou

Ler *o* livro

Uma abordagem que tentamos foi pegar o livro azul e toda semana discutir um capítulo junto de exercícios.

Amo esse livro, mas ele exige muito comprometimento. O livro é bem detalhado na explicação de conceitos, o que obviamente é ótimo, mas significa que o progresso é lento e uniforme - não é para todo mundo.

Descobri que apenas um pequeno número de pessoas pegaria o capítulo X para ler e faria os exercícios, enquanto que a maioria não.

LEARN GO WITH TESTS

LEARN THE GO PROGRAMMING LANGUAGE AND
TEST DRIVEN DEVELOPMENT

Chris James & other cool people

Figure 1:

Figure 2:

Resolver alguns problemas

Katas são divertidos, mas geralmente se limitam ao escopo de aprender uma linguagem; é improvável que você use goroutines para resolver um kata.

Outro problema é quando você tem níveis diferentes de entusiasmo. Algumas pessoas aprendem mais da linguagem que outras e, quando demonstram o que já fizeram, confundem essas pessoas apresentando funcionalidades que as outras ainda não conhecem.

Isso acaba tornando o aprendizado bem *desestruturado* e *específico*.

O que funcionou

De longe, a forma mais eficaz foi apresentar os conceitos da linguagem aos poucos lendo o go by example, explorando-o com exemplos e discutindo-o como um grupo. Essa abordagem foi bem mais interativa do que "leia o capítulo X como lição de casa".

Com o tempo, a equipe ganhou uma base sólida da *gramática* da linguagem para que conseguíssemos começar a desenvolver sistemas.

Para mim, é semelhante à ideia de praticar escalas quando se tenta aprender a tocar violão.

Não importa quão artístico você seja; é improvável que você crie músicas boas sem entender os fundamentos e praticando os mecanismos.

O que funcionou para mim

Quando *eu* aprendo uma nova linguagem de programação, costumo começar brincando em um REPL, mas hora ou outra preciso de mais estrutura.

O que eu gosto de fazer é explorar conceitos e então solidificar as ideias com testes. Testes certificam de que o código que escrevi está correto e documentam a funcionalidade que aprendi.

Usando minha experiência de aprendizado em grupo e a minha própria, vou tentar criar algo que seja útil para outras equipes. Aprender os conceitos escrevendo testes pequenos para que você possa usar suas habilidades de desenvolvimento de software e entregar sistemas ótimos.

Para quem isso foi feito

- Pessoas que se interessam em aprender Go.
- Pessoas que já sabem Go, mas querem explorar testes com TDD.

O que vamos precisar

- Um computador!
- Go instalado
- Um editor de texto
- Experiência com programação. Entendimento de conceitos como `if`, variáveis, funções etc.
- Se sentir confortável com o terminal

Traduzido com <3 por

- Davi Marcondes Moreira

github twitter

- Diego Nascimento

github twitter linkedin

- Edmilton Neves

site github twitter linkedin

- Jéssica Paz

site github twitter linkedin

- Lauren Ferreira

site github twitter linkedin

- Rafael Acioly

github twitter linkedin

Feedback

- Crie issues/submita PRs aqui ou [me envie um tweet em @quii](https://twitter.com/quii).
- Para a versão em português, submita um PR aqui ou entre em contato comigo pelo meu site.

MIT license

Logo criado por egonelbre Que estrela!

Por que criar testes unitários e como fazê-los dar certo

Vejam um vídeo meu falando sobre esse assunto

Se não gostar muito de vídeos, aqui vai o artigo relacionado a isso.

Software

A promessa do software é que ele pode mudar. É por isso que é chamado de _soft_ware: é mais maleável se comparado ao hardware. Uma boa equipe de engenharia deve ser um componente incrível para uma empresa, criando sistemas que podem evoluir com um negócio para manter seu valor de entrega.

Então por que somos tão ruins nisso? Quantos projetos que você ouve falar sobre que ultrapassam o nível da falha? Ou viram "legado" e precisam ser totalmente recriados (e a reescrita também acaba falhando)!

Mas como é que um software "falha"? Não dá para ele apenas ser modificado até estar correto? É isso que prometemos!

Muita gente costuma escolher o Go para criar sistemas porque a linguagem teve várias decisões que evitam que o software vire legado.

- Comparado à minha antiga vida de Scala onde descrevi como é fácil acabar se dando mal com a linguagem, o Go tem apenas 25 palavras-chave. *Muitos* sistemas podem ser criados a partir da biblioteca padrão e alguns outros pacotes pequenos. O que se espera é que com Go você possa escrever código, voltar a vê-lo 6 meses depois e ele ainda fazer sentido.

- As ferramentas relacionadas a testes, benchmarking, linting e shipping são incríveis se comparadas à maioria das alternativas.
- A biblioteca padrão é brilhante.
- Velocidade de compilação muito rápida para loops de feedback mais frequentes.
- A famigerada promessa da compatibilidade. Parece que Go vai receber `generics` e outras funcionalidades no futuro, mas os mantenedores prometeram que mesmo o código Go que você escreveram cinco anos atrás ainda vai compilar e funcionar. Eu literalmente passei semanas atualizando um projeto em Scala da versão 2.8 para a 2.10.

Com todas essas propriedades ótimas, ainda podemos acabar criando sistemas terríveis. Por isso, precisamos aplicar lições de engenharia de software que se aplicam independente do quão maravilhosa (ou não) sua linguagem seja.

Em 1974, um engenheiro de software esperto chamado Manny Lehman escreveu as leis de Lehman para a evolução do software.

> As leis descrevem um equilíbrio entre o desenvolvimento de software em uma ponta e a diminuição do progresso em outra.

É importante entender esses extremos para não acabar em um ciclo infinito de entregar sistemas que se tornam em legado e precisam ser reescritos novamente.

Lei da Mudança Contínua

> Qualquer software utilizado no mundo real precisa se adaptar ou vai se tornar cada vez mais obsoleto.

Parece óbvio que um software *precisa* mudar ou acaba se tornando menos útil, mas quantas vezes isso é ignorado?

Muitas equipes são incentivadas a entregar um projeto em uma data específica e passar para o próximo projeto. Se o software tiver "sorte", vai acabar na mão de outro grupo de pessoas para mantê-lo, mas é claro que nenhuma dessas pessoas o escreveu.

As pessoas se preocupam em escolher uma framework que vai ajudá-las a "entregar rapidamente", mas não focam na longevidade do sistema em termos de como precisa ser evoluído.

Mesmo se você for um engenheiro de software incrível, ainda vai cair na armadilha de não saber que futuro aguarda seu software. Já que o negócio muda, o código brilhante que você escreveu já não vai mais ser relevante.

Lehman estava contudo nos anos 70, porque nos deu outra lei para quebrarmos a cabeça.

Lei da Complexidade Crescente

> Enquanto o softwate evolui, sua complexidade aumenta. A não ser que um esforço seja investido para reduzi-la.

O que ele diz aqui é que não podemos ter equipes de software para funcionar apenas como fábricas de funcionalidades, inserindo mais e mais funcionalidades no software para que ele possa sobreviver a longo prazo.

Nós **temos** que lidar com a complexidade do sistema conforme o conhecimento do nosso domínio muda.

Refatoração

Existem *diversas* facetas na engenharia de software que mantêm um software maleável, como:

- Capacitação do desenvolvimento
- Em termos gerais, código "bom". Separação sensível de responsabilidades, etc
- Habilidades de comunicação
- Arquitetura
- Observabilidade
- Implantabilidade
- Testes automatizados
- Retornos de feedback

Vou focar na refatoração. Quantas vezes você já ouviu a frase "precisamos refatorar isso"? Provavelmente dita para uma pessoa desenvolvedora em seu primeiro dia de programação sem pensar duas vezes.

De onde essa frase vem? Por que refatorar é diferente de escrever código?

Sei que eu e muitas outras pessoas só *pensaram* que estavam refatorando, mas estávamos cometendo um erro.

Martin Fowler descreve como as pessoas entendem a refatoração errada aqui.

> No entanto, o termo "refatoração" costuma ser utilizado de forma inapropriada. Se alguém fala que um sistema ficará quebrado por alguns dias enquanto está sendo refatorado, pode ter certeza que eles não estão refatorando.

Então o que é refatoração?

Fatoração

Quando estudava matemática na escola, você provavelmente aprendeu fatoração. Aqui vai um exemplo bem simples:

- Calcule `1/2 + 1/4`

Para fazer isso você `fatora` os denominadores (você também pode conhecer como MMC, mínimo múltiplo comum), transformando a expressão em `2/4 + 1/4` que então pode se transformar em `3/4`.

Podemos tirar algumas lições importantes disso. Quando `fatoramos a expressão`, `não mudamos o que ela faz`. Ambas as expressões são iguais a `3/4`, mas facilitamos a forma como trabalhamos com esse resultado; trocar `1/2` por `2/4` torna nosso "domínio" mais fácil.

Quando refatora seu código, você tenta encontrar formas de tornar seu código mais fácil de entender e "encaixar" no seu entendimento atual do que o sistema precisa fazer. Mas é extremamente importante que **o comportamento do código não seja alterado**.

Exemplo em Go

Aqui está uma função que cumprimenta `nome` em uma `linguagem` específica:

```
func Ola(nome, linguagem string) string {

  if linguagem == "br" {
     return "Olá, " + nome
  }

  if linguagem == "fr" {
     return "Bonjour, " + nome
  }

  // e mais várias linguagens

  return "Hello, " + nome
}
```

Não é bom ter várias condicionais `if` e temos uma duplicação que concatena um cumprimento específico da linguagem com `,` e o `nome`. Logo, vou refatorar o código.

```
func Ola(nome, linguagem string) string {
      return fmt.Sprintf(
          "%s, %s",
          cumprimento(linguagem),
          nome,
      )
}

var cumprimentos = map[string]string {
  br: "Olá",
```

```
  fr: "Bonjour",
  // etc..
}

func cumprimento(linguagem string) string {
  cumprimento, existe := cumprimentos[linguagem]

  if existe {
     return cumprimento
  }

  return "Hello"
}
```

A natureza dessa refatoração não é tão importante. O que importa é que não mudei o comportamento do código.

Quando estiver refatorando, você pode fazer o que quiser: adicionar interfaces, tipos novos, funções, métodos etc. A única regra é que você não mude o comportamento do software.

Quando estiver refatorando o código, seu comportamento não deve ser modificado

Isso é muito importante. Se estiver mudando o comportamento enquanto refatora, você vai estar fazendo *duas* coisas de uma vez. Como engenheiros de software, aprendemos a dividir o sistema em diferentes arquivos/pacotes/funções/etc porque sabemos que tentar entender algo enorme e acoplado é difícil.

Não queremos ter que pensar sobre muitas coisas ao mesmo tempo porque é aí que cometemos erros. Já vi tantos esforços de refatoração falharem pelas pessoas que estavam desenvolvendo darem um passo maior que a perna.

Quando fazia fatorações nas aulas de matemática com papel e caneta, eu precisava verificar manualmente que não havia mudado o significado das expressões na minha cabeça. Como sabemos que não estamos mudando o comportamento quando refatoramos as coisas no código, especialmente em um sistema que não é tão simples?

As pessoas que escolhem não escrever testes vão depender do teste manual. Para quem não trabalha em um projeto pequeno, isso vai ser uma tremenda perda de tempo e não vai escalar a longo prazo.

Para ter uma refatoração segura, você precisa escrever testes unitários, porque eles te dão:

- Confiança de que você pode mudar o código sem se preocupar com mudar seu comportamento

- Documentação para humanos sobre como o sistema deve se comportar
- Feedback mais rápido e confiável que o teste manual

Exemplo em Go

Um teste unitário para a nossa função `Ola` pode ser feito assim:

```
func TestOla(t *testing.T) {
  obtido := Ola("Chris", br)
  esperado := "Olá, Chris"

  if obtido != esperado {
     t.Errorf("obtido '%s' esperado '%s'", obtido, esperado)
  }
}
```

Na linha de comando, posso executar `go test` e obter feedback imediato se minha refatoração alterou o comportamento da função. Na prática, é melhor aprender aonde fica o botão mágico que vai executar seus testes dentro do seu editor/IDE (ou rodar os testes sempre que salvar o arquivo).

Você deve entrar em uma rotina em que acaba fazendo:

- Refatorar uma parte pequena
- Executar testes
- Repetir

Tudo dentro de um ciclo de feedback contínuo para que você não caia em uma cilada e cometa erros.

Ter um projeto onde os seus principais comportamentos são testados unicamente e te dão feedback em menos de um segundo traz uma relação forte de segurança para refatorar sempre que for necessário. Isso nos ajuda a gerenciar a complexidade crescente que Lehman descreve.

Se testes unitários são tão bons, por que há resistência em escrevê-los?

De um lado, é possível ver pessoas (como eu) dizendo que testes unitários são importantes para a saúde do seu sistema a longo prazo, porque eles certificam que você possa continuar refatorando com confiança.

Do outro lado, é possível ver pessoas descrevendo experiências com testes unitários que na verdade *dificultaram* a refatoração.

Se pergunte o seguinte: com qual frequência você precisa mudar seus testes quando refatora? Estive em diversos projetos com boa cobertura de testes e mesmo assim os engenheiros estavam relutantes em refatorar por causa do esforço perceptível de alterar testes.

Esse é o oposto do que prometemos!

Por que isso acontece?

Imagine que te pediram para desenvolver um quadrado e você chegou à conclusão que seria necessário unir dois triângulos.

Figure 3: Dois triângulos retângulos formando um quadrado

Escrevemos nossos testes unitários nos baseando no nosso quadrado para ter certeza de que os lados são iguais e depois escrevemos alguns testes em relação aos nossos triângulos. Queremos ter certeza de que nossos triângulos são renderizados corretamente, então afirmamos que os ângulos somados dos triângulos dão 180 graus, ou verificamos que os dois são criados, etc etc. A cobertura de testes é muito importante e escrever esses testes é bem fácil, então por que não?

Algumas semanas depois, a Lei da Mudança Contínua bate no seu sistema e uma nova pessoa desenvolvedora faz algumas mudanças. Ela acredita que seria melhor se os quadrados fossem formados por dois retângulos ao invés dos dois triângulos.

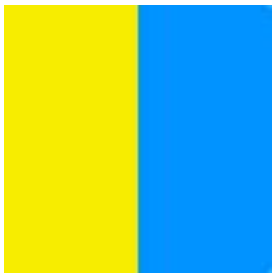

Figure 4: Dois retângulos formando um quadrado

Ela tenta fazer essa refatoração e percebe que alguns testes falharam. Ela quebrou algum comportamento realmente importante aqui? Agora ela tem que investigar esses testes de triângulo e entender o que está acontecendo.

Na verdade, não é tão importante que o quadrado seja formado por triângulo, mas **nossos testes fizeram com que isso parecesse mais importante do que deveria em relação aos detalhes da nossa implementação**

Favorecer o comportamento do teste ao invés do detalhe da implementação

Quando ouço pessoas reclamando sobre testes unitários, frequentemente o motivo é que eles estão em um nível errado de abstração. Eles testam detalhes da implementação, testando coisas muito específicas ou fazendo muitos mocks.

Acredito que isso deriva de uma falta de entendimento do que testes unitários são e perseguem métricas vaidosas (cobertura de testes).

Se estou apenas testando o comportamento, não deveríamos apenas escrever testes de sistema/caixa preta? Esses tipos de testes geram muito valor em termos de verificar as principais jornadas do usuário, mas costumam ser difíceis de escrever e lentos para rodar. Por esse motivos, eles não são muito úteis para a *refatoração* porque o ciclo de feedback é lento. Além disso, os testes de caixa preta tendem a não te ajudar muito com as causas de origem comparados aos testes unitários.

Logo, *qual* é o nível de abstração correto?

Escrevendo testes unitários de forma efetiva é um problema de design

Deixando testes de lado por um momento, é desejável "unidades" independentes e desacopladas dentro do seu sistema, centradas em torno de conceitos essenciais do seu domínio.

Gosto de imaginar essas unidades tão simples quanto blocos de Lego que têm APIs coerentes e que eu possa combinar com outros blocos para criar sistemas maiores. Por baixo dessas APIs pode haver várias coisas (tipos, funções etc) colaborando para fazê-las funcionar conforme esperado.

Por exemplo: se estiver escrevendo um banco em Go, você deve ter um pacote "conta". Ele vai te apresentar uma API que não vaza detalhes da implementação e é fácil de ser integrado.

Se tiver essas unidades que seguem essas propriedades, você consegue escrever testes unitários para suas APIs públicas. *Por definição*, esses testes só podem testar os comportamentos importantes. Por baixo dos panos dessas unidades, fico livre para refatorar a implementação o quanto eu precisar e os testes para a maior parte dela não devem me atrapalhar.

Mas são testes unitários, mesmo?

SIM. Testes unitários são feitos para "unidades", como já descrevi. Eles *nunca* devem ser feitos para uma classe/função/seja lá o que for.

Conclusão

Falamos sobre

- Refatoração
- Testes unitários
- Desenvolvimento de unidade

O que podemos começar a ver é que essas facetas do desenvolvimento de software reforçam uma à outra.

Refatoração

- Nos dá sinais sobre nossos testes unitários. Se precisamos fazer validações manuais, precisamos de mais testes. Se testes estão falhando incorretamente, então nossos testes estão no nível errado de abstração (ou não têm valor e precisam ser deletados).
- Nos ajuda a lidar com as complexidades dentro e entre nossas unidades.

Testes unitários

- Nos dão a garantia para refatoração.
- Verificam e documentam o comportamento de nossas unidades.

Unidades (bem definidas)

- Facilitam a escrita de testes unitários *significativos*.
- Facilitam a refatoração.

Há um processo que nos ajuda a alcançar um ponto onde podemos refatorar nosso código para lidar com a complexidade e manter nossos sistemas maleáveis?

Por que Desenvolvimento Orientado a Testes (TDD)

Algumas pessoas levam as citações de Lehman sobre como o software deve mudar a sério demais e elaboram sistemas complexos demais, gastando muito tempo tentando prever o impossível para criar o sistema extensível "perfeito" e acabam entendendo da forma errada e chegando a lugar nenhum.

Isso vem da época das trevas do software onde um time de analistas costumava perder seis meses escrevendo um documento de requerimentos e a equipe de arquitetura perdia outros seis meses para desenvolvê-lo e alguns anos depois o projeto inteiro falhava.

Eu disse que era uma época das trevas, mas isso ainda acontece!

O movimento ágil nos ensina que precisamos trabalhar de forma iterativa, começando com pouca coisa e evoluindo o software para que tenhamos retorno rápido do design do nosso software e como ele trabalha com usuários reais; o TDD reforça essa abordagem.

O TDD aborda as leis que Lehman fala sobre e outras lições difíceis aprendidas no decorrer da história encorajando uma metodologia de refatoração constante e entrega contínua.

Etapas pequenas

- Escrever um teste pequeno para uma unidade do comportamento desejado
- Verificar que o teste falha com um erro claro (vermelho)
- Escrever o mínimo de código para fazer o teste passar (verde)
- Refatorar (azul)
- Repetir

Conforme você pratica, essa mentalidade vai se tornar natural e rápida.

Você vai esperar que esse ciclo de feedback não leve muito tempo e se sentir desconfortável se estiver em um estado em que seu sistema não está "verde", já que isso pode indicar que você pode ter deixado algo passar.

Você sempre vai desenvolver de forma a criar funcionalidades pequenas & úteis confortavelmente reforçadas pelo feedback dos seus testes.

Resumindo

- O ponto forte do software é que podemos mudá-lo. A *maioria* dos software requer mudança com o tempo de formas imprevisíveis; não tente pensar muito à frente porque é difícil prever o futuro.
- Ao invés disso, precisamos criar nosso software de forma que ele possa se manter maleável. Para mudar o software precisamos refatorá-lo conforme ele evolui, ou vai acabar virando uma bagunça.
- Um bom conjunto de testes pode te ajudar a refatorar mais rápido e de forma menos estressante.
- Escrever bons testes unitários é um problema de design. Logo, pense em estruturar seu código de forma que ele tenha unidades significativas que possam ser unidas como blocos de Lego.

- O TDD pode ajudar e te forçar a desenvolver softwares bem fatorados continuamente, reforçados por testes para te ajudar com futuros trabalhos que podem chegar.

Instalação do Go: defina seu ambiente para produtividade

As instruções oficiais de instalação do Go estão disponíveis aqui.

Esse guia vai presumir que você está usando um gerenciador de pacotes como Homebrew, Chocolatey, Apt ou yum.

Para propósitos de demonstração, vamos te mostrar o procedimento de instalação para o OSX usando Homebrew.

Instalação

Mac OSX

O processo de instalação é bem simples. Primeiro, o que você precisa fazer é executar o comando abaixo pra instalar o homebrew (brew). O Brew depende do Xcode, então você deve se certificar de instalá-lo primeiro.

```
xcode-select --install
```

Depois, execute o comando a seguir para instalar o homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/ins
```

Agora você consegue instalar o Go:

```
brew install go
```

Siga todas as instruções recomendadas pelo seu gerenciador de pacotes. ***Nota*** *cada grupo de instruções varia de sistema operacional para sistema operacional.*

Você pode verificar a instalação com:

```
$ go version
go version go1.10 darwin/amd64
```

Linux

O processo de instalação é bem simples. Primeiro você precisa escolher e baixar a versão do Go que você deseja instalar. Para isso acesse o site oficial da linguagem e copie o da versão desejada (Ex.: https://dl.google.com/go/go1.13.linux-amd64.tar.gz). Recomendamos instalar sempre a versão mais atual.

Para baixá-lo execute o seguinte comando no seu terminal.

```
# selecione a versão  que você deseja instalar, no nosso exemplo estamos utilizando a versão
VERSAO_GO=1.13
cd ~
curl -O "https://dl.google.com/go/go${VERSAO_GO}.linux-amd64.tar.gz"
```

Agora descompacte os arquivos com o seguinte comando.

```
tar xvf "go${VERSAO_GO}.linux-amd64.tar.gz"
```

E em seguida, mova os arquivos para o diretório de binário do seu usuário.

```
sudo mv go /usr/local
```

Agora teste a sua instalação.

```
go version
go version go1.13 linux/amd64
```

Windows

Para usuários de Windows existem duas formas de instalação, através de um arquivo ZIP que requer que você configure algumas variáveis de ambiente ou uma arquivo MSI que faz toda a configuração automaticamente. Primeiro faça download da versão que você deseja instalar. Para isso acesse o site oficial da linguagem e copie o arquivo da versão desejada (Ex.: https://dl.google.com/go/go1.13.1.windows-amd64.msi). Recomendamos instalar sempre a versão mais atual.

Instalação via MSI

Abra o arquivo MSI e siga os passos da instalação. Por padrão o instalador adiciona o Go na pasta `C:\Go`.

O instalador adiciona o caminho `C:\Go\bin` na variável de ambiente "Path" e cria a variável de usuário "GOPATH" com o caminho `C:\Users\%USER%\go`.

Instalação via ZIP

Extraia os arquivos do arquivo ZIP no diretório de sua preferência.

Adicione na variável de ambiente "Path" o caminho para a pasta "bin" dos arquivos de Go. Na busca do menu Iniciar, digite "Variáveis" escolha a opção "Editar as variáveis de ambiente do sistema. Na aba"Avançado" clique em "Variáveis de Ambiente", localize a variável "Path" e clique em Editar > Novo e preencha com o caminho escolhido (Ex: `C:/minha-pasta-go/bin`).

Quando ocorre alteração nas variáveis de ambiente no Windows é necessário reiniciar o sistema.

Nos próximos passos vamos configurar o ambiente Go. As instruções abaixo valem tanto para sistema operacional OSX quanto para o Linux. O ambiente

Windows pode requisitar configurações a mais e por isso é importante seguir a documentação oficial.

O Ambiente Go

O Go divide opiniões.

Por convenção, todo o código Go é colocado dentro de apenas um workspace (pasta). Esse workspace pode estar em qualquer lugar da sua máquina. Se você não especificar, o Go vai definir o $HOME/go como workspace padrão. Ele é identificado (e modificado) pela variável de ambiente GOPATH.

Você precisa definir a variável de ambiente para que possa utilizar futuramente em scripts, shells etc.

Atualize seu .bash_profile para conter os seguintes `exports`:

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

Nota você deve abrir um novo terminal para definir essas variáveis de ambiente.

O Go presume que seu workspace contenha uma estrutura de diretórios específica.

Ele coloca seus arquivos em três diretórios: todo o código-fonte fica em `src`, os objetos dos pacotes ficam em `pkg` e os programas compilados são colocados em `bin`. É possível criar esses diretórios com o comando a seguir:

```
mkdir -p $GOPATH/src $GOPATH/pkg $GOPATH/bin
```

Agora você é capaz de usar o *go get* para que o `src/package/bin` seja instalado corretamente no diretório $GOPATH/xxx apropriado.

Editor Go

A escolha de editor é bem pessoal. Você pode já ter um de sua preferência que tem suporte a Go. Se não tiver, leve em consideração um Editor como o Visual Studio Code, que tem um suporte exceptional à linguagem.

Você pode instalá-lo com o comando a seguir:

```
brew cask install visual-studio-code
```

Confirme que o VS Code foi instalado corretamente executando o seguinte comando:

```
code .
```

O VS Code é lançado com poucos softwares habilidados. Você pode habilitar novos softwares instalando extensões. Para adicionar o suporte a Go, você deve

instalar uma extensão. Existem várias disponíveis para o VS Code, mas uma excepcional é a do Luke Hoban. Instale-a da forma a seguir:

```
code --install-extension ms-vscode.go
```

Quando abrir um arquivo Go pela primeira vez no VS Code, ele vai indicar que ferramentas de análises estão faltando. Clique no botão para instalá-las. A lista de ferramentas que são instaladas (e usadas) pelo VS Code estão disponíveis aqui.

Debugger do Go

Uma boa opção para debugar seus programas em Go (que é integrado com o VS Code) é o Delve. Ele pode ser instalado da seguinte maneira usando `go get`:

```
go get -u github.com/go-delve/delve/cmd/dlv
```

Linter do Go

Uma melhoria sob o linter padrão pode ser configurada usando o GolangCI-Lint.

Que pode ser instalada da seguinte forma:

```
go get -u github.com/golangci/golangci-lint/cmd/golangci-lint
```

Refatoração e suas ferramentas

Uma grande ênfase nesse livro é dada na importância da refatoração.

Suas ferramentas podem te ajudar a fazer uma refatoração com maior confiança.

Você deve ter familiaridade o suficiente com seu editor para performar as ações a seguir com uma simples combinação de teclas:

- **Extrair/alinhar variável**. Ser capaz de pegar valores mágicos e dar um nome a eles vai simplificar seu código rapidamente.
- **Extrair método/função**. É crucial ser capaz de tirar uma seção do código e extrair funções/métodos.
- **Renomear**. Você deve se sentir capaz de renomear símbolos no decorrer dos arquivos com confiança.
- **go fmt**. O Go tem um formatador nativo chamado `go fmt`. Seu editor deve executar esse comando a cada vez que salvar o arquivo.
- **Executar testes**. Não precisa nem dizer que você deve ser capaz de fazer todos os pontos acima e então re-executar seus testes rapidamente para certificar que sua refatoração não quebrou nada.

Além disso, para te ajudar a trabalhar com seu código, você deve ser capaz de:

- **Verificar a assinatura da função**. Nunca tenha dúvida sobre a forma de chamar uma função em Go. Sua IDE deve descrever uma função em termos de sua documentação, seus parâmetros e o que ela retorna.
- **Ver a definição da função**. Se não tiver certeza sobre como uma função funciona, você deve ser capaz de ir para o código fonte de descobrir por si facilmente.
- **Encontrar usos de um símbolo**. Ser capaz de ver o contexto de uma função sendo chamada pode te ajudar com o processo de refatoração.

Dominar suas ferramentas vai te ajudar a concentrar no código e reduzir a troca de contexto.

Resumindo

Nesse ponto você já deve ter o Go instalado, um editor disponível e algumas ferramentas básicas configuradas. O Go tem um ecossistema enorme de produtos feitos por outras pessoas. Identificamos alguns componentes úteis aqui, mas você pode encontrar uma lista mais completa no Awesome Go.

Olá, mundo

Você pode encontrar todos os códigos para esse capítulo aqui

É comum o primeiro programa em uma nova linguagem ser um *Olá, mundo*.

No capítulo anterior, discutimos sobre como Go pode ser pouco flexível em relação a onde colocar seus arquivos.

Crie um diretório no seguinte caminho: `$GOPATH/src/github.com/{seu-lindo-nome-de-usuario}/ola`.

Se estiver num ambiente baseado em *unix* , seu nome de usuário do sistema operacional for "ariel" e tiver motivação em seguir as convenções do Go sobre `$GOPATH` (que é a maneira mais fácil de configurar) você pode executar `mkdir -p $GOPATH/src/github.com/ariel/ola`.

Crie um arquivo chamado `ola.go` no diretório mencionado e escreva o código abaixo. Para executá-lo, basta digitar `go run ola.go` no console.

```
package main

import "fmt"

func main() {
    fmt.Println("Olá, mundo")
}
```

Como isso funciona?

Quando você escreve um programa em Go, há um pacote `main` definido com uma função(`func`) `main` (principal) dentro dele. Os pacotes são maneiras de agrupar códigos escritos em Go.

A palavra reservada `func` é utilizada para que você defina uma função com um nome e um conteúdo.

Ao usar `import "fmt"`, estamos importando um pacote que contém a função `Println` que será utilizada para imprimir (escrever) um valor na tela.

Como testar isso?

Como você testaria isso? É bom separar seu "domínio"(suas regras de negócio) do resto do mundo (efeitos colaterais). A função `fmt.Println` é um efeito colateral (que está imprimindo um valor no ***stdout*** *[saída padrão do terminal]*) e a string que estamos enviando para dentro dela é nosso domínio.

Então, vamos separar essas referências para ficar mais fácil para testarmos.

```
package main

import "fmt"

func Ola() string {
    return "Olá, mundo"
}

func main() {
    fmt.Println(Ola())
}
```

Criamos uma nova função usando `func`, mas dessa vez adicionamos outra palavra reservada `string` na sua definição. Isso significa que essa função terá como retorno uma `string` (*cadeia de caracteres*).

Agora, criaremos outro arquivo chamado `ola_test.go` onde iremos escrever um teste para a nossa função `Ola`.

```
package main

import "testing"

func TestOla(t *testing.T) {
    resultado := Ola()
    esperado := "Olá, mundo"

    if resultado != esperado {
```

```
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

Antes de explicar, vamos rodar o código. Execute `go test` no seu terminal. Ele deve passar! Para verificar, tente quebrar o teste de alguma forma mudando a string `esperado`.

Perceba que você não precisa usar várias frameworks (ou bibliotecas) de testes e complicar as coisas tentando instalá-las. Tudo o que você precisa está pronto na linguagem e a sintaxe é a mesma para o resto dos códigos que você irá escrever.

Escrevendo testes

Escrever um teste é como escrever uma função, com algumas regras:

- Precisa estar em um arquivo com um nome parecido com `xxx_test.go`
- A função de teste precisa começar com a palavra `Test`
- A função de teste recebe um único argumento, que é `t *testing.T`

Por enquanto é o bastante para saber que o nosso `t` do tipo `*testing.T` é a nossa porta de entrada para a ferramenta de testes e assim você poderá utilizar o `t.Fail()` quando precisar relatar um erro.

Abordando alguns novos tópicos:

`if`

Instruções `if` em Go são muito parecidas com as de outras linguagens.

Declarando variáveis

Estamos declarando algumas variáveis com a sintaxe `nomeDaVariavel := valor`, que nos permite reutilizar alguns valores nos nossos testes de maneira legível.

`t.Errorf`

Estamos chamando o *método* `Errorf` em nosso `t` que irá imprimir uma mensagem e falhar o teste. O sufixo `f` no final de `Errorf` representa que podemos formatar e montar uma string com valores inseridos dentro de valores de preenchimentos `%s`. Quando fazemos um teste falhar, devemos ser bastante claros com o que aconteceu.

Iremos explorar a diferença entre métodos e funções depois.

go doc

Outra funcionalidade importante do Go é sua documentação. Você pode ver a documentação na sua máquina rodando `godoc -http :8000`. Se acessar localhost:8000/pkg no seu navegador, verá todos os pacotes instalados no seu sistema.

A vasta biblioteca padrão da linguagem tem uma documentação excelente com exemplos. Deve valer a pena dar uma olhada em http://localhost:8000/pkg/testing/ para verificar o que está disponível para você.

Olá, VOCÊ

Agora que temos um teste, podemos iterar sobre nosso software de maneira segura.

No último exemplo, escrevemos o teste somente *depois* do código ser escrito apenas para que você pudesse ter um exemplo de como escrever um teste e declarar uma função. A partir de agora, *escreveremos os testes primeiro*.

Nosso próximo requisito é nos deixar especificar quem recebe a saudação.

Vamos começar especificando esses requisitos em um teste. Estamos praticando TDD (Desenvolvimento Orientado a Testes) de forma bastante simples e que nos permite ter certeza que nosso teste está *testando* o que precisamos. Quando você escreve testes retroativamente existe o risco que seu teste possa continuar passando mesmo que o código não esteja funcionando como esperado.

```
package main

import "testing"

func TestOla(t *testing.T) {
    resultado := Ola("Chris")
    esperado := "Olá, Chris"

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

Agora, rodando `go test`, deve ter aparecido um erro de compilação:

```
./ola_test.go:6:18: too many arguments in call to Ola
    have (string)
    want ()
```

```
./ola_test.go:6:18: argumentos demais na chamada para Ola
    tem (string)
    quer ()
```

Quando estiver usando uma linguagem estaticamente tipada como Go, é importante *dar atenção ao compilador*. O compilador entende como seu código deve se encaixar, não delegando essa função para você.

Neste caso, o compilador está te falando o que você precisa fazer para continuar. Temos que mudar a nossa função `Ola` para receber um argumento.

Edite a função `Ola` para que um argumento do tipo string seja aceito:

```
func Ola(nome string) string {
    return "Olá, mundo"
}
```

Se tentar rodar seus testes novamente, seu arquivo `main.go` irá falhar durante a compilação porque você não está passando um argumento. Passe "mundo" como argumento para fazer o teste passar.

```
func main() {
    fmt.Println(Ola("mundo"))
}
```

Agora, quando for rodar seus testes, você verá algo parecido com isso:

```
ola_test.go:10: resultado 'Olá, mundo', esperado 'Olá, Chris'
```

Finalmente temos um programa que compila, mas que não está satisfazendo os requisitos de acordo com o teste.

Vamos, então, fazer o teste passar usando o argumento `nome` e concatenar com `Olá,`

```
func Ola(nome string) string {
    return "Olá, " + nome
}
```

Quando você rodar os testes, eles irão passar. É comum como parte do ciclo do TDD *refatorar* o nosso código agora.

Uma nota sobre versionamento de código

Nesse ponto, se você estiver usando um versionamento de código (que você deveria estar fazendo!) eu faria um `commit` do código no estado atual. Agora, temos um software funcional suportado por um teste.

No entanto, eu *não faria* um push para a master, pois planejo refatorar em breve. É legal fazer um commit nesse ponto porque você pode se perder com a refatoração. Fazendo um commit você pode sempre voltar para a última versão funcional do seu software.

Não tem muita coisa para refatorar aqui, mas podemos introduzir outro recurso da linguagem: *constantes*.

Constantes

Constantes podem ser definidas como o exemplo abaixo:

```
const prefixoOlaPortugues = "Olá, "
```

Agora, podemos refatorar nosso código:

```
const prefixoOlaPortugues = "Olá, "

func Ola(nome string) string {
    return prefixoOlaPortugues + nome
}
```

Depois da refatoração, rode novamente os seus testes para ter certeza que você não quebrou nada.

Constantes devem melhorar a performance da nossa aplicação, assim como evitar que você crie uma string `"Ola, "` para cada vez que `Ola` é chamado.

Para esclarecer, o aumento de performance é incrivelmente insignificante para esse exemplo! Mas vale a pena pensar em criar constantes para capturar o significado dos valores e, às vezes, para ajudar no desempenho.

Olá, mundo... novamente

O próximo requisito é: quando nossa função for chamada com uma string vazia, ela precisa imprimir o valor padrão "Olá, mundo", ao invés de "Olá,".

Começaremos escrevendo um novo teste que irá falhar

```
func TestOla(t *testing.T) {

    t.Run("diz olá para as pessoas", func(t *testing.T) {
        resultado := Ola("Chris")
        esperado := "Olá, Chris"

        if resultado != esperado {
            t.Errorf("resultado '%s', esperado '%s'", got, want)
        }
    })

    t.Run("diz 'Olá, mundo' quando uma string vazia for passada", func(t *testing.T) {
        resultado := Ola("")
        esperado := "Olá, mundo"

        if resultado != esperado {
            t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
        }
```

```
    })
}
```

Aqui nós estamos apresentando outra ferramenta em nosso arsenal de testes, os *subtestes*. Às vezes, é útil agrupar testes em torno de uma "coisa" e, em seguida, ter *subtestes* descrevendo diferentes cenários.

O benefício dessa abordagem é que você poderá construir um código que pode ser compartilhado por outros testes.

Há um código repetido quando verificamos se a mensagem é o que esperamos.

A refatoração não vale *apenas* para o código de produção!

É importante que seus testes *sejam especificações claras* do que o código precisa fazer.

Podemos e devemos refatorar nossos testes.

```
func TestOla(t *testing.T) {
    verificaMensagemCorreta := func(t *testing.T, resultado, esperado string) {
        t.Helper()
        if resultado != esperado {
            t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
        }
    }

    t.Run("diz olá para as pessoas", func(t *testing.T) {
        resultado := Ola("Chris")
        esperado := "Olá, Chris"
        verificaMensagemCorreta(t, resultado, esperado)
    })

    t.Run("'Mundo' como padrão para 'string' vazia", func(t *testing.T) {
        resultado := Ola("")
        esperado := "Olá, Mundo"
        verificaMensagemCorreta(t, resultado, esperado)
    })
}
```

O que fizemos aqui?

Refatoramos nossa asserção em uma função. Isso reduz a duplicação e melhora a legibilidade de nossos testes. No Go, você pode declarar funções dentro de outras funções e atribui-las a variáveis. Você pode chamá-las, assim como as funções normais. Precisamos passar `t * testing.T` como parâmetro para que possamos dizer ao código de teste que ele falhará quando necessário.

`t.Helper ()` é necessário para dizermos ao conjunto de testes que este é um método auxiliar. Ao fazer isso, quando o teste falhar, o número da linha relatada

estará em nossa chamada de função, e não dentro do nosso auxiliar de teste. Isso ajudará outros desenvolvedores a rastrear os problemas com maior facilidade. Se você ainda não entendeu, comente, faça um teste falhar e observe a saída do teste.

Agora que temos um teste bem escrito falhando, vamos corrigir o código usando um `if`.

```
const prefixoOlaPortugues = "Olá, "

func Ola(nome string) string {
    if nome == "" {
        nome = "Mundo"
    }
    return prefixoOlaPortugues + nome
}
```

Se executarmos nossos testes, veremos que ele satisfaz o novo requisito e não quebramos acidentalmente a outra funcionalidade.

De volta ao controle de versão

Agora estamos felizes com o código. Eu adicionaria mais um commit ao anterior para que possamos verificar o quão adorável ficou o nosso código com os testes.

Disciplina

Vamos repassar o ciclo novamente:

- Escrever um teste
- Compilar o código sem erros
- Rodar o teste, ver o teste falhar e certificar que a mensagem de erro faz sentido
- Escrever a quantidade mínima de código para o teste passar
- Refatorar

Este ciclo pode parecer tedioso, mas se manter nesse ciclo de feedback é importante.

Ele não apenas garante que você tenha *testes relevantes*, como também ajuda a *projetar um bom software* refatorando-o com a segurança dos testes.

Ver a falha no teste é uma verificação importante porque também permite que você veja como é a mensagem de erro. Para quem programa, pode ser muito difícil trabalhar com uma base de código que, quando há falha nos testes, não dá uma ideia clara de qual é o problema.

Assegurando que seus testes sejam rápidos e configurando suas ferramentas para que a execução de testes seja simples, você pode entrar em um estado de fluxo

ao escrever seu código.

Ao não escrever testes, você está comprometendo-se a verificar manualmente seu código executando o software que interrompe seu estado de fluxo, o que não economiza tempo, especialmente a longo prazo.

Continue! Mais requisitos

Caramba, temos mais requisitos. Agora precisamos suportar um segundo parâmetro, especificando o idioma da saudação. Se for passado um idioma que não reconhecemos, use como padrão o português.

Devemos ter certeza de que podemos usar o TDD para aprimorar essa funcionalidade facilmente!

Escreva um teste para um usuário, passando espanhol. Adicione-o ao conjunto de testes existente.

```
    t.Run("em espanhol", func(t *testing.T) {
        resultado := Ola("Elodie", "espanhol")
        esperado := "Hola, Elodie"
        verificaMensagemCorreta(t, resultado, esperado)
    })
```

Lembre-se de não trapacear! *Primeiro os testes*. Quando você tenta executar o teste, o compilador deve reclamar porque está chamando `Ola` com dois argumentos ao invés de um.

```
./ola_test.go:27:19: too many arguments in call to Ola
    have (string, string)
    want (string)
```

Acerte os problemas de compilação, adicionando um novo argumento do tipo `string` ao método `Ola`:

```
func Ola(nome string, idioma string) string {
    if nome == "" {
        nome = "Mundo"
    }
    return prefixoOlaPortugues + nome
}
```

Quando você tentar executar o teste novamente, ele se queixará da função não ter recebido argumentos o suficiente para `Ola` nos seus outros testes em `ola.go`:

```
./ola.go:15:19: not enough arguments in call to Ola
    have (string)
    want (string, string)
```

Corrija-os passando `strings` vazia. Agora todos os seus testes devem compilar *e* passar, além do nosso novo cenário:

```
ola_test.go:29: resultado 'Olá, Elodie', esperado 'Hola, Elodie'
```

Podemos usar `if` aqui para verificar se o idioma é igual a "espanhol" e, em caso afirmativo, alterar a mensagem:

```
func Ola(nome string, idioma string) string {
    if nome == "" {
        nome = "Mundo"
    }

    if idioma == "Espanhol" {
        return "Hola, " + nome
    }

    return prefixoOlaPortugues + nome
}
```

Os testes devem passar agora.

Agora é hora de *refatorar*. Você verá alguns problemas no código, sequências de caracteres "mágicas", algumas das quais são repetidas. Tente refatorar você mesmo, a cada alteração, execute novamente os testes para garantir que sua refatoração não esteja quebrando nada.

```
const espanhol = "espanhol"
const prefixoOlaPortugues = "Olá, "
const prefixoOlaEspanhol = "Hola, "

func Ola(nome string, idioma string) string {
    if nome == "" {
        nome = "Mundo"
    }

    if idioma == espanhol {
        return prefixoOlaEspanhol + nome
    }

    return prefixoOlaPortugues + nome
}
```

Francês

- Escreva um teste que verifique que quando passamos o idioma `"francês"`, obtemos `"Bonjour, "`
- Veja o teste falhar, verifique se a mensagem de erro é fácil de ler
- Faça a mínima alteração de código o suficiente para que o teste passe

Você pode ter escrito algo parecido com isso:

```
func Ola(nome string, idioma string) string {
    if nome == "" {
        nome = "Mundo"
    }

    if idioma == espanhol {
        return prefixoOlaEspanhol + nome
    }

    if idioma == frances {
        return prefixoOlaFrances + nome
    }

    return prefixoOlaPortugues + nome
}
```

switch

Quando você tem muitas instruções `if` verificando um valor específico, é comum usar uma instruçãoswitch. Podemos usar o `switch` para refatorar o código facilitando a leitura e a sua extensão, caso desejarmos adicionar suporte a mais idiomas posteriormente.

```
func Ola(nome string, idioma string) string {
    if nome == "" {
        nome = "Mundo"
    }

    prefixo := prefixoOlaPortugues

    switch idioma {
    case frances:
        prefixo = prefixoOlaFrances
    case espanhol:
        prefixo = prefixoOlaEspanhol
    }

    return prefixo + nome
}
```

Faça um teste para incluir agora uma saudação no idioma de sua escolha e você deve ver como é simples estender nossa *fantástica* função.

Uma...última...refatoração?

Você pode achar que talvez nossa função esteja ficando um pouco grande. A refatoração mais simples para isso seria extrair algumas funcionalidades para outra função.

```
func Ola(nome string, idioma string) string {
    if nome == "" {
        nome = "Mundo"
    }

    return prefixodeSaudacao(idioma) + nome
}

func prefixodeSaudacao(idioma string) (prefixo string) {
    switch idioma {
    case frances:
        prefixo = prefixoOlaFrances
    case espanhol:
        prefixo = prefixoOlaEspanhol
    default:
        prefixo = prefixoOlaPortugues
    }
    return
}
```

Alguns novos conceitos:

- Em nossa assinatura de função, criamos um valor de retorno chamado `(prefixo string)`.
- Isso criará uma variável chamada `prefixo` na nossa função.
- Lhe será atribuído o valor "zero". Isso dependendo do tipo, por exemplo, para `int` será `0` e para strings será `""`.
  - Você pode retornar o que quer que esteja definido, apenas chamando `return` ao invés de `return prefixo`.
- Isso será exibido no `go doc` para sua função, para que possa tornar a intenção do seu código mais clara.
- `default` será escolhido caso o valor recebido não corresponda a nenhuma das outras instruções `case` do `switch`.
- O nome da função começa com uma letra minúscula. As funções públicas em *Go* começam com uma letra maiúscula e as privadas, com minúsculas. Não queremos que as partes internas do nosso algoritmo sejam expostas ao mundo, portanto tornamos essa função privada.

Resumindo

Quem imaginaria que você poderia tirar tanto proveito de um `Olá, mundo`?

Até agora você deve ter alguma compreensão de:

Algumas das sintaxes da linguagem *Go* para:

- Escrever testes
- Declarar funções, com argumentos e tipos de retorno
- `if`, `const` e `switch`
- Declarar variáveis e constantes

O processo TDD e *por que* as etapas são importantes

- *Escreva um teste que falhe e veja-o falhar*, para que saibamos que escrevemos um teste *relevante* para nossos requisitos e vimos que ele produz uma *descrição da falha fácil de entender*
- Escrever a menor quantidade de código para fazer o teste passar, para que saibamos que temos software funcionando
- *Em seguida*, refatorar, tendo a segurança de nossos testes para garantir que tenhamos um código bem feito e fácil de trabalhar

No nosso caso, passamos de `Ola()` para `Ola("nome")`, para `Ola ("nome","Francês ")` em etapas pequenas e fáceis de entender.

Naturalmente, isso é trivial comparado ao software do "mundo real", mas os princípios ainda permanecem. O TDD é uma habilidade que precisa de prática para se desenvolver. No entanto, você será muito mais facilidade em escrever software sendo capaz de dividir os problemas em pedaços menores que possa testar.

Inteiros

Você pode encontrar todos os códigos desse capítulo aqui

Inteiros funcionam como é de se esperar. Vamos escrever uma função de soma para testar algumas coisas. Crie um arquivo de teste chamado `adicionador_test.go` e escreva o seguinte código.

nota: Os arquivos-fonte de Go devem ter apenas um `package`(pacote) por diretório, verifique se os arquivos estão organizados separadamente. Aqui tem uma boa explicação sobre isso (em inglês).

Escreva o teste primeiro

```
package inteiros

import "testing"
```

```
func TestAdicionador(t *testing.T) {
    soma := Adiciona(2, 2)
    esperado := 4

    if soma != esperado {
        t.Errorf("esperado '%d', resultado '%d'", esperado, soma)
    }
}
```

Você deve ter notado que estamos usando `%d` como string de formatação, em vez de `%s`. Isso porque queremos que ele imprima um valor inteiro e não uma string. Observe também que não estamos mais usando o pacote `main`, em vez disso, definimos um pacote chamado `inteiros`, pois o nome sugere que ele agrupará funções para trabalhar com números inteiros, como Adiciona.

Tente executar o teste

Execute o test com `go test`

Inspecione o erro de compilação

```
./adicionador_test.go:6:9: undefined: Adiciona
```

Escreva a quantidade mínima de código para o teste rodar e verifique o erro na saída do teste

Escreva apenas o suficiente de código para satisfazer o compilador *e nada mais* - lembre-se de que queremos verificar se nossos testes falham pelo motivo certo.

```
package inteiros

func Adiciona(x, y int) int {
    return 0
}
```

Quando você tem mais de um argumento do mesmo tipo (no nosso caso dois inteiros) ao invés de ter `(x int, y int)` você pode encurtá-lo para `(x, y int)`.

Agora execute os testes. Devemos ficar felizes que o teste esteja relatando corretamente o que está errado.

```
adicionador_test.go:10: esperado '4', resultado '0'
```

Você deve ter percebido que aprendemos sobre o *valor de retorno nomeado* na última seção, mas não estamos usando aqui. Ele geralmente deve ser usado quando o significado do resultado não está claro no contexto. No nosso caso, é muito claro que a função `Adiciona` irá adicionar os parâmetros. Você pode consultar esta wiki para mais detalhes.

Escreva código o suficiente para fazer o teste passar

No sentido estrito de TDD, devemos escrever a *quantidade mínima de código para fazer o teste passar*. Uma pessoa pretenciosa pode fazer isso:

```
func Adiciona(x, y int) int {
    return 4
}
```

Ah hah! Frustração mais uma vez! TDD é uma farsa, né?

Poderíamos escrever outro teste, com números diferentes para forçar o teste a falhar, mas isso parece um jogo de gato e rato.

Quando estivermos mais familiarizados com a sintaxe do Go, apresentarei uma técnica chamada Testes Baseados em Propriedade, que interromperá a irritação das pessoas e ajudará a encontrar bugs.

Por enquanto, vamos corrigi-lo corretamente:

```
func Adiciona(x, y int) int {
    return x + y
}
```

Se você executar os testes novamente, eles devem passar.

Refatoração

Não há muitas melhorias que possamos fazer aqui.

Anteriormente, vimos como nomear o argumento de retorno que aparece na documentação e também na maioria dos editores de código.

Isso é ótimo porque ajuda na usabilidade do código que você está escrevendo. É preferível que um usuário possa entender o uso de seu código apenas observando a assinatura de tipo e a documentação.

Você pode adicionar documentação em funções escrevendo comentários, e elas aparecerão no Go Doc como quando você olha a documentação da biblioteca padrão.

```
// Adiciona recebe dois inteiros e retorna a soma deles
func Adiciona(x, y int) int {
    return x + y
}
```

Exemplos

Se realmente quer ir além, você pode fazer exemplos. Você encontrará muitos exemplos na documentação da biblioteca padrão.

Muitas vezes, exemplos encontrados fora da base de código, como um arquivo readme, ficam desatualizados e incorretos em comparação com o código real, porque eles não são verificados.

Os exemplos de Go são executados da mesma forma que os testes, para que você possa ter certeza de que eles refletem o que o código realmente faz.

Exemplos são compilados (e opcionalmente executados) como parte do conjunto de testes de um pacote.

Como nos testes comuns, os exemplos são funções que residem nos arquivos _test.go de um pacote. Adicione a seguinte função ExampleAdiciona no arquivo `adicionador_test.go`.

```
func ExampleAdiciona() {
    soma := Adiciona(1, 5)
    fmt.Println(soma)
    // Output: 6
}
```

> obs: As palavras Example e Output foram mantidas em inglês para a execução correta do código.

(Se o seu editor não importar os pacotes automaticamente, a etapa de compilação irá falhar porque você não colocou o `import "fmt"` no `adicionador_test.go`. É altamente recomendável que você pesquise como ter esses tipos de erros corrigidos automaticamente em qualquer editor que esteja usando.)

Se o seu código mudar fazendo com que o exemplo não seja mais válido, você vai ter um erro de compilação.

Executando os testes do pacote, podemos ver que a função de exemplo é executada sem a necessidade de ajustes:

```
$ go test -v
=== RUN   TestAdicionador
--- PASS: TestAdicionador (0.00s)
=== RUN   ExampleAdiciona
--- PASS: ExampleAdiciona (0.00s)
```

Note que a função de exemplo não será executada se você remover o comentário "// Output: 6". Embora a função seja compilada, ela não será executada.

Ao adicionar este trecho de código, o exemplo aparecerá na documentação dentro do `godoc`, tornando seu código ainda mais acessível.

Para ver como isso funciona, execute `godoc -http=:6060` e navegue para `http://localhost:6060/pkg/`

Aqui você vai ver uma lista de todos os pacotes em seu `$GOPATH`. Então, supondo que tenha escrito esse código em algum lugar como `$GOPATH/src/github.com/{seu_id}`, você poderá encontrar uma documentação com seus exemplos.

Se você publicar seu código com exemplos em uma URL pública, poderá compartilhar a documentação em godoc.org. Por exemplo, aqui está a API finalizada deste capítulo https://godoc.org/github.com/larien/learn-go-with-tests/inteiros/v2.

Resumindo

Falamos sobre:

- Mais práticas do fluxo de trabalho de TDD
- Inteiros, adição
- Escrever melhores documentações para que os usuários do nosso código possam entender seu uso rapidamente
- Exemplos de como usar nosso código, que são verificados como parte de nossos testes

Iteração

Você pode encontrar todo o código desse capítulo aqui

Para fazer coisas repetidamente em Go, você precisará do `for`. Go não possui nenhuma palavra chave do tipo `while`, `do` ou `until`. Você pode usar apenas `for`, o que é uma coisa boa!

Vamos escrever um teste para uma função que repete um caractere 5 vezes.

Não há nenhuma novidade até aqui, então tente escrever você mesmo para praticar.

Escreva o teste primeiro

```
package iteracao

import "testing"

func TestRepetir(t *testing.T) {
    repeticoes := Repetir("a")
    esperado := "aaaaa"

    if repeticoes != esperado {
        t.Errorf("esperado '%s' mas obteve '%s'", esperado, repeticoes)
    }
}
```

Execute o teste

```
./repetir_test.go:6:14: undefined: Repetir
```

Escreva a quantidade mínima de código para o teste rodar e verifique o erro na saída

Mantenha a disciplina! Você não precisa saber nada de diferente agora para fazer o teste falhar apropriadamente.

Tudo o que foi feito até agora é o suficiente para compilar, para que você possa verificar se escreveu o teste corretamente.

```
package iteracao

func Repetir(caractere string) string {
    return ""
}
```

Não é legal saber que você já conhece o bastante em Go para escrever testes para problemas simples? Isso significa que agora você pode mexer no código de produção o quanto quiser sabendo que ele se comportará da maneira que você desejar.

```
repetir_test.go:10: esperado 'aaaaa' mas obteve ''
```

Escreva código o suficiente para fazer o teste passar

A sintaxe do `for` é muito fácil de lembrar e segue a maioria das linguagens baseadas em `C`:

```
func Repetir(caractere string) string {
    var repeticoes string
    for i := 0; i < 5; i++ {
        repeticoes = repeticoes + caractere
    }
    return repeticoes
}
```

Ao contrário de outras linguagens como `C`, `Java` ou `Javascript`, não há parênteses ao redor dos três componentes do `for`. No entanto, as chaves `{ }` são obrigatórias.

Execute o teste e ele deverá passar.

Variações adicionais do loop `for` podem ser vistas aqui.

Refatoração

Agora é hora de refatorarmos e apresentarmos outro operador de atribuição: o `+=`.

```
const quantidadeRepeticoes = 5

func Repetir(caractere string) string {
    var repeticoes string
    for i := 0; i < quantidadeRepeticoes; i++ {
        repeticoes += caractere
    }
    return repeticoes
}
```

O operador adicionar & atribuir `+=` adiciona o valor que está à direita no valor que esta à esquerda e atribui o resultado ao valor da esquerda. Também funciona com outros tipos, como por exemplo, inteiros (`integer`).

Benchmarking

Escrever benchmarks em Go é outro recurso disponível nativamente na linguagem e é tão facil quanto escrever testes.

```
func BenchmarkRepetir(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Repetir("a")
    }
}
```

Você notará que o código é muito parecido com um teste.

O `testing.B` dará a você acesso a `b.N`.

Quando o benchmark é rodado, ele executa `b.N` vezes e mede quanto tempo leva.

A quantidade de vezes que o código é executado não deve importar para você. O framework irá determinar qual valor é "bom" para que você consiga ter resultados decentes.

Para executar o benchmark, digite `go test -bench=.` no terminal (ou se estiver executando do PowerShell do Windows, `go test-bench="."`)

```
goos: darwin
goarch: amd64
pkg: github.com/larien/learn-go-with-tests/primeiros-passos-com-go/iteracao/v4
10000000           136 ns/op
PASS
```

`136 ns/op` significa que nossa função demora cerca de 136 nanossegundos para ser executada (no meu computador). E isso é ótimo! Para chegar a esse resultado ela foi executada 10000000 (10 milhões de vezes) vezes.

NOTA por padrão, o benchmark é executado sequencialmente.

Exercícios para praticar

- Altere o teste para que a função possa especificar quantas vezes o caractere deve ser repetido e então corrija o código para passar no teste.
- Escreva `ExampleRepetir` para documentar sua função.
- Veja também o pacote strings. Encontre funções que você considera serem úteis e experimente-as escrevendo testes como fizemos aqui. Investir tempo aprendendo a biblioteca padrão irá te recompensar com o tempo.

Resumindo

- Mais praticás de TDD
- Aprendemos o `for`
- Aprendemos como escrever benchmarks

Arrays e slices

Você pode encontrar todos os códigos para esse capítulo aqui

Arrays te permitem armazenar diversos elementos do mesmo tipo em uma variável em uma ordem específica.

Quando você tem um array, é muito comum ter que percorrer sobre ele. Logo, vamos usar nosso recém adquirido conhecimento de `for` para criar uma função `Soma`. `Soma` vai receber um array de números e retornar o total.

Também vamos praticar nossas habilidades em TDD.

Escreva o teste primeiro

Em `soma_test.go`:

```
package main

import "testing"

func TestSoma(t *testing.T) {

    numeros := [5]int{1, 2, 3, 4, 5}
```

```
    resultado := Soma(numeros)
    esperado := 15

    if esperado != resultado {
        t.Errorf("resultado %d, esperado %d, dado %v", resultado, esperado, numeros)
    }
}
```

Arrays têm uma *capacidade fixa* que é definida quando você declara a variável. Podemos inicializar um array de duas formas:

- [N]tipo{valor1, valor2, ..., valorN}, como `numeros := [5]int{1, 2, 3, 4, 5}`
- [...]tipo{valor1, valor2, ..., valorN}, como `numbers := [...]int{1, 2, 3, 4, 5}`

Às vezes é útil também mostrarmos as entradas da função na mensagem de erro. Para isso estamos usando o formatador `%v`, que é o formato "padrão" e funciona bem com arrays.

Leia mais sobre formatação de strings aqui

Execute o teste

Ao executar `go test`, o compilador vai falhar com `./soma_test.go:10:15: undefined: Soma`

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhado

Em `soma.go`:

```
package main

func Soma(numeros [5]int) int {
    return 0
}
```

Agora seu teste deve falhar com uma *mensagem clara de erro*:

```
soma_test.go:13: resultado 0, esperado 15, dado [1 2 3 4 5]
```

Escreva código o suficiente para fazer o teste passar

```
func Soma(numeros [5]int) int {
    soma := 0
```

```
    for i := 0; i < 5; i++ {
        soma += numeros[i]
    }
    return soma
}
```

Para receber o valor de um array em uma posição específica, basta usar a sintaxe `array[índice]`. Nesse caso, estamos usando o `for` para percorrer cada posição do array (que tem 5 posições) e somar cada valor na variável `soma`.

Refatoração

Vamos apresentar o `range` para nos ajudar a limpar o código:

```
func Soma(numeros [5]int) int {
    soma := 0
    for _, numero := range numeros {
        soma += numero
    }
    return soma
}
```

O `range` permite que você percorra um array. Sempre que é chamado, retorna dois valores: o índice e o valor. Decidimos ignorar o valor índice usando `_` *blank identifier*.

Arrays e seus tipos

Uma propriedade interessante dos arrays é que seu tamanho é relacionado ao seu tipo. Se tentar passar um `[4]int` dentro da função que espera `[5]int`, ela não vai compilar. Elas são de tipos diferentes e é a mesma coisa que tentar passar uma `string` para uma função que espera um `int`.

Você pode estar pensando que é bastante complicado que arrays tenham tamanho fixo, não é? Só que na maioria das vezes, você provavelmente não vai usá-los!

O Go tem *slices*, em que você não define o tamanho da coleção e, graças a isso, pode ter qualquer tamanho.

O próprio requerimento será somar coleções de tamanhos variados.

Escreva o teste primeiro

Agora vamos usar o tipo slice que nos permite ter coleções de qualquer tamanho. A sintaxe é bem parecida com a dos arrays e você só precisa omitir o tamanho quando declará-lo.

`meuSlice := []int{1,2,3}` ao invés de `meuArray := [3]int{1,2,3}`

```
func TestSoma(t *testing.T) {

    t.Run("coleção de 5 números", func(t *testing.T) {
        numeros := [5]int{1, 2, 3, 4, 5}

        resultado := Soma(numeros)
        esperado := 15

        if resultado != esperado {
            t.Errorf("resultado %d, want %d, dado %v", resultado, esperado, numeros)
        }
    })

    t.Run("coleção de qualquer tamanho", func(t *testing.T) {
        numeros := []int{1, 2, 3}

        resultado := Soma(numeros)
        esperado := 6

        if resultado != esperado {
            t.Errorf("resultado %d, esperado %d, dado %v", resultado, esperado, numeros)
        }
    })

}
```

Execute o teste

Isso não vai compilar.

```
./soma_test.go:22:13: cannot use numbers (type []int) as type [5]int
in argument to Soma
```

```
não é possível usar números (tipo []int) como tipo [5]int no
argumento para Soma
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhado

Para resolver o problema, podemos:

- Alterar a API existente mudando o argumento de `Soma` para um slice ao invés de um array.Quando fazemos isso, vamos saber que podemos ter arruinado do dia de alguém, porque nosso *outro* teste não vai compilar!

- Criar uma nova função

No nosso caso, mais ninguém está usando nossa função. Logo, ao invés de ter duas funções para manter, vamos usar apenas uma.

```
func Soma(numeros []int) int {
    soma := 0
    for _, numero := range numeros {
        soma += numero
    }
    return soma
}
```

Se tentar rodar os testes eles ainda não vão compilar. Você vai ter que alterar o primeiro teste e passar um slice ao invés de um array.

Escreva código o suficiente para fazer o teste passar

Nesse caso, para arrumar os problemas de compilação, tudo o que precisamos fazer aqui é fazer os testes passarem!

Refatoração

Nós já refatoramos a função `Soma` e tudo o que fizemos foi mudar os arrays para slices. Logo, não há muito o que fazer aqui. Lembre-se que não devemos abandonar nosso código de teste na etapa de refatoração e precisamos fazer alguma coisa aqui.

```
func TestSoma(t *testing.T) {

    t.Run("coleção de 5 números", func(t *testing.T) {
        numeros := []int{1, 2, 3, 4, 5}

        resultado := Soma(numeros)
        esperado := 15

        if resultado != esperado {
            t.Errorf("resultado %d, esperado %d, dado, %v", resultado, esperado, numeros)
        }
    })

    t.Run("coleção de qualquer tamanho", func(t *testing.T) {
        numeros := []int{1, 2, 3}

        resultado := Soma(numeros)
        esperado := 6
```

```
        if resultado != esperado {
            t.Errorf("resultado %d, esperado %d, dado %v", resultado, esperado, numeros)
        }
    })

}
```

É importante questionar o valor dos seus testes. Ter o máximo de testes possível não deve ser o objetivo e sim ter o máximo de *confiança* possível na sua base de código. Ter testes demais pode se tornar um problema real e só adiciona mais peso na manutenção. **Todo teste tem um custo**.

No nosso caso, dá para perceber que ter dois testes para essa função é redundância. Se funciona para um slice de determindo tamanho, é muito provável que funciona para um slice de qualquer tamanho (dentro desse escopo).

A ferramenta de testes nativa do Go tem a funcionalidade de cobertura de código que te ajuda a identificar áreas do seu código que você não cobriu. Já adianto que ter 100% de cobertura não deve ser seu objetivo; é apenas uma ferramenta para te dar uma ideia da sua cobertura. De qualquer forma, se você aplicar o TDD, é bem provável que chegue bem perto dos 100% de cobertura.

Tente executar `go test -cover` no terminal.

Você deve ver:

```
PASS
coverage: 100.0% of statements
```

Agora apague um dos testes e verifique a cobertura novamente.

Agora que estamos felizes com nossa função bem testada, você deve salvar seu trabalho incrível com um commit antes de partir para o próximo desafio.

Precisamos de uma nova função chamada `SomaTudo`, que vai receber uma quantidade variável de slices e devolver um novo slice contendo as somas de cada slice recebido.

Por exemplo:

`SomaTudo([]int{1,2}, []int{0,9})` deve retornar `[]int{3, 9}`

ou

`SomaTudo([]int{1,1,1})` deve retornar `[]int{3}`

Escreva o teste primeiro

```
func TestSomaTudo(t *testing.T) {

    resultado := SomaTudo([]int{1,2}, []int{0,9})
```

```
    esperado := []int{3, 9}

    if resultado != esperado {
        t.Errorf("resultado %v esperado %v", resultado, esperado)
    }
}
```

Execute o teste

```
./soma_test.go:23:9: undefined: SomaTudo
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhado

Precisamos definir o SomaTudo de acordo com o que nosso teste precisa.

O Go te permite escrever *funções variádicas* em que a quantidade de argumentos podem variar.

```
func SomaTudo(numerosParaSomar ...[]int) (somas []int) {
    return
}
```

Pode tentar compilar, mas nossos testes não vão funcionar!

```
./soma_test.go:26:9: invalid operation: got != want (slice can only
be compared to nil)
```

```
operação inválida: recebido != esperado (slice só pode ser comparado
a nil
```

O Go não te deixa usar operadores de igualdade com slices. *É possível* escrever uma função que percorre cada slice `recebido` e `esperado` e verificar seus valores, mas por praticidade podemos usar o `reflect.DeepEqual` que é útil para verificar se *duas variáveis* são iguais.

```
func TestSomaTudo(t *testing.T) {

    recebido := SomaTudo([]int{1,2}, []int{0,9})
    esperado := []int{3, 9}

    if !reflect.DeepEqual(recebido, esperado) {
        t.Errorf("recebido %v esperado %v", recebido, esperado)
    }
}
```

(coloque `import reflect` no topo do seu arquivo para ter acesso ao `DeepEqual`)

É importante saber que o `reflect.DeepEqual` não tem "segurança de tipos", ou seja, o código vai compilar mesmo se você tiver feito algo estranho. Para ver isso em ação, altere o teste temporariamente para:

```
func TestSomaTudo(t *testing.T) {

    recebido := SomaTudo([]int{1,2}, []int{0,9})
    esperado := "joao"

    if !reflect.DeepEqual(recebido, esperado) {
        t.Errorf("recebido %v, esperado %v", recebido, esperado)
    }
}
```

O que fizemos aqui foi comparar um `slice` com uma `string`. Isso não faz sentido, mas o teste compila! Logo, apesar de ser uma forma simples de comparar slices (e outras coisas), você deve tomar cuidado quando for usar o `reflect.DeepEqual`.

Volte o teste da forma como estava e execute-o. Você deve ter a saída do teste com uma mensagem tipo:

```
soma_test.go:30: recebido [], esperado [3 9]
```

Escreva código o suficiente para fazer o teste passar

O que precisamos fazer é percorrer as variáveis recebidas como argumento, calcular a soma com nossa função `Soma` de antes e adicioná-la ao slice que vamos retornar:

```
func SomaTudo(numerosParaSomar ...[]int) (somas []int) {
    quantidadeDeNumeros := len(numerosParaSomar)
    somas = make([]int, quantidadeDeNumeros)

    for i, numeros := range numerosParaSomar {
        somas[i] = Soma(numeros)
    }

    return
 }
```

Muitas coisas novas para aprender!

Há uma nova forma de criar um slice. O `make` te permite criar um slice com uma capacidade inicial de `len` de `numerosParaSomar` que precisamos percorrer.

Você pode indexar slices como arrays com `meuSlice[N]` para obter seu valor ou designá-lo a um novo valor com `=`.

Agora o teste deve passar.

Refatoração

Como mencionado, slices têm uma capacidade. Se você tiver um slice com uma capacidade de 2 e tentar fazer uma atribuição como `meuSlice[10] = 1`, vai receber um erro em *tempo de execução*.

No entanto, você pode usar a função `append`, que recebe um slice e um novo valor e retorna um novo slice com todos os itens dentro dele.

```
func SomaTudo(numerosParaSomar ...[]int) []int {
    var somas []int
    for _, numeros := range numerosParaSomar {
        somas = append(somas, Soma(numeros))
    }

    return somas
}
```

Nessa implementação, nos preocupamos menos sobre capacidade. Começamos com um slice vazio `somas` e o anexamos ao resultado de `Soma` enquanto percorremos as variáveis recebidas como argumento.

Nosso próprio requisito é alterar o `SomaTudo` para `SomaTodoOResto`, onde agora calcula os totais de todos os "finais" de cada slice. O final de uma coleção é todos os itens com exceção do primeiro (a "cabeça").

Escreva o teste primeiro

```
func TestSomaTodoOResto(t *testing.T) {
    resultado := SomaTodoOResto([]int{1,2}, []int{0,9})
    esperado := []int{2, 9}

    if !reflect.DeepEqual(resultado, esperado) {
        t.Errorf("resultado %v, esperado %v", resultado, esperado)
    }
}
```

Execute o teste

```
./soma_test.go:26:9: undefined: SomaTodoOResto
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhado

Renomeie a função para `SomaTodoOResto` e volte a executar o teste.

```
soma_test.go:30: resultado [3 9], esperado [2 9]
```

Escreva código o suficiente para fazer o teste passar

```
func SomaTodoOResto(numerosParaSomar ...[]int) []int {
    var somas []int
    for _, numeros := range numerosParaSomar {
        final := numeros[1:]
        somas = append(somas, Soma(final))
    }

    return somas
}
```

Slices podem ser "fatiados"! A sintaxe usada é `slice[inicio:final]`. Se você omitir o valor de um dos lados dos `:` ele captura tudo do lado omitido. No nosso caso, quando usamos `numeros[1:]`, estamos dizendo "pegue da posição 1 até o final". É uma boa ideia investir um tempo escrevend outros testes com slices e brincar com o operador slice para criar mais familiaridade com ele.

Refatoração

Não tem muito o que refatorar dessa vez.

O que acha que aconteceria se você passar um slice vazio para a nossa função? Qual é o "final" de um slice vazio? O que acontece quando você fala para o Go capturar todos os elementos de `meuSliceVazio[1:]`?

Escreva o teste primeiro

```
func TestSomaTodoOResto(t *testing.T) {

    t.Run("faz as somas de alguns slices", func(t *testing.T) {
        resultado := SomaTodoOResto([]int{1,2}, []int{0,9})
        esperado := []int{2, 9}

        if !reflect.DeepEqual(resultado, esperado) {
            t.Errorf("resultado %v, esperado %v", resultado, esperado)
        }
    })

    t.Run("soma slices vazios de forma segura", func(t *testing.T) {
        resultado := SomaTodoOResto([]int{}, []int{3, 4, 5})
        esperado := []int{0, 9}
```

```
        if !reflect.DeepEqual(resultado, esperado) {
            t.Errorf("resultado %v, esperado %v", resultado, esperado)
        }
    })

}
```

Execute o teste

```
panic: runtime error: slice bounds out of range [recovered]
    panic: runtime error: slice bounds out of range
```

```
pânico: erro em tempo de execução: fora da capacidade do slice
```

Oh, não! É importante perceber que o test *foi compilado*, esse é um erro em tempo de execução. Erros em tempo de compilação são nossos amigos, porque nos ajudam a escrever softwares que funcionam. Erros em tempo de execução são nosso inimigos, porque afetam nossos usuários.

Escreva código o suficiente para fazer o teste passar

```
func SomaTodoOResto(numerosParaSomar ...[]int) []int {
    var somas []int
    for _, numeros := range numerosParaSomar {
        if len(numeros) == 0 {
            somas = append(somas, 0)
        } else {
            final := numeros[1:]
            somas = append(somas, Soma(final))
        }
    }

    return somas
}
```

Refatoração

Nossos testes têm código repetido em relação à asserção de novo. Vamos encapsular isso em uma função:

```
func TestSomaTodoOResto(t *testing.T) {

    verificaSomas := func(t *testing.T, resultado, esperado []int) {
        t.Helper()
        if !reflect.DeepEqual(resultado, esperado) {
```

```
            t.Errorf("resultado %v, esperado %v", resultado, esperado)
        }
    }

    t.Run("faz a soma do resto", func(t *testing.T) {
        resultado := SomaTodoOResto([]int{1, 2}, []int{0, 9})
        esperado := []int{2, 9}
        verificaSomas(t, resultado, esperado)
    })

    t.Run("soma slices vazios de forma segura", func(t *testing.T) {
        resultado := SomaTodoOResto([]int{}, []int{3, 4, 5})
        esperado := []int{0, 9}
        verificaSomas(t, resultado, esperado)
    })

}
```

Um efeito colateral útil disso é que adiciona um pouco de segurança de tipos no nosso código. Se uma pessoa espertinha adicionar um novo teste com `verificaSomas(t, resultado, "luisa")` o compilador vai pará-lo antes que algo errado aconteça.

```
$ go test
./soma_test.go:52:21: cannot use "luisa" (type string) as type []int in argument to verifica
```

```
não é possível usar "luisa" (tipo string) como tipo []int no
argumento para verificaSomas
```

Resumindo

Falamos sobre:

- Arrays
- Slices
- Várias formas de criá-las
- Como eles têm uma capacidade *fixa*, mas é posível criar novos slices de antigos usando `append`
- Como "fatiar" slices!
- `len` obtém o tamanho de um array ou slice
- Ferramenta de cobertura de testes
- `reflect.DeepEqual` e por que é útil, mas pode diminuir a segurança de tipos do seu código

Usamos slices e arrays com inteiros, mas eles também funcionam com qualquer outro tipo, incluindo até os próprios arrays/slices. Logo, você pode declarar uma variável de `[][]string` se precisar.

Dê uma olhada no post sobre slices no blog de Go para saber mais sobre slices. Tente escrever mais testes para demonstrar o que você aprendeu com a leitura.

Outra forma útil de brincar com Go ao invés de escrever testes é o Go playground. Você pode testar mais coisas lá e você pode compartilhar seu código facilmente se precisar tirar dúvidas. Criei um exemplo com um slice para testar lá.

Estruturas, métodos e interfaces

Você pode encontrar todos os códigos desse capítulo aqui

Suponha que precisamos de algum código de geometria para calcular o perímetro de um retângulo dado uma altura e largura. Podemos escrever uma função `Perimetro(largura float64, altura float64)`, onde `float64` representa números em ponto flutuante como `123.45`.

O ciclo de TDD deve ser mais familiar para você agora.

Escreva o teste primeiro

```
func TestPerimetro(t *testing.T) {
    resultado := Perimetro(10.0, 10.0)
    esperado := 40.0

    if resultado != esperado {
        t.Errorf("resultado %.2f esperado %.2f", resultado, esperado)
    }
}
```

Viu a nova string de formatação? O `f` é para nosso `float64` e o `.2` significa imprimir duas casas decimais.

Execute o teste

```
./formas_test.go:6:9: undefined: Perimetro
```

```
indefinido: Perimetro
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhando

```
func Perimetro(largura float64, altura float64) float64 {
    return 0
}
```

Resulta em `formas_test.go:10: resultado 0, esperado 40`.

Escreva código o suficiente para fazer o teste passar

```
func Perimetro(largura float64, altura float64) float64 {
    return 2 * (largura + altura)
}
```

Por enquanto, tudo fácil. Agora vamos criar uma função chamada `Area(largura, altura float64)` que retorna a área de um retângulo.

Tente fazer isso sozinho, segundo o ciclo de TDD.

Você deve terminar com os testes como estes:

```
func TestPerimetro(t *testing.T) {
    resultado := Perimetro(10.0, 10.0)
    esperado := 40.0

    if resultado != esperado {
        t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
    }
}

func TestArea(t *testing.T) {
    resultado := Area(12.0, 6.0)
    esperado := 72.0

    if resultado != esperado {
        t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
    }
}
```

E código como este:

```
func Perimetro(largura float64, altura float64) float64 {
    return 2 * (largura + altura)
}

func Area(largura float64, altura float64) float64 {
    return largura * altura
}
```

Refatoração

Nosso código faz o trabalho, mas não contém nada explícito sobre retângulos. Uma pessoa descuidada poderia tentar passar a largura e altura de um triângulo para esta função sem perceber que ela retornará uma resposta errada.

Podemos apenas dar para a função um nome mais específico como `AreaDoRetangulo`. Uma solução mais limpa é definir nosso próprio *tipo*

chamado `Retangulo` que encapsula este conceito para nós.

Podemos criar um tipo simples usando uma **struct** (estrutura). Uma struct é apenas uma coleção nomeada de campos onde você pode armazenar dados.

Declare uma `struct` assim:

```
type Retangulo struct {
    Largura float64
    Altura  float64
}
```

Agora vamos refatorar os testes para usar `Retangulo` em vez de um simples `float64`.

```
func TestPerimetro(t *testing.T) {
    retangulo := Retangulo{10.0, 10.0}
    resultado := Perimetro(retangulo)
    esperado := 40.0

    if resultado != esperado {
        t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
    }
}

func TestArea(t *testing.T) {
    retangulo := Retangulo{12.0, 6.0}
    resultado := Area(retangulo)
    esperado := 72.0

    if resultado != esperado {
        t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
    }
}
```

Lembre de rodar seus testes antes de tentar corrigir. Você deve ter erro útil como:

```
./formas_test.go:7:18: not enough arguments in call to Perimetro
    have (Retangulo)
    esperado (float64, float64)
```

Você pode acessar os campos de uma `struct` com a sintaxe `minhaStruct.campo`.

Mude as duas funções para corrigir o teste.

```
func Perimetro(retangulo Retangulo) float64 {
    return 2 * (retangulo.Largura + retangulo.Altura)
}

func Area(retangulo Retangulo) float64 {
```

```
    return retangulo.Largura * retangulo.Altura
}
```

Espero que você concorde que passar um `Retangulo` para a função mostra nossa intenção com mais clareza, mas existem mais benefícios em usar `structs` que já vamos entender.

Nosso próximo requisito é escrever uma função `Area` para círculos.

Escreva o teste primeiro

```
func TestArea(t *testing.T) {
    t.Run("retângulos", func(t *testing.T) {
        retangulo := Retangulo{12.0, 6.0}
        resultado := Area(retangulo)
        esperado := 72.0

        if resultado != esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
        }
    })

    t.Run("círculos", func(t *testing.T) {
        circulo := Circulo{10}
        resultado := Area(circulo)
        esperado := 314.1592653589793

        if resultado != esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
        }
    })
}
```

Execute o teste

```
./formas_test.go:28:13: undefined: Circulo
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhando

Precisamos definir nosso tipo `Circulo`.

```
type Circulo struct {
    Raio float64
}
```

Agora rode os testes novamente.

```
./formas_test.go:29:14: cannot use circulo (type Circulo) as type
Retangulo in argument to Area
```

Algumas linguagens de programação permitem você fazer algo como:

```
func Area(circulo Circulo) float64 { ... }
func Area(retangulo Retangulo) float64 { ... }
```

Mas em Go você não pode:

```
./formas.go:20:32: Area redeclared in this block
```

Temos duas escolhas:

- Podemos ter funções com o mesmo nome declaradas em *pacotes* diferentes. Então, poderíamos criar nossa `Area(Circulo)` em um novo *pacote*, só que isso parece um exagero aqui.
- Em vez disso, podemos definir *métodos* em nosso mais novo tipo definido.

O que são métodos?

Até agora só escrevemos *funções*, mas temos usado alguns métodos. Quando chamamos `t.Errorf`, nós chamamos o método `Errorf` na instância de nosso `t` (`testing.T`).

Um método é uma função com um receptor. Uma declaração de método vincula um identificador e o nome do método a um método e associa o método com o tipo base do receptor.

Métodos são muito parecidos com funções, mas são chamados invocando-os em uma instância de um tipo específico.

Enquanto você chama funções onde quiser, como por exemplo em `Area(retangulo)`, você só pode chamar métodos em "coisas" específicas.

Um exemplo ajudará. Então, vamos mudar nossos testes primeiro para chamar métodos em vez de funções, e, em seguida, corrigir o código.

```
func TestArea(t *testing.T) {
    t.Run("retângulos", func(t *testing.T) {
        retangulo := Retangulo{12.0, 6.0}
        resultado := retangulo.Area()
        esperado := 72.0

        if resultado != esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
        }
    })

    t.Run("círculos", func(t *testing.T) {
```

```
        circulo := Circulo{10}
        resultado := circulo.Area()
        esperado := 314.1592653589793

        if resultado != esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
        }
    })
}
```

Se rodarmos os testes agora, recebemos:

```
./formas_test.go:19:19: retangulo.Area undefined (type Retangulo has no field or method Area)
./formas_test.go:29:16: circulo.Area undefined (type Circulo has no field or method Area)
```

> type Circulo has no field or method Area

Gostaria de reforçar o quão grandioso o compilador é. É muito importante ter tempo para ler lentamente as mensagens de erro que você recebe, pois isso te ajudará a longo prazo.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhando

Vamos adicionar alguns métodos para nossos tipos:

```
type Retangulo struct {
    Largura  float64
    Altura float64
}

func (r Retangulo) Area() float64  {
    return 0
}

type Circulo struct {
    Raio float64
}

func (c Circulo) Area() float64  {
    return 0
}
```

A sintaxe para declaração de métodos é quase a mesma que usamos para funções e isso acontece porque eles são muito parecidos. A única diferença é a sintaxe para o método receptor: `func (nomeDoReceptor TipoDoReceptor) NomeDoMetodo(argumentos)`.

Quando seu método é chamado em uma variável desse tipo, você tem sua referência para o dado através da variável `nomeDoReceptor`. Em muitas outras linguagens de programação isto é feito implicitamente e você acessa o receptor através de `this`.

É uma convenção em Go que a variável receptora seja a primeira letra do tipo em minúsculo.

```
r Retangulo
```

Se você executar novamente os testes, eles devem compilar e dar alguma saída do teste falhando.

Escreva código suficiente para fazer o teste passar

Agora vamos fazer nossos testes de retângulo passarem corrigindo nosso novo método.

```
func (r Retangulo) Area() float64  {
    return r.Largura * r.Altura
}
```

Se você executar novamente os testes, aqueles de retângulo devem passar, mas os de círculo ainda falham.

Para fazer a função `Area` de círculo passar, vamos emprestar a constante `Pi` do pacote `math` (lembre-se de importá-lo).

```
func (c Circulo) Area() float64  {
    return math.Pi * c.Raio * c.Raio
}
```

Refatoração

Existe duplicação em nossos testes.

Tudo o que queremos fazer é pegar uma coleção de *formas*, chamar o método `Area()` e então verificar o resultado.

Queremos ser capazes de escrever um tipo de função `verificaArea` que permita passar tanto `Retangulo` quanto `Circulo`, mas falhe ao compilar se tentarmos passar algo que não seja uma *forma*.

Com Go, podemos trabalhar dessa forma com **interfaces**.

Interfaces são um conceito muito poderoso em linguagens de programação estaticamente tipadas, como Go, porque permitem que você crie funções que podem ser usadas com diferentes tipos e permite a criação de código altamente desacoplado, mantendo ainda a segurança de tipos.

Vamos apresentar isso refatorando nossos testes.

```
func TestArea(t *testing.T) {
    verificaArea := func(t *testing.T, forma Forma, esperado float64) {
        t.Helper()
        resultado := forma.Area()

        if resultado != esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, esperado)
        }
    }

    t.Run("retângulos", func(t *testing.T) {
        retangulo := Retangulo{12.0, 6.0}
        verificaArea(t, retangulo, 72.0)
    })

    t.Run("círculos", func(t *testing.T) {
        circulo := Circulo{10}
        verificaArea(t, circulo, 314.1592653589793)
    })
}
```

Estamos criando uma função auxiliar como fizemos em outros exercícios, mas desta vez estamos pedindo que uma `Forma` seja passada. Se tentarmos chamá-la com algo que não seja uma *forma*, não vai compilar.

Como algo se torna uma *forma*? Precisamos apenas falar para o Go o que é uma `Forma` usando uma declaração de interface.

```
type Forma interface {
    Area() float64
}
```

Estamos criando um novo `tipo`, assim como fizemos com `Retangulo` e `Circulo`, mas desta vez é uma `interface` em vez de uma `struct`.

Uma vez adicionado isso ao código, os testes passarão.

Peraí, como assim?

A interface em Go bem diferente das interfaces na maioria das outras linguagens de programação. Normalmente você tem que escrever um código para dizer que `meu tipo Foo implementa a interface Bar`.

Só que no nosso caso:

- `Retangulo` tem um método chamado `Area` que retorna um `float64`, então satisfaz a interface `Forma`.
- `Circulo` tem um método chamado `Area` que retorna um `float64`, então satisfaz a interface `Forma`.

- `string` não tem esse método, então não satisfaz a interface.
- etc.

Em Go a **resolução de interface é implícita**. Se o tipo que você passar combinar com o que a interface está esperando, o código será compilado.

Desacoplando

Veja como nossa função auxiliar não precisa se preocupar se a *forma* é um `Retangulo` ou um `Circulo` ou um `Triangulo`. Ao declarar uma interface, a função auxiliar está *desacoplada* de tipos concretos e tem apenas o método que precisa para fazer o trabalho.

Este tipo de abordagem - de usar interfaces para declarar **somente o que você precisa** - é muito importante no desenvolvimento de software e será coberto mais detalhadamente nas próximas seções.

Refatoração adicional

Agora que você conhece as `structs`, podemos apresentar os "table driven tests" (testes orientados por tabela).

Table driven tests são úteis quando você quer construir uma lista de casos de testes que podem ser testados da mesma forma.

```
func TestArea(t *testing.T) {
    testesArea := []struct {
        forma    Forma
        esperado float64
    }{
        {Retangulo{12, 6}, 72.0},
        {Circulo{10}, 314.1592653589793},
    }

    for _, tt := range testesArea {
        resultado := tt.forma.Area()
        if resultado != tt.esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, tt.esperado)
        }
    }
}
```

A única sintaxe nova aqui é a criação de uma "struct anônima", `testesArea`. Estamos declarando um slice de structs usando `[]struct` com dois campos, o `forma` e o `esperado`. Então preenchemos o slice com os casos.

Depois iteramos sobre eles assim como fazemos com qualquer outro slice, usando os campos da struct para executar nossos testes.

Dá para perceber como será muito fácil para uma pessoa inserir uma nova forma, implementar `Area` e então adicioná-la nos casos de teste. Além disso, se for encontrada uma falha em `Area`, é muito fácil adicionar um novo caso de teste para verificar antes de corrigi-la.

Testes baseados em tabela podem ser um item valioso em sua caixa de ferramentas, mas tenha certeza de que você precisa da sintaxe extra nos testes. Se você deseja testar várias implementações de uma interface ou se o dado passado para uma função tem muitos requisitos diferentes que precisam de testes, eles podem servir bem.

Vamos demonstrar tudo isso adicionando e testando outra forma; um triângulo.

Escreva o teste primeiro

Adicionar um teste para nossa nova forma é muito fácil. Simplesmente adicione `{Triangulo{12, 6}, 36.0},` à nossa lista.

```
func TestArea(t *testing.T) {
    testesArea := []struct {
        forma    Forma
        esperado float64
    }{
        {Retangulo{12, 6}, 72.0},
        {Circulo{10}, 314.1592653589793},
        {Triangulo{12, 6}, 36.0},
    }

    for _, tt := range testesArea {
        resultado := tt.forma.Area()
        if resultado != tt.esperado {
            t.Errorf("resultado %.2f, esperado %.2f", resultado, tt.esperado)
        }
    }
}
```

Execute o teste

Lembre-se, continue tentando executar o teste e deixe o compilador guiá-lo em direção a solução.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste falhando

```
./formas_test.go:25:4: undefined: Triangulo
```

Ainda não definimos `Triangulo`:

```
type Triangulo struct {
    Base   float64
    Altura float64
}
```

Tente novamente:

```
./formas_test.go:25:8: cannot use Triangulo literal (type Triangulo) as type Forma in field valu
    Triangulo does not implement Forma (missing Area method)
```

`Triangulo não implementa Forma (método Area faltando)`

Isso nos diz que não podemos usar um `Triangulo` como uma `Forma` porque ele não tem um método `Area()`, então adicione uma implementação vazia para fazermos o teste funcionar:

```
func (t Triangulo) Area() float64 {
    return 0
}
```

Finalmente o código compilou e temos o nosso erro:

```
formas_test.go:31: resultado 0.00, esperado 36.00
```

Escreva código suficiente para fazer o teste passar

```
func (t Triangulo) Area() float64 {
    return (t.Base * t.Altura) * 0.5
}
```

E nossos testes passaram!

Refatoração

Novamente, a implementação está boa, mas nossos testes podem ser melhorados.

Quando você lê isso:

```
{Retangulo{12, 6}, 72.0},
{Circulo{10}, 314.1592653589793},
{Triangulo{12, 6}, 36.0},
```

Não está tão claro o que todos os números representam e você deve ter o objetivo de escrever testes que sejam fáceis de entender.

Até agora você viu uma sintaxe para criar instâncias de structs como `MinhaStruct{valor1, valor2}`, mas você pode opcionalmente nomear esses campos.

Vamos ver como isso funciona:

```
        {forma: Retangulo{largura: 12, altura: 6}, esperado: 72.0},
        {forma: Circulo{Raio: 10}, esperado: 314.1592653589793},
        {forma: Triangulo{Base: 12, altura: 6}, esperado: 36.0},
```

Em Test-Driven Development by Example Kent Beck refatora alguns testes para um ponto e afirma:

> O teste é lido de forma mais clara, como se fosse uma afirmação da verdade, **não uma sequência de operações**

(ênfase minha)

Agora nossos testes (pelo menos a lista de casos) fazem afirmações da verdade sobre formas e suas áreas.

Garanta que a saída do seu teste seja útil

Lembra anteriormente quando implementamos `Triangulo` e tivemos um teste falhando? Ele imprimiu `formas_test.go:31: resultado 0.00 esperado, 36.00`.

Nós sabíamos que estava relacionado ao `Triangulo` porque estávamos trabalhando nisso, mas e se uma falha escorregasse para o sistema em um dos 20 casos na tabela? Como alguém saberia qual caso falhou? Não parece ser uma boa experiência. Ela teria que olhar caso a caso para encontrar qual deles está falhando de fato.

Podemos mudar nossa mensagem de erro para `%#v resultado %.2f, esperado %.2f`. A string de formatação `%#v` irá imprimir nossa struct com os valores em seu campo para que as pessoas possam ver imediatamente as propriedades que estão sendo testadas.

Para melhorar a legibilidade de nossos futuros casos de teste, podemos renomear o campo `esperado` para algo mais descritivo como `temArea`.

Uma dica final com testes guiados por tabela é usar `t.Run` e renomear os casos de teste.

Envolvendo cada caso em um `t.Run` você terá uma saída de testes mais limpa em caso de falhas, além de imprimir o nome do caso.

```
--- FAIL: TestArea (0.00s)
    --- FAIL: TestArea/Retangulo (0.00s)
     formas_test.go:33: main.Retangulo{Largura:12, Altura:6} resultado 72.00, esperado 72.10
```

E você pode rodar testes específicos dentro de sua tabela com `go test -run TestArea/Retangulo`.

Aqui está o código final do nosso teste que captura isso:

```
func TestArea(t *testing.T) {
    testesArea := []struct {
```

```
        nome    string
        forma   Forma
        temArea float64
    }{
        {nome: "Retângulo", forma: Retangulo{Largura: 12, Altura: 6}, temArea: 72.0},
        {nome: "Círculo", forma: Circulo{Raio: 10}, temArea: 314.1592653589793},
        {nome: "Triângulo", forma: Triangulo{Base: 12, Altura: 6}, temArea: 36.0},
    }

    for _, tt := range testesArea {
        t.Run(tt.nome, func(t *testing.T) {
            resultado := tt.forma.Area()
            if resultado != tt.temArea {
                t.Errorf("%#v resultado %.2f, esperado %.2f", tt.forma, resultado, tt.temAre
            }
        })
    }
}
```

Resumo

Esta foi mais uma prática de TDD, iterando em nossas soluções para problemas matemáticos básicos e aprendendo novos recursos da linguagem motivados por nossos testes.

- Declarar structs para criar seus próprios tipos de dados permite agrupar dados relacionados e torna a intenção do seu código mais clara.
- Declarar interfaces permite que você possa definir funções que podem ser usadas por diferentes tipos (polimorfismo paramétrico).
- Adicionar métodos permite que você possa adicionar funcionalidades aos seus tipos de dados e implementar interfaces.
- Testes baseados em tabela permite que você torne suas asserções mais claras e seus testes mais fáceis de estender e manter.

Este foi um capítulo importante porque agora começamos a definir nossos próprios tipos. Em linguagens estaticamente tipadas como Go, conseguir projetar seus próprios tipos é essencial para construir software que seja fácil de entender, compilar e testar.

Interfaces são uma ótima ferramenta para ocultar a complexidade de outras partes do sistema. Em nosso caso, o *código* de teste auxiliar não precisou conhecer a forma exata que estava afirmando, apenas como "pedir" pela sua área.

Conforme você se familiariza com Go, começa a ver a força real das interfaces e da biblioteca padrão.

Você aprenderá sobre as interfaces definidas na biblioteca padrão que são usadas *em todo lugar* e, implementando-as em relação aos seus próprios tipos, você pode

reutilizar rapidamente muitas das ótimas funcionalidades.

Ponteiros e erros

Você pode encontrar todos os códigos deste capítulo aqui

Aprendemos sobre estruturas na última seção, o que nos possibilita capturar valores com conceito relacionado.

Em algum momento talvez você deseje utilizar estruturas para gerenciar valores, expondo métodos que permita aos usuários mudá-los de um jeito que você possa controlar.

Fintechs amam Go e uhh bitcoins? Então vamos mostrar um sistema bancário incrível que podemos construir.

Vamos construir uma estrutura de `Carteira` que possamos depositar `Bitcoin`.

Escreva o teste primeiro

```
func TestCarteira(t *testing.T) {
    carteira := Carteira{}

    carteira.Depositar(10)

    resultado := carteira.Saldo()
    esperado := 10

    if resultado != esperado {
        t.Errorf("resultado %s, esperado %s", resultado, esperado)
    }
}
```

No exemplo anterior acessamos campos diretamente pelo nome. Entretanto, na nossa *carteira super protegida*, não queremos expor o valor interno para o resto do mundo. Queremos controlar o acesso por meio de métodos.

Execute o teste

```
./carteira_test.go:7:12: undefined: Carteira
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

O compilador não sabe o que uma `Carteira` é, então vamos declará-la.

```
type Carteira struct { }
```

Agora que declaramos nossa carteira, tente rodar o teste novamente:

```
./carteira_test.go:9:8: carteira.Depositar undefined (type Carteira has no field or method D
./carteira_test.go:11:15: carteira.Saldo undefined (type Carteira has no field or method Sal
```

Precisamos definir estes métodos.

Lembre-se de apenas fazer o necessário para fazer os testes rodarem. Precisamos ter certeza que nossos testes falhem corretamente com uma mensagem de erro clara.

```
func (c Carteira) Depositar(quantidade int) {

}

func (c Carteira) Saldo() int {
    return 0
}
```

Se essa sintaxe não for familiar, dê uma lida na seção de estruturas.

Os testes agora devem compilar e rodar:

```
carteira_test.go:15: resultado 0, esperado 10
```

Escreva código o suficiente para fazer o teste passar

Precisaremos de algum tipo de variável de *saldo* em nossa estrutura para guardar o valor:

```
type Carteira struct {
    saldo int
}
```

Em Go, se uma variável, tipo, função e etc, começam com uma letra minúsculo, então esta será privada para *outros pacotes que não seja o que a definiu*.

No nosso caso, queremos que apenas nossos métodos sejam capazes de manipular os valores.

Lembre-se que podemos acessar o valor interno do campo `saldo` usando a variável "receptora".

```
func (c Carteira) Depositar(quantidade int) {
    c.saldo += quantidade
}

func (c Carteira) Saldo() int {
    return c.saldo
}
```

Com a nossa carreira em Fintechs segura, rode os testes para nos aquecermos para passarmos no teste.

```
carteira_test.go:15: resultado 0, esperado 10
```

????

Ok, isso é confuso. Parece que nosso código deveria funcionar, pois adicionamos nosso novo valor ao saldo e o método Saldo deveria retornar o valor atual.

Em Go, **quando uma função ou um método é invocado, os argumentos são *copiados***.

Quando `func (c Carteira) Depositar(quantidade int)` é chamado, o `c` é uma cópia do valor de qualquer lugar que o método tenha sido chamado.

Sem focar em Ciência da Computação, quando criamos um valor (como uma carteira), esse valor é alocado em algum lugar da memória. Você pode descobrir o *endereço* desse bit de memória usando `&meuValor`.

Experimente isso adicionando alguns prints no código:

```
func TestCarteira(t *testing.T) {
    carteira := Carteira{}

    carteira.Depositar(10)

    resultado := carteira.Saldo()

    fmt.Printf("O endereço do saldo no teste é %v \n", &carteira.saldo)

    esperado := 10

    if resultado != esperado {
        t.Errorf("resultado %d, esperado %d", resultado, esperado)
    }
}

func (c Carteira) Depositar(quantidade int) {
    fmt.Printf("O endereço do saldo no Depositar é %v \n", &c.saldo)
    c.saldo += quantidade
}
```

O `\n` é um caractere de escape queeadiciona uma nova linha após imprimir o endereço de memória. Conseguimos acessar o ponteiro para algo com o símbolo de endereço `&`.

Agora rode o teste novamente:

```
O endereço do saldo no Depositar é 0xc420012268
O endereço do saldo no teste é is 0xc420012260
```

Podemos ver que os endereços dos dois saldos são diferentes. Então, quando mudamos o valor de um dos saldos dentro do código, estamos trabalhando em uma cópia do que veio do teste. Portanto, o saldo no teste não é alterado.

Podemos consertar isso com *ponteiros*. Ponteiros nos permitem *apontar* para alguns valores e então mudá-los. Então, em vez de termos uma cópia da Carteira, usamos um ponteiro para a carteira para que possamos alterá-la.

```
func (c *Carteira) Depositar(quantidade int) {
    c.saldo += quantidade
}

func (c *Carteira) Saldo() int {
    return c.saldo
}
```

A diferença é que o tipo do argumento é `*Carteira` em vez de `Carteira` que você pode ler como "um ponteiro para uma carteira".

Rode novamente os testes e eles devem passar.

Refatoração

Dissemos que estávamos fazendo uma carteira Bitcoin, mas até agora não os mencionamos. Estamos usando `int` porque é um bom tipo para contar coisas!

Parece um pouco exagerado criar uma `struct` para isso. `int` é o suficiente nesse contexto, mas não é descritivo o suficiente.

Go permite criarmos novos tipos a partir de tipos existentes.

A sintaxe é `type MeuNome TipoOriginal`

```
type Bitcoin int

type Carteira struct {
    saldo Bitcoin
}

func (c *Carteira) Depositar(quantidade Bitcoin) {
    c.saldo += quantidade
}

func (c *Carteira) Saldo() Bitcoin {
    return c.saldo
}
```

```
func TestCarteira(t *testing.T) {

    carteira := Carteira{}
```

```
    carteira.Depositar(Bitcoin(10))

    resultado := carteira.Saldo()

    esperado := Bitcoin(10)

    if resultado != esperado {
            t.Errorf("resultado %d, esperado %d", resultado, esperado)
        }
}
```

Para criarmos `Bitcoin`, basta usar a sintaxe `Bitcoin(999)`.

Ao fazermos isso, estamos criando um novo tipo e podemos declarar *métodos* nele. Isto pode ser muito útil quando queremos adicionar funcionalidades de domínios específicos a tipos já existentes.

Vamos implementar um Stringer para o Bitcoin:

```
type Stringer interface {
        String() string
}
```

Essa interface é definida no pacote `fmt` e permite definir como seu tipo é impresso quando utilizado com o operador de string `%s` em prints.

```
func (b Bitcoin) String() string {
    return fmt.Sprintf("%d BTC", b)
}
```

Como podemos ver, a sintaxe para criar um método em um tipo definido por nós é a mesma que a utilizada em uma struct.

Agora precisamos atualizar nossas impressões de strings no teste para que usem `String()`.

```
    if resultado != esperado {
        t.Errorf("resultado %s, esperado %s", resultado, esperado)
    }
```

Para ver funcionando, quebre o teste de propósito para que possamos ver:

```
carteira_test.go:18: resultado 10 BTC, esperado 20 BTC
```

Isto deixa mais claro o que está acontecendo em nossos testes.

O próximo requisito é criar uma função de `Retirar`.

Escreva o teste primeiro

É basicamente o aposto da função `Depositar()`:

```
func TestCarteira(t *testing.T) {
    t.Run("Depositar", func(t *testing.T) {
        carteira := Carteira{}

        carteira.Depositar(Bitcoin(10))

        resultado := carteira.Saldo()

        esperado := Bitcoin(10)

        if resultado != esperado {
            t.Errorf("resultado %s, esperado %s", resultado, esperado)
        }
    })

    t.Run("Retirar", func(t *testing.T) {
        carteira := Carteira{saldo: Bitcoin(20)}

        carteira.Retirar(Bitcoin(10))

        resultado := carteira.Saldo()

        esperado := Bitcoin(10)

        if resultado != esperado {
            t.Errorf("resultado %s, esperado %s", resultado, esperado)
        }
    })
}
```

Execute o teste

```
./carteira_test.go:26:9: carteira.Retirar undefined (type Carteira
has no field or method Retirar)
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
func (c *Carteira) Retirar(quantidade Bitcoin) {
}
```

```
carteira_test.go:33: resultado 20 BTC, esperado 10 BTC
```

Escreva código o suficiente para fazer o teste passar

```
func (c *Carteira) Retirar(quantidade Bitcoin) {
    c.saldo -= quantidade
}
```

Refatoração

Há algumas duplicações em nossos testes, vamos refatorar isso.

```
func TestCarteira(t *testing.T) {
    confirmaSaldo := func(t *testing.T, carteira Carteira, valorEsperado Bitcoin) {
        t.Helper()
        resultado := carteira.Saldo()

        if resultado != esperado {
            t.Errorf("resultado %s, esperado %s", resultado, esperado)
        }
    }

    t.Run("Depositar", func(t *testing.T) {
        carteira := Carteira{}
        carteira.Depositar(Bitcoin(10))
        confirmaSaldo(t, carteira, Bitcoin(10))
    })

    t.Run("Retirar", func(t *testing.T) {
        carteira := Carteira{saldo: Bitcoin(20)}
        carteira.Retirar(10)
        confirmaSaldo(t, carteira, Bitcoin(10))
    })
}
```

O que aconteceria se você tentasse `Retirar` mais do que há de saldo na conta? Por enquanto, nossos requisitos são assumir que não há nenhum tipo de cheque-especial.

Como sinalizamos um problema quando estivermos usando `Retirar` ?

Em Go, se você quiser indicar um erro, sua função deve retornar um `err` para que quem a chamou possar verificá-lo e tratá-lo.

Vamos tentar fazer isso em um teste.

Escreva o teste primeiro

```
t.Run("Retirar com saldo insuficiente", func(t *testing.T) {
    saldoInicial := Bitcoin(20)
    carteira := Carteira{saldoInicial}
    erro := carteira.Retirar(Bitcoin(100))

    confirmaSaldo(t, carteira, saldoInicial)

    if erro == nil {
        t.Error("Esperava um erro mas nenhum ocorreu")
    }
})
```

Queremos que `Retirar` retorne um erro se tentarmos retirar mais do que temos e o saldo deverá continuar o mesmo.

Verificamos se um erro foi retornado falhando o teste se o valor for `nil`.

`nil` é a mesma coisa que `null` de outras linguagens de programação.

Erros podem ser `nil`, porque o tipo do retorno de `Retirar` vai ser `error`, que é uma interface. Se você vir uma função que tem argumentos ou retornos que são interfaces, eles podem ser nulos.

Do mesmo jeito que `null`, se tentarmos acessar um valor que é `nil`, isso irá disparar um **pânico em tempo de execução**. Isso é ruim! Devemos ter certeza que tratamos os valores nulos.

Execute o teste

```
./carteira_test.go:31:25: carteira.Retirar(Bitcoin(100)) used as value
```

Talvez não esteja tão claro, mas nossa intenção era apenas invocar a função `Retirar` e ela nunca irá retornar um valor pois o saldo será diretamente subtraído com o ponteiro e a função deve apenas retornar o erro (se houver). Para fazer compilar, precisaremos mudar a função para que retorne um tipo.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
func (c *Carteira) Retirar(quantidade Bitcoin) error {
    c.saldo -= quantidade
    return nil
}
```

Novamente, é muito importante escrever apenas o suficiente para compilar. Corrigimos o método `Retirar` para retornar `error` e por enquanto temos que retornar *alguma coisa*, então vamos apenas retornar `nil` .

Escreva código o suficiente para fazer o teste passar

```
func (c *Carteira) Retirar(quantidade Bitcoin) error {
    if quantidade > c.saldo {
        return errors.New("eita")
    }

    c.saldo -= quantidade
    return nil
}
```

Lembre-se de importar `errors`.

`errors.New` cria um novo `error` com a mensagem escolhida.

Refatoração

Vamos fazer um método auxiliar de teste para nossa verificação de erro para deixar nosso teste mais legível.

```
confirmaErro := func(t *testing.T, erro error) {
    t.Helper()
    if erro == nil {
        t.Error("esperava um erro, mas nenhum ocorreu.")
    }
}
```

E em nosso teste:

```
t.Run("Retirar com saldo insuficiente", func(t *testing.T) {
    saldoInicial := Bitcoin(20)
    carteira := Carteira{saldoInicial}
    erro := carteira.Retirar(Bitcoin(100))

    confirmaSaldo(t, carteira, saldoInicial)
    confirmaErro(t, erro)
})
```

Espero que, ao retornamos um erro do tipo "eita", você pense que *devêssemos* deixar mais claro o que ocorreu, já que esta não parece uma informação útil para nós.

Assumindo que o erro enfim foi retornado para o usuário, vamos atualizar nosso teste para verificar o tipo espcífico de mensagem de erro ao invés de apenas

verificar se um erro existe.

Escreva o teste primeiro

Atualize nosso helper para comparar com uma `string`:

```
confirmarErro := func(t *testing.T, valor error, valorEsperado string) {
    t.Helper()
    if resultado == nil {
        t.Fatal("esperava um erro, mas nenhum ocorreu")
    }

    if resultado != esperado {
        t.Errorf("resultado %s, esperado %s", resultado, esperado)
    }
}
```

E então atualize o invocador:

```
t.Run("Retirar saldo insuficiente", func(t *testing.T) {
    saldoInicial := Bitcoin(20)
    carteira := Carteira{saldoInicial}
    erro := carteira.Retirar(Bitcoin(100))

    confirmaSaldo(t, carteira, saldoInicial)
    confirmaErro(t, erro, "não é possível retirar: saldo insuficiente")
})
```

Usamos o `t.Fatal` que interromperá o teste se for chamado. Isso é feito porque não queremos fazer mais asserções no erro retornado, se não houver um. Sem isso, o teste continuaria e causaria erros por causa do ponteiro `nil`.

Execute o teste

```
carteira_test.go:61: erro resultado 'eita', erro esperado 'não é
possível retirar: saldo insuficiente'
```

Escreva código o suficiente para fazer o teste passar

```
func (c *Carteira) Retirar(quantidade Bitcoin) error {

    if quantidade > c.saldo {
        return errors.New("não é possível retirar: saldo insuficiente")
    }

    c.saldo -= quantidade
```

```
    return nil
}
```

Refatoração

Temos duplicação da mensagem de erro tanto no código de teste quanto no código de `Retirar`.

Seria chato se o teste falhasse por alguém ter mudado a mensagem do erro e é muito detalhe para o nosso teste. Nós não *necessariamente* nos importamos qual mensagem é exatamente, apenas que algum tipo de erro significativo sobre a função é retornado dada uma certa condição.

Em Go, erros são valores, então podemos refatorar isso para ser uma variável e termos apenas uma fonte da verdade.

```
var ErroSaldoInsuficiente = errors.New("não é possível retirar: saldo insuficiente")

func (c *Carteira) Retirar(amount Bitcoin) error {

    if quantidade > c.saldo {
        return ErroSaldoInsuficiente
    }

    c.saldo -= quantidade
    return nil
}
```

A palavra-chave `var` no escopo do arquivo nos permite definir valores globais para o pacote.

Esta é uma mudança positiva, pois agora nossa função `Retirar` parece mais limpa.

Agora, podemos refatorar nosso código para usar este valor ao invés de uma string específica.

```
func TestCarteira(t *testing.T) {
    t.Run("Depositar", func(t *testing.T) {
        carteira := Carteira{}
        carteira.Depositar(Bitcoin(10))

        confirmaSaldo(t, carteira, Bitcoin(10))
    })

    t.Run("Retirar com saldo suficiente", func(t *testing.T) {
        carteira := Carteira{Bitcoin(20)}
        erro := carteira.Retirar(Bitcoin(10))
```

```
        confirmaSaldo(t, carteira, Bitcoin(10))
        confirmaErroInexistente(t, erro)
    })

    t.Run("Retirar com saldo insuficiente", func(t *testing.T) {
        saldoInicial := Bitcoin(20)
        carteira := Carteira{saldoInicial}
        erro := carteira.Retirar(Bitcoin(100))

        confirmaSaldo(t, carteira, saldoInicial)
        confirmaErro(t, erro, ErroSaldoInsuficiente)
    })
}

func confirmaSaldo(t *testing.T, carteira Carteira, esperado Bitcoin) {
    t.Helper()
    resultado := carteira.Saldo()

    if resultado != esperado {
        t.Errorf("resultado %s, esperado %s", resultado, esperado)
    }
}

func confirmaErro(t *testing.T, resultado error, esperado error) {
    t.Helper()
    if resultado == nil {
        t.Fatal("esperava um erro, mas nenhum ocorreu")
    }

    if resultado != esperado {
        t.Errorf("erro resultado %s, erro esperado %s", resultado, esperado)
    }
}
```

Agora está mais fácil dar continuidade ao nosso teste.

Nós apenas movemos os métodos auxiliares para fora da função principal de teste. Logo, quando alguém abrir o arquivo, começará lendo nossas asserções primeiro ao invés desses métodos auxiliares.

Outra propriedade útil de testes é que eles nos ajudam a entender o uso *real* do nosso código e assim podemos fazer códigos mais compreensivos. Podemos ver aqui que um desenvolvedor pode simplesmente chamar nosso código e fazer uma comparação de igualdade a `ErroSaldoInsuficiente`, e então agir de acordo.

Erros não verificados

Embora o compilador do Go ajude bastante, há coisas que você pode acabar errando e o tratamento de erro pode se tornar complicado.

Há um cenário que nós não testamos. Para descobri-lo, execute o comando a seguir no terminal para instalar o `errcheck`, um dos muitos linters disponíveis em Go.

```
go get -u github.com/kisielk/errcheck
```

Então, dentro do diretório do seu código, execute `errcheck ..`

Você deve receber algo assim:

```
carteira_test.go:17:18: carteira.Retirar(Bitcoin(10))
```

O que isso está nos dizendo é que não verificamos o erro sendo retornado naquela linha de código. Aquela linha de código, no meu computador, corresponde para o nosso cenário normal de retirada, porque não verificamos que se `Retirar` é bem sucedido quando um erro *não* é retornado.

Aqui está o código de teste final que resolve isto.

```
func TestCarteira(t *testing.T) {
    t.Run("Depositar", func(t *testing.T) {
        carteira := Carteira{}
        carteira.Depositar(Bitcoin(10))

        confirmaSaldo(t, carteira, Bitcoin(10))
    })

    t.Run("Retirar com saldo suficiente", func(t *testing.T) {
        carteira := Carteira{Bitcoin(20)}
        erro := carteira.Retirar(Bitcoin(10))

        confirmaSaldo(t, carteira, Bitcoin(10))
        confirmaErroInexistente(t, erro)
    })

    t.Run("Retirar com saldo insuficiente", func(t *testing.T) {
        saldoInicial := Bitcoin(20)
        carteira := Carteira{saldoInicial}
        erro := carteira.Retirar(Bitcoin(100))

        confirmaSaldo(t, carteira, saldoInicial)
        confirmaErro(t, erro, ErroSaldoInsuficiente)
    })
}
```

```
func confirmaSaldo(t *testing.T, carteira Carteira, esperado Bitcoin) {
    t.Helper()
    resultado := carteira.Saldo()

    if resultado != esperado {
        t.Errorf("resultado %s, esperado %s", resultado, esperado)
    }
}

func confirmaErroInexistente(t *testing.T, resultado error) {
    t.Helper()
    if resultado != nil {
        t.Fatal("erro inesperado recebido")
    }
}

func confirmaErro(t *testing.T, resultado error, esperado error) {
    t.Helper()
    if resultado == nil {
        t.Fatal("esperava um erro, mas nenhum ocorreu")
    }

    if resultado != esperado {
        t.Errorf("erro resultado %s, erro esperado %s", resultado, esperado)
    }
}
```

Resumo

Ponteiros

- Go copia os valores quando são passados para funções/métodos. Então, se estiver escrevendo uma função que precise mudar o estado, você precisará de um ponteiro para o valor que você quer mudar.
- O fato de que Go pega um cópia dos valores é muito útil na maior parte do tempo, mas às vezes você não vai querer que o seu sistema faça cópia de alguma coisa. Nesse caso, você precisa passar uma referência. Podemos, por exemplo, ter dados muito grandes, ou coisas que você talvez pretenda ter apenas uma instância (como conexões a banco de dados).

nil

- Ponteiros podem ser `nil`.

- Quando uma função retorna um ponteiro para algo, você precisa ter certeza de verificar se ele é `nil` ou isso vai gerar uma exceção em tempo de execução, já que o compilador não te consegue te ajudar nesses casos.
- Útil para quando você quer descrever um valor que pode estar faltando.

Erros

- Erros são a forma de sinalizar falhas na execução de uma função/método.
- Analisando nossos testes, concluímos que buscar por uma string em um erro poderia resultar em um teste não muito confiável. Então, refatoramos para usar um valor significativo, que resultou em um código mais fácil de ser testado e concluímos que também seria mais fácil para usuários de nossa API.
- Este não é o fim do assunto de tratamento de erros. Você pode fazer coisas mais sofisticadas, mas esta é apenas uma introdução. Capítulos posteriores vão abordar mais estratégias.
- Não somente verifique os erros, trate-os graciosamente

Crie novos tipos a partir de existentes

- Útil para adicionar domínios mais específicos a valores
- Permite implementar interfaces

Ponteiros e erros são uma grande parte de escrita em Go que você precisa estar confortável. Por sorte, *na maioria das vezes* o compilador irá ajudar se você fizer algo errado. É só tomar um tempinho lendo a mensagem de erro.

Maps

Você pode encontrar todos os códigos para esse capítulo aqui

Em arrays e slices, vimos como armazenar valores em ordem. Agora, vamos descobrir uma forma de armazenar itens por uma `key` (chave) e procurar por ela rapidamente.

Maps te permitem armazenar itens de forma parecida com a de um dicionário. Você pode pensar na `chave` como a palavra e o `valor` como a definição. E tem forma melhor de aprender sobre maps do que criar seu próprio dicionário?

Primeiro, vamos presumir que já temos algumas palavras com suas definições no dicionário. Se procurarmos por uma palavra, o dicionário deve retornar sua definição.

Escreva o teste primeiro

Em `dicionario_test.go`

```
package main

import "testing"

func TestBusca(t *testing.T) {
    dicionario := map[string]string{"teste": "isso é apenas um teste"}

    resultado := Busca(dictionary, "teste")
    esperado := "isso é apenas um teste"

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s', dado '%s'", resultado, esperado, "test")
    }
}
```

Declarar um map é bem parecido com declarar um array. A diferença é que começa com a palavra-chave `map` e requer dois tipos. O primeiro é o tipo da chave, que é escrito dentro de `[]`. O segundo é o tipo do valor, que vai logo após o `[]`.

O tipo da chave é especial. Só pode ser um tipo comparável, porque sem a habilidade de dizer se duas chaves são iguais, não temos como ter certeza de que estamos obtendo o valor correto. Tipos comparáveis são explicados com detalhes na especificação da linguagem (em inglês).

O tipo do valor, por outro lado, pode ser o tipo que quiser. Pode até ser outro map.

O restante do teste já deve ser familiar para você.

Execute o teste

Ao executar `go test`, o compilador vai falhar com `./dicionario_test.go:8:9: undefined: Busca`.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Em `dicionario.go`:

```
package main

func Busca(dicionario map[string]string, palavra string) string {
```

```
    return ""
}
```

Agora seu teste vai falhar com uma *mensagem de erro clara*:

```
dicionario_test.go:12: resultado '', esperado 'isso é apenas um
teste', dado 'teste'.
```

Escreva código o suficiente para fazer o teste passar

```
func Busca(dicionario map[string]string, palavra string) string {
    return dicionario[palavra]
}
```

Obter um valor de um map é igual a obter um valor de um array: `map[chave]`.

Refatoração

```
func TestBusca(t *testing.T) {
    dicionario := map[string]string{"teste": "isso é apenas um teste"}

    resultado := Busca(dicionario, "teste")
    esperado := "isso é apenas um teste"

    comparaStrings(t, resultado, esperado)
}

func comparaStrings(t *testing.T, resultado, esperado string) {
    t.Helper()

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s', dado '%s'", resultado, esperado, "teste")
    }
}
```

Decidi criar um helper `comparaStrings` para tornar a implementação mais genérica.

Usando um tipo personalizado

Podemos melhorar o uso do nosso dicionário criando um novo tipo baseado no map e transformando a `Busca` em um método.

Em `dicionario_test.go`:

```
func TestBusca(t *testing.T) {
    dicionario := Dictionary{"teste": "isso é apenas um teste"}
```

```
    resultado := dictionary.Busca("teste")
    esperado := "isso é apenas um teste"

    comparaStrings(t, resultado, esperado)
}
```

Começamos a usar o tipo `Dicionario`, que ainda não definimos. Depois disso, chamamos `Busca` da instância de `Dicionario`.

Não precisamos mudar o `comparaStrings`.

Em `dicionario.go`:

```
type Dicionario map[string]string

func (d Dicionario) Busca(palavra string) string {
    return d[palavra]
}
```

Aqui criamos um tipo `Dicionario` que trabalha em cima da abstração de `map`. Com o tipo personalizado definido, podemos criar o método `Busca`.

Escreva o teste primeiro

A busca básica foi bem fácil de implementar, mas o que acontece se passarmos uma palavra que não está no nosso dicionário?

Com o código atual, não recebemos nada de volta. Isso é bom porque o programa continua a ser executado, mas há uma abordagem melhor. A função pode reportar que a palavra não está no dicionário. Dessa forma, o usuário não fica se perguntando se a palavra não existe ou se apenas não existe definição para ela (isso pode não parecer tão útil para um dicionário. No entanto, é um caso que pode ser essencial em outros casos de uso).

```
func TestBusca(t *testing.T) {
    dicionario := Dicionario{"teste": "isso é apenas um teste"}

    t.Run("palavra conhecida", func(t *testing.T) {
        resultado, _ := dicionario.Busca("teste")
        esperado := "isso é apenas um teste"

        comparaStrings(t, resultado, esperado)
    })

    t.Run("palavra desconhecida", func(t *testing.T) {
        _, resultado := dicionario.Busca("desconhecida")

        if err == nil {
```

```
            t.Fatal("é esperado que um erro seja obtido.")
        }
    })
}
```

A forma de lidar com esse caso no Go é retornar um segundo argumento que é do tipo `Error`.

Erros podem ser convertidos para uma string com o método `.Error()`, o que podemos fazer quando passarmos para a asserção. Também estamos protegendo o `comparaStrings` com `if` para certificar que não chamemos `.Error()` quando o erro for `nil`.

Execute o teste

Isso não vai compilar.

```
./dictionary_test.go:18:10: assignment mismatch: 2 variables but 1
values
```

```
incompatibilidade de atribuição: 2 variáveis, mas 1 valor
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
func (d Dicionario) Busca(palavra string) (string, error) {
    return d[palavra], nil
}
```

Agora seu teste deve falhar com uma mensagem de erro muito mais clara.

```
dictionary_test.go:22: expected to get an error.
```

```
erro esperado.
```

Escreva código o suficiente para fazer o teste passar

```
func (d Dicionario) Busca(palavra string) (string, error) {
    definicao, existe := d[palavra]
    if !existe {
        return "", errors.New("não foi possível encontrar a palavra que você procura")
    }

    return definicao, nil
}
```

Para fazê-lo passar, estamos usando uma propriedade interessante ao percorrer o map. Ele pode retornar dois valores. O segundo valor é uma boleana que indica se a chave foi encontrada com sucesso.

Essa propriedade nos permite diferenciar entre uma palavra que não existe e uma palavra que simplesmente não tem uma definição.

Refatoração

```
var ErrNaoEncontrado = errors.New("não foi possível encontrar a palavra que você procura")

func (d Dicionario) Busca(palavra string) (string, error) {
    definicao, existe := d[palavra]
    if !existe {
        return "", ErrNaoEncontrado
    }

    return definicao, nil
}
```

Podemos nos livrar do "erro mágico" na nossa função de `Busca` extraindo-o para dentro de uma variável. Isso também nos permite ter um teste melhor.

```
t.Run("palavra desconhecida", func(t *testing.T) {
    _, resultado := dicionario.Busca("desconhecida")

    comparaErro(t, resultado, ErrNaoEncontrado)
})

func comparaErro(t *testing.T, resultado, esperado error) {
    t.Helper()

    if resultado != esperado {
        t.Errorf("resultado erro '%s', esperado '%s'", resultado, esperado)
    }
}
```

Conseguimos simplificar nosso teste criando um novo helper e começando a usar nossa variável `ErrNaoEncontrado` para que nosso teste não falhe se mudarmos o texto do erro no futuro.

Escreva o teste primeiro

Temos uma ótima maneira de buscar no dicionário. No entanto, não temos como adicionar novas palavras nele.

```
func TestAdiciona(t *testing.T) {
    dicionario := Dicionario{}
    dicionario.Adiciona("teste", "isso é apenas um teste")

    esperado := "isso é apenas um teste"
    resultado, err := dicionario.Busca("teste")
    if err != nil {
        t.Fatal("não foi possível encontrar a palavra adicionada:", err)
    }

    if esperado != resultado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

Nesse teste, estamos utilizando nossa função `Busca` para tornar a validação do dicionário um pouco mais fácil.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Em `dicionario.go`

```
func (d Dicionario) Adiciona(palavra, definicao string) {
}
```

Agora seu teste deve falhar.

```
dicionario_test.go:31: deveria ter encontrado palavra adicionada: não foi possível encontrar
```

Escreva código o suficiente para fazer o teste passar

```
func (d Dicionario) Adiciona(palavra, definicao string) {
    d[palavra] = definicao
}
```

Adicionar coisas a um map também é bem semelhante a um array. Você só precisar especificar uma chave e definir qual é seu valor.

Tipos Referência

Uma propriedade interessante dos maps é que você pode modificá-los sem passá-los como ponteiro. Isso é porque o `map` é um tipo referência. Isso significa que ele contém uma referência à estrutura de dado que estamos utilizando, assim como um ponteiro. Logo, quando criamos passamos o map como parâmetro, estamos

alterando o map original e não sua cópia. A estrutura de dados utilizada é uma `tabela de dispersão` ou `mapa de hash`, e você pode ler mais sobre aqui.

É muito bom ter o map como referência, porque não importa o tamanho do map, só vai haver uma cópia.

Um conceito que os tipos referência apresentam é que maps podem ser um valor `nil`. Um map `nil` se comporta como um map vazio durante a leitura, mas tentar inserir coisas em um map `nil` gera um panic em tempo de execução. Você pode saber mais sobre maps aqui (em inglês).

Além disso, você nunca deve inicializar um map vazio, como:

```
var m map[string]string
```

Ao invés disso, você pode inicializar um map vazio como fizemos lá em cima, ou usando a palavra-chave `make` para criar um map para você:

```
dicionario = map[string]string{}

// OU

dicionario = make(map[string]string)
```

Ambas as abordagens criam um `hash map` vazio e apontam um `dicionario` para ele. Assim, nos certificamos que você nunca vai obter um panic em tempo de execução.

Refatoração

Não há muito para refatorar na nossa implementação, mas podemos simplificar o teste.

```
func TestAdiciona(t *testing.T) {
    dicionario := Dicionario{}
    palavra := "teste"
    definicao := "isso é apenas um teste"

    dicionario.Adiciona(palavra, definicao)

    comparaDefinicao(t, dicionario, palavra, definicao)
}

func comparaDefinicao(t *testing.T, dicionario Dicionario, palavra, definicao string) {
    t.Helper()

    resultado, err := dicionario.Busca(palavra)
    if err != nil {
        t.Fatal("deveria ter encontrado palavra adicionada:", err)
```

```
    }

    if definicao != resultado {
        t.Errorf("resultado '%s',  esperado '%s'", resultado, definicao)
    }
}
```

Criamos variáveis para palavra e definição e movemos a comparação da definição para sua própria função auxiliar.

Nosso `Adiciona` está bom. No entanto, não consideramos o que acontece quando o valor que estamos tentando adicionar já existe!

O map não vai mostrar um erro se o valor já existe. Ao invés disso, ele vai sobrescrever o valor com o novo recebido. Isso pode ser conveniente na prática, mas torna o nome da nossa função muito menos preciso. `Adiciona` não deve modificar valores existentes. Só deve adicionar palavras novas ao nosso dicionário.

Escreva o teste primeiro

```
func TestAdiciona(t *testing.T) {
    t.Run("palavra nova", func(t *testing.T) {
        dicionario := Dicionario{}
        palavra := "teste"
        definicao := "isso é apenas um teste"

        err := dicionario.Adiciona(palavra, definicao)

        comparaErro(t, err, nil)
        comparaDefinicao(t, dicionario, palavra, definicao)
    })

    t.Run("palavra existente", func(t *testing.T) {
        palavra := "teste"
        definicao := "isso é apenas um teste"
        dicionario := Dicionario{palavra: definicao}
        err := dicionario.Add(palavra, "teste novo")

        comparaErro(t, err, ErrPalavraExistente)
        comparaDefinicao(t, dicionario, palavra, definicao)
    })
}
```

Para esse teste, fizemos `Adiciona` devolver um erro, que estamos validando com uma nova variável de erro, `ErrPalavraExistente`. Também modificamos o teste anterior para verificar um erro `nil`.

Execute o teste

Agora o compilador vai falhar porque não estamos devolvendo um valor para `Adiciona`.

```
./dicionario_test.go:30:13: dicionario.Adiciona(palavra, definicao) used as value
./dicionario_test.go:41:13: dicionario.Adiciona(palavra, "teste novo") used as value
```

`usado como valor`

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Em `dicionario.go`:

```
var (
    ErrNaoEncontrado = errors.New("não foi possível encontrar a palavra que você procura")
    ErrPalavraExistente = errors.New("não é possível adicionar a palavra pois ela já existe"
)

func (d Dicionario) Adiciona(palavra, definicao string) error {
    d[palavra] = definicao
    return nil
}
```

Agora temos mais dois erros. Ainda estamos modificando o valor e retornando um erro `nil`.

```
dicionario_test.go:43: resultado erro '%!s(<nil>)', esperado 'não é possível adicionar a pal
dicionario_test.go:44: resultado 'teste novo', esperado 'isso é apenas um teste'
```

Escreva código o suficiente para fazer o teste passar

```
func (d Dicionario) Adiciona(palavra, definicao string) error {
    _, err := d.Busca(palavra)
    switch err {
    case ErrNaoEncontrado:
        d[palavra] = definicao
    case nil:
        return ErrPalavraExistente
    default:
        return err

    }

    return nil
}
```

Aqui estamos usando a declaração `switch` para coincidir com o erro. Usar o `switch` dessa forma dá uma segurança a mais, no caso de `Busca` retornar um erro diferente de `ErrNaoEncontrado`.

Refatoração

Não temos muito o que refatorar, mas já que nossos erros estão aumentando, podemos fazer algumas modificações.

```
const (
    ErrNaoEncontrado = ErrDicionario("não foi possível encontrar a palavra que você procura'
    ErrPalavraExistente = ErrDicionario("não é possível adicionar a palavra pois ela já exis
)

type ErrDicionario string

func (e ErrDicionario) Error() string {
    return string(e)
}
```

Tornamos os erros constantes; para isso, tivemos que criar nosso próprio tipo `ErrDicionario` que implementa a interface `error`. Você pode ler mais sobre nesse artigo excelente escrito por Dave Cheney (em inglês). Resumindo, isso torna os erros mais reutilizáveis e imutáveis.

Agora, vamos criar uma função que `Atualiza` a definição de uma palavra.

Escreva o teste primeiro

```
func TestUpdate(t *testing.T) {
    palavra := "teste"
    definicao := "isso é apenas um teste"
    dicionario := Dicionario{palavra: definicao}
    novaDefinicao := "nova definição"

    dicionario.Atualiza(palavra, novaDefinicao)

    comparaDefinicao(t, dicionario, palavra, novaDefinicao)
}
```

`Atualiza` é bem parecido com `Adiciona` e será nossa próxima implementação.

Execute o teste

```
./dicionario_test.go:53:2: dicionario.Atualiza undefined (type Dicionario has no field or me
```

```
dicionario.Atualiza não definido (tipo Dicionario não tem nenhum
campo ou método chamado Atualiza
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Já sabemos como lidar com um erro como esse. Precisamos definir nossa função.

```
func (d Dicionario) Atualiza(palavra, definicao string) {}
```

Feito isso, somos capazes de ver o que precisamos para mudar a definição da palavra.

```
dicionario_test.go:55: resultado 'isso é apenas um teste', esperado 'nova definição'
```

Escreva código o suficiente para fazer o teste passar

Já vimos como fazer essa implementação quando corrigimos o problema com `Adiciona`. Logo, vamos implementar algo bem parecido com `Adiciona`.

```
func (d Dicionario) Atualiza(palavra, definicao string) {
    d[palavra] = definicao
}
```

Não é necessário fazer refatorar nada, já que foi uma mudança simples. No entanto, agora temos o mesmo problema com `Adiciona`. Se passarmos uma palavra nova, `Atualiza` vai adicioná-la no dicionário.

Escreva o teste primeiro

```
    t.Run("palavra existente", func(t *testing.T) {
        palavra := "teste"
        definicao := "isso é apenas um teste"
        novaDefinicao := "nova definição"
        dicionario := Dicionario{palavra: definicao}
        err := dicionario.Atualiza(palavra, novaDefinicao)

        comparaErro(t, err, nil)
        comparaDefinicao(t, dicionario, palavra, novaDefinicao)
    })

    t.Run("palavra nova", func(t *testing.T) {
        palavra := "teste"
        definicao := "isso é apenas um teste"
        dicionario := Dicionario{}
```

```
        err := dicionario.Atualiza(palavra, definicao)

        comparaErro(t, err, ErrPalavraInexistente)
    })
```

Criamos um outro tipo de erro para quando a palavra não existe. Também modificamos o `Atualiza` para retornar um valor `error`.

Execute o teste

```
./dicionario_test.go:53:16: dicionario.Atualiza(palavra, "teste novo") used as value
./dicionario_test.go:64:16: dicionario.Atualiza(palavra, definicao) used as value
./dicionario_test.go:66:23: undefined: ErrPalavraInexistente
```

Agora recebemos três erros, mas sabemos como lidar com eles.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
const (
    ErrNaoEncontrado = ErrDicionario("não foi possível encontrar a palavra que você procura'
    ErrPalavraExistente = ErrDicionario("não é possível adicionar a palavra pois ela já exis
    ErrPalavraInexistente = ErrDicionario("não foi possível atualizar a palavra pois ela não
)

func (d Dicionario) Atualiza(palavra, definicao string) error {
    d[palavra] = definicao
    return nil
}
```

Adicionamos nosso próprio tipo erro e retornamos um erro `nil`.

Com essas mudanças, agora temos um erro muito mais claro:

```
dicionario_test.go:66: resultado erro '%!s(<nil>)', esperado 'não foi possível atualizar a p
```

Escreva código o suficiente para fazer o teste passar

```
func (d Dicionario) Atualiza(palavra, definicao string) error {
    _, err := d.Busca(palavra)
    switch err {
    case ErrNaoEncontrado:
        return ErrPalavraInexistente
    case nil:
        d[palavra] = definicao
    default:
```

```
        return err

    }

    return nil
}
```

Essa função é quase idêntica à `Adiciona`, com exceção de que trocamos quando atualizamos o `dicionario` e quando retornamos um erro.

Nota sobre a declaração de um novo erro para Atualiza

Poderíamos reutilizar `ErrNaoEncontrado` e não criar um novo erro. No entanto, geralmente é melhor ter um erro preciso para quando uma atualização falhar.

Ter erros específicos te dá mais informação sobre o que deu errado. Segue um exemplo em uma aplicação web:

> Você pode redirecionar o usuário quando o `ErrNaoEncontrado` é encontrado, mas mostrar uma mensagem de erro só quando `ErrPalavraInexistente` é encontrado.

Agora, vamos criar uma função que `Deleta` uma palavra no dicionário.

Escreva o teste primeiro

```
func TestDeleta(t *testing.T) {
    palavra := "teste"
    dicionario := Dicionario{palavra: "definição de teste"}

    dicionario.Deleta(palavra)

    _, err := dicionario.Busca(palavra)
    if err != ErrNaoEncontrado {
        t.Errorf("espera-se que '%s' seja deletado", palavra)
    }
}
```

Nosso teste cria um `Dicionario` com uma palavra e depois verifica se a palavra foi removida.

Execute o teste

Executando `go test` obtemos:

```
./dicionario_test.go:74:6: dicionario.Deleta undefined (type Dicionario has no field or meth
```

```
dicionario.Deleta não definido (tipo Dicionario não tem campo ou
método Deleta)
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
func (d Dicionario) Deleta(palavra string) {

}
```

Depois que adicionamos isso, o teste nos diz que não estamos deletando a palavra.

```
dicionario_test.go:78: espera-se que 'teste' seja deletado
```

Escreva código o suficiente para fazer o teste passar

```
func (d Dicionario) Deleta(palavra string) {
    delete(d, palavra)
}
```

Go tem uma função nativa chamada `delete` que funciona em maps. Ela leva dois argumentos: o primeiro é o map e o segundo é a chave a ser removida.

A função `delete` não retorna nada, e baseamos nosso método `Deleta` nesse conceito. Já que deletar um valor não tem nenhum efeito, diferentemente dos nossos métodos `Atualiza` e `Adiciona`, não precisamos complicar a API com erros.

Resumo

Nessa seção, falamos sobre muita coisa. Criamos uma API CRUD (Criar, Ler, Atualizar e Deletar) completa para nosso dicionário. No decorrer do processo, aprendemos como:

- Criar maps
- Buscar por itens em maps
- Adicionar novos itens aos maps
- Atualizar itens em maps
- Deletar itens de um map
- Aprendemos mais sobre erros
  - Como criar erros que são constantes
  - Escrever encapsuladores de erro

Injeção de dependência

Você pode encontrar todos os códigos para esse capítulo aqui

Presume-se que você tenha lido a seção de `structs` antes, já que será necessário saber um pouco sobre interfaces para entender este capítulo.

Há muitos mal entendidos relacionados à injeção de dependência na comunidade de programação. Se tudo der certo, esse guia vai te mostrar que:

- Você não precisa de uma framework
- Não torna seu design complexo demais
- Facilita seus testes
- Permite que você escreva funções ótimas para propósitos diversos.

Queremos criar uma função que cumprimenta alguém, assim como a que fizemos no capítulo Olá, mundo, mas dessa vez vamos testar o *print de verdade*.

Para recapitular, a função era parecida com isso:

```
func Cumprimenta(nome string) {
    fmt.Printf("Olá, %s", nome)
}
```

Mas como podemos testar isso? Chamar `fmt.Printf` imprime na saída, o que torna a captura com a ferramenta de testes bem difícil para nós.

O que precisamos fazer é sermos capazes de **injetar** (que é só uma palavra chique para passar) a dependência de impressão.

Nossa função não precisa se preocupar com *onde* ou *como* a impressão acontece, então vamos aceitar uma *interface* ao invés de um tipo concreto.

Se fizermos isso, podemos mudar a implementação para imprimir algo que controlamos para poder testá-lo. Na "vida real", você iria injetar em algo que escreve na saída.

Se dermos uma olhada no código fonte do `fmt.Printf`, podemos ver uma forma de começar:

```
// Printf retorna o número de bytes escritos e algum erro de escrita encontrado.
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}
```

Interessante! Por baixo dos panos, o `Printf` só chama o `Fprintf` passando o `os.Stdout`.

O que exatamente *é* um `os.Stdout`? O que o `Fprintf` espera que passe para ele como primeiro argumento?

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
    p := newPrinter()
```

```
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

Um `io.Writer`:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Quanto mais você escreve código em Go, mais vai perceber que essa interface aparece bastante, pois é uma ótima interface de uso geral para "colocar esses dados em algum lugar".

Logo, sabemos que por baixo dos panos estamos usando o `Writer` para enviar nosso cumprimento para algum lugar. Vamos usar essa abstração existente para tornar nosso código testável e mais reutilizável.

Escreva o teste primeiro

```
func TestCumprimenta(t *testing.T) {
    buffer := bytes.Buffer{}
    Cumprimenta(&buffer, "Chris")

    resultado := buffer.String()
    esperado := "Olá, Chris"

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

O tipo `buffer` do pacote `bytes` implementa a interface `Writer`.

Logo, vamos utilizá-lo no nosso teste para enviá-lo como nosso `Writer` e depois podemos verificar o que foi escrito nele quando chamamos `Cumprimenta`.

Execute o teste

O teste não vai compilar:

```
./id_test.go:10:7: too many arguments in call to Cumprimenta
    have (*bytes.Buffer, string)
    want (string)
```

```
./id_test.go:10:7: muitos argumentos na chamada de Cumprimenta
    obteve (*bytes.Buffer, string)
    esperado (string)
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Preste atenção no compilador e corrija o problema.

```
func Cumprimenta(escritor io.Writer, nome string) {
    fmt.Printf("Olá, %s", nome)
}
```

```
Olá, Chris id_test.go:16: resultado '', esperado 'Olá, Chris'
```

O teste falha. Note que o nome está sendo impresso, mas está indo para a saída.

Escreva código o suficiente para fazer o teste passar

Use o escritor para enviar o cumprimento para o buffer no nosso teste. Lembre-se que o `fmt.Fprintf` é parecido com o `fmt.Printf`, com a diferença de que leva um `Writer` em que a string é enviada, enquanto que o`fmt.Printf` redireciona para a saída por padrão.

```
func Cumprimenta(escritor io.Writer, nome string) {
    fmt.Fprintf(escritor, "Olá, %s", nome)
}
```

Agora o teste vai passar.

Refatoração

Antes, o compilador nos disse para passar um ponteiro para um `bytes.Buffer`. Isso está tecnicamente correto, mas não é muito útil.

Para demonstrar isso, tente utilizar a função `Cumprimenta` em uma aplicação Go onde queremos que imprima na saída.

```
func main() {
    Cumprimenta(os.Stdout, "Elodie")
}
```

```
./id.go:14:7: cannot use os.Stdout (type *os.File) as type
*bytes.Buffer in argument to Cumprimenta
```

```
não é possível utilizar os.Stdout (tipo *os.File) como tipo
*bytes.Buffer no argumento para Cumprimenta
```

Como discutimos antes, o `fmt.Fprintf` te permite passar um `io.Writer`, que sabemos que o `os.Stdout` e `bytes.Buffer` implementam.

Se mudarmos nosso código para usar uma interface de propósito mais geral, podemos usá-la tanto nos testes quanto na nossa aplicação.

```
package main

import (
    "fmt"
    "os"
    "io"
)

func Cumprimenta(escritor io.Writer, nome string) {
    fmt.Fprintf(escritor, "Olá, %s", nome)
}

func main() {
    Cumprimenta(os.Stdout, "Elodie")
}
```

Mais sobre io.Writer

Quais outros lugares podemos escrever dados usando `io.Writer`? Para qual propósito geral nossa função `Cumprimenta` é feita?

A internet

Execute o seguinte:

```
package main

import (
    "fmt"
    "io"
    "net/http"
)

func Cumprimenta(escritor io.Writer, nome string) {
    fmt.Fprintf(escritor, "Olá, %s", nome)
}

func HandlerMeuCumprimento(w http.ResponseWriter, r *http.Request) {
    Cumprimenta(w, "mundo")
}
```

```
func main() {
    err := http.ListenAndServe(":5000", http.HandlerFunc(HandlerMeuCumprimento))

    if err != nil {
        fmt.Println(err)
    }
}
```

Execute o programa e vá para http://localhost:5000. Você verá sua função de cumprimento ser utilizada.

Falaremos sobre servidores HTTP em um próximo capítulo, então não se preocupe muito com os detalhes.

Quando se cria um handler HTTP, você recebe um `http.ResponseWriter` e o `http.Request` que é usado para fazer a requisição. Quando implementa seu servidor, você *escreve* sua resposta usando o escritor.

Você deve ter adivinhado que o `http.ResponseWriter` também implementa o `io.Writer` e é por isso que podemos reutilizar nossa função `Cumprimenta` dentro do nosso handler.

Resumo

Nossa primeira rodada de código não foi fácil de testar porque escrevemos dados em algum lugar que não podíamos controlar.

Graças aos nossos testes, refatoramos o código para que pudéssemos controlar para *onde* os dados eram escritos **injetando uma dependência** que nos permitiu:

- **Testar nosso código**: se você não consegue testar uma função *de forma simples*, geralmente é porque dependências estão acopladas em uma função *ou* estado global. Se você tem um pool de conexão global da base de dados, por exemplo, é provável que seja difícil testar e vai ser lento para ser executado. A injeção de dependência te motiva a injetar em uma dependência da base de dados (através de uma interface), para que você possa criar um mock com algo que você possa controlar nos seus testes.
- *Separar nossas preocupações*, desacoplando *onde os dados vão* de *como gerá-los*. Se você já achou que um método/função tem responsabilidades demais (gerando dados *e* escrevendo na base de dados? Lidando com requisições HTTP *e* aplicando lógica a nível de domínio?), a injeção de dependência provavelmente será a ferramenta que você precisa.
- **Permitir que nosso código seja reutilizado em contextos diferentes**: o primeiro contexto "novo" do nosso código pode ser usado dentro dos testes. No entanto, se alguém quiser testar algo novo com nossa função, a pessoa pode injetar suas próprias dependências.

Mas e o mock? Ouvi falar que precisa disso para trabalhar com injeção de dependência e que também é do demonho

Vamos falar mais sobre mocks depois (e não é do demonho). Você mocka para substituir coisas reais que você injeta com uma versão falsa que você pode controlar e examinar nos seus testes. No entanto, no nosso caso a biblioteca padrão já tinha algo pronto para usarmos.

A biblioteca padrão do Go é muito boa, leve um tempo para estudá-la

Ao termos familiaridade com a interface `io.Writer`, somos capazes de usar `bytes.Buffer` no nosso teste como nosso `Writer` para que depois possamos usar outros `Writer` da biblioteca padrão para usar na nossa função em uma aplicação de linha de comando ou em um servidor web.

Quanto mais familiar você for com a biblioteca padrão, mais vai ver essas interfaces de propósito geral que você pode reutilizar no seu próprio código para tornar o software reutilizável em vários contextos diferentes.

Esse exemplo teve grande influência de um capítulo de A Linguagem de Programação Go. Logo, se gostou, vá adquiri-lo!

Mocks

Você pode encontrar todos os códigos para esse capítulo aqui

Te pediram para criar um programa que conta a partir de 3, imprimindo cada número em uma linha nova (com um segundo de intervalo entre cada uma) e quando chega a zero, imprime "Vai!" e sai.

```
3
2
1
Vai!
```

Vamos resolver isso escrevendo uma função chamada `Contagem` que vamos colocar dentro de um programa `main` e se parecer com algo assim:

```
package main

func main() {
    Contagem()
}
```

Apesar de ser um programa simples, para testá-lo completamente vamos precisar, como de costume, de uma abordagem *iterativa* e *orientada a testes*.

Mas o que quero dizer com iterativa? Precisamos ter certeza de que tomamos os menores passos que pudermos para ter um *software* útil.

Não queremos passar muito tempo com código que vai funcionar hora ou outra após alguma implementação mirabolante, porque é assim que os desenvolvedores caem em armadilhas. **É importante ser capaz de dividir os requerimentos da menor forma que conseguir para você ter um *software funcionando*.**

Podemos separar essa tarefa da seguinte forma:

- Imprimir 3
- Imprimir de 3 para Vai!
- Esperar um segundo entre cada linha

Escreva o teste primeiro

Nosso software precisa imprimir para a saída. Vimos como podemos usar a injeção de dependência para facilitar nosso teste na seção anterior.

```
func TestContagem(t *testing.T) {
    buffer := &bytes.Buffer{}

    Contagem(buffer)

    resultado := buffer.String()
    esperado := "3"

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

Se tiver dúvidas sobre o `buffer`, leia a seção anterior novamente.

Sabemos que nossa função `Contagem` precisa escrever dados em algum lugar e o `io.Writer` é a forma de capturarmos essa saída como uma interface em Go.

- Na `main`, vamos enviar o `os.Stdout` como parâmetro para nossos usuários verem a contagem regressiva impressa no terminal.
- No teste, vamos enviar o `bytes.Buffer` como parâmetro para que nossos testes possam capturar que dado está sendo gerado.

Execute o teste

```
./contagem_test.go:11:2: undefined: Contagem
```

```
indefinido: Contagem
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Defina `Contagem`:

```
func Contagem() {}
```

Tente novamente:

```
./contagem_test.go:11:11: too many arguments in call to Countdown
    have (*bytes.Buffer)
    want ()
```

`argumentos demais na chamada para Contagem`

O compilador está te dizendo como a assinatura da função deve ser, então é só atualizá-la.

```
func Contagem(saida *bytes.Buffer) {}
```

```
contagem_test.go:17: resultado '', esperado '3'
```

Perfeito!

Escreva código o suficiente para fazer o teste passar

```
func Contagem(saida *bytes.Buffer) {
    fmt.Fprint(saida, "3")
}
```

Estamos usando `fmt.Fprint`, o que significa que ele recebe um `io.Writer` (como `*bytes.Buffer`) e envia uma `string` para ele. O teste deve passar.

Refatoração

Agora sabemos que, apesar do `*bytes.Buffer` funcionar, seria melhor ter uma interface de propósito geral ao invés disso.

```
func Contagem(saida io.Writer) {
    fmt.Fprint(saida, "3")
}
```

Execute os testes novamente e eles devem passar.

Só para finalizar, vamos colocar nossa função dentro da `main` para que possamos executar o software para nos assegurarmos de que estamos progredindo.

```
package main

import (
    "fmt"
    "io"
```

```
    "os"
)

func Contagem(saida io.Writer) {
    fmt.Fprint(saida, "3")
}

func main() {
    Contagem(os.Stdout)
}
```

Execute o programa e surpreenda-se com seu trabalho.

Apesar de parecer simples, essa é a abordagem que recomendo para qualquer projeto. **Escolher uma pequena parte da funcionalidade e fazê-la funcionar do começo ao fim com apoio de testes.**

Depois, precisamos fazer o software imprimir 2, 1 e então "Vai!".

Escreva o teste primeiro

Após investirmos tempo e esforço para fazer o principal funcionar, podemos iterar nossa solução com segurança e de forma simples. Não vamos mais precisar parar e executar o programa novamente para ter confiança de que ele está funcionando, desde que a lógica esteja testada.

```
func TestContagem(t *testing.T) {
    buffer := &bytes.Buffer{}

    Contagem(buffer)

    resultado := buffer.String()
    esperado := `3
2
1
Vai!`
    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

A sintaxe de aspas simples é outra forma de criar uma `string`, mas te permite colocar coisas como linhas novas, o que é perfeito para nosso teste.

Execute o teste

```
contagem_test.go:21: resultado '3', esperado '3
        2
        1
        Vai!'
```

Escreva código o suficiente para fazer o teste passar

```
func Contagem(saida io.Writer) {
    for i := 3; i > 0; i-- {
        fmt.Fprintln(saida, i)
    }
    fmt.Fprint(saida, "Go!")
}
```

Usamos um laço `for` fazendo contagem regressiva com `i--` e depois `fmt.Fprintln` para imprimir a `saida` com nosso número seguro por um caracter de nova linha. Finalmente, usamos o `fmt.Fprint` para enviar "Vai!" no final.

Refatoração

Não há muito para refatorar além de transformar alguns valores mágicos em constantes com nomes descritivos.

```
const ultimaPalavra = "Go!"
const inicioContagem = 3

func Contagem(saida io.Writer) {
    for i := inicioContagem; i > 0; i-- {
        fmt.Fprintln(saida, i)
    }
    fmt.Fprint(saida, ultimaPalavra)
}
```

Se executar o programa agora, você deve obter a saída desejada, mas não tem uma contagem regressiva dramática com as pausas de 1 segundo.

Go te permite obter isso com `time.Sleep`. Tente adicionar essa função ao seu código.

```
func Contagem(saida io.Writer) {
    for i := inicioContagem; i > 0; i-- {
        time.Sleep(1 * time.Second)
        fmt.Fprintln(saida, i)
    }
```

```
    time.Sleep(1 * time.Second)
    fmt.Fprint(saida, ultimaPalavra)
}
```

Se você executar o programa, ele funciona conforme esperado.

Mock

Os testes ainda vão passar e o software funciona como planejado, mas temos alguns problemas:

- Nossos testes levam 4 segundos para rodar.
  - Todo conteúdo gerado sobre desenvolvimento de software enfatiza a importância de loops de feedback rápidos.
  - **Testes lentos arruinam a produtividade do desenvolvedor**.
  - Imagine se os requerimentos ficam mais sofisticados, gerando a necessidade de mais testes. É viável adicionar 4s para cada teste novo de `Contagem`?
- Não testamos uma propriedade importante da nossa função.

Temos uma dependência no `Sleep` que precisamos extrair para podermos controlá-la nos nossos testes.

Se conseguirmos *mockar* o `time.Sleep`, podemos usar a *injeção de dependências* para usá-lo ao invés de um `time.Sleep` "de verdade", e então podemos **verificar as chamadas** para certificar de que estão corretas.

Escreva o teste primeiro

Vamos definir nossa dependência como uma interface. Isso nos permite usar um Sleeper *de verdade* em `main` e um *sleeper spy* nos nossos testes. Usar uma interface na nossa função `Contagem` é essencial para isso e dá certa flexibilidade à função que a chamar.

```
type Sleeper interface {
    Sleep()
}
```

Tomei uma decisão de design que nossa função `Contagem` não seria responsável por quanto tempo o sleep leva. Isso simplifica um pouco nosso código, pelo menos por enquanto, e significa que um usuário da nossa função pode configurar a duração desse tempo como preferir.

Agora precisamos criar um *mock* disso para usarmos nos nossos testes.

```
type SleeperSpy struct {
    Chamadas int
}
```

```
func (s *SleeperSpy) Sleep() {
    s.Chamadas++
}
```

Spies (espiões) são um tipo de *mock* em que podemos gravar como uma dependência é usada. Eles podem gravar os argumentos definidos, quantas vezes são usados etc. No nosso caso, vamos manter o controle de quantas vezes `Sleep()` é chamada para verificá-la no nosso teste.

Atualize os testes para injetar uma dependência no nosso Espião e verifique se o sleep foi chamado 4 vezes.

```
func TestContagem(t *testing.T) {
    buffer := &bytes.Buffer{}
    sleeperSpy := &SleeperSpy{}

    Contagem(buffer, sleeperSpy)

    resultado := buffer.String()
    esperado := `3
2
1
Vai!`

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }

    if sleeperSpy.Chamadas != 4 {
        t.Errorf("não houve chamadas suficientes do sleeper, esperado 4, resultado %d", slee
    }
}
```

Execute o teste

```
too many arguments in call to Contagem
    have (*bytes.Buffer, *SpySleeper)
    want (io.Writer)
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Precisamos atualizar a `Contagem` para aceitar nosso `Sleeper`:

```
func Contagem(saida io.Writer, sleeper Sleeper) {
    for i := inicioContagem; i > 0; i-- {
        time.Sleep(1 * time.Second)
        fmt.Fprintln(saida, i)
    }

    time.Sleep(1 * time.Second)
    fmt.Fprint(saida, ultimaPalavra)
}
```

Se tentar novamente, nossa `main` não vai mais compilar pelo mesmo motivo:

```
./main.go:26:11: not enough arguments in call to Contagem
    have (*os.File)
    want (io.Writer, Sleeper)
```

Vamos criar um sleeper *de verdade* que implementa a interface que precisamos:

```
type SleeperPadrao struct {}

func (d *SleeperPadrao) Sleep() {
    time.Sleep(1 * time.Second)
}
```

Podemos usá-lo na nossa aplicação real, como:

```
func main() {
    sleeper := &SleeperPadrao{}
    Contagem(os.Stdout, sleeper)
}
```

Escreva código o suficiente para fazer o teste passar

Agora o teste está compilando, mas não passando. Isso acontece porque ainda estamos chamando o `time.Sleep` ao invés da injetada. Vamos arrumar isso.

```
func Contagem(saida io.Writer, sleeper Sleeper) {
    for i := inicioContagem; i > 0; i-- {
        sleeper.Sleep()
        fmt.Fprintln(saida, i)
    }

    sleeper.Sleep()
    fmt.Fprint(saida, ultimaPalavra)
}
```

O teste deve passar sem levar 4 segundos.

Ainda temos alguns problemas

Ainda há outra propriedade importante que não estamos testando.

A `Contagem` deve ter uma pausa para cada impressão, como por exemplo:

- `Pausa`
- `Imprime N`
- `Pausa`
- `Imprime N-1`
- `Pausa`
- `Imprime Vai!`
- etc

Nossa alteração mais recente só verifica se o software teve 4 pausas, mas essas pausas poderiam ocorrer fora de ordem.

Quando escrevemos testes, se não estiver confiante de que seus testes estão te dando confiança o suficiente, quebre-o (mas certifique-se de que você salvou suas alterações antes)! Mude o código para o seguinte:

```
func Contagem(saida io.Writer, sleeper Sleeper) {
    for i := inicioContagem; i > 0; i-- {
        sleeper.Pausa()
        fmt.Fprintln(saida, i)
    }

    for i := inicioContagem; i > 0; i-- {
        fmt.Fprintln(saida, i)
    }

    sleeper.Pausa()
    fmt.Fprint(saida, ultimaPalavra)
}
```

Se executar seus testes, eles ainda vão passar, apesar da implementação estar errada.

Vamos usar o spy novamente com um novo teste para verificar se a ordem das operações está correta.

Temos duas dependências diferentes e queremos gravar todas as operações delas em uma lista. Logo, vamos criar *um spy para ambas*.

```
type SpyContagemOperacoes struct {
    Chamadas []string
}

func (s *SpyContagemOperacoes) Pausa() {
    s.Chamadas = append(s.Chamadas, pausa)
}
```

```
func (s *SpyContagemOperacoes) Write(p []byte) (n int, err error) {
    s.Chamadas = append(s.Chamadas, escrita)
    return
}

const escrita = "escrita"
const pausa = "pausa"
```

Nosso `SpyContagemOperacoes` implementa tanto o `io.Writer` quanto o `Sleeper`, gravando cada chamada em um slice. Nesse teste, temos preocupação apenas na ordem das operações, então apenas gravá-las em uma lista de operações nomeadas é suficiente.

Agora podemos adicionar um subteste no nosso conjunto de testes.

```
t.Run("pausa antes de cada impressão", func(t *testing.T) {
        spyImpressoraSleep := &SpyContagemOperacoes{}
        Contagem(spyImpressoraSleep, spyImpressoraSleep)

        esperado := []string{
            pausa,
            escrita,
            pausa,
            escrita,
            pausa,
            escrita,
            pausa,
            escrita,
        }

        if !reflect.DeepEqual(esperado, spyImpressoraSleep.Chamadas) {
            t.Errorf("esperado %v chamadas, resultado %v", esperado, spyImpressoraSleep.Cha
        }
    })
```

Esse teste deve falhar. Volte o código que quebramos para a versão correta e agora o novo teste deve passar.

Agora temos dois spies no `Sleeper`. O próximo passo é refatorar nosso teste para que um teste o que está sendo impresso e o outro se certifique de que estamos pausando entre as impressões. Por fim, podemos apagar nosso primeiro spy, já que não é mais utilizado.

```
func TestContagem(t *testing.T) {

    t.Run("imprime 3 até Vai!", func(t *testing.T) {
        buffer := &bytes.Buffer{}
        Contagem(buffer, &SpyContagemOperacoes{})
```

```
        resultado := buffer.String()
        esperado := `3
2
1
Vai!`

        if resultado != esperado {
            t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
        }
    })

    t.Run("pausa antes de cada impressão", func(t *testing.T) {
        spyImpressoraSleep := &SpyContagemOperacoes{}
        Contagem(spyImpressoraSleep, spyImpressoraSleep)

        esperado := []string{
            pausa,
            escrita,
            pausa,
            escrita,
            pausa,
            escrita,
            pausa,
            escrita,
        }

        if !reflect.DeepEqual(esperado, spyImpressoraSleep.Chamadas) {
            t.Errorf("esperado %v chamadas, resultado %v", esperado, spyImpressoraSleep.Cha
        }
    })
}
```

Agora temos nossa função e suas duas propriedades testadas adequadamente.

Extendendo o Sleeper para se tornar configurável

Uma funcionalidade legal seria o `Sleeper` ser configurável.

Escreva o teste primeiro

Agora vamos criar um novo tipo para `SleeperConfiguravel` que aceita o que precisamos para configuração e teste.

```
type SleeperConfiguravel struct {
    duracao time.Duration
```

```
    pausa   func(time.Duration)
}
```

Estamos usando a `duracao` para configurar o tempo de pausa e `pausa` como forma de passar uma função de pausa. A assinatura de `sleep` é a mesma de `time.Sleep`, nos permitindo usar `time.Sleep` na nossa implementação real e um spy nos nossos testes.

```
type TempoSpy struct {
    duracaoPausa time.Duration
}

func (t *TempoSpy) Pausa(duracao time.Duration) {
    t.duracaoPausa = duracao
}
```

Definindo nosso spy, podemos criar um novo teste para o sleeper configurável.

```
func TestSleeperConfiguravel(t *testing.T) {
    tempoPausa := 5 * time.Second

    tempoSpy := &TempoSpy{}
    sleeper := SleeperConfiguravel{tempoPausa, tempoSpy.Pausa}
    sleeper.Pausa()

    if tempoSpy.duracaoPausa != tempoPausa {
        t.Errorf("deveria ter pausado por %v, mas pausou por %v", tempoPausa, tempoSpy.durac
    }
}
```

Não há nada de novo nesse teste e seu funcionamento é bem semelhante aos testes com mock anteriores.

Execute o teste

```
sleeper.Pausa undefined (type SleeperConfiguravel has no field or method Pausa, but does hav
```

```
sleeper.Pausa não definido (tipo SleeperConfiguravel não tem campo
ou método Pausa, mas tem o método sleep
```

Você deve ver uma mensagem de erro bem clara indicando que não temos um método `Pausa` criado no nosso `SleeperConfiguravel`.

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
func (c *SleeperConfiguravel) Pausa() {
}
```

Com nossa nova função `Pausa` implementada, ainda há um teste falhando.

```
contagem_test.go:56: deveria ter pausado por 5s, mas pausou por 0s
```

Escreva código o suficiente para fazer o teste passar

Tudo o que precisamos fazer agora é implementar a função `Pausa` para o `SleeperConfiguravel`.

```
func (s *SleeperConfiguravel) Pausa() {
    s.pausa(s.duracao)
}
```

Com essa mudança, todos os testes devem voltar a passar.

Limpeza e refatoração

A última coisa que precisamos fazer é de fato usar nosso `SleeperConfiguravel` na função main.

```
func main() {
    sleeper := &SleeperConfiguravel{1 * time.Second, time.Sleep}
    Contagem(os.Stdout, sleeper)
}
```

Se executarmos os testes e o programa manualmente, podemos ver que todo o comportamento permanece o mesmo.

Já que estamos usando o `SleeperConfiguravel`, é seguro deletar o `SleeperPadrao`.

Mas o mock não é do demonho?

Você já deve ter ouvido que o mock é do mal. Quase qualquer coisa no desenvolvimento de software pode ser usada para o mal, assim como o DRY.

As pessoas acabam chegando numa fase ruim em que não *dão atenção aos próprios testes* e *não respeitam a etapa de refatoração.*

Se seu código de mock estiver ficando complicado ou você tem que mockar muita coisa para testar algo, você deve *prestar mais atenção* a essa sensação ruim e pensar sobre o seu código. Geralmente isso é sinal de que:

- A coisa que você está testando está tendo que fazer coisas demais
  - Modularize a função para que faça menos coisas
- Suas dependências estão muito desacopladas
  - Pense e uma forma de consolidar algumas das dependências em um módulo útil
- Você está se preocupando demais com detalhes de implementação

– Dê prioridade em testar o comportamento esperado ao invés da implementação

Normalmente, muitos pontos de mock são sinais de *abstração ruim* no seu código.

As pessoas costumam pensar que essa é uma fraqueza no TDD, mas na verdade é um ponto forte. Testes mal desenvolvidos são resultado de código ruim. Código bem desenvolvido é fácil de ser testado.

Só que mocks e testes ainda estão dificultando minha vida!

Já se deparou com a situação a seguir?

- Você quer refatorar algo
- Para isso, você precisa mudar vários testes
- Você duvida do TDD e cria um post no Medium chamado "Mock é prejudicial"

Isso costuma ser um sinal de que você está testando muito *detalhe de implementação*. Tente fazer de forma que esteja testando *comportamentos úteis*, a não ser que a implementação seja tão importante que a falta dela possa fazer o sistema quebrar.

Às vezes é difícil saber *qual nível* testar exatamente, então aqui vai algumas ideias e regras que tento seguir:

-**A definição de refatoração é que o código muda, mas o comportamento permanece o mesmo**. Se você decidiu refatorar alguma coisa, na teoria você deve ser capaz de salvar seu código sem que o teste mude. Então, quando estiver escrevendo um teste, pergunte para si: - Estou testando o comportamento que quero ou detalhes de implementação? - Se fosse refatorar esse código, eu teria que fazer muitas mudanças no meu teste?

- Apesar do Go te deixar testar funções privadas, eu evitaria fazer isso, já que funções privadas costumam ser detalhes de implementação.
- Se o teste estiver com **3 mocks, esse é um sinal de alerta** - hora de repensar no design.
- Use spies com cuidado. Spies te deixam ver a parte interna do algoritmo que você está escrevendo, o que pode ser bem útil, mas significa que há um acoplamento maior entre o código do teste e a implementação. **Certifique-se de que você realmente precisa desses detalhes se você vai colocar um spy neles**.

Como sempre, regras no desenvolvimento de software não são realmente regras e podem haver exceções. O artigo do Uncle Bob sobre "Quando mockar" (em inglês) tem alguns pontos excelentes.

Resumo

Mais sobre abordagem TDD

- Quando se deparar com exemplos menos comuns, divida o problema em "linhas verticais finas". Tente chegar em um ponto onde você tem *software em funcionamento com o apoio de testes* o mais rápido possível, para evitar cair em armadilhas e se perder.
- Quando tiver uma parte do software em funcionamento, deve ser mais fácil *iterar com etapas pequenas* até chegar no software que você precisa.

  "Quando usar o desenvolvimento iterativo? Apenas em projetos que você quer obter sucesso."

Martin Fowler.

Mock

- **Sem o mock, partes importantes do seu código não serão testadas**. No nosso caso, não seríamos capazes de testar se nosso código pausava em cada impressão, mas existem inúmeros exemplos. Chamar um serviço que *pode* falhar? Querer testar seu sistema em um estado em particular? É bem difícil testar esses casos sem mock.
- Sem mocks você pode ter que definir bancos de dados e outras dependências externas só para testar regras de negócio simples. Seus testes provavelmente ficarão mais lentos, resultando em **loops de feedback lentos**.
- Ter que se conectar a um banco de dados ou webservice para testar algo vai tornar seus testes **frágeis** por causa da falta de segurança nesses serviços.

Uma vez que a pessoa aprende a mockar, é bem fácil testar pontos demais de um sistema em termos da *forma que ele funciona* ao invés *do que ele faz*. Sempre tenha em mente o **valor dos seus testes** e qual impacto eles teriam em uma refatoração futura.

Nesse artigo sobre mock, falamos sobre **spies**, que são um tipo de mock. Aqui estão diferentes tipos de mocks. O Uncle Bob explica os tipos em um artigo bem fácil de ler (em inglês). Nos próximos capítulos, vamos precisar escrever código que depende de outros para obter dados, que é aonde vou mostrar os **Stubs** em ação.

Concorrência

Você pode encontrar todos os códigos para esse capítulo aqui

A questão é a seguinte: um colega escreveu uma função, `VerificaWebsites`, que verifica o status de uma lista de URLs.

```go
package concorrencia

type VerificadorWebsite func(string) bool

func VerificaWebsites(vw VerificadorWebsite, urls []string) map[string]bool {
    resultados := make(map[string]bool)

    for _, url := range urls {
        resultados[url] = vw(url)
    }

    return resultados
}
```

Ela retorna um map de cada URL verificado com um valor booleano - `true` para uma boa resposta, `false` para uma resposta ruim.

Você também tem que passar um `VerificadorWebsite` como parâmetro, que leva um URL e retorna um boleano. Isso é usado pela função que verifica todos os websites.

Usando a injeção de dependência, conseguimos testar a função sem fazer chamadas HTTP de verdade, tornando o teste seguro e rápido.

Aqui está o teste que escreveram:

```go
package concurrency

import (
    "reflect"
    "testing"
)

func mockVerificadorWebsite(url string) bool {
    if url == "waat://furhurterwe.geds" {
        return false
    }
    return true
}

func TestVerificaWebsites(t *testing.T) {
    websites := []string{
        "http://google.com",
        "http://blog.gypsydave5.com",
        "waat://furhurterwe.geds",
    }

    esperado := map[string]bool{
```

```
        "http://google.com":          true,
        "http://blog.gypsydave5.com": true,
        "waat://furhurterwe.geds":    false,
    }

    resultado := VerificaWebsites(mockVerificadorWebsite, websites)

    if !reflect.DeepEqual(esperado, resultado) {
        t.Fatalf("esperado %v, resultado %v", esperado, resultado)
    }
}
```

A função que está em produção está sendo usada para verificar centenas de websites. Só que seu colega começou a reclamar que está lento demais, e pediram sua ajuda para melhorar a velocidade dele.

Escreva o teste primeiro

Vamos usar um teste de benchmark para testar a velocidade de `VerificaWebsites` para que possamos ver o efeito das nossas alterações.

```
package concorrencia

import (
    "testing"
    "time"
)

func slowStubVerificadorWebsite(_ string) bool {
    time.Sleep(20 * time.Millisecond)
    return true
}

func BenchmarkVerificaWebsites(b *testing.B) {
    urls := make([]string, 100)
    for i := 0; i < len(urls); i++ {
        urls[i] = "uma url"
    }

    for i := 0; i < b.N; i++ {
        VerificaWebsites(slowStubVerificadorWebsite, urls)
    }
}
```

O benchmark testa `VerificaWebsites` usando um slice de 100 URLs e usa uma nova implementação falsa de `VerificadorWebsite`. `slowStubVerificadorWebsite` é intencionalmente lento. Ele usa um `time.Sleep` para esperar exatamente 20

milissegundos e então retorna verdadeiro.

Quando executamos o benchmark com `go test -bench=.` (ou, se estiver no Powershell do Windows, `go test -bench="."`):

```
pkg: github.com/larien/learn-go-with-tests/concorrencia/v1
BenchmarkVerificaWebsites-4               1        2249228637 ns/op
PASS
ok      github.com/larien/learn-go-with-tests/concorrencia/v1        2.268s
```

`VerificaWebsites` teve uma marca de 2249228637 nanosegundos - pouco mais de dois segundos.

Vamos torná-lo mais rápido.

Escreva código o suficiente para fazer o teste passar

Agora finalmente podemos falar sobre concorrência que, apenas para fins dessa situação, significa "fazer mais do que uma coisa ao mesmo tempo". Isso é algo que fazemos naturalmente todo dia.

Por exemplo, hoje de manhã fiz uma xícara de chá. Coloquei a chaleira no fogo e, enquanto esperava a água ferver, tirei o leite da geladeira, tirei o chá do armário, encontrei minha xícara favorita, coloquei o saquinho do chá e, quando a chaleira ferveu a água, coloquei a água na xícara.

O que eu *não fiz* foi colocar a chaleira no fogo e então ficar sem fazer nada só esperando a chaleira ferver a água, para depois fazer todo o restante quando a água tivesse fervido.

Se conseguir entender por que é mais rápido fazer chá da primeira forma, então você é capaz de entender como vamos tornar o `VerificaWebsites` mais rápido. Ao invés de esperar por um website responder antes de enviar uma requisição para o próximo website, vamos dizer para nosso computador fazer a próxima requisição enquanto espera pela primeira.

Normalmente, em Go, quando chamamos uma função `fazAlgumaCoisa()`, esperamos que ela retorne alguma coisa (mesmo se não tiver valor para retornar, ainda esperamos que ela termine). Chamamos essa operação de *bloqueante* - espera algo acabar para terminar seu trabalho. Uma operação que não bloqueia no Go vai rodar em um *processo* separado, chamado de *goroutine*. Pense no processo como uma leitura de uma página de código Go de cima para baixo, 'entrando' em cada função quando é chamado para ler o que essa página faz. Quando um processo separado começa, é como se outro leitor começasse a ler o interior da função, deixando o leitor original continuar lendo a página.

Para dizer ao Go começar uma nova goroutine, transformamos a chamada de função em uma declaração `go` colocando a palavra-chave `go` na frente da função: `go fazAlgumaCoisa()`.

```
package concurrency

type VerificadorWebsite func(string) bool

func VerificaWebsites(vw VerificadorWebsite, urls []string) map[string]bool {
    resultados := make(map[string]bool)

    for _, url := range urls {
        go func() {
            resultados[url] = vw(url)
        }()
    }

    return resultados
}
```

Já que a única forma de começar uma goroutine é colocar `go` na frente da chamada de função, costumamos usar *funções anônimas* quando queremos iniciar uma goroutine. Uma função anônima literal é bem parecida com uma declaração de função normal, mas (obviamente) sem um nome. Você pode ver uma acima no corpo do laço `for`.

Funções anônimas têm várias funcionalidades que as tornam úteis, duas das quais estamos usando acima. Primeiramente, elas podem ser executadas assim que fazemos sua declaração - que é o `()` no final da função anônima. Em segundo lugar, elas mantém acesso ao escopo léxico em que são definidas - todas as variáveis que estão disponíveis no ponto em que a função anônima é declarada também estão disponíveis no corpo da função.

O corpo da função anônima acima é quase o mesmo da função no laço utilizada anteriormente. A única diferença é que cada iteração do loop vai iniciar uma nova goroutine, concorrente com o processo atual (a função `VerificadorWebsite`), e cada uma vai adicionar seu resultado ao map de resultados.

```
--- FAIL: TestVerificaWebsites (0.00s)
        VerificaWebsites_test.go:31: esperado map[http://google.com:true http://blog.gypsyda
FAIL
exit status 1
FAIL    github.com/larien/learn-go-with-tests/concorrencia/v2        0.010s
```

Uma breve visita ao universo paralelo...

Você pode não ter obtido esse resultado. Você pode obter uma mesnagem de pânico, que vamos falar sobre em breve. Não se preocupe se isso aparecer para você, basta você executar o teste até você *de fato* receber o resultado acima. Ou faça de conta que você recebeu. Escolha sua. Boas vindas à concorrência:

quando não for trabalhada da forma correta, é difícil prever o que vai acontecer. Não se preocupe, é por isso que estamos escrevendo testes: para nos ajudar a saber quando estamos trabalhando com concorrência de forma previsível.

... e estamos de volta.

Acabou que os testes originais do `VerificadorWebsite` agora estão devolvendo um map vazio. O que deu de errado?

Nenhuma das goroutines que nosso loop `for` iniciou teve tempo de adicionar seu resultado ao map `resultados`; a função `VerificadorWebsite` é rápida demais para eles, e por isso retorna o map vazio.

Para consertar isso, podemos apenas esperar enquanto todas as goroutines fazem seu trabalho, para depois retornar. Dois segundos devem servir, certo?

```
package concurrency

import "time"

type VerificadorWebsite func(string) bool

func VerificaWebsites(vw VerificadorWebsite, urls []string) map[string]bool {
    resultados := make(map[string]bool)

    for _, url := range urls {
        go func() {
            resultados[url] = vw(url)
        }()
    }

    time.Sleep(2 * time.Second)

    return resultados
}
```

Agora, quando os testes forem executados, você vai ver (ou não - leia a mensagem no início do tópico):

```
--- FAIL: TestVerificaWebsites (0.00s)
        VerificaWebsites_test.go:31: esperado map[http://google.com:true http://blog.gypsyda
FAIL
exit status 1
FAIL    github.com/larien/learn-go-with-tests/concorrencia/v1        0.010s
```

Isso não é muito bom - por que só um resultado? Podemos arrumar isso aumentando o tempo de espera - pode tentar se preferir. Não vai funcionar. O problema aqui é que a variável `url` é reutilizada para cada iteração do laço `for` - ele recebe um valor novo de `urls` a cada vez. Mas cada uma das goroutines

tem uma referência para a variável `url` - eles não têm sua própria cópia independente. Logo, *todas* estão escrevendo o valor que `url` tem no final da iteração - o último URL. E é por isso que o resultado que obtemos é a última URL.

Para consertar isso:

```
package concorrencia

import (
    "time"
)

type VerificadorWebsite func(string) bool

func VerificaWebsites(vw VerificadorWebsite, urls []string) map[string]bool {
    resultados := make(map[string]bool)

    for _, url := range urls {
        go func(u string) {
            resultados[u] = vw(u)
        }(url)
    }

    time.Sleep(2 * time.Second)

    return resultados
}
```

Ao passar cada função anônima como parâmetro para a URL - como `u` - e chamar a função anônima com `url` como argumento, nos certificamos de que o valor de `u` está fixado como o valor de `url` para cada iteração do laço de `url` e não pode ser modificado.

Agora, se você tiver sorte, vai obter:

```
PASS
ok      github.com/larien/learn-go-with-tests/concorrencia/v1        2.012s
```

No entanto, se não tiver sorte (isso é mais provável se estiver rodando o código com o benchmark, já que haverá mais tentativas):

```
fatal error: concurrent map writes

goroutine 37 [running]:
runtime.throw(0x6d74f3, 0x15)
    /usr/local/go/src/runtime/panic.go:608 +0x72 fp=0xc000034718 sp=0xc0000346e8 pc=0x42d4e2
runtime.mapassign_faststr(0x67dbe0, 0xc000082660, 0x6d33cb, 0x7, 0x0)
    /usr/local/go/src/runtime/map_faststr.go:275 +0x3bf fp=0xc000034780 sp=0xc000034718 pc=0
github.com/larien/learn-go-with-tests/concorrencia/v2.VerificaWebsites.func1(0x6e6580, 0xc00
    /home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebsit
```

```
runtime.goexit()
    /usr/local/go/src/runtime/asm_amd64.s:1333 +0x1 fp=0xc0000347c8 sp=0xc0000347c0 pc=0x45c
created by github.com/larien/learn-go-with-tests/concorrencia/v2.VerificaWebsites
    /home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebsit

        ... e mais um monte de linhas assustadoras ...
```

Isso pode ser enorme e assustador, mas tudo o que precisamos fazer é respirar com calma e ler o stacktrace: `fatal error: concurrent map writes` (erro fatal: escrita concorrente no map). Às vezes, quando executamos nossos testes, duas das goroutines escrevem no map `resultados` ao mesmo tempo. Maps em Go não gostam quando mais de uma coisa tenta escrever algo neles ao mesmo tempo, então o `erro fatal` é gerado.

Essa é uma *condição de corrida*, um bug que aparece quando a saída do nosso software depende do timing e da sequência de eventos que não temos controle sobre. Por não termos controle exato sobre quando cada goroutine escreve no map `resultados`, ficamos vulneráveis à situação de duas goroutines escreverem nele ao mesmo tempo.

O Go nos ajuda a encontrar condições de corrida com seu *detector de corrida* nativo. Para habilitar essa funcionalidade, execute os testes com a flag `race`: `go test -race`.

Você deve ver uma saída parecida com essa:

```
==================
WARNING: DATA RACE
Write at 0x00c000120089 by goroutine 6:
  reflect.typedmemmove()
      /usr/local/go/src/runtime/mbarrier.go:177 +0x0
  reflect.Value.MapIndex()
      /usr/local/go/src/reflect/value.go:1124 +0x2ae
  reflect.deepValueEqual()
      /usr/local/go/src/reflect/deepequal.go:118 +0x13be
  reflect.DeepEqual()
      /usr/local/go/src/reflect/deepequal.go:196 +0x2f0
  github.com/larien/learn-go-with-tests/concorrencia/v2.TestVerificaWebsites()
      /home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebs
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:827 +0x162

Previous write at 0x00c000120089 by goroutine 8:
  github.com/larien/learn-go-with-tests/concorrencia/v2.VerificaWebsites.func1()
      /home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebs

Goroutine 6 (running) created at:
  testing.(*T).Run()
```

```
      /usr/local/go/src/testing/testing.go:878 +0x659
  testing.runTests.func1()
      /usr/local/go/src/testing/testing.go:1119 +0xa8
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:827 +0x162
  testing.runTests()
      /usr/local/go/src/testing/testing.go:1117 +0x4ee
  testing.(*M).Run()
      /usr/local/go/src/testing/testing.go:1034 +0x2ee
  main.main()
      _testmain.go:44 +0x221

Goroutine 8 (finished) created at:
  github.com/larien/learn-go-with-tests/concorrencia/v2.VerificaWebsites()
      /home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebs
  github.com/larien/learn-go-with-tests/concorrencia/v2.TestVerificaWebsites()
      /home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebs
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:827 +0x162
==================
```

Os detalhes ainda assim são bem difíceis de serem lidos - mas o `WARNING: DATA RACE` (CUIDADO: CONDIÇÃO DE CORRIDA) é bem claro. Lendo o corpo do erro podemos ver duas goroutines diferentes performando escritas em um map:

```
Write at 0x00c000120089 by goroutine 6:
```

está escrevendo no mesmo bloco de memória que:

```
Previous write at 0x00c000120089 by goroutine 8:
```

Além disso, conseguimos ver a linha de código onde a escrita está acontecendo:

```
/home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebsites.g
+0x97
```

e a linha de código onde as goroutines 6 e 7 foram iniciadas:

```
/home/larien/go/src/github.com/larien/learn-go-with-tests/concorrencia/v2/VerificaWebsites.g
+0xb2
```

Tudo o que você precisa saber está impresso no seu terminal - tudo o que você tem que fazer é ser paciente o bastante para lê-lo.

Canais

Podemos resolver essa condição de corrida coordenando nossas goroutines usando *canais*. Canais são uma estrutura de dados em Go que pode receber e enviar valores. Essas operações, junto de seus detalhes, permitem a comunicação entre processos diferentes.

Nesse caso, queremos pensar sobre a comunicação entre o processo pai e cada uma das goroutines criadas por ele de forma que façam o trabalho de executar a função `VerificadorWebsite` com a URL.

```
package concurrency

type VerificadorWebsite func(string) bool
type resultado struct {
    string
    bool
}

func VerificaWebsites(vw VerificadorWebsite, urls []string) map[string]bool {
    resultados := make(map[string]bool)
    canalResultado := make(chan resultado)

    for _, url := range urls {
        go func(u string) {
            canalResultado <- resultado{u, vw(u)}
        }(url)
    }

    for i := 0; i < len(urls); i++ {
        resultado := <-canalResultado
        resultados[resultado.string] = resultado.bool
    }

    return resultados
}
```

Junto do map `resultados`, agora temos um `canalResultados`, que criamos da mesma forma usando `make`. O `chan resultado` é o tipo do canal - um canal de `resultado`. O tipo novo, `resultado`, foi criado para associar o retorno de `VerificadorWebsite` com a URL sendo verificada - é uma estrutura que contém uma `string` e um `bool`. Já que não precisamos que nenhum valor tenha um nome, cada um deles é anônimo dentro da struct; isso pode ser útil quando for difícil saber que nome dar a um valor.

Agora que iteramos pelas URLs, ao invés de escrever no `map` diretamente, enviamos uma struct `resultado` para cada chamada de `vw` para o `canalResultado` com uma *sintaxe de envio*. Essa sintaxe usa o operador `<-`, usando um canal à esquerda e um valor à direita:

```
// Sintaxe de envio
canalResultado <- resultado{u, vw(u)}
```

O próximo laço `for` itera uma vez sobre cada uma das URLs. Dentro, estamos usando uma *expressão de recebimento*, que atribui um valor recebido por um

canal a uma variável. Essa expressão também usa o operador `<-`, mas com os dois operandos ao posições invertidas: o canal agora fica à direita e a variável que está recebendo o valor dele fica à esquerda:

```
// Expressão recebida
resultado := <-canalResultado
```

E depois usamos o `resultado` recebido para atualizar o map.

Ao enviar os resultados para um canal, podemos controlar o timing de cada escrita dentro do map `resultados`, garantindo que só aconteça uma por vez. Apesar de cada uma das chamadas de `vw` e cada envio ao canal resultado estar acontecendo em paralelo dentro de seu próprio processo, cada resultado está sendo resolvido de cada vez enquanto tiramos o valor do canal resultado com a expressão recebida.

Paralelizamos um pedaço do código que queríamos tornar mais rápida, enquanto mantivemos a parte que não pode acontecer em paralelo ainda acontecendo linearmente. E comunicamos diversos processos envolvidos utilizando canais.

Agora podemos executar o benchmark:

```
pkg: github.com/larien/learn-go-with-tests/concorrencia/v3
BenchmarkVerificaWebsites-8              100          23406615 ns/op
PASS
ok      github.com/larien/learn-go-with-tests/concorrencia/v3        2.377s
```

23406615 nanossegundos - 0.023 segundos, cerca de 100 vezes mais rápida que a função original. Um sucesso enorme.

Resumo

Esse exercício foi um pouco mais leve na parte do TDD que o restante. Levamos um bom tempo refatorando a função `VerificaWebsites`; as entradas e saídas não mudaram, ela apenas ficou mais rápida. Mas, com os testes que já tinhamos escrito, assim como com o benchmark que escrevemos, fomos capazes de refatorar o `VerificaWebsites` de forma que mantivéssemos a confiança de que o software ainda estava funcionando, enquanto demonstramos que ela realmente havia ficado mais rápida.

Tornando as coisas mais rápidas, aprendemos sobre:

- *goroutines*, a unidade básica de concorrência em Go, que nos permite verificar mais do que um site ao mesmo tempo.
- *funções anônimas*, que usamos para iniciar cada um dos processos concorrentes que verificam os sites.
- *canais*, para nos ajudar a organizar e controlar a comunicação entre diferentes processos, nos permitindo evitar um bug de *condição de corrida*.

- *o detector de corrida*, que nos ajudou a desvendar problemas com código concorrente.

Torne-o rápido

Uma formulação da forma ágil de desenvolver software, erroneamente atribuida a Kent Beck, é:

> Faça funcionar, faça da forma certa, torne-o rápido (em inglês)

Onde 'funcionar' é fazer os testes passarem, 'forma certa' é refatorar o código e 'tornar rápido' é otimizar o código para, por exemplo, tornar sua execução rápida. Só podemos 'torná-lo rápido' quando fizermos funcionar da forma certa. Tivemos sorte que o código que estudamos já estava funcionando e não precisava ser refatorado. Nunca devemos tentar 'torná-lo rápido' antes das outras duas etapas terem sido feitas, porque:

> Otimização prematura é a raiz de todo o mal – Donald Knuth

Select

Você pode encontrar todos os códigos desse capítulo aqui

Te pediram para fazer uma função chamada `Corredor` que recebe duas URLs que "competirão" entre si através de uma chamada HTTP GET onde a primeira URL a responder será retornada. Se nenhuma delas responder dentro de 10 segundos a função deve retornar um `erro`.

Para isso, vamos utilizar:

- `net/http` para chamadas HTTP.
- `net/http/httptest` para nos ajudar a testar.
- goroutines.
- `select` para sincronizar processos.

Escreva o teste primeiro

Vamos começar com algo simples.

```
func TestCorredor(t *testing.T) {
    URLLenta := "http://www.facebook.com"
    URLRapida := "http://www.quii.co.uk"

    esperado := URLRapida
    resultado := Corredor(URLLenta, urlRapida)

    if resultado != esperado {
```

```
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}
```

Sabemos que não está perfeito e que existem problemas, mas é um bom início. É importante não perder tanto tempo deixando as coisas perfeitas de primeira.

Execute o teste

```
./corredor_test.go:14:9: undefined: Corredor
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

```
func Corredor(a, b string) (vencedor string) {
    return
}
```

```
corredor_test.go:25: resultado '', esperado 'http://www.quii.co.uk'
```

Escreva código suficiente para que o teste passe

```
func Corredor(a, b string) (vencedor string) {
    inicioA := time.Now()
    http.Get(a)
    duracaoA := time.Since(inicioA)

    inicioB := time.Now()
    http.Get(b)
    duracaoB := time.Since(inicioB)

    if duracaoA < duracaoB {
        return a
    }

    return b
}
```

Para cada URL:

1. Usamos `time.Now()` para marcar o tempo antes de tentarmos pegar a `URL`.
2. Então usamos `http.Get` para tentar capturar os conteúdos da `URL`. Essa função retorna `http.Response` e um `erro`, mas não temos interesse nesses valores.

3. `time.Since` pega o tempo inicial e retorna a diferença na forma de `time.Duration`.

Feito isso, podemos simplesmente comparar as durações e ver qual é mais rápida.

Problemas

Isso pode ou não fazer com que o teste passe para você. O problema é que estamos acessando sites reais para testar nossa lógica.

Testar códigos que usam HTTP é tão comum que Go tem ferramentas na biblioteca padrão para te ajudar a testá-los.

Nos capítulos de mock e injeção de dependências, falamos sobre como idealmente não queremos depender de serviços externos para testar nosso código, pois:

- Podem ser lentos
- Podem ser inconsistentes
- Não conseguimos testar casos extremos

Na biblioteca padrão, existe um pacote chamado `net/http/httptest` onde é possível simular um servidor HTTP facilmente.

Vamos alterar nosso teste para usar essas simulações para termos servidores confiáveis para testar sob nosso controle.

```
func TestCorredor(t *testing.T) {

    servidorLento := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http
        time.Sleep(20 * time.Millisecond)
        w.WriteHeader(http.StatusOK)
    }))

    servidorRapido := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *htt
        w.WriteHeader(http.StatusOK)
    }))

    URLLenta := servidorLento.URL
    URLRapida := servidorRapido.URL

    esperado := URLRapida
    resultado := Corredor(URLLenta, URLRapida)

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }

    servidorLento.Close()
```

```
    servidorRapido.Close()
}
```

A sintaxe pode parecer um pouco complicada, mas não tenha pressa.

`httptest.NewServer` recebe um `http.HandlerFunc` que vamos enviar para uma função *anônima*.

`http.HandlerFunc` é um tipo que se parece com isso: `type HandlerFunc func(ResponseWriter, *Requisicao)`.

Tudo o que assinatura diz é que ela precisa de uma função que recebe um `ResponseWriter` e uma `Requisição`, o que não é novidade para um servidor HTTP.

Acontece que não existe nenhuma mágica aqui, **também é assim que você escreveria um servidor HTTP real em Go**. A única diferença é que estamos utilizando ele dentro de um `httptest.NewServer` ,o que facilita seu uso em testes por ele encontrar uma porta aberta para escutar e você poder fechá-la quando estiverem concluídos dentro dos próprios testes.

Dentro de nossos dois servidores, fazemos com que um deles tenha um `time.Sleep` quando receber a requisição para torná-lo propositalmente mais lento que o outro. Ambos os servidores, então, devolvem uma resposta `OK` com `w.WriteHeader(http.StatusOK)` a quem realizou a chamada.

Se você rodar o teste novamente, ele definitivamente irá passar e deve ser mais rápido. Brinque com os **sleeps** para quebrar o teste propositalmente.

Refatoração

Temos algumas duplicações tanto em nosso código de produção quanto em nosso código de teste.

```
func Corredor(a, b string) (vencedor string) {
    duracaoA := medirTempoDeResposta(a)
    duracaoB := medirTempoDeResposta(b)

    if duracaoA < duracaoB {
        return a
    }

    return b
}

func medirTempoDeResposta(URL string) time.Duration {
    inicio := time.Now()
    http.Get(URL)
    return time.Since(inicio)
}
```

Essa "enxugada" torna nosso código `Corredor` bem mais legível.

```
func TestCorredor(t *testing.T) {

    servidorLento := criarServidorComAtraso(20 * time.Millisecond)
    servidorRapido := criarServidorComAtraso(0 * time.Millisecond)

    defer servidorLento.Close()
    defer servidorRapido.Close()

    URLLenta := servidorLento.URL
    URLRapida := servidorRapido.URL

    esperado := URLRapida
    resultado := Corredor(URLLenta, URLRapida)

    if resultado != esperado {
        t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
    }
}

func criarServidorComAtraso(atraso time.Duration) *httptest.Server {
    return httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request)
        time.Sleep(atraso)
        w.WriteHeader(http.StatusOK)
    }))
}
```

Fizemos a refatoração criando nossos servidores falsos numa função chamada `criarServidorComAtraso` para remover alguns códigos desnecessários do nosso teste e reduzir repetições.

`defer`

Ao chamar uma função com o prefixo `defer`, ela será chamada *após o término da função que a contém.*

Às vezes você vai precisar liberar recursos, como fechar um arquivo ou, como no nosso caso, fechar um servidor para que esse não continue escutando a uma porta.

Utilizamos o `defer` quando queremos que a função seja executada no final de uma função, mas mantendo essa instrução próxima de onde o servidor foi criado para facilitar a vida das pessoas que forem ler o código futuramente.

Nossa refatoração é uma melhoria e uma solução razoável dados os recursos de Go que vimos até aqui, mas podemos deixar essa solução ainda mais simples.

Sincronizando processos

- Por que estamos testando a velocidade dos sites sequencialmente quando Go é ótimo com concorrência? Devemos conseguir verificar ambos ao mesmo tempo.
- Não nos preocupamos com o *tempo exato de resposta* das requisições, apenas queremos saber qual retorna primeiro.

Para fazer isso, vamos apresentar uma nova construção chamada `select` que nos ajudará a sincronizar os processos de forma mais fácil e clara.

```
func Corredor(a, b string) (vencedor string) {
    select {
    case <-ping(a):
        return a
    case <-ping(b):
        return b
    }
}

func ping(URL string) chan bool {
    ch := make(chan bool)
    go func() {
        http.Get(URL)
        ch <- true
    }()
    return ch
}
```

`ping`

Definimos a função `ping` que cria um `chan bool` e a retorna.

No nosso caso, não nos *importamos* com o tipo enviado no canal, *só queremos enviar um sinal* para dizer que terminamos, então booleanos já servem.

Dentro da mesma função, iniciamos a goroutine que enviará um sinal a esse canal uma vez que a função `http.Get(URL)` tenha sido finalizada.

`select`

Se você se lembrar do capítulo de concorrência, é possível esperar os valores serem enviados a um canal com `variavel := <-ch`. Isso é uma chamada *bloqueante*, pois está aguardando por um valor.

O que o `select` te permite fazer é aguardar *múltiplos* canais. O primeiro a enviar um valor "vence" e o código abaixo do `case` é executado.

Nós usamos `ping` em nosso `select` para configurar um canal para cada uma de nossas `URL`s. Qualquer um que enviar para esse canal primeiro vai ter seu código executado no `select`, que resultará nessa `URL` sendo retornada (que consequentemente será a vencedora).

Após essas mudanças, a intenção por trás de nosso código fica bem clara e sua implementação efetivamente mais simples.

Limites de tempo

Nosso último requisito era retornar um erro se o `Corredor` demorar mais que 10 segundos.

Escreva o teste primeiro

```
t.Run("retorna um erro se o servidor não responder dentro de 10s", func(t *testing.T) {
    servidorA := criarServidorComAtraso(11 * time.Second)
    servidorB := criarServidorComAtraso(12 * time.Second)

    defer servidorA.Close()
    defer servidorB.Close()

    _, err := Corredor(servidorA.URL, servidorB.URL)

    if err == nil {
        t.Error("esperava um erro, mas não obtive um")
    }
})
```

Fizemos nossos servidores de teste demorarem mais que 10s para retornar para exercitar esse cenário e agora estamos esperando que `Corredor` retorne dois valores: a URL vencedora (que ignoramos nesse teste com `_`) e um `erro`.

Execute o teste

```
./corredor_test.go:37:10: assignment mismatch: 2 variables but 1
values
```

Escreva a menor quantidade de código para rodar o teste e verifique a saída do teste que falhou

```
func Corredor(a, b string) (vencedor string, erro error) {
    select {
    case <-ping(a):
```

```
        return a, nil
    case <-ping(b):
        return b, nil
    }
}
```

Alteramos a assinatura de `Corredor` para retornar o vencedor e um `erro`. Retornamos `nil` para nossos casos de sucesso.

O compilador vai reclamar sobre seu *primeiro teste* esperar apenas um valor, então altere essa linha para `obteve, _ := Corredor(urlLenta, urlRapida)`. Sabendo disso devemos verificar se *não* obteremos um erro em nosso caso de sucesso.

Se executar isso agora, o teste irá falhar após 11 segundos.

```
--- FAIL: TestCorredor (12.00s)
   --- FAIL: TestCorredor/retorna_um_erro_se_o_teste_não_responder_dentro_de_10s (12.00s)
        corredor_test.go:40: esperava um erro, mas não obtive um.
```

Escreva código o suficiente para fazer o teste passar

```
func Corredor(a, b string) (vencedor string, erro error) {
    select {
    case <-ping(a):
        return a, nil
    case <-ping(b):
        return b, nil
    case <-time.After(10 * time.Second):
        return "", fmt.Errorf("tempo limite de espera excedido para %s e %s", a, b)
    }
}
```

`time.After` é uma função muito útil quando usamos `select`. Embora não ocorra em nosso caso, você pode escrever um código que bloqueia para sempre se os canais que o `select` estiver ouvindo nunca retornarem um valor. `time.After` retorna um `chan` (como `ping`) e te enviará um sinal após a quantidade de tempo definida.

Para nós isso é perfeito; se `a` ou `b` conseguir retornar teremos um vencedor, mas se chegar a 10 segundos nosso `time.After` nos enviará um sinal e retornaremos um `erro`.

Testes lentos

O problema que temos é que esse teste demora 10 segundos para rodar. Para uma lógica tão simples, isso não parece ótimo.

O que podemos fazer é deixar esse esgotamento de tempo configurável. Então, em nosso teste, podemos ter um tempo bem curto e, quando utilizado no mundo real, esse tempo ser definido para 10 segundos.

```
func Corredor(a, b string, tempoLimite time.Duration) (vencedor string, erro error) {
    select {
    case <-ping(a):
        return a, nil
    case <-ping(b):
        return b, nil
    case <-time.After(tempoLimite):
        return "", fmt.Errorf("tempo limite de espera excedido para %s e %s", a, b)
    }
}
```

Nosso teste não irá compilar pois não fornecemos um tempo de expiração.

Antes de nos apressar para adicionar esse valor padrão a ambos os testes, vamos *ouvi-los*.

- Nos importamos com o tempo excedido em nosso caso de teste de sucesso?
- Os requisitos foram explícitos sobre o tempo limite?

Dado esse conhecimento, vamos fazer uma pequena refatoração para ser simpático aos nossos testes e aos usuários de nosso código.

```
var limiteDeDezSegundos = 10 * time.Second

func Corredor(a, b string) (vencedor string, erro error) {
    return Configuravel(a, b, limiteDeDezSegundos)
}

func Configuravel(a, b string, tempoLimite time.Duration) (vencedor string, erro error) {
    select {
    case <-ping(a):
        return a, nil
    case <-ping(b):
        return b, nil
    case <-time.After(tempoLimite):
        return "", fmt.Errorf("tempo limite de espera excedido para %s e %s", a, b)
    }
}
```

Nossos usuários e nosso primeiro teste podem utilizar `Corredor` (que usa `Configuravel` por baixo dos panos) e nosso caminho triste pode usar `Configuravel`.

```
func TestCorredor(t *testing.T) {
    t.Run("compara a velocidade de servidores, retornando o endereço do mais rápido", func(t
        servidorLento := criarServidorComAtraso(20 * time.Millisecond)
```

```
        servidorRapido := criarServidorComAtraso(0 * time.Millisecond)

        defer servidorLento.Close()
        defer servidorRapido.Close()

        URLLenta := servidorLento.URL
        URLRapida := servidorRapido.URL

        esperado := URLRapida
        resultado, err := Corredor(URLLenta, URLRapida)

        if err != nil {
            t.Fatalf("não esperava um erro, mas obteve um %v", err)
        }

        if resultado != esperado {
            t.Errorf("resultado '%s', esperado '%s'", resultado, esperado)
        }
    })

    t.Run("retorna um erro se o servidor não responder dentro de 10s", func(t *testing.T) {
        servidor := criarServidorComAtraso(25 * time.Millisecond)

        defer servidor.Close()

        _, err := Configuravel(servidor.URL, servidor.URL, 20*time.Millisecond)

        if err == nil {
            t.Error("esperava um erro, mas não obtive um")
        }
    })
}
```

Adicionei uma verificação final ao primeiro teste para saber se não pegamos um `erro`.

Resumo

`select`

- Ajuda você a escutar vários canais.
- Às vezes você pode precisar incluir `time.After` em um de seus `cases` para prevenir que seu sistema fique bloqueado para sempre.

`httptest`

- Uma forma conveniente de criar servidores de teste para que se tenha testes confiáveis e controláveis.
- Usa as mesmas interfaces que servidores `net/http` reais, o que torna seu sistema consistente e gera menos coisas para você aprender.

Reflection

Você pode encontrar todo o código para esse capítulo aqui

Do Twitter

> Desafio Golang: escreva uma função `percorre(x interface{}, fn func(string))` que recebe uma struct `x` e chama `fn` para todos os campos string encontrados dentro dela. nível de dificuldade: recursão.

Para fazer isso vamos precisar usar `reflection` (reflexão).

> A reflexão em computação é a habilidade de um programa examinar sua própria estrutura, particularmente através de tipos; é uma forma de metaprogramação. Também é uma ótima fonte de confusão.

De The Go Blog: Reflection

O que é `interface`?

Aproveitamos a segurança de tipos que o Go nos ofereceu em termos de funções que funcionam com tipos conhecidos, como `string`, `int` e nossos próprios tipos como `ContaBancaria`.

Isso significa que de praxe temos documentação e o compilador vai reclamar se você tentar passar o tipo errado para uma função.

Só que você pode se deparar com situações em que quer escrever uma função, mas não sabe o tipo da variável em tempo de compilação.

Go nos permite contornar isso com o tipo `interface{}`, que você pode relacionar com *qualquer* tipo.

Logo, `percorre(x interface{}, fn func(string))` aceitará qualquer valor para `x`.

Então por que não usar `interface` para tudo e ter funções bem flexíveis?

- Quando utiliza uma função que usa `interface`, você perde a segurança de tipos. E se você quisesse passar `Foo.bar` do tipo `string` para uma função, mas ao invés disso passa `Foo.baz` do tipo `int`? O compilador não vai ser capaz de informar seu erro. Você também não tem ideia *do que* pode passar para uma função. Saber que uma função recebe um `ServicoDeUsuario`, por exemplo, é muito útil.

Resumindo, só use reflexão quando realmente precisar.

Se quiser funções polimórficas, considere desenvolvê-la em torno de uma interface (não `interface{}`, só para esclarecer) para que os usuários possam usar sua função com vários tipos se implementarem os métodos que você precisar para a sua função funcionar.

Nossa função vai precisar ser capaz de trabalhar com várias coisas diferentes. Como sempre, vamos usar uma abordagem iterativa, escrevendo testes para cada coisa nova que quisermos dar suporte e refatorando ao longo do caminho até finalizarmos.

Escreva o teste primeiro

Vamos chamar nossa função com uma estrutura que tem um campo string dentro (`x`). Depois, podemos espiar a função (`fn`) passada para ela para ver se ela foi chamada.

```
func TestPercorre(t *testing.T) {

    esperado := "Chris"
    var resultado []string

    x := struct {
        Nome string
    }{esperado}

    percorre(x, func(entrada string) {
        resultado = append(resultado, entrada)
    })

    if len(resultado) != 1 {
        t.Errorf("número incorreto de chamadas de função: resultado %d, esperado %d", len(re
    }
}
```

- Queremos armazenar um slice de strings (`resultado`) que armazena quais strings foram passadas dentro de `fn` pelo `percorre`. Algumas vezes,

nos capítulos anteriores, criamos tipos dedicados para isso para espionar chamadas de função/método, mas nesse caso vamos apenas passá-lo em uma função anônima para `fn` que acaba em `resultado`.
- Usamos uma `struct` anônima com um campo `Nome` do tipo string para partir para caminho "feliz" e mais simples.
- Finalmente, chamamos `percorre` com `x` e o espião e por enquanto só verificamos o tamanho de `resultado`. Teremos mais precisão nas nossas verificações quando tivermos algo bem básico funcionando.

Tente executar o teste

```
./reflection_test.go:21:2: undefined: percorre
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Precisamos definir `percorre`.

```
func percorre(x interface{}, fn func(entrada string)) {

}
```

Execute o teste novamente:

```
=== RUN   TestPercorre
--- FAIL: TestPercorre (0.00s)
   reflection_test.go:19: número incorreto de chamadas de função: resultado 0, esperado 1
FAIL
```

Escreva código o suficiente para fazer o teste passar

Agora podemos chamar o espião com qualquer string para fazer o teste passar.

```
func percorre(x interface{}, fn func(entrada string)) {
    fn("Ainda não acredito que o Brasil perdeu de 7 a 1")
}
```

Agora o teste deve estar passando. A próxima coisa que vamos precisar fazer é criar uma verificação mais específica do que está sendo chamado dentro do nosso `fn`.

Escreva o teste primeiro

Adicione o código a seguir para o teste existente para verificar se a string passada para `fn` está correta:

```
if resultado[0] != esperado {
    t.Errorf("resultado '%s', esperado '%s'", resultado[0], esperado)
}
```

Execute o teste

```
=== RUN   TestPercorre
--- FAIL: TestPercorre (0.00s)
   reflection_test.go:23: resultado 'Ainda não acredito que o Brasil perdeu de 7 a 1', esperado '
FAIL
```

Escreva código o suficiente para fazer o teste passar

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := reflect.ValueOf(x) // ValorDe
    campo := valor.Field(0)     // Campo
    fn(campo.String())
}
```

Esse código está *pouco seguro e muito frágil*, mas lembre-se que nosso objetivo quando estamos no "vermelho" (os testes estão falhando) é escrever a menor quantidade de código possível. Depois escrevemos mais testes para resolver nossas lacunas.

Precisamos usar o reflection para verificar as propriedades de `x`.

No pacote reflect existe uma função chamada `ValueOf` que retorna um `Value` (valor) de determinada variável. Isso nos permite inspecionar um valor, inclusive seus campos usados nas próximas linhas.

Então podemos presumir coisas bem otimistas sobre o valor passado:

- Podemos procurar pelo primeiro e único campo, mas pode não haver nenhum campo, o que causaria um pânico.
- Depois podemos chamar `String()` que tetorna o valor subjacente como string, mas sabemos que vai dar errado se o campo for de algum tipo que não uma string.

Refatoração

Nosso código está passando pelo caso simples, mas sabemos que nosso código tem várias falhas.

Vamos escrever alguns testes onde passamos valores diferentes e verificaremos o array de strings com que `fn` foi chamado.

Precisamos refatorar nosso teste em um teste orientado por tabelas para tornar esse processo mais fácil para continuarmos testando novas situações.

```
func TestPercorre(t *testing.T) {

    casos := []struct {
        Nome              string
        Entrada           interface{}
        ChamadasEsperadas []string
    }{
        {
            "Struct com um campo string",
            struct {
                Nome string
            }{"Chris"},
            []string{"Chris"},
        },
    }

    for _, teste := range casos {
        t.Run(teste.Nome, func(t *testing.T) {
            var resultado []string
            percorre(teste.Entrada, func(entrada string) {
                resultado = append(resultado, entrada)
            })

            if !reflect.DeepEqual(resultado, teste.ChamadasEsperadas) {
                t.Errorf("resultado %v, esperado %v", resultado, teste.ChamadasEsperadas)
            }
        })
    }
}
```

Agora podemos adicionar uma situação facilmente para ver o que acontece se tivermos mais de um campo string.

Escreva o teste primeiro

Adicione o cenário a seguir nos `casos`.

```
{
    "Struct com dois campos tipo string",
    struct {
        Nome   string
        Cidade string
    }{"Chris", "Londres"},
    []string{"Chris", "Londres"},
}
```

Execute o teste

```
=== RUN   TestPercorre/Struct_com_dois_campos_string
    --- FAIL: TestPercorre/Struct_com_dois_campos_string (0.00s)
     reflection_test.go:40: resultado [Chris], esperado [Chris Londres]
```

Escreva código o suficiente para fazer o teste passar

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := reflect.ValueOf(x)

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)
        fn(campo.String())
    }
}
```

`valor` tem um método chamado `NumField` que retorna a quantidade de campos no valor. Isso nos permite iterar sobre os campos e chamar `fn`, o que faz nosso teste passar.

Refatoração

Não parece haver nenhuma refatoração óbvia aqui que pode melhorar nosso código, então vamos continuar.

A próxima falha em `percorre` é que ela presume que todo campo é uma `string`. Vamos escrever um teste para esse caso.

Escreva o teste primeiro

Inclua o seguinte cenário:

```
{
    "Struct sem campo tipo string",
    struct {
        Nome  string
        Idade int
    }{"Chris", 33},
    []string{"Chris"},
}
```

Execute o teste

```
=== RUN   TestPercorre/Struct_sem_campo_tipo_string
```

```
--- FAIL: TestPercorre/Struct_with_noStruct_sem_campo_tipo_stringn_string_field (0.00s)
    reflection_test.go:46: resutado [Chris <int Value>], esperado [Chris]
```

Escreva código o suficiente para fazer o teste passar

Precisamos verificar que o tipo do campo é uma `string`.

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := reflect.ValueOf(x)

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)

        if campo.Kind() == reflect.String { // Tipo
            fn(campo.String())
        }
    }
}
```

Podemos verificar seu tipo chamando a função `Kind`.

Refatoração

Parece que o código ainda está razoável por enquanto.

O próximo caso é: e se o valor não for uma `struct` "única"? Em outras palavras, o que acontece se tivermos uma `struct` com alguns campos aninhados?

Escreva o teste primeiro

Estivemos usando a sintaxe de estrutura anônima para declarar tipos conforme precisávamos para nossos testes, então poderíamos continuar a fazer isso, como:

```
{
    "Campos aninhados",
    struct {
        Nome string
        Perfil struct {
            Idade  int
            Cidade string
        }
    }{"Chris", struct {
        Idade  int
        Cidade string
    }{33, "Londres"}},
```

```
    []string{"Chris", "Londres"},
},
```

Mas podemos ver que quando você usa estruturas anônimas cada vez mais aninhadas, a sintaxe fica um pouco bagunçada. Há uma proposta para fazer isso de forma que a sintaxe seja mais agradável.

Vamos apenas refatorar isso criando um tipo conhecido para esse caso e referenciá-lo no nosso teste. Não é aconselhável colocar código do teste fora do teste, mas as pessoas devem ser capazes de encontrar essas estruturas procurando por sua definição.

Inclua as seguintes declarações de tipos no seu arquivo de teste:

```
type Pessoa struct {
    Nome   string
    Perfil Perfil
}

type Perfil struct {
    Idade  int
    Cidade string
}
```

Agora podemos adicionar isso aos nossos casos ficarem bem mais legíveis que antes:

```
{
    "Campos aninhados",
    Pessoa{
        "Chris",
        Perfil{33, "Londres"},
    },
    []string{"Chris", "Londres"},
}
```

Execute o teste

```
=== RUN   TestPercorre/Campps_aninhados
    --- FAIL: TestPercorre/Campps_aninhados (0.00s)
     reflection_test.go:54: resultado [Chris], esperado [Chris Londres]
```

O problema é que estamos apenas iterando sobre os campos no primeiro nível da hierarquia de tipos.

Escreva código o suficiente para fazer o teste passar

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := reflect.ValueOf(x)

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)

        if campo.Kind() == reflect.String {
            fn(campo.String())
        }

        if campo.Kind() == reflect.Struct {
            percorre(campo.Interface(), fn)
        }
    }
}
```

A solução é bem simples. Inspecionamos seu tipo novamente e se for uma estrutura apenas chamamos `percorre` novamente na nossa estrutura de dentro.

Refatoração

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := reflect.ValueOf(x)

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)

        switch campo.Kind() {
        case reflect.String:
            fn(campo.String())
        case reflect.Struct:
            percorre(campo.Interface(), fn)
        }
    }
}
```

Quando você está fazendo uma comparação de mesmo valor mais de uma vez, *geralmente* refatorar as condições dentro de um `switch` vai melhorar a legibilidade e tornar seu código mais fácil de estender.

E se o valor passado na estrutura for um ponteiro?

Escreva o teste primeiro

Inclua esse caso:

```
{
    "Ponteiros para coisas",
    &Pessoa{
        "Chris",
        Perfil{33, "Londres"},
    },
    []string{"Chris", "Londres"},
}
```

Execute o teste

```
=== RUN   TestPercorre/Ponteiros_para_coisas
panic: reflect: call of reflect.Value.NumField on ptr Value [recovered]
    panic: reflect: call of reflect.Value.NumField on ptr Value
```

Escreva código o suficiente para fazer o teste passar

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := reflect.ValueOf(x)

    if valor.Kind() == reflect.Ptr {
        valor = valor.Elem()
    }

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)

        switch campo.Kind() {
        case reflect.String:
            fn(campo.String())
        case reflect.Struct:
            percorre(campo.Interface(), fn)
        }
    }
}
```

Não é possível usar o `NumField` em um ponteiro `Value` e precisamos extrair o valor antes disso usando `Elem()`.

Refatoração

Vamos encapsular a responsabilidade de extrair o `reflect.Value` de determinada `interface{}` para uma função.

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)

        switch campo.Kind() {
        case reflect.String:
            fn(campo.String())
        case reflect.Struct:
            percorre(campo.Interface(), fn)
        }
    }
}

func obtemValor(x interface{}) reflect.Value {
    valor := reflect.ValueOf(x)

    if valor.Kind() == reflect.Ptr {
        valor = valor.Elem()
    }

    return valor
}
```

Isso acaba adicionando *mais* código, mas me parece que o nível de abstração está correto.

- Obter o `reflect.Value` de `x` para que eu possa inspecioná-lo, não me importa de qual forma.
- Iterar pelos campos, fazendo o que for necessário dependendo de seu tipo.

Depois precisamos lidar com os slices.

Escreva o teste primeiro

```
{
    "Slices",
    []Perfil{
        {33, "Londres"},
        {34, "Reykjavík"},
    },
```

```
    []string{"Londres", "Reykjavík"},
}
```

Execute o teste

```
=== RUN   TestPercorre/Slices
panic: reflect: call of reflect.Value.NumField on slice Value [recovered]
    panic: reflect: call of reflect.Value.NumField on slice Value
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Esse caso se parece bastante com o do ponteiro acima, pois estamos chamar `NumField` em nosso `reflect.Value`, mas não há um por não ser uma struct.

Escreva código o suficiente para fazer o teste passar

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    if valor.Kind() == reflect.Slice {
        for i:=0; i< valor.Len(); i++ {
            percorre(valor.Index(i).Interface(), fn)
        }
        return
    }

    for i := 0; i < valor.NumField(); i++ {
        campo := valor.Field(i)

        switch campo.Kind() {
        case reflect.String:
            fn(campo.String())
        case reflect.Struct:
            percorre(campo.Interface(), fn)
        }
    }
}
```

Refatoração

Isso funciona, mas está bagunçado. Não se preocupe, pois temos cada pedaço de código coberto por testes e podemos brincar da forma que quisermos.

Se formos pensar um pouco abstradamente, queremos chamar `percorre` em:

- Cada campo de uma estrutura
- Cada *coisa* de um slice

No momento nosso código faz isso, mas não reflete muito bem. Precisamos ter uma verificação no início da função para certificar se é um slice (com um `return` para parar a execução do restante do código) e se não for, só vamos presumir que é uma estrutura.

Vamos retrabalhar o código para verificar o tipo *primeiro* para depois fazermos o que importa.

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    switch valor.Kind() {
    case reflect.Struct:
        for i:=0; i<valor.NumField(); i++ {
            percorre(valor.Field(i).Interface(), fn)
        }
    case reflect.Slice:
        for i:=0; i<valor.Len(); i++ {
            percorre(valor.Index(i).Interface(), fn)
        }
    case reflect.String:
        fn(valor.String())
    }
}
```

Parece muito melhor! Se for uma estrutura ou um slice, iteramos sobre seus valores chamando `percorre` para cada um. Por outro lado, se for um `reflect.String`, podemos apenas chamar `fn`.

Ainda assim me parece que poderia ficar melhor. Há repetição da operação de iterar sobre campos/valores e chamar `percorre` sendo que conceitualmente são a mesma coisa.

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    quantidadeDeValores := 0
    var obtemCampo func(int) reflect.Value

    switch valor.Kind() {
    case reflect.String:
        fn(valor.String())
    case reflect.Struct:
        quantidadeDeValores = valor.NumField()
        obtemCampo = valor.Field
```

```
    case reflect.Slice:
        quantidadeDeValores = valor.Len()
        obtemCampo = valor.Index
    }

    for i := 0; i < quantidadeDeValores; i++ {
        percorre(obtemCampo(i).Interface(), fn)
    }
}
```

Se o `valor` for um `reflect.String`, chamamos `fn` normalmente.

Se for outra coisa, nosso `switch` vai extrair duas coisas dependendo do tipo:

- Quantos campos existem
- Como extrair o `Value` (`Field` [campo] ou `Index` [índice])

Uma vez que determinamos esses pontos, podemos iterar pela `quantidadeDeValores` chamando `percorre` com o resultado da função `getField`.

A partir disso, lidar com arrays deve ser simples.

Escreva o teste primeiro

Inclua o caso:

```
{
    "Arrays",
    [2]Perfil{
        {33, "Londres"},
        {34, "Reykjavík"},
    },
    []string{"Londres", "Reykjavík"},
}
```

Execute o teste

```
=== RUN   TestPercorre/Arrays
    --- FAIL: TestPercorre/Arrays (0.00s)
     reflection_test.go:78: resultado [], esperado [Londres Reykjavík]
```

Escreva código o suficiente para fazer o teste passar

Podemos resolver o caso dos arrays da mesma forma que os slices, basta adicioná-los com uma vírgula:

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    quantidadeDeValores := 0
    var obtemCampo func(int) reflect.Value

    switch valor.Kind() {
    case reflect.String:
        fn(valor.String())
    case reflect.Struct:
        quantidadeDeValores = valor.NumField()
        obtemCampo = valor.Field
    case reflect.Slice, reflect.Array:
        quantidadeDeValores = valor.Len()
        obtemCampo = valor.Index
    }

    for i := 0; i < quantidadeDeValores; i++ {
        percorre(obtemCampo(i).Interface(), fn)
    }
}
```

O último tipo que queremos lidar é o `map`.

Escreva o teste primeiro

```
{
    "Maps",
    map[string]string{
        "Foo": "Bar",
        "Baz": "Boz",
    },
    []string{"Bar", "Boz"},
},
```

Execute o teste

```
=== RUN   TestPercorre/Maps
    --- FAIL: TestPercorre/Maps (0.00s)
        reflection_test.go:86: resultado [], esperado [Bar Boz]
```

Escreva código o suficiente para fazer o teste passar

Novamente, se pensar um pouco de forma abstrata, percebe-se que o `map` é bem parecido com a `struct`, mas as chaves são desconhecidas em tempo de compilação.

Again if you think a little abstractly you can see that `map` is very similar to `struct`, it's just the keys are unknown at compile time.

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    quantidadedeValores := 0
    var obtemCampo func(int) reflect.Value

    switch valor.Kind() {
    case reflect.String:
        fn(valor.String())
    case reflect.Struct:
        quantidadeDeValores = valor.NumField()
        obtemCampo = valor.Field
    case reflect.Slice, reflect.Array:
        quantidadeDeValores = valor.Len()
        obtemCampo = valor.Index
    case reflect.Map:
        for _, chave := range valor.MapKeys() {
            percorre(valor.MapIndex(chave).Interface(), fn)
        }
    }

    for i := 0; i< quantidadeDeValores; i++ {
        percorre(obtemCampo(i).Interface(), fn)
    }
}
```

No entanto, por design, não é possível obter os valores de um map por índice. Só é possível fazer isso pela *chave*, que, caramba, acaba com a nossa abstração.

Refatoração

Como se sente agora? Parecia que essa era uma boa abstração naquele momento, mas agora o código parece um pouco bagunçado.

Está tudo bem! Refatoração é uma jornada e às vezes vamos cometer erros. Um ponto importante do TDD é que ele nos dá a liberdade de testar esse tipo de coisa.

Graças aos testes implmentados a cada etapa, essa situação não é irreversível de forma alguma. Vamos apenas voltar a como estava antes da refatoração.

```
func percorre(x interface{}, fn func(entrada string)) {
    valor := obtemValor(x)

    percorreValor := func(valor reflect.Value) {
        percorre(valor.Interface(), fn)
    }

    switch valor.Kind() {
    case reflect.String:
        fn(valor.String())
    case reflect.Struct:
        for i := 0; i < valor.NumField(); i++ {
            percorreValor(valor.Field(i))
        }
    case reflect.Slice, reflect.Array:
        for i := 0; i < valor.Len(); i++ {
            percorreValor(valor.Index(i))
        }
    case reflect.Map:
        for _, chave := range valor.MapKeys() {
            percorreValor(valor.MapIndex(chave))
        }
    }
}
```

Apresentamos o `percorreValor`, que encapsula chamadas para `percorre` dentro do nosso `switch` para que só tenham que extrair os `reflect.Value` de `valor`.

Um último problema

Lembre que maps em Go não têm ordem garantida. Logo, às vezes os testes irão falhar porque verificamos as chamadas de `fn` em uma ordem específica.

Para arrumar isso, precisaremos mover nossa verificação com os maps para um novo teste onde não nos importamos com a ordem.

```
t.Run("com maps", func(t *testing.T) {
    mapA := map[string]string{
        "Foo": "Bar",
        "Baz": "Boz",
    }

    var resultado []string
    percorre(mapA, func(entrada string) {
```

```
        resultado = append(resultado, entrada)
    })

    verificaSeContem(t, resultado, "Bar")
    verificaSeContem(t, resultado, "Boz")
})
```

Essa é a definição de `verificaSeContem`:

```
func verificaSeContem(t *testing.T, palheiro []string, agulha string) {
    contem := false
    for _, x := range palheiro {
        if x == agulha {
            contem = true
        }
    }
    if !contem {
        t.Errorf("esperava-se que %+v contivesse '%s', mas não continha", palheiro, agulha)
    }
}
```

Resumo

- Apresentamos alguns dos conceitos do pacote `reflect`.
- Usamos recursão para percorrer estruturas de dados arbitrárias.
- Houve uma reflexão quanto a uma refatoração ruim, mas não há por que se preocupar muito com isso. Isso não deve ser um problema muito grande se trabalharmos com testes de forma iterativa.
- Esse capítulo só cobre um aspecto pequeno de reflexão. O blog do Go tem um artigo excelente cobrindo mais detalhes.
- Agora que você tem conhecimento sobre reflexão, faça o possível para evitá-lo.

Sync

Você pode encontrar todo o código para esse capítulo aqui

Queremos fazer um contador que é seguro para ser usado concorrentemente.

Vamos começar com um contador não seguro e verificar se seu comportamento funciona em um ambiente com apenas uma *thread*.

Em seguida, vamos testar sua falta de segurança com várias *goroutines* tentando usar o contador dentro dos testes e consertar essa falha.

Escreva o teste primeiro

Queremos que nossa API nos dê um método para incrementar o contador e depois recupere esse valor.

```
func TestContador(t *testing.T) {
    t.Run("incrementar o contador 3 vezes resulta no valor 3", func(t *testing.T) {
        contador := Contador{}
        contador.Incrementa()
        contador.Incrementa()
        contador.Incrementa()

        if contador.Valor() != 3 {
        t.Errorf("resultado %d, esperado %d", contador.Valor(), 3)
    }
    })
}
```

Tente rodar o teste

```
./sync_test.go:9:14: undefined: Contador
```

Escreva o mínimo de código possível para fazer o teste rodar e verifique a saída do teste que tiver falhado

Vamos definir `Contador`.

```
type Contador struct {

}
```

Tente rodar o teste de novo e ele falhará com o seguinte erro:

```
./sync_test.go:14:10: contador.Incrementa undefined (type Contador has no field or method Incre
./sync_test.go:18:13: contador.Valor undefined (type Contador has no field or method Valor)
```

Então, para finalmente fazer o teste rodar, podemos definir esses métodos:

```
func (c *Contador) Incrementa() {

}

func (c *Contador) Valor() int {
    return 0
}
```

Agora tudo deve rodar e falhar:

```
=== RUN   TestContador
=== RUN  TestContador/incrementar_o_contador_3_vezes_resulta_no_valor_3
--- FAIL: TestContador (0.00s)
   --- FAIL: TestContador/incrementar_o_contador_3_vezes_resulta_no_valor_3 (0.00s)
        sync_test.go:27: resultado 0, esperado 3
```

Escreva código o suficiente para fazer o teste passar

Isso deve ser simples para *experts* em Go como nós. Precisamos criar uma instância do tipo Contador e incrementá-lo com cada chamada de `Incrementa`.

```
type Contador struct {
    valor int
}

func (c *Contador) Incrementa() {
    c.valor++
}

func (c *Contador) Valor() int {
    return c.valor
}
```

Refatoração

Não há muito o que refatorar, mas já que iremos escrever mais testes em torno do `Contador`, vamos escrever uma pequena função de asserção `verificaContador` para que o teste fique um pouco mais legível.

```
t.Run("incrementar o contador 3 vezes resulta no valor 3", func(t *testing.T) {
    contador := Contador{}
    contador.Incrementa()
    contador.Incrementa()
    contador.Incrementa()

    verificaContador(t, contador, 3)
})

func verificaContador(t *testing.T, resultado Contador, esperado int) {
    t.Helper()
    if resultado.Valor() != esperado {
        t.Errorf("resultado %d, esperado %d", resultado.Valor(), esperado)
    }
}
```

Próximos passos

Isso foi muito fácil, mas agora temos um requerimento que é: o programa precisa ser seguro o suficiente para ser usado em um ambiente com acesso concorrente. Vamos precisar criar um teste para exercitar isso.

Escreva o teste primeiro

```
t.Run("roda concorrentemente em segurança", func(t *testing.T) {
    contagemEsperada := 1000
    contador := NovoContador()

    var wg sync.WaitGroup
    wg.Add(contagemEsperada)

    for i := 0; i < contagemEsperada; i++ {
        go func(w *sync.WaitGroup) {
            contador.Incrementa()
            w.Done()
        }(&wg)
    }
    wg.Wait()

    verificaContador(t, contador, contagemEsperada)
})
```

Isso vai iterar até a nossa `contagemEsperada` e disparar uma *goroutine* para chamar `contador.Incrementa()` a cada iteração.

Estamos usando `sync.WaitGroup`, que é uma maneira simples de sincronizar processos concorrentes.

> Um WaitGroup aguarda por uma coleção de *goroutines* terminar seu processamento. A *goroutine* principal faz a chamada para o `Add` definir o número de *goroutines* que serão esperadas. Então, cada uma das *goroutines* é executada e chama `Done` quando termina sua execução. Ao mesmo tempo, `Wait` pode ser usado para bloquear a execução até que todas as *goroutines* tenham terminado.

Ao esperar por `wg.Wait()` terminar sua execução antes de fazer nossas asserções, podemos ter certeza que todas as nossas *goroutines* tentaram chamar o `Incrementa` no `Contador`.

Tente rodar o teste

```
=== RUN   TestContador/roda_concorrentemente_em_seguranca
--- FAIL: TestContador (0.00s)
```

```
   --- FAIL: TestContador/roda_concorrentemente_em_seguranca (0.00s)
        sync_test.go:26: resultado 939, esperado 1000
FAIL
```

O teste *provavelmente* vai falhar com um número diferente, mas de qualquer forma demonstra que não roda corretamente quando várias *goroutines* tentam mudar o valor do contador ao mesmo tempo.

Escreva código o suficiente para fazer o teste passar

Uma solução simples é adicionar uma trava ao nosso `Contador`, um `Mutex`.

> Um Mutex é uma trava de exclusão mútua. O valor zero de um Mutex é um Mutex destravado.

```
type Contador struct {
    mu sync.Mutex
    valor int
}

func (c *Contador) Incrementa() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.valor++
}
```

Isso significa que qualquer *goroutine* chamando `Incrementa` vai receber a trava em `Contador` se for a primeira chamando essa função. Todas as outras *goroutines* vão ter que esperar por essa primeira execução até que ele esteja `Unlock`, ou destravado, antes de ganhar o acesso à instância de `Contador` alterada pela primeira chamada de função.

Agora, se você rodar o teste novamente, ele deve funcionar porque cada uma das *goroutines* tem que esperar até que seja sua vez antes de fazer alguma mudança.

Já vi outros exemplos em que o `sync.Mutex` está embutido dentro da struct.

Você pode ver exemplos como esse:

```
type Contador struct {
    sync.Mutex
    valor int
}
```

Há quem diga que isso torna o código um pouco mais elegante.

```
func (c *Contador) Incrementa() {
    c.Lock()
```

```
    defer c.Unlock()
    c.valor++
}
```

Isso *parece* legal, mas, apesar de programação ser uma área altamente subjetiva, isso é **feio e errado**.

Às vezes as pessoas esquecem que tipos embutidos significam que os métodos daquele tipo se tornam *parte da interface pública*; e você geralmente não quer isso. Não se esqueçam que devemos ter muito cuidado com as nossas APIs públicas. O momento que tornamos algo público é o momento que outros códigos podem acoplar-se a ele e queremos evitar acoplamentos desnecessários.

Expôr `Lock` e `Unlock` é, no seu melhor caso, muito confuso e, no seu pior caso, potencialmente perigoso para o seu software se quem chamar o seu tipo começar a chamar esses métodos diretamente.

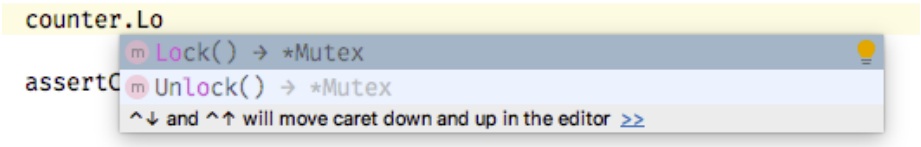

Figure 5: Demonstração de como um usuário dessa API pode chamar erroneamente o estado da trava

Isso parece uma péssima ideia.

Copiando mutexes

Nossos testes passam, mas nosso código ainda é um pouco perigoso.

Se você rodar `go vet` no seu código, deve receber um erro similar ao seguinte:

```
sync/v2/sync_test.go:16: call of verificaContador copies lock valor: v1.Contador contains sync.
sync/v2/sync_test.go:39: verificaContador passes lock by valor: v1.Contador contains sync.Mute
```

Uma rápida olhada na documentação do `sync.Mutex` nos diz o porquê:

> Um Mutex não deve ser copiado depois do primeiro uso.

Quando passamos nosso `Contador` (por valor) para `verificaContador`, ele vai tentar criar uma cópia do mutex.

Para resolver isso, devemos passar um ponteiro para o nosso `Contador`. Vamos, então, mudar a assinatura de `verificaContador`.

```
func verificaContador(t *testing.T, resultado *Contador, esperado int)
```

Nossos testes não vão mais compilar porque estamos tentando passar um `Contador` ao invés de um `*Contador`. Para resolver isso, é melhor criar um

construtor que mostra aos usuários da nossa API que seria melhor ele mesmo não inicializar seu tipo.

```
func NovoContador() *Contador {
    return &Contador{}
}
```

Use essa função em seus testes quando for inicializar o `Contador`.

Resumo

Falamos sobre algumas coisas do pacote sync:

- `Mutex` nos permite adicionar travas aos nossos dados
- `WaitGroup` é uma maneira de esperar as *goroutines* terminarem suas tarefas

Quando usar travas em vez de *channels* e *goroutines*?

Anteriormente falamos sobre *goroutines* no primeiro capítulo sobre concorrência que nos permite escrever código concorrente e seguro, então por que usar travas? A wiki do Go tem uma página dedicada para esse tópico: Mutex ou Channel?

> Um erro comum de um iniciante em Go é usar demais os *channels* e *goroutines* apenas porque é possível e/ou porque é divertido. Não tenha medo de usar um `sync.Mutex` se for uma solução melhor para o seu problema. Go é pragmático em deixar você escolher as ferramentas que melhor resolvem o seu problema e não te força em um único estilo de código.

Resumindo:

- **Use channels quando for passar a propriedade de um dado**
- **Use mutexes para gerenciar estados**

go vet

Não se esqueça de usar `go vet` nos seus scripts de *build* porque ele pode te alertar a respeito de bugs mais sutis no seu código antes que eles atinjam seus pobres usuários.

Não use códigos embutidos apenas porque é conveniente

- Pense a respeito do efeito que embutir códigos tem na sua API pública.
- Você *realmente* quer expôr esses métodos e ter pessoas acoplando o código próprio delas a ele?

- Mutexes podem se tornar um desastre de maneiras muito imprevisíveis e estranhas. Imagine um código inesperado destravando um mutex quando não deveria? Isso causaria erros muito estranhos que seriam muito difíceis de encontrar.

Contexto

Você pode encontrar todo o código para esse capítulo aqui

Softwares geralmente iniciam processos de longa duração e de uso intensivo de recursos (muitas vezes em goroutines). Se a ação que causou isso é cancelada ou falha por algum motivo, você precisa parar esses processos de forma consistente dentro da sua aplicação.

Se você não gerenciar isso, sua aplicação Go tão ágil da qual você tem tanto orgulho pode começar a ter problemas de desempenho difíceis de depurar.

Neste capítulo vamos usar o pacote `context` para nos ajudar a gerenciar processos de longa duração.

Vamos começar com um exemplo clássico de um servidor web que, quando iniciado, abre um processo de longa execução que vai buscar alguns dados para devolver em uma resposta.

Colocaremos em prática um cenário em que um usuário cancela a requisição antes que os dados possam ser recuperados e faremos com que o processo seja instruído a desistir.

Criei um código no caminho feliz para começarmos. Aqui está o código do nosso servidor.

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, store.Fetch())
    }
}
```

A função `Server` (servidor) recebe uma `Store` (armazenamento) e nos retorna um `http.HandlerFunc`. Store está definida como:

```
type Store interface {
    Fetch() string
}
```

A função retornada chama o método `Fetch` (busca) da `Store` para obter os dados e escrevê-los na resposta.

Nós temos um stub correspondente para `Store` que usamos em um teste.

```
type StubStore struct {
    response string
```

```
}

func (s *StubStore) Fetch() string {
    return s.response
}

func TestHandler(t *testing.T) {
    data := "olá, mundo"
    svr := Server(&StubStore{data})

    request := httptest.NewRequest(http.MethodGet, "/", nil)
    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if response.Body.String() != data {
        t.Errorf(`resultado "%s", esperado "%s"`, response.Body.String(), data)
    }
}
```

Agora que temos um caminho feliz, queremos fazer um cenário mais realista onde a `Store` não consiga finalizar o `Fetch` antes que o usuário cancele a requisição.

Escreva o teste primeiro

Nosso handler precisará de uma maneira de dizer à `Store` para cancelar o trabalho, então atualize a interface.

```
type Store interface {
    Fetch() string
    Cancel()
}
```

Precisaremos ajustar nosso spy para que leve algum tempo para retornar `data` e uma maneira de saber que foi dito para cancelar. Nós também o renomearemos para `SpyStore`, pois agora vamos observar a forma como ele é chamado. Ele terá que adicionar `Cancel` como um método para implementar a interface `Store`.

```
type SpyStore struct {
    response string
    cancelled bool
}

func (s *SpyStore) Fetch() string {
    time.Sleep(100 * time.Millisecond)
    return s.response
}
```

```
func (s *SpyStore) Cancel() {
    s.cancelled = true
}
```

Vamos adicionar um novo teste onde cancelamos a requisição antes de 100 milissegundos e verificamos a store para ver se ela é cancelada.

```
t.Run("avisa a store para cancelar o trabalho se a requisição for cancelada", func(t *testin
      store := &SpyStore{response: data}
      svr := Server(store)

      request := httptest.NewRequest(http.MethodGet, "/", nil)

      cancellingCtx, cancel := context.WithCancel(request.Context())
      time.AfterFunc(5 * time.Millisecond, cancel)
      request = request.WithContext(cancellingCtx)

      response := httptest.NewRecorder()

      svr.ServeHTTP(response, request)

      if !store.cancelled {
          t.Errorf("store não foi avisada para cancelar")
      }
  })
```

Do blog da Google novamente:

> O pacote context fornece funções para derivar novos valores de contexto dos já existentes. Estes valores formam uma árvore: quando um contexto é cancelado, todos os contextos derivados dele também são cancelados.

É importante que você derive seus contextos para que os cancelamentos sejam propagados através da pilha de chamadas (*call stack*) para uma determinada requisição.

O que fazemos é derivar um novo `cancellingCtx` da nossa requisição que nos retorna uma função `cancel`. Nós então programamos que a função seja chamada em 5 milissegundos usando `time.AfterFunc`. Por fim, usamos este novo contexto em nossa requisição chamando `request.WithContext`.

Execute o teste

O teste falha como seria de esperar.

```
--- FAIL: TestServer (0.00s)
    --- FAIL: TestServer/avisa_a_store_para_cancelar_o_trabalho_se_a_requisicao_for_cancelad
```

```
        context_test.go:62: store no foi avisada para cancelar
```

Escreva código o suficiente para fazer o teste passar

Lembre-se de ser disciplinado com o TDD. Escreva a quantidade *mínima* de código para fazer nosso teste passar.

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        store.Cancel()
        fmt.Fprint(w, store.Fetch())
    }
}
```

Isto faz com que este teste passe, mas não parece tão bom! Certamente não deveríamos estar cancelando a `Store` antes do fetch em *cada requisição*.

Ao ser disciplinado ele destacou uma falha em nossos testes, isso é uma coisa boa!

Vamos precisar atualizar nosso teste de caminho feliz para verificar que ele não será cancelado.

```
t.Run("retorna dados da store", func(t *testing.T) {
    store := SpyStore{response: data}
    svr := Server(&store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)
    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if response.Body.String() != data {
        t.Errorf(`resultado "%s", esperado "%s"`, response.Body.String(), data)
    }

    if store.cancelled {
        t.Error("não deveria ter cancelado a store")
    }
})
```

Execute ambos os testes. O teste do caminho feliz deve agora estar falhando e somos forçados a fazer uma implementação mais sensata.

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()

        data := make(chan string, 1)
```

```
        go func() {
            data <- store.Fetch()
        }()

        select {
        case d := <-data:
            fmt.Fprint(w, d)
        case <-ctx.Done():
            store.Cancel()
        }
    }
}
```

O que fizemos aqui?

`context` tem um método `Done()` que retorna um canal que recebe um sinal quando o context estiver "done" (finalizado) ou "cancelled" (cancelado). Queremos ouvir esse sinal e chamar `store.Cancel` se o obtivermos, mas queremos ignorá-lo se a nossa `Store` conseguir finalizar o `Fetch` antes dele.

Para gerenciar isto, executamos o `Fetch` em uma goroutine e ele irá escrever o resultado em um novo channel `data`. Nós então usamos `select` para efetivamente correr para os dois processos assíncronos e então escrevemos uma resposta ou cancelamos com `Cancel`.

Refatoração

Podemos refatorar um pouco o nosso código de teste fazendo métodos de verificação no nosso spy.

```
func (s *SpyStore) assertWasCancelled() {
    s.t.Helper()
    if !s.cancelled {
        s.t.Errorf("store não foi avisada para cancelar")
    }
}

func (s *SpyStore) assertWasNotCancelled() {
    s.t.Helper()
    if s.cancelled {
        s.t.Errorf("store foi avisada para cancelar")
    }
}
```

Lembre-se de passar o `*testing.T` ao criar o spy.

```
func TestServer(t *testing.T) {
    data := "olá, mundo"

    t.Run("retorna dados da store", func(t *testing.T) {
        store := &SpyStore{response: data, t: t}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)
        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        if response.Body.String() != data {
            t.Errorf(`recebi "%s", quero "%s"`, response.Body.String(), data)
        }

        store.assertWasNotCancelled()
    })

    t.Run("avisa a store para cancelar o trabalho se a requisição for cancelada", func(t *te
        store := &SpyStore{response: data, t: t}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)

        cancellingCtx, cancel := context.WithCancel(request.Context())
        time.AfterFunc(5*time.Millisecond, cancel)
        request = request.WithContext(cancellingCtx)

        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        store.assertWasCancelled()
    })
}
```

Esta abordagem é boa, mas é idiomática?

Faz sentido para o nosso servidor web estar preocupado com o cancelamento manual da `Store`? E se a `Store` também depender de outros processos de execução lenta? Nós teremos que ter certeza que a `Store.Cancel` propagará corretamente o cancelamento para todos os seus dependentes.

Um dos pontos principais do `context` é que é uma maneira consistente de oferecer cancelamento.

Da documentação do go

> As requisições de entrada para um servidor devem criar um Context e as chamadas de saída para servidores devem aceitar um Context. A cadeia de chamadas de função entre eles deve propagar o Context, substituindo-o opcionalmente por um Context derivado criado usando WithCancel, WithDeadline, WithTimeout ou WithValue. Quando um Context é cancelado, todos os Contexts derivados dele também são cancelados.

Do blog da Google novamente:

> Na Google, exigimos que os programadores Go passem um parâmetro Context como o primeiro argumento para cada função no caminho de chamada entre requisições de entrada e saída. Isto permite que o código Go desenvolvido por muitas equipes diferentes interopere bem. Ele fornece um controle simples sobre timeouts e cancelamentos e garante que valores críticos, como credenciais de segurança, transitem corretamente pelos programas Go.

(Pare por um momento e pense nas ramificações de cada função tendo que enviar um context e a ergonomia disso.)

Se sentindo um pouco desconfortável? Bom. Vamos tentar seguir essa abordagem e, em vez disso, passar o `context` para nossa `Store` e deixá-la ser responsável. Dessa maneira, ela também pode passar o `context` para os seus dependentes e eles também podem ser responsáveis por se pararem.

Escreva o teste primeiro

Teremos de alterar os nossos testes existentes, uma vez que as suas responsabilidades estão mudando. As únicas coisas que nosso handler é responsável agora é certificar-se que emite um contexto à `Store` em cascata (downstream) e que trata o erro que virá da `Store` quando é cancelada.

Vamos atualizar nossa interface `Store` para mostrar as novas responsabilidades.

```
type Store interface {
    Fetch(ctx context.Context) (string, error)
}
```

Apague o código dentro do nosso handler por enquanto:

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
    }
}
```

Atualize nosso `SpyStore`:

```
type SpyStore struct {
    response string
    t        *testing.T
```

```
}

func (s *SpyStore) Fetch(ctx context.Context) (string, error) {
    data := make(chan string, 1)

    go func() {
        var result string
        for _, c := range s.response {
            select {
            case <-ctx.Done():
                s.t.Log("spy store foi cancelado")
                return
            default:
                time.Sleep(10 * time.Millisecond)
                result += string(c)
            }
        }
        data <- result
    }()

    select {
    case <-ctx.Done():
        return "", ctx.Err()
    case res := <-data:
        return res, nil
    }
}
```

Temos que fazer nosso spy agir como um método real que funciona com o `context`.

Estamos simulando um processo lento onde construímos o resultado lentamente adicionando a string, caractere por caractere em uma goroutine. Quando a goroutine termina seu trabalho, ela escreve a string no channel `data`. A goroutine escuta o `ctx.Done` e irá parar o trabalho se um sinal for enviado nesse channel.

Finalmente o código usa outro `select` para esperar que a goroutine termine seu trabalho ou que o cancelamento ocorra.

É semelhante à nossa abordagem de antes onde usamos as primitivas de concorrência do Go para fazerem dois processos assíncronos disputarem um contra o outro para determinar o que retornamos.

Você usará uma abordagem similar ao escrever suas próprias funções e métodos que aceitam um `context`, por isso certifique-se de que está entendendo o que está acontecendo.

Nós removemos a referência ao `ctx` dos campos do `SpyStore` porque não é mais interessante para nós. Estamos estritamente testando o comportamento agora,

que preferimos em comparação aos detalhes da implementação dos testes, como "você passou um determinado valor para a função `foo`".

Finalmente podemos atualizar nossos testes. Comente nosso teste de cancelamento para que possamos corrigir o teste do caminho feliz primeiro.

```
t.Run("retorna dados da store", func(t *testing.T) {
    store := &SpyStore{response: data, t: t}
    svr := Server(store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)
    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if response.Body.String() != data {
        t.Errorf(`resultado "%s", esperado "%s"`, response.Body.String(), data)
    }
})
```

Execute o teste

```
=== RUN   TestServer/retorna_dados_da_store
--- FAIL: TestServer (0.00s)
    --- FAIL: TestServer/retorna_dados_da_store (0.00s)
        context_test.go:22: resultado "", esperado "olá, mundo"
```

Escreva código o suficiente para fazer o teste passar

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        data, _ := store.Fetch(r.Context())
        fmt.Fprint(w, data)
    }
}
```

O nosso caminho feliz deve estar… feliz. Agora podemos corrigir o outro teste.

Escreva o teste primeiro

Precisamos testar que não escrevemos qualquer tipo de resposta no caso de erro. Infelizmente o `httptest.ResponseRecorder` não tem uma maneira de descobrir isso, então teremos que usar nosso próprio spy para testar.

```
type SpyResponseWriter struct {
    written bool
```

```
}

func (s *SpyResponseWriter) Header() http.Header {
    s.written = true
    return nil
}

func (s *SpyResponseWriter) Write([]byte) (int, error) {
    s.written = true
    return 0, errors.New("não implementado")
}

func (s *SpyResponseWriter) WriteHeader(statusCode int) {
    s.written = true
}
```

Nosso `SpyResponseWriter` implementa `http.ResponseWriter` para que possamos usá-lo no teste.

```
t.Run("avisa a store para cancelar o trabalho se a requisição for cancelada", func(t *testin
    store := &SpyStore{response: data, t: t}
    svr := Server(store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)

    cancellingCtx, cancel := context.WithCancel(request.Context())
    time.AfterFunc(5*time.Millisecond, cancel)
    request = request.WithContext(cancellingCtx)

    response := &SpyResponseWriter{}

    svr.ServeHTTP(response, request)

    if response.written {
        t.Error("uma resposta não deveria ter sido escrita")
    }
})
```

Execute o teste

```
=== RUN   TestServer
=== RUN  TestServer/avisa_a_store_para_cancelar_o_trabalho_se_a_requisicao_for_cancelada
--- FAIL: TestServer (0.01s)
   --- FAIL: TestServer/avisa_a_store_para_cancelar_o_trabalho_se_a_requisicao_for_cancelada
       context_test.go:47: uma resposta não deveria ter sido escrita
```

Escreva código o suficiente para fazer o teste passar

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        data, err := store.Fetch(r.Context())

        if err != nil {
            return // todo: registre o erro como você quiser
        }

        fmt.Fprint(w, data)
    }
}
```

Podemos ver depois disso que o código do servidor se tornou simplificado, pois não é mais explicitamente responsável pelo cancelamento. Ele simplesmente passa o `context` e confia nas funções em cascata (downstream) para respeitar qualquer cancelamento que possa ocorrer.

Resumo

Sobre o que falamos

- Como testar um handler HTTP que teve a requisição cancelada pelo cliente.
- Como usar o contexto para gerenciar o cancelamento.
- Como escrever uma função que aceita `context` e o usa para se cancelar usando goroutines, `select` e canais.
- Seguir as diretrizes da Google a respeito de como controlar o cancelamento propagando o contexto escopado da requisição através da sua pilha de chamadas (*call stack*).
- Como levar seu próprio spy para `http.ResponseWriter` se você precisar dele.

E quanto ao context.Value?

Michal Štrba e eu temos uma opinião semelhante.

> Se você usar o ctx.Value na minha empresa (inexistente), você está demitido

Alguns engenheiros têm defendido a passagem de valores através do `context` porque *parece conveniente*.

A conveniência é muitas vezes a causa do código ruim.

O problema com `context.Values` é que ele é apenas um mapa não tipado para que você não tenha nenhum tipo de segurança e você tem que lidar com ele não realmente contendo seu valor. Você tem que criar um acoplamento de chaves de mapa de um módulo para outro e se alguém muda alguma coisa começar a quebrar.

Resumindo, **se uma função necessita de alguns valores, coloque-os como parâmetros tipados em vez de tentar obtê-los a partir de `context.Value`**. Isto torna-o estaticamente verificado e documentado para que todos o vejam.

Mas...

Por outro lado, pode ser útil incluir informações que sejam ortogonais a uma requisição em um contexto, como um identificador único. Potencialmente esta informação não seria necessária para todas as funções da sua pilha de chamadas (*call stack*) e tornaria as suas assinaturas funcionais muito confusas.

Jack Lindamood diz que **Context.Value deve informar, não controlar**

> O conteúdo do context.Value é para os mantenedores e não para os usuários. Ele nunca deve ser uma entrada necessária para resultados documentados ou esperados.

Material adicional

- Gostei muito de ler Context should go away for Go 2 por Michal Štrba. Seu argumento é que ter que passar o `context` em toda parte é um indicador que está apontando a uma deficiência na linguagem a respeito do cancelamento. Ele diz que seria melhor se isso fosse resolvido de alguma forma no nível de linguagem, em vez de em um nível de biblioteca. Até que isso aconteça, você precisará do `context` se quiser gerenciar processos de longa duração.
- O blog do Go descreve ainda mais a motivação para trabalhar com `context` e tem alguns exemplos

Introdução

Finalizada a seção dos *Primeiros passos com Go*, você já deve possuir uma base sólida sobre os principais recursos da linguagem Go e como utilizar o TDD durante o seu processo de desenvolvimento.

Nossos próximos passos vão envolver o desenvolvimento de uma aplicação. Nessa seção, todo capitulo irá depender da funcionalidade implementada pelo seu antecessor, por isso evite pulá-los.

Aqui novos conceitos serão introduzidos para facilitar a escrita de grandes aplicações e a maior parte desse projeto será realizada utilizando bibliotecas padrões da linguagem Go.

Até o final dessa seção você deve ter obtido um entendimento sólido de como escrever aplicações em Go com o apoio de testes.

- HTTP server - Vamos criar uma API que aceita requisições HTTP.
- Respostas em JSON e roteamentos - Iremos evoluir nossa API para retornar objetos JSON e vamos explorar como fazer roteamentos.
- IO - Vamos salvar e ler dados de arquivos. Também vamos ordenar esses dados.
- Linha de comando - Vamos criar uma aplicação que vai ser utilizada por linha de comando no terminal, para entendermos como podemos suportar múltiplas plataformas.
- Eventos - Vamos agendar alguns eventos de processamento que irão acontecer dependendo do horário que usuário utilizou a aplicação.

Servidor HTTP

You can find all the code for this chapter here

You have been asked to create a web server where users can track how many games players have won.

- `GET /players/{name}` should return a number indicating the total number of wins
- `POST /players/{name}` should record a win for that name, incrementing for every subsequent `POST`

We will follow the TDD approach, getting working software as quickly as we can and then making small iterative improvements until we have the solution. By taking this approach we

- Keep the problem space small at any given time
- Don't go down rabbit holes
- If we ever get stuck/lost, doing a revert wouldn't lose loads of work.

Red, green, refactor

Throughout this book, we have emphasised the TDD process of write a test & watch it fail (red), write the *minimal* amount of code to make it work (green) and then refactor.

This discipline of writing the minimal amount of code is important in terms of the safety TDD gives you. You should be striving to get out of "red" as soon as you can.

Kent Beck describes it as:

> Make the test work quickly, committing whatever sins necessary in process.

You can commit these sins because you will refactor afterwards backed by the safety of the tests.

What if you don't do this?

The more changes you make while in red, the more likely you are to add more problems, not covered by tests.

The idea is to be iteratively writing useful code with small steps, driven by tests so that you don't fall into a rabbit hole for hours.

Chicken and egg

How can we incrementally build this? We can't `GET` a player without having stored something and it seems hard to know if `POST` has worked without the `GET` endpoint already existing.

This is where *mocking* shines.

- `GET` will need a `PlayerStore` *thing* to get scores for a player. This should be an interface so when we test we can create a simple stub to test our code without needing to have implemented any actual storage code.
- For `POST` we can *spy* on its calls to `PlayerStore` to make sure it stores players correctly. Our implementation of saving won't be coupled to retrieval.
- For having some working software quickly we can make a very simple in-memory implementation and then later we can create an implementation backed by whatever storage mechanism we prefer.

Write the test first

We can write a test and make it pass by returning a hard-coded value to get us started. Kent Beck refers this as "Faking it". Once we have a working test we can then write more tests to help us remove that constant.

By doing this very small step, we can make the important start of getting an overall project structure working correctly without having to worry too much about our application logic.

To create a web server in Go you will typically call ListenAndServe.

```
func ListenAndServe(addr string, handler Handler) error
```

This will start a web server listening on a port, creating a goroutine for every request and running it against a `Handler`.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

It has one function which expects two arguments, the first being where we *write our response* and the second being the HTTP request that was sent to us.

Let's write a test for a function `PlayerServer` that takes in those two arguments. The request sent in will be to get a player's score, which we expect to be `"20"`.

```
func TestGETPlayers(t *testing.T) {
    t.Run("returns Pepper's score", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodGet, "/players/Pepper", nil)
        response := httptest.NewRecorder()

        PlayerServer(response, request)

        got := response.Body.String()
        want := "20"

        if got != want {
            t.Errorf("got '%s', want '%s'", got, want)
        }
    })
}
```

In order to test our server, we will need a `Request` to send in and we'll want to *spy* on what our handler writes to the `ResponseWriter`.

- We use `http.NewRequest` to create a request. The first argument is the request's method and the second is the request's path. The `nil` argument refers to the request's body, which we don't need to set in this case.
- `net/http/httptest` has a spy already made for us called `ResponseRecorder` so we can use that. It has many helpful methods to inspect what has been written as a response.

Try to run the test

```
./server_test.go:13:2: undefined: PlayerServer
```

Write the minimal amount of code for the test to run and check the failing test output

The compiler is here to help, just listen to it.

Define `PlayerServer`

```
func PlayerServer() {}
```

Try again

```
./server_test.go:13:14: too many arguments in call to PlayerServer
    have (*httptest.ResponseRecorder, *http.Request)
    want ()
```

Add the arguments to our function

```
import "net/http"

func PlayerServer(w http.ResponseWriter, r *http.Request) {

}
```

The code now compiles and the test fails

```
=== RUN   TestGETPlayers/returns_Pepper's_score
    --- FAIL: TestGETPlayers/returns_Pepper's_score (0.00s)
        server_test.go:20: got '', want '20'
```

Write enough code to make it pass

From the DI chapter, we touched on HTTP servers with a `Greet` function. We learned that net/http's `ResponseWriter` also implements io `Writer` so we can use `fmt.Fprint` to send strings as HTTP responses.

```
func PlayerServer(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "20")
}
```

The test should now pass.

Complete the scaffolding

We want to wire this up into an application. This is important because

- We'll have *actual working software*, we don't want to write tests for the sake of it, it's good to see the code in action.
- As we refactor our code, it's likely we will change the structure of the program. We want to make sure this is reflected in our application too as part of the incremental approach.

Create a new file for our application and put this code in.

```
package main

import (
```

```
    "log"
    "net/http"
)

func main() {
    handler := http.HandlerFunc(PlayerServer)
    if err := http.ListenAndServe(":5000", handler); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

So far all of our application code has been in one file, however, this isn't best practice for larger projects where you'll want to separate things into different files.

To run this, do `go build` which will take all the `.go` files in the directory and build you a program. You can then execute it with `./myprogram`.

`http.HandlerFunc`

Earlier we explored that the `Handler` interface is what we need to implement in order to make a server. *Typically* we do that by creating a `struct` and make it implement the interface. However the use-case for structs is for holding data but *currently* we have no state, so it doesn't feel right to be creating one.

HandlerFunc lets us avoid this.

> The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler that calls f.

```
type HandlerFunc func(ResponseWriter, *Request)
```

So we use this to wrap our `PlayerServer` function so that it now conforms to `Handler`.

`http.ListenAndServe(":5000"...)`

`ListenAndServe` takes a port to listen on a `Handler`. If the port is already being listened to it will return an `error` so we are using an `if` statement to capture that scenario and log the problem to the user.

What we're going to do now is write *another* test to force us into making a positive change to try and move away from the hard-coded value.

Write the test first

We'll add another subtest to our suite which tries to get the score of a different player, which will break our hard-coded approach.

```
t.Run("returns Floyd's score", func(t *testing.T) {
    request, _ := http.NewRequest(http.MethodGet, "/players/Floyd", nil)
    response := httptest.NewRecorder()

    PlayerServer(response, request)

    got := response.Body.String()
    want := "10"

    if got != want {
        t.Errorf("got '%s', want '%s'", got, want)
    }
})
```

You may have been thinking

> Surely we need some kind of concept of storage to control which player gets what score. It's weird that the values seem so arbitrary in our tests.

Remember we are just trying to take as small as steps as reasonably possible, so we're just trying to break the constant for now.

Try to run the test

```
=== RUN   TestGETPlayers/returns_Pepper's_score
    --- PASS: TestGETPlayers/returns_Pepper's_score (0.00s)
=== RUN   TestGETPlayers/returns_Floyd's_score
    --- FAIL: TestGETPlayers/returns_Floyd's_score (0.00s)
        server_test.go:34: got '20', want '10'
```

Write enough code to make it pass

```
func PlayerServer(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    if player == "Pepper" {
        fmt.Fprint(w, "20")
        return
    }

    if player == "Floyd" {
```

```
        fmt.Fprint(w, "10")
        return
    }
}
```

This test has forced us to actually look at the request's URL and make a decision. So whilst in our heads, we may have been worrying about player stores and interfaces the next logical step actually seems to be about *routing*.

If we had started with the store code the amount of changes we'd have to do would be very large compared to this. **This is a smaller step towards our final goal and was driven by tests**.

We're resisting the temptation to use any routing libraries right now, just the smallest step to get our test passing.

`r.URL.Path` returns the path of the request and then we are using slice syntax to slice it past the final slash after `/players/`. It's not very robust but will do the trick for now.

Refactor

We can simplify the `PlayerServer` by separating out the score retrieval into a function

```
func PlayerServer(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    fmt.Fprint(w, GetPlayerScore(player))
}

func GetPlayerScore(name string) string {
    if name == "Pepper" {
        return "20"
    }

    if name == "Floyd" {
        return "10"
    }

    return ""
}
```

And we can DRY up some of the code in the tests by making some helpers

```
func TestGETPlayers(t *testing.T) {
    t.Run("returns Pepper's score", func(t *testing.T) {
        request := newGetScoreRequest("Pepper")
        response := httptest.NewRecorder()
```

```
        PlayerServer(response, request)

        assertResponseBody(t, response.Body.String(), "20")
    })

    t.Run("returns Floyd's score", func(t *testing.T) {
        request := newGetScoreRequest("Floyd")
        response := httptest.NewRecorder()

        PlayerServer(response, request)

        assertResponseBody(t, response.Body.String(), "10")
    })
}

func newGetScoreRequest(name string) *http.Request {
    req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/players/%s", name), nil)
    return req
}

func assertResponseBody(t *testing.T, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("response body is wrong, got '%s' want '%s'", got, want)
    }
}
```

However, we still shouldn't be happy. It doesn't feel right that our server knows the scores.

Our refactoring has made it pretty clear what to do.

We moved the score calculation out of the main body of our handler into a function `GetPlayerScore`. This feels like the right place to separate the concerns using interfaces.

Let's move our function we re-factored to be an interface instead

```
type PlayerStore interface {
    GetPlayerScore(name string) int
}
```

For our `PlayerServer` to be able to use a `PlayerStore`, it will need a reference to one. Now feels like the right time to change our architecture so that our `PlayerServer` is now a `struct`.

```
type PlayerServer struct {
    store PlayerStore
}
```

Finally, we will now implement the `Handler` interface by adding a method to our new struct and putting in our existing handler code.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]
    fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

The only other change is we now call our `store.GetPlayerStore` to get the score, rather than the local function we defined (which we can now delete).

Here is the full code listing of our server

```
type PlayerStore interface {
    GetPlayerScore(name string) int
}

type PlayerServer struct {
    store PlayerStore
}

func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]
    fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

Fix the issues

This was quite a few changes and we know our tests and application will no longer compile, but just relax and let the compiler work through it.

```
./main.go:9:58: type PlayerServer is not an expression
```

We need to change our tests to instead create a new instance of our `PlayerServer` and then call its method `ServeHTTP`.

```
func TestGETPlayers(t *testing.T) {
    server := &PlayerServer{}

    t.Run("returns Pepper's score", func(t *testing.T) {
        request := newGetScoreRequest("Pepper")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertResponseBody(t, response.Body.String(), "20")
    })

    t.Run("returns Floyd's score", func(t *testing.T) {
```

```
        request := newGetScoreRequest("Floyd")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertResponseBody(t, response.Body.String(), "10")
    })
}
```

Notice we're still not worrying about making stores *just yet*, we just want the compiler passing as soon as we can.

You should be in the habit of prioritising having code that compiles and then code that passes the tests.

By adding more functionality (like stub stores) whilst the code isn't compiling, we are opening ourselves up to potentially *more* compilation problems.

Now `main.go` won't compile for the same reason.

```
func main() {
    server := &PlayerServer{}

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

Finally, everything is compiling but the tests are failing

```
=== RUN   TestGETPlayers/returns_the_Pepper's_score
panic: runtime error: invalid memory address or nil pointer dereference [recovered]
   panic: runtime error: invalid memory address or nil pointer dereference
```

This is because we have not passed in a `PlayerStore` in our tests. We'll need to make a stub one up.

```
type StubPlayerStore struct {
    scores map[string]int
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
    score := s.scores[name]
    return score
}
```

A `map` is a quick and easy way of making a stub key/value store for our tests. Now let's create one of these stores for our tests and send it into our `PlayerServer`.

```
func TestGETPlayers(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{
```

```
            "Pepper": 20,
            "Floyd":  10,
        },
    }
    server := &PlayerServer{&store}

    t.Run("returns Pepper's score", func(t *testing.T) {
        request := newGetScoreRequest("Pepper")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertResponseBody(t, response.Body.String(), "20")
    })

    t.Run("returns Floyd's score", func(t *testing.T) {
        request := newGetScoreRequest("Floyd")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertResponseBody(t, response.Body.String(), "10")
    })
}
```

Our tests now pass and are looking better. The *intent* behind our code is clearer now due to the introduction of the store. We're telling the reader that because we have *this data in a `PlayerStore`* that when you use it with a `PlayerServer` you should get the following responses.

Run the application

Now our tests are passing the last thing we need to do to complete this refactor is to check if our application is working. The program should start up but you'll get a horrible response if you try and hit the server at `http://localhost:5000/players/Pepper`.

The reason for this is that we have not passed in a `PlayerStore`.

We'll need to make an implementation of one, but that's difficult right now as we're not storing any meaningful data so it'll have to be hard-coded for the time being.

```
type InMemoryPlayerStore struct{}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return 123
```

```
}

func main() {
    server := &PlayerServer{&InMemoryPlayerStore{}}

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

If you run `go build` again and hit the same URL you should get `"123"`. Not great, but until we store data that's the best we can do.

We have a few options as to what to do next

- Handle the scenario where the player doesn't exist
- Handle the `POST /players/{name}` scenario
- It didn't feel great that our main application was starting up but not actually working. We had to manually test to see the problem.

Whilst the `POST` scenario gets us closer to the "happy path", I feel it'll be easier to tackle the missing player scenario first as we're in that context already. We'll get to the rest later.

Write the test first

Add a missing player scenario to our existing suite

```
t.Run("returns 404 on missing players", func(t *testing.T) {
    request := newGetScoreRequest("Apollo")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    got := response.Code
    want := http.StatusNotFound

    if got != want {
        t.Errorf("got status %d want %d", got, want)
    }
})
```

Try to run the test

```
=== RUN   TestGETPlayers/returns_404_on_missing_players
   --- FAIL: TestGETPlayers/returns_404_on_missing_players (0.00s)
        server_test.go:56: got status 200 want 404
```

Write enough code to make it pass

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    w.WriteHeader(http.StatusNotFound)

    fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

Sometimes I heavily roll my eyes when TDD advocates say "make sure you just write the minimal amount of code to make it pass" as it can feel very pedantic.

But this scenario illustrates the example well. I have done the bare minimum (knowing it is not correct), which is write a `StatusNotFound` on **all responses** but all our tests are passing!

By doing the bare minimum to make the tests pass it can highlight gaps in your tests. In our case, we are not asserting that we should be getting a `StatusOK` when players *do* exist in the store.

Update the other two tests to assert on the status and fix the code.

Here are the new tests

```
func TestGETPlayers(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{
            "Pepper": 20,
            "Floyd":  10,
        },
    }
    server := &PlayerServer{&store}

    t.Run("returns Pepper's score", func(t *testing.T) {
        request := newGetScoreRequest("Pepper")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusOK)
        assertResponseBody(t, response.Body.String(), "20")
    })

    t.Run("returns Floyd's score", func(t *testing.T) {
        request := newGetScoreRequest("Floyd")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)
```

```
        assertStatus(t, response.Code, http.StatusOK)
        assertResponseBody(t, response.Body.String(), "10")
    })

    t.Run("returns 404 on missing players", func(t *testing.T) {
        request := newGetScoreRequest("Apollo")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusNotFound)
    })
}

func assertStatus(t *testing.T, got, want int) {
    t.Helper()
    if got != want {
        t.Errorf("did not get correct status, got %d, want %d", got, want)
    }
}

func newGetScoreRequest(name string) *http.Request {
    req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/players/%s", name), nil)
    return req
}

func assertResponseBody(t *testing.T, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("response body is wrong, got '%s' want '%s'", got, want)
    }
}
```

We're checking the status in all our tests now so I made a helper `assertStatus` to facilitate that.

Now our first two tests fail because of the 404 instead of 200, so we can fix `PlayerServer` to only return not found if the score is 0.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
```

```
    }

    fmt.Fprint(w, score)
}
```

Storing scores

Now that we can retrieve scores from a store it now makes sense to be able to store new scores.

Write the test first

```
func TestStoreWins(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{},
    }
    server := &PlayerServer{&store}

    t.Run("it returns accepted on POST", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodPost, "/players/Pepper", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusAccepted)
    })
}
```

For a start let's just check we get the correct status code if we hit the particular route with POST. This lets us drive out the functionality of accepting a different kind of request and handling it differently to `GET /players/{name}`. Once this works we can then start asserting on our handler's interaction with the store.

Try to run the test

```
=== RUN   TestStoreWins/it_returns_accepted_on_POST
    --- FAIL: TestStoreWins/it_returns_accepted_on_POST (0.00s)
     server_test.go:70: did not get correct status, got 404, want 202
```

Write enough code to make it pass

Remember we are deliberately committing sins, so an `if` statement based on the request's method will do the trick.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    if r.Method == http.MethodPost {
        w.WriteHeader(http.StatusAccepted)
        return
    }

    player := r.URL.Path[len("/players/"):]

    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}
```

Refactor

The handler is looking a bit muddled now. Let's break the code up to make it easier to follow and isolate the different functionality into new functions.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    switch r.Method {
    case http.MethodPost:
        p.processWin(w)
    case http.MethodGet:
        p.showScore(w, r)
    }

}

func (p *PlayerServer) showScore(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}
```

```
func (p *PlayerServer) processWin(w http.ResponseWriter) {
    w.WriteHeader(http.StatusAccepted)
}
```

This makes the routing aspect of `ServeHTTP` a bit clearer and means our next iterations on storing can just be inside `processWin`.

Next, we want to check that when we do our `POST /players/{name}` that our `PlayerStore` is told to record the win.

Write the test first

We can accomplish this by extending our `StubPlayerStore` with a new `RecordWin` method and then spy on its invocations.

```
type StubPlayerStore struct {
    scores   map[string]int
    winCalls []string
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
    score := s.scores[name]
    return score
}

func (s *StubPlayerStore) RecordWin(name string) {
    s.winCalls = append(s.winCalls, name)
}
```

Now extend our test to check the number of invocations for a start

```
func TestStoreWins(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{},
    }
    server := &PlayerServer{&store}

    t.Run("it records wins when POST", func(t *testing.T) {
        request := newPostWinRequest("Pepper")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusAccepted)

        if len(store.winCalls) != 1 {
            t.Errorf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
```

```
        }
    })
}

func newPostWinRequest(name string) *http.Request {
    req, _ := http.NewRequest(http.MethodPost, fmt.Sprintf("/players/%s", name), nil)
    return req
}
```

Try to run the test

```
./server_test.go:26:20: too few values in struct initializer
./server_test.go:65:20: too few values in struct initializer
```

Write the minimal amount of code for the test to run and check the failing test output

We need to update our code where we create a `StubPlayerStore` as we've added a new field

```
store := StubPlayerStore{
    map[string]int{},
    nil,
}
```

```
--- FAIL: TestStoreWins (0.00s)
    --- FAIL: TestStoreWins/it_records_wins_when_POST (0.00s)
        server_test.go:80: got 0 calls to RecordWin want 1
```

Write enough code to make it pass

As we're only asserting the number of calls rather than the specific values it makes our initial iteration a little smaller.

We need to update `PlayerServer`'s idea of what a `PlayerStore` is by changing the interface if we're going to be able to call `RecordWin`.

```
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
}
```

By doing this `main` no longer compiles

```
./main.go:17:46: cannot use InMemoryPlayerStore literal (type *InMemoryPlayerStore) as type Pla
   *InMemoryPlayerStore does not implement PlayerStore (missing RecordWin method)
```

The compiler tells us what's wrong. Let's update `InMemoryPlayerStore` to have that method.

```
type InMemoryPlayerStore struct{}

func (i *InMemoryPlayerStore) RecordWin(name string) {}
```

Try and run the tests and we should be back to compiling code - but the test is still failing.

Now that `PlayerStore` has `RecordWin` we can call it within our `PlayerServer`

```
func (p *PlayerServer) processWin(w http.ResponseWriter) {
    p.store.RecordWin("Bob")
    w.WriteHeader(http.StatusAccepted)
}
```

Run the tests and it should be passing! Obviously `"Bob"` isn't exactly what we want to send to `RecordWin`, so let's further refine the test.

Write the test first

```
t.Run("it records wins on POST", func(t *testing.T) {
    player := "Pepper"

    request := newPostWinRequest(player)
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertStatus(t, response.Code, http.StatusAccepted)

    if len(store.winCalls) != 1 {
        t.Fatalf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
    }

    if store.winCalls[0] != player {
        t.Errorf("did not store correct winner got '%s' want '%s'", store.winCalls[0], playe
    }
})
```

Now that we know there is one element in our `winCalls` slice we can safely reference the first one and check it is equal to `player`.

Try to run the test

```
=== RUN   TestStoreWins/it_records_wins_on_POST
    --- FAIL: TestStoreWins/it_records_wins_on_POST (0.00s)
```

```
server_test.go:86: did not store correct winner got 'Bob' want 'Pepper'
```

Write enough code to make it pass

```
func (p *PlayerServer) processWin(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}
```

We changed `processWin` to take `http.Request` so we can look at the URL to extract the player's name. Once we have that we can call our `store` with the correct value to make the test pass.

Refactor

We can DRY up this code a bit as we're extracting the player name the same way in two places

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    switch r.Method {
    case http.MethodPost:
        p.processWin(w, player)
    case http.MethodGet:
        p.showScore(w, player)
    }
}

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}
```

Even though our tests are passing we don't really have working software. If you try and run `main` and use the software as intended it doesn't work because we haven't got round to implementing `PlayerStore` correctly. This is fine though; by focusing on our handler we have identified the interface that we need, rather than trying to design it up-front.

We *could* start writing some tests around our `InMemoryPlayerStore` but it's only here temporarily until we implement a more robust way of persisting player scores (i.e. a database).

What we'll do for now is write an *integration test* between our `PlayerServer` and `InMemoryPlayerStore` to finish off the functionality. This will let us get to our goal of being confident our application is working, without having to directly test `InMemoryPlayerStore`. Not only that, but when we get around to implementing `PlayerStore` with a database, we can test that implementation with the same integration test.

Integration tests

Integration tests can be useful for testing that larger areas of your system work but you must bear in mind:

- They are harder to write
- When they fail, it can be difficult to know why (usually it's a bug within a component of the integration test) and so can be harder to fix
- They are sometimes slower to run (as they often are used with "real" components, like a database)

For that reason, it is recommended that you research *The Test Pyramid*.

Write the test first

In the interest of brevity, I am going to show you the final refactored integration test.

```
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
    store := InMemoryPlayerStore{}
    server := PlayerServer{&store}
    player := "Pepper"

    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))

    response := httptest.NewRecorder()
    server.ServeHTTP(response, newGetScoreRequest(player))
    assertStatus(t, response.Code, http.StatusOK)
```

```
    assertResponseBody(t, response.Body.String(), "3")
}
```

- We are creating our two components we are trying to integrate with: `InMemoryPlayerStore` and `PlayerServer`.
- We then fire off 3 requests to record 3 wins for `player`. We're not too concerned about the status codes in this test as it's not relevant to whether they are integrating well.
- The next response we do care about (so we store a variable `response`) because we are going to try and get the `player`'s score.

Try to run the test

```
--- FAIL: TestRecordingWinsAndRetrievingThem (0.00s)
   server_integration_test.go:24: response body is wrong, got '123' want '3'
```

Write enough code to make it pass

I am going to take some liberties here and write more code than you may be comfortable with without writing a test.

This is allowed! We still have a test checking things should be working correctly but it is not around the specific unit we're working with (`InMemoryPlayerStore`).

If I were to get stuck in this scenario, I would revert my changes back to the failing test and then write more specific unit tests around `InMemoryPlayerStore` to help me drive out a solution.

```
func NewInMemoryPlayerStore() *InMemoryPlayerStore {
    return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct{
    store map[string]int
}

func (i *InMemoryPlayerStore) RecordWin(name string) {
    i.store[name]++
}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return i.store[name]
}
```

- We need to store the data so I've added a `map[string]int` to the `InMemoryPlayerStore` struct
- For convenience I've made `NewInMemoryPlayerStore` to initialise the store, and updated the integration test to use it (`store := NewInMemoryPlayerStore()`)
- The rest of the code is just wrapping around the `map`

The integration test passes, now we just need to change `main` to use `NewInMemoryPlayerStore()`

```
package main

import (
    "log"
    "net/http"
)

func main() {
    server := &PlayerServer{NewInMemoryPlayerStore()}

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

Build it, run it and then use `curl` to test it out.

- Run this a few times, change the player names if you like `curl -X POST http://localhost:5000/players/Pepper`
- Check scores with `curl http://localhost:5000/players/Pepper`

Great! You've made a REST-ish service. To take this forward you'd want to pick a data store to persist the scores longer than the length of time the program runs.

- Pick a store (Bolt? Mongo? Postgres? File system?)
- Make `PostgresPlayerStore` implement `PlayerStore`
- TDD the functionality so you're sure it works
- Plug it into the integration test, check it's still ok
- Finally plug it into `main`

Wrapping up

`http.Handler`

- Implement this interface to create web servers
- Use `http.HandlerFunc` to turn ordinary functions into `http.Handler`s

- Use `httptest.NewRecorder` to pass in as a `ResponseWriter` to let you spy on the responses your handler sends
- Use `http.NewRequest` to construct the requests you expect to come in to your system

Interfaces, Mocking and DI

- Lets you iteratively build the system up in smaller chunks
- Allows you to develop a handler that needs a storage without needing actual storage
- TDD to drive out the interfaces you need

Commit sins, then refactor (and then commit to source control)

- You need to treat having failing compilation or failing tests as a red situation that you need to get out of as soon as you can.
- Write just the necessary code to get there. *Then* refactor and make the code nice.
- By trying to do too many changes whilst the code isn't compiling or the tests are failing puts you at risk of compounding the problems.
- Sticking to this approach forces you to write small tests, which means small changes, which helps keep working on complex systems manageable.

JSON, roteamento and embedding

You can find all the code for this chapter here

In the previous chapter we created a web server to store how many games players have won.

Our product owner has a new requirement; to have a new endpoint called `/league` which returns a list of all players stored. She would like this to be returned as JSON.

Here is the code we have so far

```
// server.go
package main

import (
    "fmt"
    "net/http"
)
```

```
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
}

type PlayerServer struct {
    store PlayerStore
}

func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    switch r.Method {
    case http.MethodPost:
        p.processWin(w, player)
    case http.MethodGet:
        p.showScore(w, player)
    }
}

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}
```

```
// InMemoryPlayerStore.go
package main

func NewInMemoryPlayerStore() *InMemoryPlayerStore {
    return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct {
    store map[string]int
}
```

```
func (i *InMemoryPlayerStore) RecordWin(name string) {
    i.store[name]++
}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return i.store[name]
}
```

```
// main.go
package main

import (
    "log"
    "net/http"
)

func main() {
    server := &PlayerServer{NewInMemoryPlayerStore()}

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

You can find the corresponding tests in the link at the top of the chapter.

We'll start by making the league table endpoint.

Write the test first

We'll extend the existing suite as we have some useful test functions and a fake `PlayerStore` to use.

```
func TestLeague(t *testing.T) {
    store := StubPlayerStore{}
    server := &PlayerServer{&store}

    t.Run("it returns 200 on /league", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodGet, "/league", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusOK)
    })
}
```

Before worrying about actual scores and JSON we will try and keep the changes small with the plan to iterate toward our goal. The simplest start is to check we can hit `/league` and get an `OK` back.

Try to run the test

```
=== RUN   TestLeague/it_returns_200_on_/league
panic: runtime error: slice bounds out of range [recovered]
    panic: runtime error: slice bounds out of range

goroutine 6 [running]:
testing.tRunner.func1(0xc42010c3c0)
  /usr/local/Cellar/go/1.10/libexec/src/testing/testing.go:742 +0x29d
panic(0x1274d60, 0x1438240)
  /usr/local/Cellar/go/1.10/libexec/src/runtime/panic.go:505 +0x229
github.com/quii/learn-go-with-tests/json-and-io/v2.(*PlayerServer).ServeHTTP(0xc420048d30, 0
  /Users/quii/go/src/github.com/quii/learn-go-with-tests/json-and-io/v2/server.go:20 +0xec
```

Your `PlayerServer` should be panicking like this. Go to the line of code in the stack trace which is pointing to `server.go`.

```
player := r.URL.Path[len("/players/"):]
```

In the previous chapter, we mentioned this was a fairly naive way of doing our routing. What is happening is it's trying to split the string of the path starting at an index beyond `/league` so it is `slice bounds out of range`.

Write enough code to make it pass

Go has a built-in routing mechanism called ServeMux (request multiplexer) which lets you attach `http.Handler`s to particular request paths.

Let's commit some sins and get the tests passing in the quickest way we can, knowing we can refactor it with safety once we know the tests are passing.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    router := http.NewServeMux()

    router.Handle("/league", http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
    }))

    router.Handle("/players/", http.HandlerFunc(func(w http.ResponseWriter, r *http.Request)
        player := r.URL.Path[len("/players/"):]

        switch r.Method {
```

```
        case http.MethodPost:
            p.processWin(w, player)
        case http.MethodGet:
            p.showScore(w, player)
        }
    }))

    router.ServeHTTP(w, r)
}
```

- When the request starts we create a router and then we tell it for `x` path use `y` handler.
- So for our new endpoint, we use `http.HandlerFunc` and an *anonymous function* to `w.WriteHeader(http.StatusOK)` when `/league` is requested to make our new test pass.
- For the `/players/` route we just cut and paste our code into another `http.HandlerFunc`.
- Finally, we handle the request that came in by calling our new router's `ServeHTTP` (notice how `ServeMux` is *also* an `http.Handler`?)

The tests should now pass.

Refactor

`ServeHTTP` is looking quite big, we can separate things out a bit by refactoring our handlers into separate methods.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    router.ServeHTTP(w, r)
}

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
}

func (p *PlayerServer) playersHandler(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    switch r.Method {
    case http.MethodPost:
        p.processWin(w, player)
    case http.MethodGet:
```

```
        p.showScore(w, player)
    }
}
```

It's quite odd (and inefficient) to be setting up a router as a request comes in and then calling it. What we ideally want to do is have some kind of `NewPlayerServer` function which will take our dependencies and do the one-time setup of creating the router. Each request can then just use that one instance of the router.

```
type PlayerServer struct {
    store  PlayerStore
    router *http.ServeMux
}

func NewPlayerServer(store PlayerStore) *PlayerServer {
    p := &PlayerServer{
        store,
        http.NewServeMux(),
    }

    p.router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    p.router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    return p
}

func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    p.router.ServeHTTP(w, r)
}
```

- `PlayerServer` now needs to store a router.
- We have moved the routing creation out of `ServeHTTP` and into our `NewPlayerServer` so this only has to be done once, not per request.
- You will need to update all the test and production code where we used to do `PlayerServer{&store}` with `NewPlayerServer(&store)`.

One final refactor

Try changing the code to the following.

```
type PlayerServer struct {
    store  PlayerStore
    http.Handler
}

func NewPlayerServer(store PlayerStore) *PlayerServer {
```

```
    p := new(PlayerServer)

    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    p.Handler = router

    return p
}
```

Finally make sure you **delete** `func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request)` as it is no longer needed!

Embedding

We changed the second property of `PlayerServer`, removing the named property `router http.ServeMux` and replaced it with `http.Handler`; this is called *embedding*.

> Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to "borrow" pieces of an implementation by embedding types within a struct or interface.
>
> Effective Go - Embedding

What this means is that our `PlayerServer` now has all the methods that `http.Handler` has, which is just `ServeHTTP`.

To "fill in" the `http.Handler` we assign it to the `router` we create in `NewPlayerServer`. We can do this because `http.ServeMux` has the method `ServeHTTP`.

This lets us remove our own `ServeHTTP` method, as we are already exposing one via the embedded type.

Embedding is a very interesting language feature. You can use it with interfaces to compose new interfaces.

```
type Animal interface {
    Eater
    Sleeper
}
```

And you can use it with concrete types too, not just interfaces. As you'd expect if you embed a concrete type you'll have access to all its public methods and fields.

Any downsides?

You must be careful with embedding types because you will expose all public methods and fields of the type you embed. In our case, it is ok because we embedded just the *interface* that we wanted to expose (`http.Handler`).

If we had been lazy and embedded `http.ServeMux` instead (the concrete type) it would still work *but* users of `PlayerServer` would be able to add new routes to our server because `Handle(path, handler)` would be public.

When embedding types, really think about what impact that has on your public API.

It is a *very* common mistake to misuse embedding and end up polluting your APIs and exposing the internals of your type.

Now we've restructured our application we can easily add new routes and have the start of the `/league` endpoint. We now need to make it return some useful information.

We should return some JSON that looks something like this.

```
[
   {
      "Name":"Bill",
      "Wins":10
   },
   {
      "Name":"Alice",
      "Wins":15
   }
]
```

Write the test first

We'll start by trying to parse the response into something meaningful.

```
func TestLeague(t *testing.T) {
    store := StubPlayerStore{}
    server := NewPlayerServer(&store)

    t.Run("it returns 200 on /league", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodGet, "/league", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        var got []Player
```

```
        err := json.NewDecoder(response.Body).Decode(&got)

        if err != nil {
            t.Fatalf ("Unable to parse response from server '%s' into slice of Player, '%v''
        }

        assertStatus(t, response.Code, http.StatusOK)
    })
}
```

Why not test the JSON string?

You could argue a simpler initial step would be just to assert that the response body has a particular JSON string.

In my experience tests that assert against JSON strings have the following problems.

- *Brittleness.* If you change the data-model your tests will fail.
- *Hard to debug.* It can be tricky to understand what the actual problem is when comparing two JSON strings.
- *Poor intention.* Whilst the output should be JSON, what's really important is exactly what the data is, rather than how it's encoded.
- *Re-testing the standard library.* There is no need to test how the standard library outputs JSON, it is already tested. Don't test other people's code.

Instead, we should look to parse the JSON into data structures that are relevant for us to test with.

Data modelling

Given the JSON data model, it looks like we need an array of `Player` with some fields so we have created a new type to capture this.

```
type Player struct {
    Name string
    Wins int
}
```

JSON decoding

```
var got []Player
err := json.NewDecoder(response.Body).Decode(&got)
```

To parse JSON into our data model we create a `Decoder` from `encoding/json` package and then call its `Decode` method. To create a `Decoder` it needs an `io.Reader` to read from which in our case is our response spy's `Body`.

`Decode` takes the address of the thing we are trying to decode into which is why we declare an empty slice of `Player` the line before.

Parsing JSON can fail so `Decode` can return an `error`. There's no point continuing the test if that fails so we check for the error and stop the test with `t.Fatalf` if it happens. Notice that we print the response body along with the error as it's important for someone running the test to see what string cannot be parsed.

Try to run the test

```
=== RUN   TestLeague/it_returns_200_on_/league
    --- FAIL: TestLeague/it_returns_200_on_/league (0.00s)
     server_test.go:107: Unable to parse response from server '' into slice of Player, 'unexpec
```

Our endpoint currently does not return a body so it cannot be parsed into JSON.

Write enough code to make it pass

```
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    leagueTable := []Player{
        {"Chris", 20},
    }

    json.NewEncoder(w).Encode(leagueTable)

    w.WriteHeader(http.StatusOK)
}
```

The test now passes.

Encoding and Decoding

Notice the lovely symmetry in the standard library.

- To create an `Encoder` you need an `io.Writer` which is what `http.ResponseWriter` implements.
- To create a `Decoder` you need an `io.Reader` which the `Body` field of our response spy implements.

Throughout this book, we have used `io.Writer` and this is another demonstration of its prevalence in the standard library and how a lot of libraries easily work with it.

Refactor

It would be nice to introduce a separation of concern between our handler and getting the `leagueTable` as we know we're going to not hard-code that very soon.

```
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(p.getLeagueTable())
    w.WriteHeader(http.StatusOK)
}

func (p *PlayerServer) getLeagueTable() []Player{
    return []Player{
        {"Chris", 20},
    }
}
```

Next, we'll want to extend our test so that we can control exactly what data we want back.

Write the test first

We can update the test to assert that the league table contains some players that we will stub in our store.

Update `StubPlayerStore` to let it store a league, which is just a slice of `Player`. We'll store our expected data in there.

```
type StubPlayerStore struct {
    scores   map[string]int
    winCalls []string
    league []Player
}
```

Next, update our current test by putting some players in the league property of our stub and assert they get returned from our server.

```
func TestLeague(t *testing.T) {

    t.Run("it returns the league table as JSON", func(t *testing.T) {
        wantedLeague := []Player{
            {"Cleo", 32},
            {"Chris", 20},
            {"Tiest", 14},
        }

        store := StubPlayerStore{nil, nil, wantedLeague}
        server := NewPlayerServer(&store)
```

```
        request, _ := http.NewRequest(http.MethodGet, "/league", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        var got []Player

        err := json.NewDecoder(response.Body).Decode(&got)

        if err != nil {
            t.Fatalf("Unable to parse response from server '%s' into slice of Player, '%v'",
        }

        assertStatus(t, response.Code, http.StatusOK)

        if !reflect.DeepEqual(got, wantedLeague) {
            t.Errorf("got %v want %v", got, wantedLeague)
        }
    })
}
```

Try to run the test

```
./server_test.go:33:3: too few values in struct initializer
./server_test.go:70:3: too few values in struct initializer
```

Write the minimal amount of code for the test to run and check the failing test output

You'll need to update the other tests as we have a new field in `StubPlayerStore`; set it to nil for the other tests.

Try running the tests again and you should get

```
=== RUN   TestLeague/it_returns_the_league_table_as_JSON
   --- FAIL: TestLeague/it_returns_the_league_table_as_JSON (0.00s)
      server_test.go:124: got [{Chris 20}] want [{Cleo 32} {Chris 20} {Tiest 14}]
```

Write enough code to make it pass

We know the data is in our `StubPlayerStore` and we've abstracted that away into an interface `PlayerStore`. We need to update this so anyone passing us in a `PlayerStore` can provide us with the data for leagues.

```
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
    GetLeague() []Player
}
```

Now we can update our handler code to call that rather than returning a hard-coded list. Delete our method `getLeagueTable()` and then update `leagueHandler` to call `GetLeague()`.

```
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(p.store.GetLeague())
    w.WriteHeader(http.StatusOK)
}
```

Try and run the tests.

```
# github.com/quii/learn-go-with-tests/json-and-io/v4
./main.go:9:50: cannot use NewInMemoryPlayerStore() (type *InMemoryPlayerStore) as type Player
  *InMemoryPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_integration_test.go:11:27: cannot use store (type *InMemoryPlayerStore) as type Playe
  *InMemoryPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:36:28: cannot use &store (type *StubPlayerStore) as type PlayerStore in argume
  *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:74:28: cannot use &store (type *StubPlayerStore) as type PlayerStore in argume
  *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:106:29: cannot use &store (type *StubPlayerStore) as type PlayerStore in argum
  *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
```

The compiler is complaining because `InMemoryPlayerStore` and `StubPlayerStore` do not have the new method we added to our interface.

For `StubPlayerStore` it's pretty easy, just return the `league` field we added earlier.

```
func (s *StubPlayerStore) GetLeague() []Player {
    return s.league
}
```

Here's a reminder of how `InMemoryStore` is implemented.

```
type InMemoryPlayerStore struct {
    store map[string]int
}
```

Whilst it would be pretty straightforward to implement `GetLeague` "properly" by iterating over the map remember we are just trying to *write the minimal amount of code to make the tests pass.*

So let's just get the compiler happy for now and live with the uncomfortable feeling of an incomplete implementation in our `InMemoryStore`.

```
func (i *InMemoryPlayerStore) GetLeague() []Player {
    return nil
}
```

What this is really telling us is that *later* we're going to want to test this but let's park that for now.

Try and run the tests, the compiler should pass and the tests should be passing!

Refactor

The test code does not convey out intent very well and has a lot of boilerplate we can refactor away.

```
t.Run("it returns the league table as JSON", func(t *testing.T) {
    wantedLeague := []Player{
        {"Cleo", 32},
        {"Chris", 20},
        {"Tiest", 14},
    }

    store := StubPlayerStore{nil, nil, wantedLeague}
    server := NewPlayerServer(&store)

    request := newLeagueRequest()
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    got := getLeagueFromResponse(t, response.Body)
    assertStatus(t, response.Code, http.StatusOK)
    assertLeague(t, got, wantedLeague)
})
```

Here are the new helpers

```
func getLeagueFromResponse(t *testing.T, body io.Reader) (league []Player) {
    t.Helper()
    err := json.NewDecoder(body).Decode(&league)

    if err != nil {
        t.Fatalf("Unable to parse response from server '%s' into slice of Player, '%v'", boc
    }

    return
}

func assertLeague(t *testing.T, got, want []Player) {
```

```
    t.Helper()
    if !reflect.DeepEqual(got, want) {
        t.Errorf("got %v want %v", got, want)
    }
}

func newLeagueRequest() *http.Request {
    req, _ := http.NewRequest(http.MethodGet, "/league", nil)
    return req
}
```

One final thing we need to do for our server to work is make sure we return a `content-type` header in the response so machines can recognise we are returning `JSON`.

Write the test first

Add this assertion to the existing test

```
if response.Result().Header.Get("content-type") != "application/json" {
    t.Errorf("response did not have content-type of application/json, got %v", response.Resu
}
```

Try to run the test

```
=== RUN   TestLeague/it_returns_the_league_table_as_JSON
   --- FAIL: TestLeague/it_returns_the_league_table_as_JSON (0.00s)
      server_test.go:124: response did not have content-type of application/json, got map[Conte
```

Write enough code to make it pass

Update `leagueHandler`

```
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("content-type", "application/json")
    json.NewEncoder(w).Encode(p.store.GetLeague())
}
```

The test should pass.

Refactor

Add a helper for `assertContentType`.

```
const jsonContentType = "application/json"

func assertContentType(t *testing.T, response *httptest.ResponseRecorder, want string) {
    t.Helper()
    if response.Result().Header.Get("content-type") != want {
        t.Errorf("response did not have content-type of %s, got %v", want, response.Result()
    }
}
```

Use it in the test.

```
assertContentType(t, response, jsonContentType)
```

Now that we have sorted out `PlayerServer` for now we can turn our attention to `InMemoryPlayerStore` because right now if we tried to demo this to the product owner `/league` will not work.

The quickest way for us to get some confidence is to add to our integration test, we can hit the new endpoint and check we get back the correct response from `/league`.

Write the test first

We can use `t.Run` to break up this test a bit and we can reuse the helpers from our server tests - again showing the importance of refactoring tests.

```
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
    store := NewInMemoryPlayerStore()
    server := NewPlayerServer(store)
    player := "Pepper"

    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))

    t.Run("get score", func(t *testing.T) {
        response := httptest.NewRecorder()
        server.ServeHTTP(response, newGetScoreRequest(player))
        assertStatus(t, response.Code, http.StatusOK)

        assertResponseBody(t, response.Body.String(), "3")
    })

    t.Run("get league", func(t *testing.T) {
        response := httptest.NewRecorder()
        server.ServeHTTP(response, newLeagueRequest())
        assertStatus(t, response.Code, http.StatusOK)
```

```
        got := getLeagueFromResponse(t, response.Body)
        want := []Player{
            {"Pepper", 3},
        }
        assertLeague(t, got, want)
    })
}
```

Try to run the test

```
=== RUN   TestRecordingWinsAndRetrievingThem/get_league
   --- FAIL: TestRecordingWinsAndRetrievingThem/get_league (0.00s)
        server_integration_test.go:35: got [] want [{Pepper 3}]
```

Write enough code to make it pass

`InMemoryPlayerStore` is returning `nil` when you call `GetLeague()` so we'll need to fix that.

```
func (i *InMemoryPlayerStore) GetLeague() []Player {
    var league []Player
    for name, wins := range i.store {
        league = append(league, Player{name, wins})
    }
    return league
}
```

All we need to do is iterate over the map and convert each key/value to a `Player`.

The test should now pass.

Wrapping up

We've continued to safely iterate on our program using TDD, making it support new endpoints in a maintainable way with a router and it can now return JSON for our consumers. In the next chapter, we will cover persisting the data and sorting our league.

What we've covered:

- **Routing**. The standard library offers you an easy to use type to do routing. It fully embraces the `http.Handler` interface in that you assign routes to `Handler`s and the router itself is also a `Handler`. It does not have some features you might expect though such as path variables (e.g `/users/{id}`). You can easily parse this information yourself but you might want to consider looking at other routing libraries if it becomes a

burden. Most of the popular ones stick to the standard library's philosophy of also implementing `http.Handler`.
- **Type embedding**. We touched a little on this technique but you can learn more about it from Effective Go. If there is one thing you should take away from this is that it can be extremely useful but *always thinking about your public API, only expose what's appropriate.*
- **JSON deserializing and serializing**. The standard library makes it very trivial to serialise and deserialise your data. It is also open to configuration and you can customise how these data transformations work if necessary.

IO e sorting

You can find all the code for this chapter here

In the previous chapter we continued iterating on our application by adding a new endpoint `/league`. Along the way we learned about how to deal with JSON, embedding types and routing.

Our product owner is somewhat perturbed by the software losing the scores when the server was restarted. This is because our implementation of our store is in-memory. She is also not pleased that we didn't interpret the `/league` endpoint should return the players ordered by the number of wins!

The code so far

```
// server.go
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
)

// PlayerStore stores score information about players
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
    GetLeague() []Player
}

// Player stores a name with a number of wins
type Player struct {
    Name string
```

```
    Wins int
}

// PlayerServer is a HTTP interface for player information
type PlayerServer struct {
    store PlayerStore
    http.Handler
}

const jsonContentType = "application/json"

// NewPlayerServer creates a PlayerServer with routing configured
func NewPlayerServer(store PlayerStore) *PlayerServer {
    p := new(PlayerServer)

    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    p.Handler = router

    return p
}

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(p.store.GetLeague())
    w.Header().Set("content-type", jsonContentType)
    w.WriteHeader(http.StatusOK)
}

func (p *PlayerServer) playersHandler(w http.ResponseWriter, r *http.Request) {
    player := r.URL.Path[len("/players/"):]

    switch r.Method {
    case http.MethodPost:
        p.processWin(w, player)
    case http.MethodGet:
        p.showScore(w, player)
    }
}

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
    score := p.store.GetPlayerScore(player)
```

```
    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}
```

```
// InMemoryPlayerStore.go
package main

func NewInMemoryPlayerStore() *InMemoryPlayerStore {
    return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct {
    store map[string]int
}

func (i *InMemoryPlayerStore) GetLeague() []Player {
    var league []Player
    for name, wins := range i.store {
        league = append(league, Player{name, wins})
    }
    return league
}

func (i *InMemoryPlayerStore) RecordWin(name string) {
    i.store[name]++
}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return i.store[name]
}
```

```
// main.go
package main

import (
    "log"
    "net/http"
)
```

```
func main() {
    server := NewPlayerServer(NewInMemoryPlayerStore())

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

You can find the corresponding tests in the link at the top of the chapter.

Store the data

There are dozens of databases we could use for this but we're going to go for a very simple approach. We're going to store the data for this application in a file as JSON.

This keeps the data very portable and is relatively simple to implement.

It won't scale especially well but given this is a prototype it'll be fine for now. If our circumstances change and it's no longer appropriate it'll be simple to swap it out for something different because of the `PlayerStore` abstraction we have used.

We will keep the `InMemoryPlayerStore` for now so that the integration tests keep passing as we develop our new store. Once we are confident our new implementation is sufficient to make the integration test pass we will swap it in and then delete `InMemoryPlayerStore`.

Write the test first

By now you should be familiar with the interfaces around the standard library for reading data (`io.Reader`), writing data (`io.Writer`) and how we can use the standard library to test these functions without having to use real files.

For this work to be complete we'll need to implement `PlayerStore` so we'll write tests for our store calling the methods we need to implement. We'll start with `GetLeague`.

```
func TestFileSystemStore(t *testing.T) {

    t.Run("/league from a reader", func(t *testing.T) {
        database := strings.NewReader(`[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)

        store := FileSystemPlayerStore{database}

        got := store.GetLeague()
```

```
        want := []Player{
            {"Cleo", 10},
            {"Chris", 33},
        }

        assertLeague(t, got, want)
    })
}
```

We're using `strings.NewReader` which will return us a `Reader`, which is what our `FileSystemPlayerStore` will use to read data. In `main` we will open a file, which is also a `Reader`.

Try to run the test

```
# github.com/quii/learn-go-with-tests/json-and-io/v7
./FileSystemStore_test.go:15:12: undefined: FileSystemPlayerStore
```

Write the minimal amount of code for the test to run and check the failing test output

Let's define `FileSystemPlayerStore` in a new file

```
type FileSystemPlayerStore struct {}
```

Try again

```
# github.com/quii/learn-go-with-tests/json-and-io/v7
./FileSystemStore_test.go:15:28: too many values in struct initializer
./FileSystemStore_test.go:17:15: store.GetLeague undefined (type FileSystemPlayerStore has no
```

It's complaining because we're passing in a `Reader` but not expecting one and it doesn't have `GetLeague` defined yet.

```
type FileSystemPlayerStore struct {
    database io.Reader
}

func (f *FileSystemPlayerStore) GetLeague() []Player {
    return nil
}
```

One more try...

```
=== RUN   TestFileSystemStore//league_from_a_reader
    --- FAIL: TestFileSystemStore//league_from_a_reader (0.00s)
      FileSystemStore_test.go:24: got [] want [{Cleo 10} {Chris 33}]
```

Write enough code to make it pass

We've read JSON from a reader before

```
func (f *FileSystemPlayerStore) GetLeague() []Player {
    var league []Player
    json.NewDecoder(f.database).Decode(&league)
    return league
}
```

The test should pass.

Refactor

We *have* done this before! Our test code for the server had to decode the JSON from the response.

Let's try DRYing this up into a function.

Create a new file called `league.go` and put this inside.

```
func NewLeague(rdr io.Reader) ([]Player, error) {
    var league []Player
    err := json.NewDecoder(rdr).Decode(&league)
    if err != nil {
        err = fmt.Errorf("problem parsing league, %v", err)
    }

    return league, err
}
```

Call this in our implementation and in our test helper `getLeagueFromResponse` in `server_test.go`

```
func (f *FileSystemPlayerStore) GetLeague() []Player {
    league, _ := NewLeague(f.database)
    return league
}
```

We haven't got a strategy yet for dealing with parsing errors but let's press on.

Seeking problems

There is a flaw in our implementation. First of all, let's remind ourselves how `io.Reader` is defined.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

With our file, you can imagine it reading through byte by byte until the end. What happens if you try and `Read` a second time?

Add the following to the end of our current test.

```
// read again
got = store.GetLeague()
assertLeague(t, got, want)
```

We want this to pass, but if you run the test it doesn't.

The problem is our `Reader` has reached the end so there is nothing more to read. We need a way to tell it to go back to the start.

ReadSeeker is another interface in the standard library that can help.

```
type ReadSeeker interface {
    Reader
    Seeker
}
```

Remember embedding? This is an interface comprised of `Reader` and `Seeker`

```
type Seeker interface {
    Seek(offset int64, whence int) (int64, error)
}
```

This sounds good, can we change `FileSystemPlayerStore` to take this interface instead?

```
type FileSystemPlayerStore struct {
    database io.ReadSeeker
}

func (f *FileSystemPlayerStore) GetLeague() []Player {
    f.database.Seek(0, 0)
    league, _ := NewLeague(f.database)
    return league
}
```

Try running the test, it now passes! Happily for us `string.NewReader` that we used in our test also implements `ReadSeeker` so we didn't have to make any other changes.

Next we'll implement `GetPlayerScore`.

Write the test first

```
t.Run("get player score", func(t *testing.T) {
    database := strings.NewReader(`[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
```

```
    store := FileSystemPlayerStore{database}

    got := store.GetPlayerScore("Chris")

    want := 33

    if got != want {
        t.Errorf("got %d want %d", got, want)
    }
})
```

Try to run the test

```
./FileSystemStore_test.go:38:15:  store.GetPlayerScore  undefined
(type FileSystemPlayerStore has no field or method GetPlayerScore)
```

Write the minimal amount of code for the test to run and check the failing test output

We need to add the method to our new type to get the test to compile.

```
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {
    return 0
}
```

Now it compiles and the test fails

```
=== RUN   TestFileSystemStore/get_player_score
    --- FAIL: TestFileSystemStore//get_player_score (0.00s)
        FileSystemStore_test.go:43: got 0 want 33
```

Write enough code to make it pass

We can iterate over the league to find the player and return their score

```
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {

    var wins int

    for _, player := range f.GetLeague() {
        if player.Name == name {
            wins = player.Wins
            break
        }
    }
```

```
    return wins
}
```

Refactor

You will have seen dozens of test helper refactorings so I'll leave this to you to make it work

```
t.Run("/get player score", func(t *testing.T) {
    database := strings.NewReader(`[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)

    store := FileSystemPlayerStore{database}

    got := store.GetPlayerScore("Chris")
    want := 33
    assertScoreEquals(t, got, want)
})
```

Finally, we need to start recording scores with `RecordWin`.

Write the test first

Our approach is fairly short-sighted for writes. We can't (easily) just update one "row" of JSON in a file. We'll need to store the *whole* new representation of our database on every write.

How do we write? We'd normally use a `Writer` but we already have our `ReadSeeker`. Potentially we could have two dependencies but the standard library already has an interface for us `ReadWriteSeeker` which lets us do all the things we'll need to do with a file.

Let's update our type

```
type FileSystemPlayerStore struct {
    database io.ReadWriteSeeker
}
```

See if it compiles

```
./FileSystemStore_test.go:15:34: cannot use database (type *strings.Reader) as type io.ReadW
    *strings.Reader does not implement io.ReadWriteSeeker (missing Write method)
./FileSystemStore_test.go:36:34: cannot use database (type *strings.Reader) as type io.ReadW
    *strings.Reader does not implement io.ReadWriteSeeker (missing Write method)
```

It's not too surprising that `strings.Reader` does not implement `ReadWriteSeeker` so what do we do?

We have two choices

- Create a temporary file for each test. `*os.File` implements `ReadWriteSeeker`. The pro of this is it becomes more of an integration test, we're really reading and writing from the file system so it will give us a very high level of confidence. The cons are we prefer unit tests because they are faster and generally simpler. We will also need to do more work around creating temporary files and then making sure they're removed after the test.
- We could use a third party library. Mattetti has written a library filebuffer which implements the interface we need and doesn't touch the file system.

I don't think there's an especially wrong answer here, but by choosing to use a third party library I would have to explain dependency management! So we will use files instead.

Before adding our test we need to make our other tests compile by replacing the `strings.Reader` with an `os.File`.

Let's create a helper function which will create a temporary file with some data inside it

```
func createTempFile(t *testing.T, initialData string) (io.ReadWriteSeeker, func()) {
    t.Helper()

    tmpfile, err := ioutil.TempFile("", "db")

    if err != nil {
        t.Fatalf("could not create temp file %v", err)
    }

    tmpfile.Write([]byte(initialData))

    removeFile := func() {
        tmpfile.Close()
        os.Remove(tmpfile.Name())
    }

    return tmpfile, removeFile
}
```

TempFile creates a temporary file for us to use. The `"db"` value we've passed in is a prefix put on a random file name it will create. This is to ensure it won't clash with other files by accident.

You'll notice we're not only returning our `ReadWriteSeeker` (the file) but also a function. We need to make sure that the file is removed once the test is finished. We don't want to leak details of the files into the test as it's prone to error and uninteresting for the reader. By returning a `removeFile` function, we can take care of the details in our helper and all the caller has to do is run `defer`

```
cleanDatabase().

func TestFileSystemStore(t *testing.T) {

    t.Run("league from a reader", func(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)
        defer cleanDatabase()

        store := FileSystemPlayerStore{database}

        got := store.GetLeague()

        want := []Player{
            {"Cleo", 10},
            {"Chris", 33},
        }

        assertLeague(t, got, want)

        // read again
        got = store.GetLeague()
        assertLeague(t, got, want)
    })

    t.Run("get player score", func(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)
        defer cleanDatabase()

        store := FileSystemPlayerStore{database}

        got := store.GetPlayerScore("Chris")
        want := 33
        assertScoreEquals(t, got, want)
    })
}
```

Run the tests and they should be passing! There were a fair amount of changes but now it feels like we have our interface definition complete and it should be very easy to add new tests from now.

Let's get the first iteration of recording a win for an existing player

```
t.Run("store wins for existing players", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[
```

```
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
    defer cleanDatabase()

    store := FileSystemPlayerStore{database}

    store.RecordWin("Chris")

    got := store.GetPlayerScore("Chris")
    want := 34
    assertScoreEquals(t, got, want)
})
```

Try to run the test

```
./FileSystemStore_test.go:67:8: store.RecordWin undefined (type
FileSystemPlayerStore has no field or method RecordWin)
```

Write the minimal amount of code for the test to run and check the failing test output

Add the new method

```
func (f *FileSystemPlayerStore) RecordWin(name string) {

}
```

```
=== RUN   TestFileSystemStore/store_wins_for_existing_players
   --- FAIL: TestFileSystemStore/store_wins_for_existing_players (0.00s)
        FileSystemStore_test.go:71: got 33 want 34
```

Our implementation is empty so the old score is getting returned.

Write enough code to make it pass

```
func (f *FileSystemPlayerStore) RecordWin(name string) {
    league := f.GetLeague()

    for i, player := range league {
        if player.Name == name {
            league[i].Wins++
        }
    }

    f.database.Seek(0,0)
```

```
    json.NewEncoder(f.database).Encode(league)
}
```

You may be asking yourself why I am doing `league[i].Wins++` rather than `player.Wins++`.

When you `range` over a slice you are returned the current index of the loop (in our case `i`) and a *copy* of the element at that index. Changing the `Wins` value of a copy won't have any effect on the `league` slice that we iterate on. For that reason, we need to get the reference to the actual value by doing `league[i]` and then changing that value instead.

If you run the tests, they should now be passing.

Refactor

In `GetPlayerScore` and `RecordWin`, we are iterating over `[]Player` to find a player by name.

We could refactor this common code in the internals of `FileSystemStore` but to me, it feels like this is maybe useful code we can lift into a new type. Working with a "League" so far has always been with `[]Player` but we can create a new type called `League`. This will be easier for other developers to understand and then we can attach useful methods onto that type for us to use.

Inside `league.go` add the following

```
type League []Player

func (l League) Find(name string) *Player {
    for i, p := range l {
        if p.Name==name {
            return &l[i]
        }
    }
    return nil
}
```

Now if anyone has a `League` they can easily find a given player.

Change our `PlayerStore` interface to return `League` rather than `[]Player`. Try and re-run the tests, you'll get a compilation problem because we've changed the interface but it's very easy to fix; just change the return type from `[]Player` to `League`.

This lets us simplify our methods in `FileSystemStore`.

```
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {

    player := f.GetLeague().Find(name)
```

```
    if player != nil {
        return player.Wins
    }

    return 0
}

func (f *FileSystemPlayerStore) RecordWin(name string) {
    league := f.GetLeague()
    player := league.Find(name)

    if player != nil {
        player.Wins++
    }

    f.database.Seek(0, 0)
    json.NewEncoder(f.database).Encode(league)
}
```

This is looking much better and we can see how we might be able to find other useful functionality around `League` can be refactored.

We now need to handle the scenario of recording wins of new players.

Write the test first

```
t.Run("store wins for new players", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
    defer cleanDatabase()

    store := FileSystemPlayerStore{database}

    store.RecordWin("Pepper")

    got := store.GetPlayerScore("Pepper")
    want := 1
    assertScoreEquals(t, got, want)
})
```

Try to run the test

```
=== RUN   TestFileSystemStore/store_wins_for_new_players#01
   --- FAIL: TestFileSystemStore/store_wins_for_new_players#01 (0.00s)
```

```
FileSystemStore_test.go:86: got 0 want 1
```

Write enough code to make it pass

We just need to handle the scenario where `Find` returns `nil` because it couldn't find the player.

```
func (f *FileSystemPlayerStore) RecordWin(name string) {
    league := f.GetLeague()
    player := league.Find(name)

    if player != nil {
        player.Wins++
    } else {
        league = append(league, Player{name, 1})
    }

    f.database.Seek(0, 0)
    json.NewEncoder(f.database).Encode(league)
}
```

The happy path is looking ok so we can now try using our new `Store` in the integration test. This will give us more confidence that the software works and then we can delete the redundant `InMemoryPlayerStore`.

In `TestRecordingWinsAndRetrievingThem` replace the old store.

```
database, cleanDatabase := createTempFile(t, "")
defer cleanDatabase()
store := &FileSystemPlayerStore{database}
```

If you run the test it should pass and now we can delete `InMemoryPlayerStore`. `main.go` will now have compilation problems which will motivate us to now use our new store in the "real" code.

```
package main

import (
    "log"
    "net/http"
    "os"
)

const dbFileName = "game.db.json"

func main() {
    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)
```

```
    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }

    store := &FileSystemPlayerStore{db}
    server := NewPlayerServer(store)

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

- We create a file for our database.
- The 2nd argument to `os.OpenFile` lets you define the permissions for opening the file, in our case `O_RDWR` means we want to read and write *and* `os.O_CREATE` means create the file if it doesn't exist.
- The 3rd argument means sets permissions for the file, in our case, all users can read and write the file. (See superuser.com for a more detailed explanation).

Running the program now persists the data in a file in between restarts, hooray!

More refactoring and performance concerns

Every time someone calls `GetLeague()` or `GetPlayerScore()` we are reading the file from the start, and parsing it into JSON. We should not have to do that because `FileSystemStore` is entirely responsible for the state of the league; we just want to use the file at the start to get the current state and updating it when data changes.

We can create a constructor which can do some of this initialisation for us and store the league as a value in our `FileSystemStore` to be used on the reads instead.

```
type FileSystemPlayerStore struct {
    database io.ReadWriteSeeker
    league League
}

func NewFileSystemPlayerStore(database io.ReadWriteSeeker) *FileSystemPlayerStore {
    database.Seek(0, 0)
    league, _ := NewLeague(database)
    return &FileSystemPlayerStore{
        database:database,
        league:league,
    }
}
```

This way we only have to read from disk once. We can now replace all of our previous calls to getting the league from disk and just use `f.league` instead.

```
func (f *FileSystemPlayerStore) GetLeague() League {
    return f.league
}

func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {

    player := f.league.Find(name)

    if player != nil {
        return player.Wins
    }

    return 0
}

func (f *FileSystemPlayerStore) RecordWin(name string) {
    player := f.league.Find(name)

    if player != nil {
        player.Wins++
    } else {
        f.league = append(f.league, Player{name, 1})
    }

    f.database.Seek(0, 0)
    json.NewEncoder(f.database).Encode(f.league)
}
```

If you try and run the tests it will now complain about initialising `FileSystemPlayerStore` so just fix them by calling our new constructor.

Another problem

There is some more naivety in the way we are dealing with files which *could* create a very nasty bug down the line.

When we `RecordWin` we `Seek` back to the start of the file and then write the new data but what if the new data was smaller than what was there before?

In our current case, this is impossible. We never edit or delete scores so the data can only get bigger but it would be irresponsible for us to leave the code like this, it's not unthinkable that a delete scenario could come up.

How will we test for this though? What we need to do is first refactor our code so we separate out the concern of the *kind of data we write, from the writing.*

We can then test that separately to check it works how we hope.

We'll create a new type to encapsulate our "when we write we go from the beginning" functionality. I'm going to call it `Tape`. Create a new file with the following

```
package main

import "io"

type tape struct {
    file io.ReadWriteSeeker
}

func (t *tape) Write(p []byte) (n int, err error) {
    t.file.Seek(0, 0)
    return t.file.Write(p)
}
```

Notice that we're only implementing `Write` now, as it encapsulates the `Seek` part. This means our `FileSystemStore` can just have a reference to a `Writer` instead.

```
type FileSystemPlayerStore struct {
    database io.Writer
    league   League
}
```

Update the constructor to use `Tape`

```
func NewFileSystemPlayerStore(database io.ReadWriteSeeker) *FileSystemPlayerStore {
    database.Seek(0, 0)
    league, _ := NewLeague(database)

    return &FileSystemPlayerStore{
        database: &tape{database},
        league:   league,
    }
}
```

Finally, we can get the amazing payoff we wanted by removing the `Seek` call from `RecordWin`. Yes, it doesn't feel much, but at least it means if we do any other kind of writes we can rely on our `Write` to behave how we need it to. Plus it will now let us test the potentially problematic code separately and fix it.

Let's write the test where we want to update the entire contents of a file with something that is smaller than the original contents. In `tape_test.go`:

Write the test first

We'll just create a file, try and write to it using our tape, read it all again and see what's in the file

```
func TestTape_Write(t *testing.T) {
    file, clean := createTempFile(t, "12345")
    defer clean()

    tape := &tape{file}

    tape.Write([]byte("abc"))

    file.Seek(0, 0)
    newFileContents, _ := ioutil.ReadAll(file)

    got := string(newFileContents)
    want := "abc"

    if got != want {
        t.Errorf("got '%s' want '%s'", got, want)
    }
}
```

Try to run the test

```
=== RUN   TestTape_Write
--- FAIL: TestTape_Write (0.00s)
    tape_test.go:23: got 'abc45' want 'abc'
```

As we thought! It simply writes the data we want, leaving over the rest.

Write enough code to make it pass

`os.File` has a truncate function that will let us effectively empty the file. We should be able to just call this to get what we want.

Change `tape` to the following

```
type tape struct {
    file *os.File
}

func (t *tape) Write(p []byte) (n int, err error) {
    t.file.Truncate(0)
    t.file.Seek(0, 0)
```

```
    return t.file.Write(p)
}
```

The compiler will fail in a number of places where we are expecting an `io.ReadWriteSeeker` but we are sending in `*os.File`. You should be able to fix these problems yourself by now but if you get stuck just check the source code.

Once you get it refactoring our `TestTape_Write` test should be passing!

One other small refactor

In `RecordWin` we have the line `json.NewEncoder(f.database).Encode(f.league)`.

We don't need to create a new encoder every time we write, we can initialise one in our constructor and use that instead.

Store a reference to an `Encoder` in our type.

```
type FileSystemPlayerStore struct {
    database *json.Encoder
    league   League
}
```

Initialise it in the constructor

```
func NewFileSystemPlayerStore(file *os.File) *FileSystemPlayerStore {
    file.Seek(0, 0)
    league, _ := NewLeague(file)

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }
}
```

Use it in `RecordWin`.

Didn't we just break some rules there? Testing private things? No interfaces?

On testing private types

It's true that *in general* you should favour not testing private things as that can sometimes lead to your tests being too tightly coupled to the implementation; which can hinder refactoring in future.

However, we must not forget that tests should give us *confidence*.

We were not confident that our implementation would work if we added any kind of edit or delete functionality. We did not want to leave the code like that, especially if this was being worked on by more than one person who may not be aware of the shortcomings of our initial approach.

Finally, it's just one test! If we decide to change the way it works it won't be a disaster to just delete the test but we have at the very least captured the requirement for future maintainers.

Interfaces

We started off the code by using `io.Reader` as that was the easiest path for us to unit test our new `PlayerStore`. As we developed the code we moved on to `io.ReadWriter` and then `io.ReadWriteSeeker`. We then found out there was nothing in the standard library that actually implemented that apart from `*os.File`. We could've taken the decision to write our own or use an open source one but it felt pragmatic just to make temporary files for the tests.

Finally, we needed `Truncate` which is also on `*os.File`. It would've been an option to create our own interface capturing these requirements.

```
type ReadWriteSeekTruncate interface {
    io.ReadWriteSeeker
    Truncate(size int64) error
}
```

But what is this really giving us? Bear in mind we are *not mocking* and it is unrealistic for a **file system** store to take any type other than an `*os.File` so we don't need the polymorphism that interfaces give us.

Don't be afraid to chop and change types and experiment like we have here. The great thing about using a statically typed language is the compiler will help you with every change.

Error handling

Before we start working on sorting we should make sure we're happy with our current code and remove any technical debt we may have. It's an important principle to get to working software as quickly as possible (stay out of the red state) but that doesn't mean we should ignore error cases!

If we go back to `FileSystemStore.go` we have `league, _ := NewLeague(f.database)` in our constructor.

`NewLeague` can return an error if it is unable to parse the league from the `io.Reader` that we provide.

It was pragmatic to ignore that at the time as we already had failing tests. If we had tried to tackle it at the same time we would be juggling two things at

once.

Let's make it so if our constructor is capable of returning an error.

```
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {
    file.Seek(0, 0)
    league, err := NewLeague(file)

    if err != nil {
        return nil, fmt.Errorf("problem loading player store from file %s, %v", file.Name()
    }

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }, nil
}
```

Remember it is very important to give helpful error messages (just like your tests). People jokingly on the internet say most Go code is

```
if err != nil {
    return err
}
```

That is 100% not idiomatic. Adding contextual information (i.e what you were doing to cause the error) to your error messages makes operating your software far easier.

If you try and compile you'll get some errors.

```
./main.go:18:35: multiple-value NewFileSystemPlayerStore() in single-value context
./FileSystemStore_test.go:35:36: multiple-value NewFileSystemPlayerStore() in single-value co
./FileSystemStore_test.go:57:36: multiple-value NewFileSystemPlayerStore() in single-value co
./FileSystemStore_test.go:70:36: multiple-value NewFileSystemPlayerStore() in single-value co
./FileSystemStore_test.go:85:36: multiple-value NewFileSystemPlayerStore() in single-value co
./server_integration_test.go:12:35: multiple-value NewFileSystemPlayerStore() in single-value
```

In main we'll want to exit the program, printing the error.

```
store, err := NewFileSystemPlayerStore(db)

if err != nil {
    log.Fatalf("problem creating file system player store, %v ", err)
}
```

In the tests we should assert there is no error. We can make a helper to help with this.

```
func assertNoError(t *testing.T, err error) {
    t.Helper()
    if err != nil {
```

```
        t.Fatalf("didnt expect an error but got one, %v", err)
    }
}
```

Work through the other compilation problems using this helper. Finally, you should have a failing test

```
=== RUN   TestRecordingWinsAndRetrievingThem
--- FAIL: TestRecordingWinsAndRetrievingThem (0.00s)
   server_integration_test.go:14: didnt expect an error but got one, problem loading player stor
```

We cannot parse the league because the file is empty. We weren't getting errors before because we always just ignored them.

Let's fix our big integration test by putting some valid JSON in it and then we can write a specific test for this scenario.

```
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[]`)
    //etc...
```

Now all the tests are passing we need to handle the scenario where the file is empty.

Write the test first

```
t.Run("works with an empty file", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, "")
    defer cleanDatabase()

    _, err := NewFileSystemPlayerStore(database)

    assertNoError(t, err)
})
```

Try to run the test

```
=== RUN   TestFileSystemStore/works_with_an_empty_file
    --- FAIL: TestFileSystemStore/works_with_an_empty_file (0.00s)
     FileSystemStore_test.go:108: didnt expect an error but got one, problem loading player sto
```

Write enough code to make it pass

Change our constructor to the following

```
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {

    file.Seek(0, 0)
```

```
    info, err := file.Stat()

    if err != nil {
        return nil, fmt.Errorf("problem getting file info from file %s, %v", file.Name(), er
    }

    if info.Size() == 0 {
        file.Write([]byte("[]"))
        file.Seek(0, 0)
    }

    league, err := NewLeague(file)

    if err != nil {
        return nil, fmt.Errorf("problem loading player store from file %s, %v", file.Name()
    }

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }, nil
}
```

`file.Stat` returns stats on our file. This lets us check the size of the file, if it's empty we `Write` an empty JSON array and `Seek` back to the start ready for the rest of the code.

Refactor

Our constructor is a bit messy now, we can extract the initialise code into a function

```
func initialisePlayerDBFile(file *os.File) error {
    file.Seek(0, 0)

    info, err := file.Stat()

    if err != nil {
        return fmt.Errorf("problem getting file info from file %s, %v", file.Name(), err)
    }

    if info.Size()==0 {
        file.Write([]byte("[]"))
        file.Seek(0, 0)
    }
```

```
    return nil
}

func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {

    err := initialisePlayerDBFile(file)

    if err != nil {
        return nil, fmt.Errorf("problem initialising player db file, %v", err)
    }

    league, err := NewLeague(file)

    if err != nil {
        return nil, fmt.Errorf("problem loading player store from file %s, %v", file.Name(),
    }

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }, nil
}
```

Sorting

Our product owner wants `/league` to return the players sorted by their scores.

The main decision to make here is where in the software should this happen. If we were using a "real" database we would use things like `ORDER BY` so the sorting is super fast so for that reason it feels like implementations of `PlayerStore` should be responsible.

Write the test first

We can update the assertion on our first test in `TestFileSystemStore`

```
t.Run("league sorted", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
    defer cleanDatabase()

    store := FileSystemPlayerStore{database}

    got := store.GetLeague()
```

```
    want := []Player{
        {"Chris", 33},
        {"Cleo", 10},
    }

    assertLeague(t, got, want)

    // read again
    got = store.GetLeague()
    assertLeague(t, got, want)
})
```

The order of the JSON coming in is in the wrong order and our `want` will check that it is returned to the caller in the correct order.

Try to run the test

```
=== RUN   TestFileSystemStore/league_from_a_reader,_sorted
   --- FAIL: TestFileSystemStore/league_from_a_reader,_sorted (0.00s)
      FileSystemStore_test.go:46: got [{Cleo 10} {Chris 33}] want [{Chris 33} {Cleo 10}]
      FileSystemStore_test.go:51: got [{Cleo 10} {Chris 33}] want [{Chris 33} {Cleo 10}]
```

Write enough code to make it pass

```
func (f *FileSystemPlayerStore) GetLeague() League {
    sort.Slice(f.league, func(i, j int) bool {
        return f.league[i].Wins > f.league[j].Wins
    })
    return f.league
}
```

`sort.Slice`

> Slice sorts the provided slice given the provided less function.

Easy!

Wrapping up

What we've covered

- The `Seeker` interface and its relation with `Reader` and `Writer`.
- Working with files.
- Creating an easy to use helper for testing with files that hides all the messy stuff.

- `sort.Slice` for sorting slices.
- Using the compiler to help us make structural changes to the application safely.

Breaking rules

- Most rules in software engineering aren't really rules, just best practices that work 80% of the time.
- We discovered a scenario where one of our previous "rules" of not testing internal functions was not helpful for us so we broke the rule.
- It's important when breaking rules to understand the trade-off you are making. In our case, we were ok with it because it was just one test and would've been very difficult to exercise the scenario otherwise.
- In order to be able to break the rules **you must understand them first**. An analogy is with learning guitar. It doesn't matter how creative you think you are, you must understand and practice the fundamentals.

Where our software is at

- We have an HTTP API where you can create players and increment their score.
- We can return a league of everyone's scores as JSON.
- The data is persisted as a JSON file.

Linha de comando e estrutura de pacotes

You can find all the code for this chapter here

Our product owner now wants to *pivot* by introducing a second application - a command line application.

For now, it will just need to be able to record a player's win when the user types `Ruth wins`. The intention is to eventually be a tool for helping users play poker.

The product owner wants the database to be shared amongst the two applications so that the league updates according to wins recorded in the new application.

A reminder of the code

We have an application with a `main.go` file that launches an HTTP server. The HTTP server won't be interesting to us for this exercise but the abstraction it uses will. It depends on a `PlayerStore`.

```
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
    GetLeague() League
}
```

In the previous chapter, we made a `FileSystemPlayerStore` which implements that interface. We should be able to re-use some of this for our new application.

Some project refactoring first

Our project now needs to create two binaries, our existing web server and the command line app.

Before we get stuck into our new work we should structure our project to accommodate this.

So far all the code has lived in one folder, in a path looking like this

`$GOPATH/src/github.com/your-name/my-app`

In order for you to make an application in Go, you need a `main` function inside a `package main`. So far all of our "domain" code has lived inside `package main` and our `func main` can reference everything.

This was fine so far and it is good practice not to go over-the-top with package structure. If you take the time to look through the standard library you will see very little in the way of lots of folders and structure.

Thankfully it's pretty straightforward to add structure *when you need it*.

Inside the existing project create a `cmd` directory with a `webserver` directory inside that (e.g `mkdir -p cmd/webserver`).

Move the `main.go` inside there.

If you have `tree` installed you should run it and your structure should look like this

```
.
  FileSystemStore.go
  FileSystemStore_test.go
  cmd
      webserver
          main.go
  league.go
  server.go
  server_integration_test.go
  server_test.go
  tape.go
  tape_test.go
```

We now effectively have a separation between our application and the library code but we now need to change some package names. Remember when you build a Go application its package *must* be `main`.

Change all the other code to have a package called `poker`.

Finally, we need to import this package into `main.go` so we can use it to create our web server. Then we can use our library code by using `poker.FunctionName`.

The paths will be different on your computer, but it should be similar to this:

```
package main

import (
    "log"
    "net/http"
    "os"
    "github.com/quii/learn-go-with-tests/criando-uma-aplicacao/command-line/v1"
)

const dbFileName = "game.db.json"

func main() {
    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }

    store, err := poker.NewFileSystemPlayerStore(db)

    if err != nil {
        log.Fatalf("problem creating file system player store, %v ", err)
    }

    server := poker.NewPlayerServer(store)

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

The full path may seem a bit jarring, but this is how you can import *any* publicly available library into your code.

By separating our domain code into a separate package and committing it to a public repo like GitHub any Go developer can write their own code which imports that package the features we've written available. The first time you

try and run it will complain it is not existing but all you need to do is run `go get`.

In addition, users can view the documentation at godoc.org.

Final checks

- Inside the root run `go test` and check they're still passing
- Go inside our `cmd/webserver` and do `go run main.go`
- Visit `http://localhost:5000/league` and you should see it's still working

Walking skeleton

Before we get stuck into writing tests, let's add a new application that our project will build. Create another directory inside `cmd` called `cli` (command line interface) and add a `main.go` with the following

```
package main

import "fmt"

func main() {
    fmt.Println("Let's play poker")
}
```

The first requirement we'll tackle is recording a win when the user types `{PlayerName} wins`.

Write the test first

We know we need to make something called `CLI` which will allow us to `Play` poker. It'll need to read user input and then record wins to a `PlayerStore`.

Before we jump too far ahead though, let's just write a test to check it integrates with the `PlayerStore` how we'd like.

Inside `CLI_test.go` (in the root of the project, not inside `cmd`)

```
func TestCLI(t *testing.T) {
    playerStore := &StubPlayerStore{}
    cli := &CLI{playerStore}
    cli.PlayPoker()

    if len(playerStore.winCalls) !=1 {
        t.Fatal("expected a win call but didn't get any")
```

```
    }
}
```

- We can use our `StubPlayerStore` from other tests
- We pass in our dependency into our not yet existing `CLI` type
- Trigger the game by an unwritten `PlayPoker` method
- Check that a win is recorded

Try to run the test

```
# github.com/quii/learn-go-with-tests/criando-uma-aplicacao/command-line/v2
./cli_test.go:25:10: undefined: CLI
```

Write the minimal amount of code for the test to run and check the failing test output

At this point, you should be comfortable enough to create our new `CLI` struct with the respective field for our dependency and add a method.

You should end up with code like this

```
type CLI struct {
    playerStore PlayerStore
}

func (cli *CLI) PlayPoker() {}
```

Remember we're just trying to get the test running so we can check the test fails how we'd hope

```
--- FAIL: TestCLI (0.00s)
    cli_test.go:30: expected a win call but didn't get any
FAIL
```

Write enough code to make it pass

```
func (cli *CLI) PlayPoker() {
    cli.playerStore.RecordWin("Cleo")
}
```

That should make it pass.

Next, we need to simulate reading from `Stdin` (the input from the user) so that we can record wins for specific players.

Let's extend our test to exercise this.

Write the test first

```
func TestCLI(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &StubPlayerStore{}

    cli := &CLI{playerStore, in}
    cli.PlayPoker()

    if len(playerStore.winCalls) < 1 {
        t.Fatal("expected a win call but didn't get any")
    }

    got := playerStore.winCalls[0]
    want := "Chris"

    if got != want {
        t.Errorf("didn't record correct winner, got '%s', want '%s'", got, want)
    }
}
```

`os.Stdin` is what we'll use in `main` to capture the user's input. It is a `*File` under the hood which means it implements `io.Reader` which as we know by now is a handy way of capturing text.

We create an `io.Reader` in our test using the handy `strings.NewReader`, filling it with what we expect the user to type.

Try to run the test

```
./CLI_test.go:12:32: too many values in struct initializer
```

Write the minimal amount of code for the test to run and check the failing test output

We need to add our new dependency into `CLI`.

```
type CLI struct {
    playerStore PlayerStore
    in io.Reader
}
```

Write enough code to make it pass

```
--- FAIL: TestCLI (0.00s)
```

```
   CLI_test.go:23: didn't record the correct winner, got 'Cleo', want 'Chris'
FAIL
```

Remember to do the strictly easiest thing first

```
func (cli *CLI) PlayPoker() {
    cli.playerStore.RecordWin("Chris")
}
```

The test passes. We'll add another test to force us to write some real code next, but first, let's refactor.

Refactor

In `server_test` we earlier did checks to see if wins are recorded as we have here. Let's DRY that assertion up into a helper

```
func assertPlayerWin(t *testing.T, store *StubPlayerStore, winner string) {
    t.Helper()

    if len(store.winCalls) != 1 {
        t.Fatalf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
    }

    if store.winCalls[0] != winner {
        t.Errorf("did not store correct winner got '%s' want '%s'", store.winCalls[0], winne
    }
}
```

Now replace the assertions in both `server_test.go` and `CLI_test.go`.

The test should now read like so

```
func TestCLI(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &StubPlayerStore{}

    cli := &CLI{playerStore, in}
    cli.PlayPoker()

    assertPlayerWin(t, playerStore, "Chris")
}
```

Now let's write *another* test with different user input to force us into actually reading it.

Write the test first

```
func TestCLI(t *testing.T) {

    t.Run("record chris win from user input", func(t *testing.T) {
        in := strings.NewReader("Chris wins\n")
        playerStore := &StubPlayerStore{}

        cli := &CLI{playerStore, in}
        cli.PlayPoker()

        assertPlayerWin(t, playerStore, "Chris")
    })

    t.Run("record cleo win from user input", func(t *testing.T) {
        in := strings.NewReader("Cleo wins\n")
        playerStore := &StubPlayerStore{}

        cli := &CLI{playerStore, in}
        cli.PlayPoker()

        assertPlayerWin(t, playerStore, "Cleo")
    })

}
```

Try to run the test

```
=== RUN   TestCLI
--- FAIL: TestCLI (0.00s)
=== RUN   TestCLI/record_chris_win_from_user_input
    --- PASS: TestCLI/record_chris_win_from_user_input (0.00s)
=== RUN   TestCLI/record_cleo_win_from_user_input
    --- FAIL: TestCLI/record_cleo_win_from_user_input (0.00s)
     CLI_test.go:27: did not store correct winner got 'Chris' want 'Cleo'
FAIL
```

Write enough code to make it pass

We'll use a `bufio.Scanner` to read the input from the `io.Reader`.

> Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.

Update the code to the following

```
type CLI struct {
    playerStore PlayerStore
    in          io.Reader
}

func (cli *CLI) PlayPoker() {
    reader := bufio.NewScanner(cli.in)
    reader.Scan()
    cli.playerStore.RecordWin(extractWinner(reader.Text()))
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins", "", 1)
}
```

The tests will now pass.

- `Scanner.Scan()` will read up to a newline.
- We then use `Scanner.Text()` to return the `string` the scanner read to.

Now that we have some passing tests, we should wire this up into `main`. Remember we should always strive to have fully-integrated working software as quickly as we can.

In `main.go` add the following and run it. (you may have to adjust the path of the second dependency to match what's on your computer)

```
package main

import (
    "fmt"
    "github.com/quii/learn-go-with-tests/criando-uma-aplicacao/command-line/v3"
    "log"
    "os"
)

const dbFileName = "game.db.json"

func main() {
    fmt.Println("Let's play poker")
    fmt.Println("Type {Name} wins to record a win")

    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }
```

```
    store, err := poker.NewFileSystemPlayerStore(db)

    if err != nil {
        log.Fatalf("problem creating file system player store, %v ", err)
    }

    game := poker.CLI{store, os.Stdin}
    game.PlayPoker()
}
```

You should get an error

```
command-line/v3/cmd/cli/main.go:32:25: implicit assignment of unexported field 'playerStore' i
command-line/v3/cmd/cli/main.go:32:34: implicit assignment of unexported field 'in' in poker.C
```

What's happening here is because we are trying to assign to the fields `playerStore` and `in` in `CLI`. These are unexported (private) fields. We *could* do this in our test code because our test is in the same package as `CLI` (`poker`). But our `main` is in package `main` so it does not have access.

This highlights the importance of *integrating your work*. We rightfully made the dependencies of our `CLI` private (because we don't want them exposed to users of `CLI`s) but haven't made a way for users to construct it.

Is there a way to have caught this problem earlier?

```
package mypackage_test
```

In all other examples so far, when we make a test file we declare it as being in the same package that we are testing.

This is fine and it means on the odd occasion where we want to test something internal to the package we have access to the unexported types.

But given we have advocated for *not* testing internal things *generally*, can Go help enforce that? What if we could test our code where we only have access to the exported types (like our `main` does)?

When you're writing a project with multiple packages I would strongly recommend that your test package name has `_test` at the end. When you do this you will only be able to have access to the public types in your package. This would help with this specific case but also helps enforce the discipline of only testing public APIs. If you still wish to test internals you can make a separate test with the package you want to test.

An adage with TDD is that if you cannot test your code then it is probably hard for users of your code to integrate with it. Using `package foo_test` will help with this by forcing you to test your code as if you are importing it like users of your package will.

Before fixing `main` let's change the package of our test inside `CLI_test.go` to `poker_test`.

If you have a well-configured IDE you will suddenly see a lot of red! If you run the compiler you'll get the following errors

```
./CLI_test.go:12:19: undefined: StubPlayerStore
./CLI_test.go:17:3: undefined: assertPlayerWin
./CLI_test.go:22:19: undefined: StubPlayerStore
./CLI_test.go:27:3: undefined: assertPlayerWin
```

We have now stumbled into more questions on package design. In order to test our software we made unexported stubs and helper functions which are no longer available for us to use in our `CLI_test` because the helpers are defined in the `_test.go` files in the `poker` package.

Do we want to have our stubs and helpers 'public'?

This is a subjective discussion. One could argue that you do not want to pollute your package's API with code to facilitate tests.

In the presentation "Advanced Testing with Go" by Mitchell Hashimoto, it is described how at HashiCorp they advocate doing this so that users of the package can write tests without having to re-invent the wheel writing stubs. In our case, this would mean anyone using our `poker` package won't have to create their own stub `PlayerStore` if they wish to work with our code.

Anecdotally I have used this technique in other shared packages and it has proved extremely useful in terms of users saving time when integrating with our packages.

So let's create a file called `testing.go` and add our stub and our helpers.

```
package poker

import "testing"

type StubPlayerStore struct {
    scores   map[string]int
    winCalls []string
    league   []Player
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
    score := s.scores[name]
    return score
}

func (s *StubPlayerStore) RecordWin(name string) {
    s.winCalls = append(s.winCalls, name)
```

```
}

func (s *StubPlayerStore) GetLeague() League {
    return s.league
}

func AssertPlayerWin(t *testing.T, store *StubPlayerStore, winner string) {
    t.Helper()

    if len(store.winCalls) != 1 {
        t.Fatalf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
    }

    if store.winCalls[0] != winner {
        t.Errorf("did not store correct winner got '%s' want '%s'", store.winCalls[0], winne
    }
}

// todo for you - the rest of the helpers
```

You'll need to make the helpers public (remember exporting is done with a capital letter at the start) if you want them to be exposed to importers of our package.

In our `CLI` test you'll need to call the code as if you were using it within a different package.

```
func TestCLI(t *testing.T) {

    t.Run("record chris win from user input", func(t *testing.T) {
        in := strings.NewReader("Chris wins\n")
        playerStore := &poker.StubPlayerStore{}

        cli := &poker.CLI{playerStore, in}
        cli.PlayPoker()

        poker.AssertPlayerWin(t, playerStore, "Chris")
    })

    t.Run("record cleo win from user input", func(t *testing.T) {
        in := strings.NewReader("Cleo wins\n")
        playerStore := &poker.StubPlayerStore{}

        cli := &poker.CLI{playerStore, in}
        cli.PlayPoker()

        poker.AssertPlayerWin(t, playerStore, "Cleo")
```

```
    })

}
```

You'll now see we have the same problems as we had in `main`

```
./CLI_test.go:15:26: implicit assignment of unexported field 'playerStore' in poker.CLI literal
./CLI_test.go:15:39: implicit assignment of unexported field 'in' in poker.CLI literal
./CLI_test.go:25:26: implicit assignment of unexported field 'playerStore' in poker.CLI literal
./CLI_test.go:25:39: implicit assignment of unexported field 'in' in poker.CLI literal
```

The easiest way to get around this is to make a constructor as we have for other types. We'll also change `CLI` so it stores a `bufio.Scanner` instead of the reader as it's now automatically wrapped at construction time.

```
type CLI struct {
    playerStore PlayerStore
    in          *bufio.Scanner
}

func NewCLI(store PlayerStore, in io.Reader) *CLI {
    return &CLI{
        playerStore: store,
        in:          bufio.NewScanner(in),
    }
}
```

By doing this, we can then simplify and refactor our reading code

```
func (cli *CLI) PlayPoker() {
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins", "", 1)
}

func (cli *CLI) readLine() string {
    cli.in.Scan()
    return cli.in.Text()
}
```

Change the test to use the constructor instead and we should be back to the tests passing.

Finally, we can go back to our new `main.go` and use the constructor we just made

```
game := poker.NewCLI(store, os.Stdin)
```

Try and run it, type "Bob wins".

Refactor

We have some repetition in our respective applications where we are opening a file and creating a `FileSystemStore` from its contents. This feels like a slight weakness in our package's design so we should make a function in it to encapsulate opening a file from a path and returning you the `PlayerStore`.

```
func FileSystemPlayerStoreFromFile(path string) (*FileSystemPlayerStore, func(), error) {
    db, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        return nil, nil, fmt.Errorf("problem opening %s %v", path, err)
    }

    closeFunc := func() {
        db.Close()
    }

    store, err := NewFileSystemPlayerStore(db)

    if err != nil {
        return nil, nil, fmt.Errorf("problem creating file system player store, %v ", err)
    }

    return store, closeFunc, nil
}
```

Now refactor both of our applications to use this function to create the store.

CLI application code

```
package main

import (
    "github.com/quii/learn-go-with-tests/criando-uma-aplicacao/command-line/v3"
    "log"
    "os"
    "fmt"
)

const dbFileName = "game.db.json"

func main() {
    store, close, err := poker.FileSystemPlayerStoreFromFile(dbFileName)
```

```
    if err != nil {
        log.Fatal(err)
    }
    defer close()

    fmt.Println("Let's play poker")
    fmt.Println("Type {Name} wins to record a win")
    poker.NewCLI(store, os.Stdin).PlayPoker()
}
```

Web server application code

```
package main

import (
    "github.com/quii/learn-go-with-tests/criando-uma-aplicacao/command-line/v3"
    "log"
    "net/http"
)

const dbFileName = "game.db.json"

func main() {
    store, close, err := poker.FileSystemPlayerStoreFromFile(dbFileName)

    if err != nil {
        log.Fatal(err)
    }
    defer close()

    server := poker.NewPlayerServer(store)

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

Notice the symmetry: despite being different user interfaces the setup is almost identical. This feels like good validation of our design so far. And notice also that `FileSystemPlayerStoreFromFile` returns a closing function, so we can close the underlying file once we are done using the Store.

Wrapping up

Package structure

This chapter meant we wanted to create two applications, re-using the domain code we've written so far. In order to do this, we needed to update our package structure so that we had separate folders for our respective `main`s.

By doing this we ran into integration problems due to unexported values so this further demonstrates the value of working in small "slices" and integrating often.

We learned how `mypackage_test` helps us create a testing environment which is the same experience for other packages integrating with your code, to help you catch integration problems and see how easy (or not!) your code is to work with.

Reading user input

We saw how reading from `os.Stdin` is very easy for us to work with as it implements `io.Reader`. We used `bufio.Scanner` to easily read line by line user input.

Simple abstractions leads to simpler code re-use

It was almost no effort to integrate `PlayerStore` into our new application (once we had made the package adjustments) and subsequently testing was very easy too because we decided to expose our stub version too.

Tempo

You can find all the code for this chapter here

The product owner wants us to expand the functionality of our command line application by helping a group of people play Texas-Holdem Poker.

Just enough information on poker

You wont need to know much about poker, only that at certain time intervals all the players need to be informed of a steadily increasing "blind" value.

Our application will help keep track of when the blind should go up, and how much it should be.

- When it starts it asks how many players are playing. This determines the amount of time there is before the "blind" bet goes up.
- There is a base amount of time of 5 minutes.
- For every player, 1 minute is added.
- e.g 6 players equals 11 minutes for the blind.
- After the blind time expires the game should alert the players the new amount the blind bet is.
- The blind starts at 100 chips, then 200, 400, 600, 1000, 2000 and continue to double until the game ends (our previous functionality of "Ruth wins" should still finish the game)

Reminder of the code

In the previous chapter we made our start to the command line application which already accepts a command of `{name} wins`. Here is what the current `CLI` code looks like, but be sure to familiarise yourself with the other code too before starting.

```
type CLI struct {
    playerStore PlayerStore
    in          *bufio.Scanner
}

func NewCLI(store PlayerStore, in io.Reader) *CLI {
    return &CLI{
        playerStore: store,
        in:          bufio.NewScanner(in),
    }
}

func (cli *CLI) PlayPoker() {
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins", "", 1)
}

func (cli *CLI) readLine() string {
    cli.in.Scan()
    return cli.in.Text()
}
```

`time.AfterFunc`

We want to be able to schedule our program to print the blind bet values at certain durations dependant on the number of players.

To limit the scope of what we need to do, we'll forget about the number of players part for now and just assume there are 5 players so we'll test that *every 10 minutes the new value of the blind bet is printed.*

As usual the standard library has us covered with `func AfterFunc(d Duration, f func()) *Timer`

> `AfterFunc` waits for the duration to elapse and then calls f in its own goroutine. It returns a `Timer` that can be used to cancel the call using its Stop method.

`time.Duration`

> A Duration represents the elapsed time between two instants as an int64 nanosecond count.

The time library has a number of constants to let you multiply those nanoseconds so they're a bit more readable for the kind of scenarios we'll be doing

```
5 * time.Second
```

When we call `PlayPoker` we'll schedule all of our blind alerts.

Testing this may be a little tricky though. We'll want to verify that each time period is scheduled with the correct blind amount but if you look at the signature of `time.AfterFunc` its second argument is the function it will run. You cannot compare functions in Go so we'd be unable to test what function has been sent in. So we'll need to write some kind of wrapper around `time.AfterFunc` which will take the time to run and the amount to print so we can spy on that.

Write the test first

Add a new test to our suite

```
t.Run("it schedules printing of blind values", func(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &poker.StubPlayerStore{}
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(playerStore, in, blindAlerter)
    cli.PlayPoker()

    if len(blindAlerter.alerts) != 1 {
        t.Fatal("expected a blind alert to be scheduled")
```

```
    }
})
```

You'll notice we've made a `SpyBlindAlerter` which we are trying to inject into our `CLI` and then checking that after we call `PlayerPoker` that an alert is scheduled.

(Remember we are just going for the simplest scenario first and then we'll iterate.)

Here's the definition of `SpyBlindAlerter`

```
type SpyBlindAlerter struct {
    alerts []struct{
        scheduledAt time.Duration
        amount int
    }
}

func (s *SpyBlindAlerter) ScheduleAlertAt(duration time.Duration, amount int) {
    s.alerts = append(s.alerts, struct {
        scheduledAt time.Duration
        amount int
    }{duration,  amount})
}
```

Try to run the test

```
./CLI_test.go:32:27: too many arguments in call to poker.NewCLI
  have (*poker.StubPlayerStore, *strings.Reader, *SpyBlindAlerter)
   want (poker.PlayerStore, io.Reader)
```

Write the minimal amount of code for the test to run and check the failing test output

We have added a new argument and the compiler is complaining. *Strictly speaking* the minimal amount of code is to make `NewCLI` accept a `*SpyBlindAlerter` but let's cheat a little and just define the dependency as an interface.

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}
```

And then add it to the constructor

```
func NewCLI(store PlayerStore, in io.Reader, alerter BlindAlerter) *CLI
```

Your other tests will now fail as they dont have a `BlindAlerter` passed in to `NewCLI`.

Spying on BlindAlerter is not relevant for the other tests so in the test file add

```
var dummySpyAlerter = &SpyBlindAlerter{}
```

Then use that in the other tests to fix the compilation problems. By labelling it as a "dummy" it is clear to the reader of the test that it is not important.

> Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.

The tests should now compile and our new test fails.

```
=== RUN   TestCLI
=== RUN   TestCLI/it_schedules_printing_of_blind_values
--- FAIL: TestCLI (0.00s)
    --- FAIL: TestCLI/it_schedules_printing_of_blind_values (0.00s)
        CLI_test.go:38: expected a blind alert to be scheduled
```

Write enough code to make it pass

We'll need to add the `BlindAlerter` as a field on our `CLI` so we can reference it in our `PlayPoker` method.

```
type CLI struct {
    playerStore PlayerStore
    in          *bufio.Reader
    alerter     BlindAlerter
}

func NewCLI(store PlayerStore, in io.Reader, alerter BlindAlerter) *CLI {
    return &CLI{
        playerStore: store,
        in:          bufio.NewReader(in),
        alerter:     alerter,
    }
}
```

To make the test pass, we can call our `BlindAlerter` with anything we like

```
func (cli *CLI) PlayPoker() {
    cli.alerter.ScheduleAlertAt(5 * time.Second, 100)
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}
```

Next we'll want to check it schedules all the alerts we'd hope for, for 5 players

Write the test first

```
t.Run("it schedules printing of blind values", func(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &poker.StubPlayerStore{}
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(playerStore, in, blindAlerter)
    cli.PlayPoker()

    cases := []struct{
        expectedScheduleTime time.Duration
        expectedAmount       int
    } {
        {0 * time.Second, 100},
        {10 * time.Minute, 200},
        {20 * time.Minute, 300},
        {30 * time.Minute, 400},
        {40 * time.Minute, 500},
        {50 * time.Minute, 600},
        {60 * time.Minute, 800},
        {70 * time.Minute, 1000},
        {80 * time.Minute, 2000},
        {90 * time.Minute, 4000},
        {100 * time.Minute, 8000},
    }

    for i, c := range cases {
        t.Run(fmt.Sprintf("%d scheduled for %v", c.expectedAmount, c.expectedScheduleTim

            if len(blindAlerter.alerts) <= i {
                t.Fatalf("alert %d was not scheduled %v", i, blindAlerter.alerts)
            }

            alert := blindAlerter.alerts[i]

            amountGot := alert.amount
            if amountGot != c.expectedAmount {
                t.Errorf("got amount %d, want %d", amountGot, c.expectedAmount)
            }

            gotScheduledTime := alert.scheduledAt
            if gotScheduledTime != c.expectedScheduleTime {
                t.Errorf("got scheduled time of %v, want %v", gotScheduledTime, c.expect
            }
        })
```

```
        }
    })
```

Table-based test works nicely here and clearly illustrate what our requirements are. We run through the table and check the `SpyBlindAlerter` to see if the alert has been scheduled with the correct values.

Try to run the test

You should have a lot of failures looking like this

```
=== RUN   TestCLI
--- FAIL: TestCLI (0.00s)
=== RUN   TestCLI/it_schedules_printing_of_blind_values
    --- FAIL: TestCLI/it_schedules_printing_of_blind_values (0.00s)
=== RUN   TestCLI/it_schedules_printing_of_blind_values/100_scheduled_for_0s
        --- FAIL: TestCLI/it_schedules_printing_of_blind_values/100_scheduled_for_0s (0.00s)
            CLI_test.go:71: got scheduled time of 5s, want 0s
=== RUN   TestCLI/it_schedules_printing_of_blind_values/200_scheduled_for_10m0s
        --- FAIL: TestCLI/it_schedules_printing_of_blind_values/200_scheduled_for_10m0s (0.0
            CLI_test.go:59: alert 1 was not scheduled [{5000000000 100}]
```

Write enough code to make it pass

```
func (cli *CLI) PlayPoker() {

    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
    for _, blind := range blinds {
        cli.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + 10 * time.Minute
    }

    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}
```

It's not a lot more complicated than what we already had. We're just now iterating over an array of `blinds` and calling the scheduler on an increasing `blindTime`

Refactor

We can encapsulate our scheduled alerts into a method just to make `PlayPoker` read a little clearer.

```
func (cli *CLI) PlayPoker() {
    cli.scheduleBlindAlerts()
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func (cli *CLI) scheduleBlindAlerts() {
    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
    for _, blind := range blinds {
        cli.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + 10*time.Minute
    }
}
```

Finally our tests are looking a little clunky. We have two anonymous structs representing the same thing, a `ScheduledAlert`. Let's refactor that into a new type and then make some helpers to compare them.

```
type scheduledAlert struct {
    at time.Duration
    amount int
}

func (s scheduledAlert) String() string {
    return fmt.Sprintf("%d chips at %v", s.amount, s.at)
}

type SpyBlindAlerter struct {
    alerts []scheduledAlert
}

func (s *SpyBlindAlerter) ScheduleAlertAt(at time.Duration, amount int) {
    s.alerts = append(s.alerts, scheduledAlert{at, amount})
}
```

We've added a `String()` method to our type so it prints nicely if the test fails

Update our test to use our new type

```
t.Run("it schedules printing of blind values", func(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &poker.StubPlayerStore{}
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(playerStore, in, blindAlerter)
    cli.PlayPoker()
```

```
    cases := []scheduledAlert {
        {0 * time.Second, 100},
        {10 * time.Minute, 200},
        {20 * time.Minute, 300},
        {30 * time.Minute, 400},
        {40 * time.Minute, 500},
        {50 * time.Minute, 600},
        {60 * time.Minute, 800},
        {70 * time.Minute, 1000},
        {80 * time.Minute, 2000},
        {90 * time.Minute, 4000},
        {100 * time.Minute, 8000},
    }

    for i, want := range cases {
        t.Run(fmt.Sprint(want), func(t *testing.T) {

            if len(blindAlerter.alerts) <= i {
                t.Fatalf("alert %d was not scheduled %v", i, blindAlerter.alerts)
            }

            got := blindAlerter.alerts[i]
            assertScheduledAlert(t, got, want)
        })
    }
})
```

Implement `assertScheduledAlert` yourself.

We've spent a fair amount of time here writing tests and have been somewhat naughty not integrating with our application. Let's address that before we pile on any more requirements.

Try running the app and it wont compile, complaining about not enough args to `NewCLI`.

Let's create an implementation of `BlindAlerter` that we can use in our application.

Create `BlindAlerter.go` and move our `BlindAlerter` interface and add the new things below

```
package poker

import (
    "time"
    "fmt"
    "os"
)
```

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}

type BlindAlerterFunc func(duration time.Duration, amount int)

func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int) {
    a(duration, amount)
}

func StdOutAlerter(duration time.Duration, amount int) {
    time.AfterFunc(duration, func() {
        fmt.Fprintf(os.Stdout, "Blind is now %d\n", amount)
    })
}
```

Remember that any *type* can implement an interface, not just `structs`. If you are making a library that exposes an interface with one function defined it is a common idiom to also expose a `MyInterfaceFunc` type.

This type will be a `func` which will also implement your interface. That way users of your interface have the option to implement your interface with just a function; rather than having to create an empty `struct` type.

We then create the function `StdOutAlerter` which has the same signature as the function and just use `time.AfterFunc` to schedule it to print to `os.Stdout`.

Update `main` where we create `NewCLI` to see this in action

```
poker.NewCLI(store, os.Stdin, poker.BlindAlerterFunc(poker.StdOutAlerter)).PlayPoker()
```

Before running you might want to change the `blindTime` increment in `CLI` to be 10 seconds rather than 10 minutes just so you can see it in action.

You should see it print the blind values as we'd expect every 10 seconds. Notice how you can still type `Shaun wins` into the CLI and it will stop the program how we'd expect.

The game wont always be played with 5 people so we need to prompt the user to enter a number of players before the game starts.

Write the test first

To check we are prompting for the number of players we'll want to record what is written to StdOut. We've done this a few times now, we know that `os.Stdout` is an `io.Writer` so we can check what is written if we use dependency injection to pass in a `bytes.Buffer` in our test and see what our code will write.

We don't care about our other collaborators in this test just yet so we've made some dummies in our test file.

We should be a little wary that we now have 4 dependencies for `CLI`, that feels like maybe it is starting to have too many responsibilities. Let's live with it for now and see if a refactoring emerges as we add this new functionality.

```
var dummyBlindAlerter = &SpyBlindAlerter{}
var dummyPlayerStore = &poker.StubPlayerStore{}
var dummyStdIn = &bytes.Buffer{}
var dummyStdOut = &bytes.Buffer{}
```

Here is our new test

```
t.Run("it prompts the user to enter the number of players", func(t *testing.T) {
    stdout := &bytes.Buffer{}
    cli := poker.NewCLI(dummyPlayerStore, dummyStdIn, stdout, dummyBlindAlerter)
    cli.PlayPoker()

    got := stdout.String()
    want := "Please enter the number of players: "

    if got != want {
        t.Errorf("got '%s', want '%s'", got, want)
    }
})
```

We pass in what will be `os.Stdout` in `main` and see what is written.

Try to run the test

```
./CLI_test.go:38:27: too many arguments in call to poker.NewCLI
  have (*poker.StubPlayerStore, *bytes.Buffer, *bytes.Buffer, *SpyBlindAlerter)
   want (poker.PlayerStore, io.Reader, poker.BlindAlerter)
```

Write the minimal amount of code for the test to run and check the failing test output

We have a new dependency so we'll have to update `NewCLI`

```
func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CLI
```

Now the *other* tests will fail to compile because they dont have an `io.Writer` being passed into `NewCLI`.

Add `dummyStdout` for the other tests.

The new test should fail like so

```
=== RUN   TestCLI
--- FAIL: TestCLI (0.00s)
=== RUN  TestCLI/it_prompts_the_user_to_enter_the_number_of_players
   --- FAIL: TestCLI/it_prompts_the_user_to_enter_the_number_of_players (0.00s)
     CLI_test.go:46: got '', want 'Please enter the number of players: '
FAIL
```

Write enough code to make it pass

We need to add our new dependency to our `CLI` so we can reference it in `PlayPoker`

```
type CLI struct {
    playerStore PlayerStore
    in          *bufio.Reader
    out         io.Writer
    alerter     BlindAlerter
}

func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CLI {
    return &CLI{
        playerStore: store,
        in:          bufio.NewReader(in),
        out:         out,
        alerter:     alerter,
    }
}
```

Then finally we can write our prompt at the start of the game

```
func (cli *CLI) PlayPoker() {
    fmt.Fprint(cli.out, "Please enter the number of players: ")
    cli.scheduleBlindAlerts()
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}
```

Refactor

We have a duplicate string for the prompt which we should extract into a constant

```
const PlayerPrompt = "Please enter the number of players: "
```

Use this in both the test code and `CLI`.

Now we need to send in a number and extract it out. The only way we'll know if it has had the desired effect is by seeing what blind alerts were scheduled.

Write the test first

```
t.Run("it prompts the user to enter the number of players", func(t *testing.T) {
    stdout := &bytes.Buffer{}
    in := strings.NewReader("7\n")
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(dummyPlayerStore, in, stdout, blindAlerter)
    cli.PlayPoker()

    got :=stdout.String()
    want := poker.PlayerPrompt

    if got != want {
        t.Errorf("got '%s', want '%s'", got, want)
    }

    cases := []scheduledAlert{
        {0 * time.Second, 100},
        {12 * time.Minute, 200},
        {24 * time.Minute, 300},
        {36 * time.Minute, 400},
    }

    for i, want := range cases {
        t.Run(fmt.Sprint(want), func(t *testing.T) {

            if len(blindAlerter.alerts) <= i {
                t.Fatalf("alert %d was not scheduled %v", i, blindAlerter.alerts)
            }

            got := blindAlerter.alerts[i]
            assertScheduledAlert(t, got, want)
        })
    }
})
```

Ouch! A lot of changes.

- We remove our dummy for StdIn and instead send in a mocked version representing our user entering 7
- We also remove our dummy on the blind alerter so we can see that the number of players has had an effect on the scheduling
- We test what alerts are scheduled

Try to run the test

The test should still compile and fail reporting that the scheduled times are wrong because we've hard-coded for the game to be based on having 5 players

```
=== RUN   TestCLI
--- FAIL: TestCLI (0.00s)
=== RUN  TestCLI/it_prompts_the_user_to_enter_the_number_of_players
   --- FAIL: TestCLI/it_prompts_the_user_to_enter_the_number_of_players (0.00s)
=== RUN  TestCLI/it_prompts_the_user_to_enter_the_number_of_players/100_chips_at_0s
     --- PASS: TestCLI/it_prompts_the_user_to_enter_the_number_of_players/100_chips_at_0s (0
=== RUN  TestCLI/it_prompts_the_user_to_enter_the_number_of_players/200_chips_at_12m0s
```

Write enough code to make it pass

Remember, we are free to commit whatever sins we need to make this work. Once we have working software we can then work on refactoring the mess we're about to make!

```
func (cli *CLI) PlayPoker() {
    fmt.Fprint(cli.out, PlayerPrompt)

    numberOfPlayers, _ := strconv.Atoi(cli.readLine())

    cli.scheduleBlindAlerts(numberOfPlayers)

    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func (cli *CLI) scheduleBlindAlerts(numberOfPlayers int) {
    blindIncrement := time.Duration(5 + numberOfPlayers) * time.Minute

    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
    for _, blind := range blinds {
        cli.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + blindIncrement
    }
}
```

- We read in the `numberOfPlayersInput` into a string
- We use `cli.readLine()` to get the input from the user and then call `Atoi` to convert it into an integer - ignoring any error scenarios. We'll need to write a test for that scenario later.
- From here we change `scheduleBlindAlerts` to accept a number of players. We then calculate a `blindIncrement` time to use to add to `blindTime` as

we iterate over the blind amounts

While our new test has been fixed, a lot of others have failed because now our system only works if the game starts with a user entering a number. You'll need to fix the tests by changing the user inputs so that a number followed by a newline is added (this is highlighting yet more flaws in our approach right now).

Refactor

This all feels a bit horrible right? Let's **listen to our tests**.

- In order to test that we are scheduling some alerts we set up 4 different dependencies. Whenever you have a lot of dependencies for a *thing* in your system, it implies it's doing too much. Visually we can see it in how cluttered our test is.
- To me it feels like **we need to make a cleaner abstraction between reading user input and the business logic we want to do**
- A better test would be *given this user input, do we call a new type `Game` with the correct number of players.*
- We would then extract the testing of the scheduling into the tests for our new `Game`.

We can refactor toward our `Game` first and our test should continue to pass. Once we've made the structural changes we want we can think about how we can refactor the tests to reflect our new separation of concerns

Remember when making changes in refactoring try to keep them as small as possible and keep re-running the tests.

Try it yourself first. Think about the boundaries of what a `Game` would offer and what our `CLI` should be doing.

For now **don't** change the external interface of `NewCLI` as we don't want to change the test code and the client code at the same time as that is too much to juggle and we could end up breaking things.

This is what I came up with:

```
// game.go
type Game struct {
    alerter BlindAlerter
    store   PlayerStore
}

func (p *Game) Start(numberOfPlayers int) {
    blindIncrement := time.Duration(5+numberOfPlayers) * time.Minute

    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
```

```
    for _, blind := range blinds {
        p.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + blindIncrement
    }
}

func (p *Game) Finish(winner string) {
    p.store.RecordWin(winner)
}

// cli.go
type CLI struct {
    playerStore PlayerStore
    in          *bufio.Reader
    out         io.Writer
    game        *Game
}

func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CLI {
    return &CLI{
        in:  bufio.NewReader(in),
        out: out,
        game: &Game{
            alerter: alerter,
            store:   store,
        },
    }
}

const PlayerPrompt = "Please enter the number of players: "

func (cli *CLI) PlayPoker() {
    fmt.Fprint(cli.out, PlayerPrompt)

    numberOfPlayersInput := cli.readLine()
    numberOfPlayers, _ := strconv.Atoi(strings.Trim(numberOfPlayersInput, "\n"))

    cli.game.Start(numberOfPlayers)

    winnerInput := cli.readLine()
    winner := extractWinner(winnerInput)

    cli.game.Finish(winner)
}

func extractWinner(userInput string) string {
```

```
    return strings.Replace(userInput, " wins\n", "", 1)
}

func (cli *CLI) readLine() string {
    cli.in.Scan()
    return cli.in.Text()
}
```

From a "domain" perspective:

- We want to `Start` a `Game`, indicating how many people are playing
- We want to `Finish` a `Game`, declaring the winner

The new `Game` type encapsulates this for us.

With this change we've passed `BlindAlerter` and `PlayerStore` to `Game` as it is now responsible for alerting and storing results.

Our `CLI` is now just concerned with:

- Constructing `Game` with its existing dependencies (which we'll refactor next)
- Interpreting user input as method invocations for `Game`

We want to try to avoid doing "big" refactors which leave us in a state of failing tests for extended periods as that increases the chances of mistakes. (If you are working in a large/distributed team this is extra important)

The first thing we'll do is refactor `Game` so that we inject it into `CLI`. We'll do the smallest changes in our tests to facilitate that and then we'll see how we can break up the tests into the themes of parsing user input and game management.

All we need to do right now is change `NewCLI`

```
func NewCLI(in io.Reader, out io.Writer, game *Game) *CLI {
    return &CLI{
        in:  bufio.NewReader(in),
        out: out,
        game: game,
    }
}
```

This feels like an improvement already. We have less dependencies and *our dependency list is reflecting our overall design goal* of CLI being concerned with input/output and delegating game specific actions to a `Game`.

If you try and compile there are problems. You should be able to fix these problems yourself. Don't worry about making any mocks for `Game` right now, just initialise *real* `Game`s just to get everything compiling and tests green.

To do this you'll need to make a constructor

```
func NewGame(alerter BlindAlerter, store PlayerStore) *Game {
    return &Game{
```

```
        alerter:alerter,
        store:store,
    }
}
```

Here's an example of one of the setups for the tests being fixed

```
stdout := &bytes.Buffer{}
in := strings.NewReader("7\n")
blindAlerter := &SpyBlindAlerter{}
game := poker.NewGame(blindAlerter, dummyPlayerStore)

cli := poker.NewCLI(in, stdout, game)
cli.PlayPoker()
```

It shouldn't take much effort to fix the tests and be back to green again (that's the point!) but make sure you fix `main.go` too before the next stage.

```
// main.go
game := poker.NewGame(poker.BlindAlerterFunc(poker.StdOutAlerter), store)
cli := poker.NewCLI(os.Stdin, os.Stdout, game)
cli.PlayPoker()
```

Now that we have extracted out `Game` we should move our game specific assertions into tests separate from CLI.

This is just an exercise in copying our `CLI` tests but with less dependencies

```
func TestGame_Start(t *testing.T) {
    t.Run("schedules alerts on game start for 5 players", func(t *testing.T) {
        blindAlerter := &poker.SpyBlindAlerter{}
        game := poker.NewTexasHoldem(blindAlerter, dummyPlayerStore)

        game.Start(5)

        cases := []poker.ScheduledAlert{
            {At: 0 * time.Second, Amount: 100},
            {At: 10 * time.Minute, Amount: 200},
            {At: 20 * time.Minute, Amount: 300},
            {At: 30 * time.Minute, Amount: 400},
            {At: 40 * time.Minute, Amount: 500},
            {At: 50 * time.Minute, Amount: 600},
            {At: 60 * time.Minute, Amount: 800},
            {At: 70 * time.Minute, Amount: 1000},
            {At: 80 * time.Minute, Amount: 2000},
            {At: 90 * time.Minute, Amount: 4000},
            {At: 100 * time.Minute, Amount: 8000},
        }

        checkSchedulingCases(cases, t, blindAlerter)
```

```
    })

    t.Run("schedules alerts on game start for 7 players", func(t *testing.T) {
        blindAlerter := &poker.SpyBlindAlerter{}
        game := poker.NewTexasHoldem(blindAlerter, dummyPlayerStore)

        game.Start(7)

        cases := []poker.ScheduledAlert{
            {At: 0 * time.Second, Amount: 100},
            {At: 12 * time.Minute, Amount: 200},
            {At: 24 * time.Minute, Amount: 300},
            {At: 36 * time.Minute, Amount: 400},
        }

        checkSchedulingCases(cases, t, blindAlerter)
    })

}

func TestGame_Finish(t *testing.T) {
    store := &poker.StubPlayerStore{}
    game := poker.NewTexasHoldem(dummyBlindAlerter, store)
    winner := "Ruth"

    game.Finish(winner)
    poker.AssertPlayerWin(t, store, winner)
}
```

The intent behind what happens when a game of poker starts is now much clearer.

Make sure to also move over the test for when the game ends.

Once we are happy we have moved the tests over for game logic we can simplify our CLI tests so they reflect our intended responsibilities clearer

- Process user input and call `Game`'s methods when appropriate
- Send output
- Crucially it doesn't know about the actual workings of how games work

To do this we'll have to make it so `CLI` no longer relies on a concrete `Game` type but instead accepts an interface with `Start(numberOfPlayers)` and `Finish(winner)`. We can then create a spy of that type and verify the correct calls are made.

It's here we realise that naming is awkward sometimes. Rename `Game` to `TexasHoldem` (as that's the *kind* of game we're playing) and the new interface will be called `Game`. This keeps faithful to the notion that our CLI is oblivious to

the actual game we're playing and what happens when you `Start` and `Finish`.

```
type Game interface {
    Start(numberOfPlayers int)
    Finish(winner string)
}
```

Replace all references to `*Game` inside `CLI` and replace them with `Game` (our new interface). As always keep re-running tests to check everything is green while we are refactoring.

Now that we have decoupled `CLI` from `TexasHoldem` we can use spies to check that `Start` and `Finish` are called when we expect them to, with the correct arguments.

Create a spy that implements `Game`

```
type GameSpy struct {
    StartedWith  int
    FinishedWith string
}

func (g *GameSpy) Start(numberOfPlayers int) {
    g.StartedWith = numberOfPlayers
}

func (g *GameSpy) Finish(winner string) {
    g.FinishedWith = winner
}
```

Replace any `CLI` test which is testing any game specific logic with checks on how our `GameSpy` is called. This will then reflect the responsibilities of CLI in our tests clearly.

Here is an example of one of the tests being fixed; try and do the rest yourself and check the source code if you get stuck.

```
    t.Run("it prompts the user to enter the number of players and starts the game", func(t *
        stdout := &bytes.Buffer{}
        in := strings.NewReader("7\n")
        game := &GameSpy{}

        cli := poker.NewCLI(in, stdout, game)
        cli.PlayPoker()

        gotPrompt := stdout.String()
        wantPrompt := poker.PlayerPrompt

        if gotPrompt != wantPrompt {
            t.Errorf("got '%s', want '%s'", gotPrompt, wantPrompt)
```

```
        }

        if game.StartCalledWith != 7 {
            t.Errorf("wanted Start called with 7 but got %d", game.StartCalledWith)
        }
    })
```

Now that we have a clean separation of concerns, checking edge cases around IO in our `CLI` should be easier.

We need to address the scenario where a user puts a non numeric value when prompted for the number of players:

Our code should not start the game and it should print a handy error to the user and then exit.

Write the test first

We'll start by making sure the game doesn't start

```
t.Run("it prints an error when a non numeric value is entered and does not start the game",
        stdout := &bytes.Buffer{}
        in := strings.NewReader("Pies\n")
        game := &GameSpy{}

        cli := poker.NewCLI(in, stdout, game)
        cli.PlayPoker()

        if game.StartCalled {
            t.Errorf("game should not have started")
        }
    })
```

You'll need to add to our `GameSpy` a field `StartCalled` which only gets set if `Start` is called

Try to run the test

```
=== RUN  TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_start_t
   --- FAIL: TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_star
        CLI_test.go:62: game should not have started
```

Write enough code to make it pass

Around where we call `Atoi` we just need to check for the error

```
numberOfPlayers, err := strconv.Atoi(cli.readLine())

if err != nil {
    return
}
```

Next we need to inform the user of what they did wrong so we'll assert on what is printed to `stdout`.

Write the test first

We've asserted on what was printed to `stdout` before so we can copy that code for now

```
gotPrompt := stdout.String()

wantPrompt := poker.PlayerPrompt + "you're so silly"

if gotPrompt != wantPrompt {
    t.Errorf("got '%s', want '%s'", gotPrompt, wantPrompt)
}
```

We are storing *everything* that gets written to stdout so we still expect the `poker.PlayerPrompt`. We then just check an additional thing gets printed. We're not too bothered about the exact wording for now, we'll address it when we refactor.

Try to run the test

```
=== RUN  TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_start_t
   --- FAIL: TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_star
     CLI_test.go:70: got 'Please enter the number of players: ', want 'Please enter the number o
```

Write enough code to make it pass

Change the error handling code

```
if err != nil {
    fmt.Fprint(cli.out, "you're so silly")
    return
}
```

Refactor

Now refactor the message into a constant like `PlayerPrompt`

```
wantPrompt := poker.PlayerPrompt + poker.BadPlayerInputErrMsg
```

and put in a more appropriate message

```
const BadPlayerInputErrMsg = "Bad value received for number of players, please try again wit
```

Finally our testing around what has been sent to `stdout` is quite verbose, let's write an assert function to clean it up.

```
func assertMessagesSentToUser(t *testing.T, stdout *bytes.Buffer, messages ...string) {
    t.Helper()
    want := strings.Join(messages, "")
    got := stdout.String()
    if got != want {
        t.Errorf("got '%s' sent to stdout but expected %+v", got, messages)
    }
}
```

Using the vararg syntax (`...string`) is handy here because we need to assert on varying amounts of messages.

Use this helper in both of the tests where we assert on messages sent to the user.

There are a number of tests that could be helped with some `assertX` functions so practice your refactoring by cleaning up our tests so they read nicely.

Take some time and think about the value of some of the tests we've driven out. Remember we don't want more tests than necessary, can you refactor/remove some of them *and still be confident it all works* ?

Here is what I came up with

```
func TestCLI(t *testing.T) {

    t.Run("start game with 3 players and finish game with 'Chris' as winner", func(t *testi
        game := &GameSpy{}
        stdout := &bytes.Buffer{}

        in := userSends("3", "Chris wins")
        cli := poker.NewCLI(in, stdout, game)

        cli.PlayPoker()

        assertMessagesSentToUser(t, stdout, poker.PlayerPrompt)
        assertGameStartedWith(t, game, 3)
        assertFinishCalledWith(t, game, "Chris")
    })

    t.Run("start game with 8 players and record 'Cleo' as winner", func(t *testing.T) {
        game := &GameSpy{}
```

```
        in := userSends("8", "Cleo wins")
        cli := poker.NewCLI(in, dummyStdOut, game)

        cli.PlayPoker()

        assertGameStartedWith(t, game, 8)
        assertFinishCalledWith(t, game, "Cleo")
    })

    t.Run("it prints an error when a non numeric value is entered and does not start the gam
        game := &GameSpy{}

        stdout := &bytes.Buffer{}
        in := userSends("pies")

        cli := poker.NewCLI(in, stdout, game)
        cli.PlayPoker()

        assertGameNotStarted(t, game)
        assertMessagesSentToUser(t, stdout, poker.PlayerPrompt, poker.BadPlayerInputErrMsg)
    })
}
```

The tests now reflect the main capabilities of CLI, it is able to read user input in terms of how many people are playing and who won and handles when a bad value is entered for number of players. By doing this it is clear to the reader what `CLI` does, but also what it doesn't do.

What happens if instead of putting `Ruth wins` the user puts in `Lloyd is a killer` ?

Finish this chapter by writing a test for this scenario and making it pass.

Wrapping up

A quick project recap

For the past 5 chapters we have slowly TDD'd a fair amount of code

- We have two applications, a command line application and a web server.
- Both these applications rely on a `PlayerStore` to record winners
- The web server can also display a league table of who is winning the most games
- The command line app helps players play a game of poker by tracking what the current blind value is.

time.Afterfunc

A very handy way of scheduling a function call after a specific duration. It is well worth investing time looking at the documentation for `time` as it has a lot of time saving functions and methods for you to work with.

Some of my favourites are

- `time.After(duration)` which return you a `chan Time` when the duration has expired. So if you wish to do something *after* a specific time, this can help.
- `time.NewTicker(duration)` returns a `Ticker` which is similar to the above in that it returns a channel but this one "ticks" every duration, rather than just once. This is very handy if you want to execute some code every `N duration`.

More examples of good separation of concerns

Generally it is good practice to separate the responsibilities of dealing with user input and responses away from domain code. You see that here in our command line application and also our web server.

Our tests got messy. We had too many assertions (check this input, schedules these alerts, etc) and too many dependencies. We could visually see it was cluttered; it is **so important to listen to your tests**.

- If your tests look messy try and refactor them.
- If you've done this and they're still a mess it is very likely pointing to a flaw in your design
- This is one of the real strengths of tests.

Even though the tests and the production code was a bit cluttered we could freely refactor backed by our tests.

Remember when you get in to these situations to always take small steps and re-run the tests after every change.

It would've been dangerous to refactor both the test code *and* the production code at the same time, so we first refactored the production code (in the current state we couldn't improve the tests much) without changing its interface so we could rely on our tests as much as we could while changing things. *Then* we refactored the tests after the design improved.

After refactoring the dependency list reflected our design goal. This is another benefit of DI in that it often documents intent. When you rely on global variables responsibilities become very unclear.

An example of a function implementing an interface

When you define an interface with one method in it you might want to consider defining a `MyInterfaceFunc` type to complement it so users can implement your interface with just a function

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}

// BlindAlerterFunc allows you to implement BlindAlerter with a function
type BlindAlerterFunc func(duration time.Duration, amount int)

// ScheduleAlertAt is BlindAlerterFunc implementation of BlindAlerter
func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int) {
    a(duration, amount)
}
```

Websockets

You can find all the code for this chapter here

In this chapter we'll learn how to use WebSockets to improve our application.

Project recap

We have two applications in our poker codebase

- *Command line app.* Prompts the user to enter the number of players in a game. From then on informs the players of what the "blind bet" value is, which increases over time. At any point a user can enter `"{Playername} wins"` to finish the game and record the victor in a store.
- *Web app.* Allows users to record winners of games and displays a league table. Shares the same store as the command line app.

Next steps

The product owner is thrilled with the command line application but would prefer it if we could bring that functionality to the browser. She imagines a web page with a text box that allows the user to enter the number of players and when they submit the form the page displays the blind value and automatically updates it when appropriate. Like the command line application the user can declare the winner and it'll get saved in the database.

On the face of it, it sounds quite simple but as always we must emphasise taking an *iterative* approach to writing software.

First of all we will need to serve HTML. So far all of our HTTP endpoints have returned either plaintext or JSON. We *could* use the same techniques we know (as they're all ultimately strings) but we can also use the html/template package for a cleaner solution.

We also need to be able to asynchronously send messages to the user saying `The blind is now *y*` without having to refresh the browser. We can use WebSockets to facilitate this.

> WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection

Given we are taking on a number of techniques it's even more important we do the smallest amount of useful work possible first and then iterate.

For that reason the first thing we'll do is create a web page with a form for the user to record a winner. Rather than using a plain form, we will use WebSockets to send that data to our server for it to record.

After that we'll work on the blind alerts by which point we will have a bit of infrastructure code set up.

What about tests for the JavaScript ?

There will be some JavaScript written to do this but I wont go in to writing tests.

It is of course possible but for the sake of brevity I wont be including any explanations for it.

Sorry folks. Lobby O'Reilly to pay me to make a "Learn JavaScript with tests".

Write the test first

First thing we need to do is serve up some HTML to users when they hit `/game`.

Here's a reminder of the pertinent code in our web server

```
type PlayerServer struct {
    store PlayerStore
    http.Handler
}

const jsonContentType = "application/json"

func NewPlayerServer(store PlayerStore) *PlayerServer {
    p := new(PlayerServer)
```

```
    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    p.Handler = router

    return p
}
```

The *easiest* thing we can do for now is check when we `GET /game` that we get a 200.

```
func TestGame(t *testing.T) {
    t.Run("GET /game returns 200", func(t *testing.T) {
        server := NewPlayerServer(&StubPlayerStore{})

        request, _ := http.NewRequest(http.MethodGet, "/game", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusOK)
    })
}
```

Try to run the test

```
--- FAIL: TestGame (0.00s)
=== RUN   TestGame/GET_/game_returns_200
    --- FAIL: TestGame/GET_/game_returns_200 (0.00s)
     server_test.go:109: did not get correct status, got 404, want 200
```

Write enough code to make it pass

Our server has a router setup so it's relatively easy to fix.

To our router add

```
router.Handle("/game", http.HandlerFunc(p.game))
```

And then write the `game` method

```
func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
}
```

Refactor

The server code is already fine due to us slotting in more code into the existing well-factored code very easily.

We can tidy up the test a little by adding a test helper function `newGameRequest` to make the request to `/game`. Try writing this yourself.

```
func TestGame(t *testing.T) {
    t.Run("GET /game returns 200", func(t *testing.T) {
        server := NewPlayerServer(&StubPlayerStore{})

        request :=  newGameRequest()
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response, http.StatusOK)
    })
}
```

You'll also notice I changed `assertStatus` to accept `response` rather than `response.Code` as I feel it reads better.

Now we need to make the endpoint return some HTML, here it is

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Let's play poker</title>
</head>
<body>
<section id="game">
    <div id="declare-winner">
        <label for="winner">Winner</label>
        <input type="text" id="winner"/>
        <button id="winner-button">Declare winner</button>
    </div>
</section>
</body>
<script type="application/javascript">

   const submitWinnerButton = document.getElementById('winner-button')
    const winnerInput = document.getElementById('winner')

    if (window['WebSocket']) {
     const conn = new WebSocket('ws://' + document.location.host + '/ws')
```

```
        submitWinnerButton.onclick = event => {
            conn.send(winnerInput.value)
        }
    }
</script>
</html>
```

We have a very simple web page

- A text input for the user to enter the winner into
- A button they can click to declare the winner.
- Some JavaScript to open a WebSocket connection to our server and handle the submit button being pressed

`WebSocket` is built into most modern browsers so we don't need to worry about bringing in any libraries. The web page wont work for older browsers, but we're ok with that for this scenario.

How do we test we return the correct markup?

There are a few ways. As has been emphasised throughout the book, it is important that the tests you write have sufficient value to justify the cost.

1. Write a browser based test, using something like Selenium. These tests are the most "realistic" of all approaches because they start an actual web browser of some kind and simulates a user interacting with it. These tests can give you a lot of confidence your system works but are more difficult to write than unit tests and much slower to run. For the purposes of our product this is overkill.
2. Do an exact string match. This *can* be ok but these kind of tests end up being very brittle. The moment someone changes the markup you will have a test failing when in practice nothing has *actually broken*.
3. Check we call the correct template. We will be using a templating library from the standard lib to serve the HTML (discussed shortly) and we could inject in the *thing* to generate the HTML and spy on its call to check we're doing it right. This would have an impact on our code's design but doesn't actually test a great deal; other than we're calling it with the correct template file. Given we will only have the one template in our project the chance of failure here seems low.

So in the book "Learn Go with Tests" for the first time, we're not going to write a test.

Put the markup in a file called `game.html`

Next change the endpoint we just wrote to the following

```
func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
    tmpl, err := template.ParseFiles("game.html")
```

```
    if err != nil {
        http.Error(w, fmt.Sprintf("problem loading template %s", err.Error()), http.StatusIn
        return
    }

    tmpl.Execute(w, nil)
}
```

`html/template` is a Go package for creating HTML. In our case we call `template.ParseFiles`, giving the path of our html file. Assuming there is no error you can then `Execute` the template, which writes it to an `io.Writer`. In our case we want it to `Write` to the internet, so we give it our `http.ResponseWriter`.

As we have not written a test, it would be prudent to manually test our web server just to make sure things are working as we'd hope. Go to `cmd/webserver` and run the `main.go` file. Visit `http://localhost:5000/game`.

You *should* have got an error about not being able to find the template. You can either change the path to be relative to your folder, or you can have a copy of the `game.html` in the `cmd/webserver` directory. I chose to create a symlink (`ln -s ../../game.html game.html`) to the file inside the root of the project so if I make changes they are reflected when running the server.

If you make this change and run again you should see our UI.

Now we need to test that when we get a string over a WebSocket connection to our server that we declare it as a winner of a game.

Write the test first

For the first time we are going to use an external library so that we can work with WebSockets.

Run `go get github.com/gorilla/websocket`

This will fetch the code for the excellent Gorilla WebSocket library. Now we can update our tests for our new requirement.

```
t.Run("when we get a message over a websocket it is a winner of a game", func(t *testing.T)
    store := &StubPlayerStore{}
    winner := "Ruth"
    server := httptest.NewServer(NewPlayerServer(store))
    defer server.Close()

    wsURL := "ws" + strings.TrimPrefix(server.URL, "http") + "/ws"

    ws, _, err := websocket.DefaultDialer.Dial(wsURL, nil)
```

```
    if err != nil {
        t.Fatalf("could not open a ws connection on %s %v", wsURL, err)
    }
    defer ws.Close()

    if err := ws.WriteMessage(websocket.TextMessage, []byte(winner)); err != nil {
        t.Fatalf("could not send message over ws connection %v", err)
    }

    AssertPlayerWin(t, store, winner)
})
```

Make sure that you have an import for the `websocket` library. My IDE automatically did it for me, so should yours.

To test what happens from the browser we have to open up our own WebSocket connection and write to it.

Our previous tests around our server just called methods on our server but now we need to have a persistent connection to our server. To do that we use `httptest.NewServer` which takes a `http.Handler` and will spin it up and listen for connections.

Using `websocket.DefaultDialer.Dial` we try to dial in to our server and then we'll try and send a message with our `winner`.

Finally we assert on the player store to check the winner was recorded.

Try to run the test

```
=== RUN   TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game
   --- FAIL: TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game (0.00s)
      server_test.go:124: could not open a ws connection on ws://127.0.0.1:55838/ws websocket: b
```

We have not changed our server to accept WebSocket connections on `/ws` so we're not shaking hands yet.

Write enough code to make it pass

Add another listing to our router

```
router.Handle("/ws", http.HandlerFunc(p.webSocket))
```

Then add our new `webSocket` handler

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    upgrader := websocket.Upgrader{
        ReadBufferSize:  1024,
        WriteBufferSize: 1024,
```

```
    }
    upgrader.Upgrade(w, r, nil)
}
```

To accept a WebSocket connection we `Upgrade` the request. If you now re-run the test you should move on to the next error.

```
=== RUN  TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game
   --- FAIL: TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game (0.00s)
        server_test.go:132: got 0 calls to RecordWin want 1
```

Now that we have a connection opened, we'll want to listen for a message and then record it as the winner.

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    upgrader := websocket.Upgrader{
        ReadBufferSize:  1024,
        WriteBufferSize: 1024,
    }
    conn, _ := upgrader.Upgrade(w, r, nil)
    _, winnerMsg, _ := conn.ReadMessage()
    p.store.RecordWin(string(winnerMsg))
}
```

(Yes, we're ignoring a lot of errors right now!)

`conn.ReadMessage()` blocks on waiting for a message on the connection. Once we get one we use it to `RecordWin`. This would finally close the WebSocket connection.

If you try and run the test, it's still failing.

The issue is timing. There is a delay between our WebSocket connection reading the message and recording the win and our test finishes before it happens. You can test this by putting a short `time.Sleep` before the final assertion.

Let's go with that for now but acknowledge that putting in arbitrary sleeps into tests **is very bad practice**.

```
time.Sleep(10 * time.Millisecond)
AssertPlayerWin(t, store, winner)
```

Refactor

We committed many sins to make this test work both in the server code and the test code but remember this is the easiest way for us to work.

We have nasty, horrible, *working* software backed by a test, so now we are free to make it nice and know we wont break anything accidentally.

Let's start with the server code.

We can move the `upgrader` to a private value inside our package because we don't need to redeclare it on every WebSocket connection request

```
var wsUpgrader = websocket.Upgrader{
    ReadBufferSize:  1024,
    WriteBufferSize: 1024,
}

func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    conn, _ := wsUpgrader.Upgrade(w, r, nil)
    _, winnerMsg, _ := conn.ReadMessage()
    p.store.RecordWin(string(winnerMsg))
}
```

Our call to `template.ParseFiles("game.html")` will run on every `GET /game` which means we'll go to the file system on every request even though we have no need to re-parse the template. Let's refactor our code so that we parse the template once in `NewPlayerServer` instead. We'll have to make it so this function can now return an error in case we have problems fetching the template from disk or parsing it.

Here's the relevant changes to `PlayerServer`

```
type PlayerServer struct {
    store PlayerStore
    http.Handler
    template *template.Template
}

const htmlTemplatePath = "game.html"

func NewPlayerServer(store PlayerStore) (*PlayerServer, error) {
    p := new(PlayerServer)

    tmpl, err := template.ParseFiles("game.html")

    if err != nil {
        return nil, fmt.Errorf("problem opening %s %v", htmlTemplatePath, err)
    }

    p.template = tmpl
    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))
    router.Handle("/game", http.HandlerFunc(p.game))
    router.Handle("/ws", http.HandlerFunc(p.webSocket))
```

```
    p.Handler = router

    return p, nil
}

func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
    p.template.Execute(w, nil)
}
```

By changing the signature of `NewPlayerServer` we now have compilation problems. Try and fix them yourself or refer to the source code if you struggle.

For the test code I made a helper called `mustMakePlayerServer(t *testing.T, store PlayerStore) *PlayerServer` so that I could hide the error noise away from the tests.

```
func mustMakePlayerServer(t *testing.T, store PlayerStore) *PlayerServer {
    server, err := NewPlayerServer(store)
    if err != nil {
        t.Fatal("problem creating player server", err)
    }
    return server
}
```

Similarly I created another helper `mustDialWS` so that I could hide nasty error noise when creating the WebSocket connection.

```
func mustDialWS(t *testing.T, url string) *websocket.Conn {
    ws, _, err := websocket.DefaultDialer.Dial(url, nil)

    if err != nil {
        t.Fatalf("could not open a ws connection on %s %v", url, err)
    }

    return ws
}
```

Finally in our test code we can create a helper to tidy up sending messages

```
func writeWSMessage(t *testing.T, conn *websocket.Conn, message string) {
    t.Helper()
    if err := conn.WriteMessage(websocket.TextMessage, []byte(message)); err != nil {
        t.Fatalf("could not send message over ws connection %v", err)
    }
}
```

Now the tests are passing try running the server and declare some winners in `/game`. You should see them recorded in `/league`. Remember that every time we get a winner we *close the connection*, you will need to refresh the page to

open the connection again.

We've made a trivial web form that lets users record the winner of a game. Let's iterate on it to make it so the user can start a game by providing a number of players and the server will push messages to the client informing them of what the blind value is as time passes.

First of all update `game.html` to update our client side code for the new requirements

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Lets play poker</title>
</head>
<body>
<section id="game">
    <div id="game-start">
        <label for="player-count">Number of players</label>
        <input type="number" id="player-count"/>
        <button id="start-game">Start</button>
    </div>

    <div id="declare-winner">
        <label for="winner">Winner</label>
        <input type="text" id="winner"/>
        <button id="winner-button">Declare winner</button>
    </div>

    <div id="blind-value"/>
</section>

<section id="game-end">
    <h1>Another great game of poker everyone!</h1>
    <p><a href="/league">Go check the league table</a></p>
</section>

</body>
<script type="application/javascript">
    const startGame = document.getElementById('game-start')

   const declareWinner = document.getElementById('declare-winner')
  const submitWinnerButton = document.getElementById('winner-button')
    const winnerInput = document.getElementById('winner')

    const blindContainer = document.getElementById('blind-value')
```

```
    const gameContainer = document.getElementById('game')
    const gameEndContainer = document.getElementById('game-end')

    declareWinner.hidden = true
    gameEndContainer.hidden = true

  document.getElementById('start-game').addEventListener('click', event => {
        startGame.hidden = true
        declareWinner.hidden = false

      const numberOfPlayers = document.getElementById('player-count').value

        if (window['WebSocket']) {
        const conn = new WebSocket('ws://' + document.location.host + '/ws')

            submitWinnerButton.onclick = event => {
                conn.send(winnerInput.value)
                gameEndContainer.hidden = false
                gameContainer.hidden = true
            }

            conn.onclose = evt => {
                blindContainer.innerText = 'Connection closed'
            }

            conn.onmessage = evt => {
                blindContainer.innerText = evt.data
            }

            conn.onopen = function () {
                conn.send(numberOfPlayers)
            }
        }
    })
</script>
</html>
```

The main changes is bringing in a section to enter the number of players and a section to display the blind value. We have a little logic to show/hide the user interface depending on the stage of the game.

Any message we receive via `conn.onmessage` we assume to be blind alerts and so we set the `blindContainer.innerText` accordingly.

How do we go about sending the blind alerts? In the previous chapter we introduced the idea of `Game` so our CLI code could call a `Game` and everything

else would be taken care of including scheduling blind alerts. This turned out to be a good separation of concern.

```
type Game interface {
    Start(numberOfPlayers int)
    Finish(winner string)
}
```

When the user was prompted in the CLI for number of players it would `Start` the game which would kick off the blind alerts and when the user declared the winner they would `Finish`. This is the same requirements we have now, just a different way of getting the inputs; so we should look to re-use this concept if we can.

Our "real" implementation of `Game` is `TexasHoldem`

```
type TexasHoldem struct {
    alerter BlindAlerter
    store   PlayerStore
}
```

By sending in a `BlindAlerter` `TexasHoldem` can schedule blind alerts to be sent to *wherever*

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}
```

And as a reminder, here is our implementation of the `BlindAlerter` we use in the CLI.

```
func StdOutAlerter(duration time.Duration, amount int) {
    time.AfterFunc(duration, func() {
        fmt.Fprintf(os.Stdout, "Blind is now %d\n", amount)
    })
}
```

This works in CLI because we *always want to send the alerts to* `os.Stdout` but this wont work for our web server. For every request we get a new `http.ResponseWriter` which we then upgrade to `*websocket.Conn`. So we cant know when constructing our dependencies where our alerts need to go.

For that reason we need to change `BlindAlerter.ScheduleAlertAt` so that it takes a destination for the alerts so that we can re-use it in our webserver.

Open BlindAlerter.go and add the parameter `to io.Writer`

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int, to io.Writer)
}

type BlindAlerterFunc func(duration time.Duration, amount int, to io.Writer)
```

```
func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int, to io.Writer)
    a(duration, amount, to)
}
```

The idea of a `StdoutAlerter` doesn't fit our new model so just rename it to `Alerter`

```
func Alerter(duration time.Duration, amount int, to io.Writer) {
    time.AfterFunc(duration, func() {
        fmt.Fprintf(to, "Blind is now %d\n", amount)
    })
}
```

If you try and compile, it will fail in `TexasHoldem` because it is calling `ScheduleAlertAt` without a destination, to get things compiling again *for now* hard-code it to `os.Stdout`.

Try and run the tests and they will fail because `SpyBlindAlerter` no longer implements `BlindAlerter`, fix this by updating the signature of `ScheduleAlertAt`, run the tests and we should still be green.

It doesn't make any sense for `TexasHoldem` to know where to send blind alerts. Let's now update `Game` so that when you start a game you declare *where* the alerts should go.

```
type Game interface {
    Start(numberOfPlayers int, alertsDestination io.Writer)
    Finish(winner string)
}
```

Let the compiler tell you what you need to fix. The change isn't so bad:

- Update `TexasHoldem` so it properly implements `Game`
- In `CLI` when we start the game, pass in our `out` property (`cli.game.Start(numberOfPlayers, cli.out)`)
- In `TexasHoldem`'s test i use `game.Start(5, ioutil.Discard)` to fix the compilation problem and configure the alert output to be discarded

If you've got everything right, everything should be green! Now we can try and use `Game` within `Server`.

Write the test first

The requirements of `CLI` and `Server` are the same! It's just the delivery mechanism is different.

Let's take a look at our `CLI` test for inspiration.

```
t.Run("start game with 3 players and finish game with 'Chris' as winner", func(t *testing.T)
    game := &GameSpy{}
```

```
    out := &bytes.Buffer{}
    in := userSends("3", "Chris wins")

    poker.NewCLI(in, out, game).PlayPoker()

    assertMessagesSentToUser(t, out, poker.PlayerPrompt)
    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, "Chris")
})
```

It looks like we should be able to test drive out a similar outcome using `GameSpy`

Replace the old websocket test with the following

```
t.Run("start a game with 3 players and declare Ruth the winner", func(t *testing.T) {
    game := &poker.GameSpy{}
    winner := "Ruth"
    server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
    ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

    defer server.Close()
    defer ws.Close()

    writeWSMessage(t, ws, "3")
    writeWSMessage(t, ws, winner)

    time.Sleep(10 * time.Millisecond)
    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, winner)
})
```

- As discussed we create a spy `Game` and pass it into `mustMakePlayerServer` (be sure to update the helper to support this).
- We then send the web socket messages for a game.
- Finally we assert that the game is started and finished with what we expect.

Try to run the test

You'll have a number of compilation errors around `mustMakePlayerServer` in other tests. Introduce an unexported variable `dummyGame` and use it through all the tests that aren't compiling

```
var (
    dummyGame = &GameSpy{}
)
```

The final error is where we are trying to pass in `Game` to `NewPlayerServer` but it doesn't support it yet

```
./server_test.go:21:38: too many arguments in call to "github.com/quii/learn-go-with-tests/Web
   have ("github.com/quii/learn-go-with-tests/WebSockets/v2".PlayerStore, "github.com/quii/le
   want ("github.com/quii/learn-go-with-tests/WebSockets/v2".PlayerStore)
```

Write the minimal amount of code for the test to run and check the failing test output

Just add it as an argument for now just to get the test running

```
func NewPlayerServer(store PlayerStore, game Game) (*PlayerServer, error) {
```

Finally!

```
=== RUN  TestGame/start_a_game_with_3_players_and_declare_Ruth_the_winner
--- FAIL: TestGame (0.01s)
   --- FAIL: TestGame/start_a_game_with_3_players_and_declare_Ruth_the_winner (0.01s)
        server_test.go:146: wanted Start called with 3 but got 0
     server_test.go:147: expected finish called with 'Ruth' but got ''
FAIL
```

Write enough code to make it pass

We need to add `Game` as a field to `PlayerServer` so that it can use it when it gets requests.

```
type PlayerServer struct {
    store PlayerStore
    http.Handler
    template *template.Template
    game Game
}
```

(We already have a method called `game` so rename that to `playGame`)

Next lets assign it in our constructor

```
func NewPlayerServer(store PlayerStore, game Game) (*PlayerServer, error) {
    p := new(PlayerServer)

    tmpl, err := template.ParseFiles("game.html")

    if err != nil {
        return nil, fmt.Errorf("problem opening %s %v", htmlTemplatePath, err)
    }

    p.game = game
```

```
    // etc
```

Now we can use our `Game` within `webSocket`.

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    conn, _ := wsUpgrader.Upgrade(w, r, nil)

    _, numberOfPlayersMsg, _ := conn.ReadMessage()
    numberOfPlayers, _ := strconv.Atoi(string(numberOfPlayersMsg))
    p.game.Start(numberOfPlayers, ioutil.Discard) //todo: Dont discard the blinds messages!

    _, winner, _ := conn.ReadMessage()
    p.game.Finish(string(winner))
}
```

Hooray! The tests pass.

We are not going to send the blind messages anywhere *just yet* as we need to have a think about that. When we call `game.Start` we send in `ioutil.Discard` which will just discard any messages written to it.

For now start the web server up. You'll need to update the `main.go` to pass a `Game` to the `PlayerServer`

```
func main() {
    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }

    store, err := poker.NewFileSystemPlayerStore(db)

    if err != nil {
        log.Fatalf("problem creating file system player store, %v ", err)
    }

    game := poker.NewTexasHoldem(poker.BlindAlerterFunc(poker.Alerter), store)

    server, err := poker.NewPlayerServer(store, game)

    if err != nil {
        log.Fatalf("problem creating player server %v", err)
    }

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
```

```
}
```

Discounting the fact we're not getting blind alerts yet, the app does work! We've managed to re-use `Game` with `PlayerServer` and it has taken care of all the details. Once we figure out how to send our blind alerts through to the web sockets rather than discarding them it *should* all work.

Before that though, let's tidy up some code.

Refactor

The way we're using WebSockets is fairly basic and the error handling is fairly naive, so I wanted to encapsulate that in a type just to remove that messyness from the server code. We may wish to revisit it later but for now this'll tidy things up a bit

```
type playerServerWS struct {
    *websocket.Conn
}

func newPlayerServerWS(w http.ResponseWriter, r *http.Request) *playerServerWS {
    conn, err := wsUpgrader.Upgrade(w, r, nil)

    if err != nil {
        log.Printf("problem upgrading connection to WebSockets %v\n", err)
    }

    return &playerServerWS{conn}
}

func (w *playerServerWS) WaitForMsg() string {
    _, msg, err := w.ReadMessage()
    if err != nil {
        log.Printf("error reading from websocket %v\n", err)
    }
    return string(msg)
}
```

Now the server code is a bit simplified

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    ws := newPlayerServerWS(w, r)

    numberOfPlayersMsg := ws.WaitForMsg()
    numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
    p.game.Start(numberOfPlayers, ioutil.Discard) //todo: Dont discard the blinds messages!

    winner := ws.WaitForMsg()
```

```
    p.game.Finish(winner)
}
```

Once we figure out how to not discard the blind messages we're done.

Let's *not* write a test!

Sometimes when we're not sure how to do something, it's best just to play around and try things out! Make sure your work is committed first because once we've figured out a way we should drive it through a test.

The problematic line of code we have is

```
p.game.Start(numberOfPlayers, ioutil.Discard) //todo: Dont discard the blinds messages!
```

We need to pass in an `io.Writer` for the game to write the blind alerts to.

Wouldn't it be nice if we could pass in our `playerServerWS` from before? It's our wrapper around our WebSocket so it *feels* like we should be able to send that to our `Game` to send messages to.

Give it a go:

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    ws := newPlayerServerWS(w, r)

    numberOfPlayersMsg := ws.WaitForMsg()
    numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
    p.game.Start(numberOfPlayers, ws)
    //etc...
```

The compiler complains

```
./server.go:71:14: cannot use ws (type *playerServerWS) as type io.Writer in argument to p.game
  *playerServerWS does not implement io.Writer (missing Write method)
```

It seems the obvious thing to do, would be to make it so `playerServerWS` *does* implement `io.Writer`. To do so we use the underlying `*websocket.Conn` to use `WriteMessage` to send the message down the websocket

```
func (w *playerServerWS) Write(p []byte) (n int, err error) {
    err = w.WriteMessage(1, p)

    if err != nil {
        return 0, err
    }

    return len(p), nil
}
```

This seems too easy! Try and run the application and see if it works.

Beforehand edit `TexasHoldem` so that the blind increment time is shorter so you can see it in action

```
blindIncrement := time.Duration(5+numberOfPlayers) * time.Second // (rather than a minute)
```

You should see it working! The blind amount increments in the browser as if by magic.

Now let's revert the code and think how to test it. In order to *implement* it all we did was pass through to `StartGame` was `playerServerWS` rather than `ioutil.Discard` so that might make you think we should perhaps spy on the call to verify it works.

Spying is great and helps us check implementation details but we should always try and favour testing the *real* behaviour if we can because when you decide to refactor it's often spy tests that start failing because they are usually checking implementation details that you're trying to change.

Our test currently opens a websocket connection to our running server and sends messages to make it do things. Equally we should be able to test the messages our server sends back over the websocket connection.

Write the test first

We'll edit our existing test.

Currently our `GameSpy` does not send any data to `out` when you call `Start`. We should change it so we can configure it to send a canned message and then we can check that message gets sent to the websocket. This should give us confidence that we have configured things correctly whilst still exercising the real behaviour we want.

```
type GameSpy struct {
    StartCalled     bool
    StartCalledWith int
    BlindAlert      []byte

    FinishedCalled   bool
    FinishCalledWith string
}
```

Add `BlindAlert` field.

Update `GameSpy Start` to send the canned message to `out`.

```
func (g *GameSpy) Start(numberOfPlayers int, out io.Writer) {
    g.StartCalled = true
    g.StartCalledWith = numberOfPlayers
    out.Write(g.BlindAlert)
}
```

This now means when we exercise `PlayerServer` when it tries to `Start` the game it should end up sending messages through the websocket if things are working right.

Finally we can update the test

```
t.Run("start a game with 3 players, send some blind alerts down WS and declare Ruth the winn
    wantedBlindAlert := "Blind is 100"
    winner := "Ruth"

    game := &GameSpy{BlindAlert: []byte(wantedBlindAlert)}
    server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
    ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

    defer server.Close()
    defer ws.Close()

    writeWSMessage(t, ws, "3")
    writeWSMessage(t, ws, winner)

    time.Sleep(10 * time.Millisecond)
    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, winner)

    _, gotBlindAlert, _ := ws.ReadMessage()

    if string(gotBlindAlert) != wantedBlindAlert {
        t.Errorf("got blind alert '%s', want '%s'", string(gotBlindAlert), wantedBlindAlert)
    }
})
```

- We've added a `wantedBlindAlert` and configured our `GameSpy` to send it to `out` if `Start` is called.
- We hope it gets sent in the websocket connection so we've added a call to `ws.ReadMessage()` to wait for a message to be sent and then check it's the one we expected.

Try to run the test

You should find the test hangs forever. This is because `ws.ReadMessage()` will block until it gets a message, which it never will.

Write the minimal amount of code for the test to run and check the failing test output

We should never have tests that hang so let's introduce a way of handling code that we want to timeout.

```
func within(t *testing.T, d time.Duration, assert func()) {
    t.Helper()

    done := make(chan struct{}, 1)

    go func() {
        assert()
        done <- struct{}{}
    }()

    select {
    case <-time.After(d):
        t.Error("timed out")
    case <-done:
    }
}
```

What `within` does is take a function `assert` as an argument and then runs it in a go routine. If/When the function finishes it will signal it is done via the `done` channel.

While that happens we use a `select` statement which lets us wait for a channel to send a message. From here it is a race between the `assert` function and `time.After` which will send a signal when the duration has occurred.

Finally I made a helper function for our assertion just to make things a bit neater

```
func assertWebsocketGotMsg(t *testing.T, ws *websocket.Conn, want string) {
    _, msg, _ := ws.ReadMessage()
    if string(msg) != want {
        t.Errorf(`got "%s", want "%s"`, string(msg), want)
    }
}
```

Here's how the test reads now

```
t.Run("start a game with 3 players, send some blind alerts down WS and declare Ruth the winn
    wantedBlindAlert := "Blind is 100"
    winner := "Ruth"

    game := &GameSpy{BlindAlert: []byte(wantedBlindAlert)}
    server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
```

```
    ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

    defer server.Close()
    defer ws.Close()

    writeWSMessage(t, ws, "3")
    writeWSMessage(t, ws, winner)

    time.Sleep(tenMS)

    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, winner)
    within(t, tenMS, func() { assertWebsocketGotMsg(t, ws, wantedBlindAlert) })
})
```

Now if you run the test...

```
=== RUN   TestGame
=== RUN  TestGame/start_a_game_with_3_players,_send_some_blind_alerts_down_WS_and_declare_Ru
--- FAIL: TestGame (0.02s)
   --- FAIL: TestGame/start_a_game_with_3_players,_send_some_blind_alerts_down_WS_and_declare
        server_test.go:143: timed out
        server_test.go:150: got "", want "Blind is 100"
```

Write enough code to make it pass

Finally we can now change our server code so it sends our WebSocket connection to the game when it starts

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    ws := newPlayerServerWS(w, r)

    numberOfPlayersMsg := ws.WaitForMsg()
    numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
    p.game.Start(numberOfPlayers, ws)

    winner := ws.WaitForMsg()
    p.game.Finish(winner)
}
```

Refactor

The server code was a very small change so there's not a lot to change here but the test code still has a `time.Sleep` call because we have to wait for our server to do its work asynchronously.

We can refactor our helpers `assertGameStartedWith` and `assertFinishCalledWith` so that they can retry their assertions for a short period before failing.

Here's how you can do it for `assertFinishCalledWith` and you can use the same approach for the other helper.

```
func assertFinishCalledWith(t *testing.T, game *GameSpy, winner string) {
    t.Helper()

    passed := retryUntil(500*time.Millisecond, func() bool {
        return game.FinishCalledWith == winner
    })

    if !passed {
        t.Errorf("expected finish called with '%s' but got '%s'", winner, game.FinishCalledW
    }
}
```

Here is how `retryUntil` is defined

```
func retryUntil(d time.Duration, f func() bool) bool {
    deadline := time.Now().Add(d)
    for time.Now().Before(deadline) {
        if f() {
            return true
        }
    }
    return false
}
```

Wrapping up

Our application is now complete. A game of poker can be started via a web browser and the users are informed of the blind bet value as time goes by via WebSockets. When the game finishes they can record the winner which is persisted using code we wrote a few chapters ago. The players can find out who is the best (or luckiest) poker player using the website's `/league` endpoint.

Through the journey we have made mistakes but with the TDD flow we have never been very far away from working software. We were free to keep iterating and experimenting.

The final chapter will retrospect on the approach, the design we've arrived at and tie up some loose ends.

We covered a few things in this chapter

WebSockets

- Convenient way of sending messages between clients and servers that does not require the client to keep polling the server. Both the client and server code we have is very simple.
- Trivial to test, but you have to be wary of the asynchronous nature of the tests

Handling code in tests that can be delayed or never finish

- Create helper functions to retry assertions and add timeouts.
- We can use go routines to ensure the assertions dont block anything and then use channels to let them signal that they have finished, or not.
- The `time` package has some helpful functions which also send signals via channels about events in time so we can set timeouts

OS Exec

You can find all the code here

keith6014 asks on reddit

> I am executing a command using os/exec.Command() which generated XML data. The command will be executed in a function called GetData().
>
> In order to test GetData(), I have some testdata which I created.
>
> In my _test.go I have a TestGetData which calls GetData() but that will use os.exec, instead I would like for it to use my testdata.
>
> What is a good way to achieve this? When calling GetData should I have a "test" flag mode so it will read a file ie GetData(mode string)?

A few things

- When something is difficult to test, it's often due to the separation of concerns not being quite right
- Dont add "test modes" into your code, instead use Dependency Injection so that you can model your dependencies and separate concerns.

I have taken the liberty of guessing what the code might look like

```
type Payload struct {
    Message string `xml:"message"`
}

func GetData() string {
    cmd := exec.Command("cat", "msg.xml")
```

```
    out, _ := cmd.StdoutPipe()
    var payload Payload
    decoder := xml.NewDecoder(out)

    // these 3 can return errors but I'm ignoring for brevity
    cmd.Start()
    decoder.Decode(&payload)
    cmd.Wait()

    return strings.ToUpper(payload.Message)
}
```

- It uses `exec.Command` which allows you to execute an external command to the process
- We capture the output in `cmd.StdoutPipe` which returns us a `io.ReadCloser` (this will become important)
- The rest of the code is more or less copy and pasted from the excellent documentation.
- We capture any output from stdout into an `io.ReadCloser` and then we `Start` the command and then wait for all the data to be read by calling `Wait`. In between those two calls we decode the data into our `Payload` struct.

Here is what is contained inside `msg.xml`

```
<payload>
    <message>Happy New Year!</message>
</payload>
```

I wrote a simple test to show it in action

```
func TestGetData(t *testing.T) {
    got := GetData()
    want := "HAPPY NEW YEAR!"

    if got != want {
        t.Errorf("got '%s', want '%s'", got, want)
    }
}
```

Testable code

Testable code is decoupled and single purpose. To me it feels like there are two main concerns for this code

1. Retrieving the raw XML data

2. Decoding the XML data and applying our business logic (in this case `strings.ToUpper` on the `<message>`)

The first part is just copying the example from the standard lib.

The second part is where we have our business logic and by looking at the code we can see where the "seam" in our logic starts; it's where we get our `io.ReadCloser`. We can use this existing abstraction to separate concerns and make our code testable.

The problem with GetData is the business logic is coupled with the means of getting the XML. To make our design better we need to decouple them

Our `TestGetData` can act as our integration test between our two concerns so we'll keep hold of that to make sure it keeps working.

Here is what the newly separated code looks like

```
type Payload struct {
    Message string `xml:"message"`
}

func GetData(data io.Reader) string {
    var payload Payload
    xml.NewDecoder(data).Decode(&payload)
    return strings.ToUpper(payload.Message)
}

func getXMLFromCommand() io.Reader {
    cmd := exec.Command("cat", "msg.xml")
    out, _ := cmd.StdoutPipe()

    cmd.Start()
    data, _ := ioutil.ReadAll(out)
    cmd.Wait()

    return bytes.NewReader(data)
}

func TestGetDataIntegration(t *testing.T) {
    got := GetData(getXMLFromCommand())
    want := "HAPPY NEW YEAR!"

    if got != want {
        t.Errorf("got '%s', want '%s'", got, want)
    }
}
```

Now that `GetData` takes its input from just an `io.Reader` we have made it testable and it is no longer concerned how the data is retrieved; people can re-use the function with anything that returns an `io.Reader` (which is extremely common). For example we could start fetching the XML from a URL instead of the command line.

```
func TestGetData(t *testing.T) {
    input := strings.NewReader(`
<payload>
    <message>Cats are the best animal</message>
</payload>`)

    got := GetData(input)
    want := "CATS ARE THE BEST ANIMAL"

    if got != want {
        t.Errorf("got '%s', want '%s'", got, want)
    }
}
```

Here is an example of a unit test for `GetData`.

By separating the concerns and using existing abstractions within Go testing our important business logic is a breeze.

Tipos de erro

You can find all the code here

Creating your own types for errors can be an elegant way of tidying up your code, making your code easier to use and test.

Pedro on the Gopher Slack asks

> If I'm creating an error like `fmt.Errorf("%s must be foo, got %s", bar, baz)`, is there a way to test equality without comparing the string value?

Let's make up a function to help explore this idea.

```
// DumbGetter will get the string body of url if it gets a 200
func DumbGetter(url string) (string, error) {
    res, err := http.Get(url)

    if err != nil {
        return "", fmt.Errorf("problem fetching from %s, %v", url, err)
    }

    if res.StatusCode != http.StatusOK {
```

```
        return "", fmt.Errorf("did not get 200 from %s, got %d", url, res.StatusCode)
    }

    defer res.Body.Close()
    body, _ := ioutil.ReadAll(res.Body) // ignoring err for brevity

    return string(body), nil
}
```

It's not uncommon to write a function that might fail for different reasons and we want to make sure we handle each scenario correctly.

As Pedro says, we *could* write a test for the status error like so.

```
t.Run("when you dont get a 200 you get a status error", func(t *testing.T) {

    svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req *http.Reque
        res.WriteHeader(http.StatusTeapot)
    }))
    defer svr.Close()

    _, err := DumbGetter(svr.URL)

    if err == nil {
        t.Fatal("expected an error")
    }

    want := fmt.Sprintf("did not get 200 from %s, got %d", svr.URL, http.StatusTeapot)
    got := err.Error()

    if got != want {
        t.Errorf(`got "%v", want "%v"`, got, want)
    }
})
```

This test creates a server which always returns `StatusTeapot` and then we use its URL as the argument to `DumbGetter` so we can see it handles non `200` responses correctly.

Problems with this way of testing

This book tries to emphasise *listen to your tests* and this test doesn't *feel* good:

- We're constructing the same string as production code does to test it
- It's annoying to read and write
- Is the exact error message string what we're *actually concerned with* ?

What does this tell us? The ergonomics of our test would be reflected on another bit of code trying to use our code.

How does a user of our code react to the specific kind of errors we return? The best they can do is look at the error string which is extremely error prone and horrible to write.

What we should do

With TDD we have the benefit of getting into the mindset of:

> How would *I* want to use this code?

What we could do for `DumbGetter` is provide a way for users to use the type system to understand what kind of error has happened.

What if `DumbGetter` could return us something like

```
type BadStatusError struct {
    URL    string
    Status int
}
```

Rather than a magical string, we have actual *data* to work with.

Let's change our existing test to reflect this need

```
t.Run("when you dont get a 200 you get a status error", func(t *testing.T) {

    svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req *http.Reque
        res.WriteHeader(http.StatusTeapot)
    }))
    defer svr.Close()

    _, err := DumbGetter(svr.URL)

    if err == nil {
        t.Fatal("expected an error")
    }

    got, isStatusErr := err.(BadStatusError)

    if !isStatusErr {
        t.Fatalf("was not a BadStatusError, got %T", err)
    }

    want := BadStatusError{URL:svr.URL, Status:http.StatusTeapot}

    if got != want {
```

```
        t.Errorf("got %v, want %v", got, want)
    }
})
```

We'll have to make `BadStatusError` implement the error interface.

```
func (b BadStatusError) Error() string {
    return fmt.Sprintf("did not get 200 from %s, got %d", b.URL, b.Status)
}
```

What does the test do?

Instead of checking the exact string of the error, we are doing a type assertion on the error to see if it is a `BadStatusError`. This reflects our desire for the *kind* of error clearer. Assuming the assertion passes we can then check the properties of the error are correct.

When we run the test, it tells us we didn't return the right kind of error

```
--- FAIL: TestDumbGetter (0.00s)
   --- FAIL: TestDumbGetter/when_you_dont_get_a_200_you_get_a_status_error (0.00s)
      error-types_test.go:56: was not a BadStatusError, got *errors.errorString
```

Let's fix `DumbGetter` by updating our error handling code to use our type

```
if res.StatusCode != http.StatusOK {
    return "", BadStatusError{URL: url, Status: res.StatusCode}
}
```

This change has had some *real positive effects*

- Our `DumbGetter` function has become simper, it's no longer concerned with the intricacies of an error string, it just creates a `BadStatusError`.
- Our tests now reflect (and document) what a user of our code *could* do if they decided they wanted to do some more sophisticated error handling than just logging. Just do a type assertion and then you get easy access to the properties of the error.
- It is still "just" an `error`, so if they choose to they can pass it up the call stack or log it like any other `error`.

Wrapping up

If you find yourself testing for multiple error conditions dont fall in to the trap of comparing the error messages.

This leads to flaky and difficult to read/write tests and it reflects the difficulties the users of your code will have if they also need to start doing things differently depending on the kind of errors that have occurred.

Always make sure your tests reflect how *you'd* like to use your code, so in this respect consider creating error types to encapsulate your kinds of errors. This makes handling different kinds of errors easier for users of your code and also makes writing your error handling code simpler and easier to read.