3

Funções

Nos primórdios da programação, formávamos nossos sistemas com rotinas e sub-rotinas. Já na era do Fortran e do PL/1, usávamos programas, subprogramas e funções. De tudo isso, apenas função prevaleceu. As funções são a primeira linha de organização em qualquer programa. Escrevê-las bem é o assunto deste capítulo.

Veja o código na Listagem 3.1. É difícil encontrar uma função grande em FitNesse[1], mas procurando um pouco mais encontramos uma. Além de ser longo, seu código é repetido, há diversas strings estranhas e muitos tipos de dados e APIs esquisitos e nada óbvios. Veja o quanto você consegue compreender nos próximos três minutos.

Listagem 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
              SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

1. [1. Ferramenta de teste de código aberto, www.fitnese.org.]

Listagem 3-1 (continuação)
HtmlUtil.java (FitNesse 20070619)

```
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Conseguiu entender a função depois desses três minutos estudando-a? Provavelmente não. Há muita coisa acontecendo lá em muitos níveis diferentes de . Há strings estranhas e chamadas a funções esquisitas misturadas com dois `if` aninhados controlados por flags.

Entretanto, com umas poucas extrações simples de métodos, algumas renomeações e um pouco de reestruturação,fui capaz de entender o propósito da função nas nove linhas da Listagem 3.2. Veja se você consegue compreender também em três minutos.

Listagem 3-2
HtmlUtil.java (refatorado)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

A menos que já estivesse estudando o FitNesse, provavelmente você não entendeu todos os detalhes.

Ainda assim você talvez tenha compreendido que essa função efetua a inclusão de algumas páginas SetUp e TearDown em uma página de teste e, então, exibir tal página em HTML. Se estiver familiarizado com o JUnit[2], você já deve ter percebido que essa função pertença a algum tipo de framework de teste voltado para a Web. E você está certo. Deduzir tal informação da Listagem 3.2 é muito fácil, mas ela está bastante obscura na Listagem 3.1.

Então, o que torna uma função como a da Listagem 3.2 fácil de ler e entender? Como fazer uma função transmitir seu propósito? Quais atributos dar às nossas funções que permitirão um leitor comum deduzir o tipo do programa ali contido?

Pequenas!

A primeira regra para funções é que elas devem ser pequenas. A segunda é que *precisam ser mais espertas do que isso*. Não tenho como justificar essa afirmação. Não tenho referências de pesquisas que mostrem que funções muito pequenas são melhores. Só posso dizer que por cerca de quatro décadas tenho criado funções de tamanhos variados. Já escrevi diversos monstros de 3.000 linhas; bastantes funções de 100 a 300 linhas; e funções que tinham apenas de 20 a 30 linhas. Essa experiência me ensinou que, ao longo de muitas tentativas e erros, as funções devem ser muito pequenas.

Na década de 1980, costumávamos dizer que uma função não deveria ser maior do que a tela.

É claro que na época usávamos as telas VT100, de 24 linhas por 80 colunas, e nossos editores usavam 4 linhas para fins gerenciamento. Hoje em dia, com fontes reduzidas e um belo e grande monitor, você consegue colocar 150 caracteres em uma linha – não se deve ultrapassar esse limite—e umas 100 linhas ou mais por tela—as funções não devem chegar a isso tudo, elas devem ter no máximo 20 linhas.

O quão pequena deve ser uma função? Em 1999, fui visitar Kent Beck em sua casa, em Oregon, EUA. Sentamo-nos e programamos um pouco juntos. Em certo ponto, ele me mostrou um simpático programa de nome Java/Swing o qual ele chamava de *Sparkle*. Ele produzia na tela um efeito visual similar a uma varinha mágica da fada madrinha do filme da Cinderela. Ao mover o mouse, faíscas (*sparkles,* em inglês) caíam do ponteiro do mouse com um belo cintilar até o fim da janela, como se houvesse gravidade na tela. Quando Kent me mostrou o código, fiquei surpreso com tamanha pequeneza das funções. Eu estava acostumado a funções que seguiam por quilômetros em programas do Swing. Cada função *neste* programa tinha apenas duas, ou três, ou quatro linhas. Esse deve ser o tamanho das suas funções[3].

O quão pequenas devem ser suas funções? Geralmente menores do que a da Listagem 3.2! Na verdade, a Listagem 3.2 deveria ser enxuta para a Listagem 3.3.

2. Ferramenta de código aberto de teste de unidade para Java, www.junit.org.

Listagem 3-3
HtmlUtil.java (refatorado novamente)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Blocos e Endentação

Aqui quero dizer que blocos dentro de instruções `if`, `else`, `while` e outros devem ter apenas uma linha. Possivelmente uma chamada de função. Além de manter a função pequena, isso adiciona um valor significativo, pois a função chamada de dentro do bloco pode receber um nome descritivo. Isso também implica que as funções não devem ser grandes e ter estruturas aninhadas. Portanto, o nível de endentação de uma função deve ser de, no máximo, um ou dois. Isso, é claro, facilita a leitura e compreensão das funções.

Faça Apenas uma Coisa

Deve ter ficado claro que a Listagem 3.1 faz muito mais de uma coisa. Ela cria buffers, pega páginas, busca por páginas herdadas, exibe caminhos, anexa strings estranhas e gera HTML, dentre outras coisas. A Listagem 3.1 vive ocupada fazendo diversas coisas diferentes. Por outro lado, a Listagem 3.3 faz apenas uma coisa simples. Ela inclui SetUp e TearDown em páginas de teste.

O conselho a seguir tem aparecido de uma forma ou de outra por 30 anos ou mais.

AS FUNÇÕES DEVEM FAZER UMA COISA. DEVEM FAZÊ-LA BEM. DEVEM FAZER APENAS ELA.

O problema dessa declaração é que é difícil saber o que é "uma coisa". A Listagem 3.3 faz uma coisa? É fácil dizer que ela faz três:

1. Determina se a página é de teste.
2. Se for, inclui SetUps e TearDowns.
3. Exibe a página em HTML.

Então, uma ou três coisas? Note que os três passos da função estão em um nível de abaixo do nome da função. Podemos descrever a função com um breve parágrafo TO[4]:

4. A linguagem LOGO usava a palavra "TO" ("PARA") da mesma forma que Ruby e Python usam "def". Portanto, toda função começa com a palavra "TO",

TO RenderPageWithSetupsAndTeardowns, verificamos se a página é de teste, se for, incluímos SetUps e TearDowns. Em ambos os casos, exibimos a página em HTML.

Se uma função faz apenas aqueles passos em um nível abaixo do nome da função, então ela está fazendo uma só coisa. Apesar de tudo, o motivo de criarmos função é para decompor um conceito maior (em outras palavras, o nome da função) em uma série de passos no próximo nível de abstração.

Deve estar claro que a Listagem 3.1 contém passos em muitos níveis diferentes de . Portanto, obviamente ela faz mais de uma coisa. Mesmo a Listagem 3.2 possui dois níveis de abstração , como comprovado pela nossa capacidade de redução. Mas ficaria muito difícil reduzir a Listagem 3.3 de modo significativo. Poderíamos colocar a instrução `if` numa função chamada `includeSetupsAndTeardownsIfTestPage`, mas isso simplesmente reformula o código, sem modificar o nível de .

Portanto, outra forma de saber se uma função faz mais de "uma coisa" é se você pode extrair outra função dela a partir de seu nome que não seja apenas uma reformulação de sua implementação (G34).

Seções Dentro de Funções

Veja a Listagem 4.7 na página 71. Note que a função `generatePrimes` está dividida em seções, como *declarações, inicializações e seleção*. Esse é um indício óbvio de estar fazendo mais de uma coisa. Não dá para, de forma significativa, dividir em seções as funções que fazem apenas uma coisa.

Um Nível de Abstração por Função

A fim de confirmar se nossas funções fazem só "uma coisa". Precisamos verificar se todas as instruções dentro da função estão no mesmo nível de abstração. É fácil ver como a Listagem 3.1 viola essa regra. Há outros conceitos lá que estão em um nível de bem alto, como o `getHtml();` outros que estão em um nível intermediário, como `String pagePathName = PathParser.render (pagePath);` e outros que estão em um nível consideravelmente baixo, como `.append("\n").`

Vários níveis de dentro de uma função sempre geram confusão. Os leitores podem não conseguir dizer se uma expressão determinada é um conceito essencial ou um mero detalhe. Pior, como janelas quebradas, uma vez misturados os detalhes aos conceitos, mais e mais detalhes tendem a se agregar dentro da função.

Ler o Código de Cima para Baixo: *Regra Decrescente*

Queremos que o código seja lido de cima para baixo, como uma narrativa[5]. Desejamos que cada função seja seguida pelas outras no próximo nível de de modo que possamos ler o programa descendo um nível de de cada vez conforme percorremos a lista de funções. Chamamos isso de *Regra Decrescente*.

Em outras palavras, queremos poder ler o programa como se fosse uma série de parágrafos *TO*, cada um descrevendo o nível atual de e fazendo referência aos parágrafos *TO* consecutivos no próximo nível abaixo.

> *Para incluir SetUps e TearDowns, incluímos os primeiros, depois o conteúdo da página de teste e, então, adicionamos os segundos. Para incluir SetUps, adicionamos o suite setup, se este for uma coleção, incluímos o setup normal. Para incluir o suite setup, buscamos na hierarquia acima a página "SuiteSetUp" e adicionamos uma instrução de inclusão com o caminho àquela página. Para procurar na hierarquia acima...*

Acaba sendo muito difícil para os programadores aprenderem a seguir essa regra e criar funções que fiquem em apenas um nível de . Mas aprender esse truque é também muito importante, pois ele é o segredo para manter as funções curtas e garantir que façam apenas "uma coisa". Fazer com que a leitura do código possa ser feita de cima para baixo como uma série de parágrafos *TO* é uma técnica eficiente para manter o nível de consistente.

Veja a listagem 3.7 no final deste capítulo. Ela mostra toda a função testableHtml refatorada de acordo com os princípios descrito aqui. Note como cada função indica a seguinte e como cada uma mantém um nível consistente de .

Estrutura Switch

É difícil criar uma estrutura `switch` pequena[6], pois mesmo uma com apenas dois cases é maior do que eu gostaria que fosse um bloco ou uma função. Também é difícil construir uma que fala apenas uma coisa. Por padrão, as estruturas `switch` sempre fazem N coisas. Infelizmente, nem sempre conseguimos evitar o uso do `switch`, mas *podemos* nos certificar se cada um está em uma classe de baixo nível e nunca é repetido. Para isso, usamos o polimorfismo.

Veja a Listagem 3.4. Ela mostra apenas uma das operações que podem depender do tipo de funcionário (*employee*, em inglês).

---

5. [KP78], p. 37.

Listagem 3-4
Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

Esta função tem vários problemas. Primeiro, ela é grande, e quando se adiciona novos tipos de funcionários ela crescerá mais ainda. Segundo, obviamente ela faz mais de uma coisa. Terceiro, ela viola o Principio da Responsabilidade Única[7] (SRP, sigla em inglês) por haver mais de um motivo para alterá-la. Quarto, ela viola o Princípio de Aberto-Fechado[8] (OCP, sigla em inglês), pois precisa ser modificada sempre que novos tipos forem adicionados. Mas, provavelmente, o pior problema com essa função é a quantidade ilimitada de outras funções que terão a mesma estrutura. Por exemplo, poderíamos ter

```
isPayday(Employee e, Date date)
```

ou

```
deliverPay(Employee e, Money pay)
```

ou um outro grupo. Todas teriam a mesma estrutura deletéria.

A solução (veja a Listagem 3.5) é inserir a estrutura switch no fundo de uma ABSTRACT FACTORY[9] e jamais deixar que alguém a veja. A factory usará o `switch` para criar instâncias apropriadas derivadas de Employee, e as funções, como `calculatePay`, `isPayday` e `deliverPay`, serão enviadas de forma polifórmica através da interface `Employee`.

Minha regra geral para estruturas switch é que são aceitáveis se aparecerem apenas uma vez, como para a criação de objetos polifórmicos, e estiverem escondidas atrás de uma relação de herança de modo que o resto do sistema não possa enxergá-la [G23]. É claro que cada caso é um caso e haverá vezes que não respeitarei uma ou mais partes dessa regra.

Listagem 3-5
`Employee e Factory`

```
public abstract class Employee {
public abstract boolean isPayday();
public abstract Money calculatePay();
public abstract void deliverPay(Money pay);
}
---------------
```

7. a. http://en.wikipedia.org/wiki/Single_responsibility_principle

Use Nomes Descritivos

Na Listagem 3.7, eu mudei o nome do exemplo de nossa função `testableHtml` para `SetupTeardownIncluder.render`, que é bem melhor, pois descreve o que a função faz. Também dei a cada método privado nomes igualmente descritivos, como `isTestable` ou `includeSetupAndTeardownPages`. É difícil superestimar o valor de bons nomes. Lembre-se do princípio de Ward: *"Você sabe que está criando um código limpo quando cada rotina que você lê é como você esperava"*. Metade do esforço para satisfazer esse princípio é escolher bons nomes para funções pequenas que fazem apenas uma coisa. Quando menor e mais centralizada for a função, mais fácil será pensar em um nome descritivo.

Não tenha medo de criar nomes extensos, pois eles são melhores do que um pequeno e enigmático. Um nome longo e descritivo é melhor do que um comentário extenso e descritivo. Use uma convenção de nomenclatura que possibilite uma fácil leitura de nomes de funções com várias palavras e, então, use estas para dar à função um nome que explique o que ela faz.

Não se preocupe com o tempo ao escolher um nome. Na verdade, você deve tentar vários nomes e, então, ler o código com cada um deles. IDEs modernas, como Eclipse ou IntelliJ, facilita a troca de nomes. Utilize uma dessas IDEs e experimente diversos nomes até encontrar um que seja bem descritivo.

Selecionar nomes descritivos esclarecerá o modelo do módulo em sua mente e lhe ajudará a melhorá-lo. É comum que ao buscar nomes adequados resulte numa boa reestruturação do código.

Seja consistente nos nomes. Use as mesmas frases, substantivos e verbos nos nomes de funções de seu módulo. Considere, por exemplo, os nomes `includeSetup-AndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. A fraseologia nesses nomes permite uma sequência de fácil dedução. Na verdade, se eu lhe mostrasse apenas a série acima, você se perguntaria: "O que aconteceu com `includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`?", como isso é "... como o que você esperava?".

Parâmetros de Funções

A quantidade ideal de parâmetros para uma função é zero (nulo). Depois vem um (mônade), seguido de dois (díade). Sempre que possível devem-se evitar três parâmetros (tríade). Para mais de três deve-se ter um motivo muito especial (políade) – mesmo assim não devem ser usados.

Parâmetros são complicados. Eles requerem bastante conceito. É por isso que me livrei de quase todos no exemplo. Considere, por exemplo, o `StringBuffer`. Poderíamos tê-lo passado como parâmetro em vez de instanciá-lo como uma variável, mas então nossos leitores teriam de interpretá-lo sempre que o vissem. Ao ler a

estória contada por pelo módulo, fica mais fácil entender `includeSetupPage()` do que `includeSetupPageInto(newPage-Content)`. O parâmetro não está no nível de que o nome função, forçando-lhe reconhecer de um detalhe (ou seja, o `StringBuffer`) que não seja importante particularmente naquele momento.

Os parâmetros são mais difíceis ainda a partir de um ponto de vista de testes. Imagine a dificuldade de escrever todos os casos de teste para se certificar de que todas as várias combinações de parâmetros funcionem adequadamente. Se não houver parâmetros, essa tarefa é simples. Se houver um, não é tão difícil assim.

Com dois, a situação fica um pouco mais desafiadora. Com mais de dois, pode ser desencorajador testar cada combinação de valores apropriados. Os parâmetros de saída são ainda mais difíceis de entender do que os de entrada. Quando lemos uma função, estamos acostumados à ideia de informações *entrando* na função através de parâmetros e *saindo* através do valor retornado. Geralmente não esperamos dados saindo através de parâmetros. Portanto, parâmetros de saída costumam nos deixar surpresos e fazer com que leiamos novamente.

Um parâmetro de entrada é a melhor coisa depois de zero parâmetro. É fácil entender `SetupTeardown-Includer.render(pageData)`. Está óbvio que renderizemos os dados no objeto `pageData`.

Formas Mônades Comuns

Há duas razões bastante comuns para se passar um único parâmetro a uma função. Você pode estar fazendo uma pergunta sobre aquele parâmetro, como em `boolean fileExists("MyFile")`. Ou você pode trabalhar naquele parâmetro, transformando-o em outra coisa e retornando-o. Por exemplo, `InputStream fileOpen("MyFile")` transforma a `String` do nome de um arquivo em um valor retornado por `InputStream`. São esses dois usos que os leitores esperam ver em uma função.

Você deve escolher nomes que tornem clara a distinção, e sempre use duas formas em um contexto consistente. (Veja a seguir *Separação comando-consulta*). Uma forma menos comum mas ainda bastante útil de um parâmetro para uma função é um *evento*. Nesta forma, há um parâmetro de entrada, mas nenhum de saída. O programa em si serve para interpretar a chamada da função como um evento, e usar o parâmetro para alterar o estado do sistema, por exemplo, `void passwordAttemptFailedNtimes(int attempts)`. Use esse tipo com cautela. Deve ficar claro para o leitor que se trata de um evento. Escolha os nomes e os contextos com atenção.

Tente evitar funções mônades que não sigam essas formas, por exemplo, `void includeSetupPageInto(StringBuffer pageText)`. Usar um parâmetro de saída em vez de um valor de retorno para uma modificação fica confuso. Se uma função vai transformar seu parâmetro de entrada, a alteração deve aparecer como o valor retornado. De fato, `StringBuffer transform(StringBuffer in)` é melhor do que `void transform-(StringBuffer out)`, mesmo que a implementação do primeiro simplesmente retorne o parâmetro de entrada. Pelo menos ele ainda segue o formato de uma modificação.

Parâmetros Lógicos

Esses parâmetros são feios. Passar um booleano para uma função certamente é uma prática horrível, pois ele complica imediatamente a assinatura do método, mostrando explicitamente que a função faz mais de uma coisa. Ela faz uma coisa se o valor for verdadeiro, e outra se for falso!

Na Listagem 3.7, não tínhamos alternativa, pois os chamadores já estavam passando aquela flag (valor booleano) como parâmetro, e eu queria limitar o escopo da refatoração à função e para baixo. Mesmo assim, a chamada do método `render(true)` é muito confusa para um leitor simples. Analisar a chamada e visualizar `render(boolean isSuite)` ajuda um pouco, mas nem tanto. Deveríamos dividir a função em duas: `renderForSuite()` e `renderForSingleTest()`.

Funções Díades

Uma função com dois parâmetros é mais difícil de entender do que uma com um (mônade). Por exemplo, é mais fácil compreender `writeField(name)` do que `writeField(output-Stream, name)`[10]. Embora o significado de ambas esteja claro, a primeira apresenta seu propósito explicitamente quando a lemos. A segunda requer uma pequena pausa até aprendermos a ignorar o primeiro parâmetro. E *isso*, é claro, acaba resultando em problemas, pois nunca devemos ignorar qualquer parte do código. O local que ignoramos é justamente aonde se esconderão os bugs.

Há casos, é claro, em que dois parâmetros são necessários como, por exemplo, em `Point p = new Point(0,0)`. Os pontos de eixos cartesianos naturalmente recebem dois parâmetros. De fato, ficaríamos surpresos se víssemos `new Point(0)`. Entretanto, os dois parâmetros neste caso *são componentes de um único valor*! Enquanto que `output-Stream` e name não são partes de um mesmo valor.

Mesmo funções díades óbvias, como `assertEquals(expected, actual)`, são problemáticas.

Quantas vezes você já colocou actual onde deveria ser expected? Os dois parâmetros não possuem uma ordem pré-determinada natural. A ordem expected, actual é uma convenção que requer prática para assimilá-la.

Díades não são ruins, e você certamente terá de usá-las. Entretanto, deve-se estar ciente de que haverá um preço a pagar e, portanto, deve-se pensar em tirar proveito dos mecanismos disponíveis a você para convertê-los em mônades. Por exemplo, você poderia tornar o método `writeField` um membro de `outputStream` de modo que pudesse dizer `outputStream.writeField(name);` tornar `outputStream` uma variável membro da classe em uso de modo que não precisasse passá-lo por parâmetro; ou extrair uma nova classe, como `FieldWriter`, que receba o outputStream em seu construtor e possua um método `write`.

Tríades

Funções que recebem três parâmetros são consideravelmente mais difíceis de entender do que as díades. A questão de ordenação, pausa e ignoração apresentam mais do que o dobro de dificuldade. Sugiro que você pense bastante antes de criar uma tríade.

Por exemplo, considere a sobrecarga comum de `assertEquals` que recebe três parâmetros: `assertEquals(message, expected, actual)`. Quantas vezes você precisou ler o parâmetro `message` e deduzir o que ele carrega? Muitas vezes já me deparei com essa tríade em particular e tive de fazer uma pausa. Na verdade, *toda vez que a vejo*, tenho de ler novamente e, então, a ignoro.

---

10. Acabei de refatorar um módulo que usava uma díade. Consegui tornar o outputStream um campo da classe e converter todas as chamadas ao writeField para

formato imbuímos os nomes dos parâmetros no nome da função. Por exemplo, pode ser melhor escrever `assertEquals` do que `assertExpectedEqualsActual(expected, actual)`, o que resolveria o problema de ter de lembrar a ordem dos parâmetros.

Evite Efeitos Colaterais

Efeitos colaterais são mentiras. Sua função promete fazer apenas uma coisa, mas ela também faz outras *coisas escondida*. Às vezes, ela fará alterações inesperadas nas variáveis de sua própria classe. Às vezes, ela adicionará as variáveis aos parâmetros passados à função ou às globais do sistema. Em ambos os casos elas são "verdades" enganosas e prejudiciais, que geralmente resultam em acoplamentos temporários estranhos e dependências.

Considere, por exemplo, a função aparentemente inofensiva na Listagem 3.6. Ela usa um algoritmo padrão para comparar um `userName (nome de usuário)` a um `password (senha)`. Ela retorna `true (verdadeiro)` se forem iguais, e `false (falso)` caso contrário. Mas há também um efeito colateral. Consegue identificá-lo?

Listagem 3-6
`UserValidator.java`

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

O efeito colateral é a chamada ao `Session.initialize()`, é claro. A função `checkPassword`, segundo seu nome, diz que verifica a senha. O nome não indica que ela inicializa a sessão. Portanto, um chamador que acredita no que diz o nome da função corre o risco de apagar os dados da sessão existente quando ele decidir autenticar do usuário.

Esse efeito colateral cria um acoplamento temporário. Isto é, `checkPassword` só poderá ser chamado em determinadas horas (em outras palavras, quando for seguro inicializar a sessão). Se for chamado fora de ordem, sem querer, os dados da sessão poderão ser perdidos. Os acoplamentos temporários são confusos, especialmente quando são um efeito colateral. Se for preciso esse tipo

de acoplamento, é preciso deixar claro no nome da função. Neste caso, poderíamos renomear a função para `checkPasswordAndInitializeSession`, embora isso certamente violaria o "fazer apenas uma única coisa".

Parâmetros de Saída

Os parâmetros são comumente interpretados como *entradas* de uma função. Se já usa o programa há alguns anos, estou certo de que você já teve de voltar e ler novamente um parâmetro que era, na verdade, de *saída*, e não de entrada. Por exemplo:

```
appendFooter(s);
```

Essa função anexa `s` como rodapé (Footer, em inglês) em algo? Ou anexa um rodapé a `s`? `s` é uma entrada ou uma saída? Não precisa olhar muito a assinatura da função para ver:

```
public void appendFooter(StringBuffer report)
```

Isso esclarece a questão, mas à custa da verificação da declaração da função. Qualquer coisa que lhe force a verificar a assinatura da função é equivalente a uma relida. Isso é uma interrupção do raciocínio e deve ser evitado.

Antes do surgimento da programação orientada a objeto, às vezes era preciso ter parâmetros de saída. Entretanto, grande parte dessa necessidade sumiu nas linguagens OO, pois seu *propósito* é servir como um parâmetro de saída. Em outras palavras, seria melhor invocar `appendFooter` como:

```
report.appendFooter();
```

De modo geral, devem-se evitar parâmetros de saída. Caso sua função precise alterar o estado de algo, faça-a mudar o estado do objeto que a pertence.

Separação comando-consulta

As funções devem fazer ou responder algo, mas não ambos. Sua função ou altera o estado de um objeto ou retorna informações sobre ele. Efetuar as duas tarefas costuma gerar confusão. Considere, por exemplo, a função abaixo:

```
public boolean set(String attribute, String value);
```

Esta função define o valor de um dado atributo e retorna `true` (verdadeiro) se obtiver êxito e `false` (falso) se tal atributo não existir. Isso leva a instruções estranhas como:

```
if (set("username", "unclebob"))...
```

Imagine isso pelo ponto de vista do leitor. O que isso significa? Está perguntando se o atributo "`username`" anteriormente recebeu o valor "`unclebob`"? Ou se "`username`" obteve êxito ao receber o valor "`unclebob`"? É difícil adivinhar baseando-se na chamada, pois não está claro se a palavra "set" é um verbo ou um adjetivo.

O intuito do autor era que set fosse um verbo, mas no contexto da estrutura `if`, *parece* um adjetivo. Portanto, a instrução lê-se "Se o atributo username anteriormente recebeu o valor `unclebob`" e não "atribua o valor `unclebob` ao atributo `username`, e se isso funcionar, então...". Poderíamos tentar resolver isso renomeando a função set para `setAndCheckIfExists`, mas não ajudaria muito para a legibilidade da estrutura `if`.

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
    ...
}
```

Prefira exceções a retorno de códigos de erro

Fazer funções retornarem códigos de erros é uma leve violação da separação comando-consulta, pois os comandos são usados como expressões de comparação em estruturas `if`.

```
if (deletePage(page) == E_OK)
```

O problema gerado aqui não é a confusão verbo/adjetivo, mas sim a criação de estruturas aninhadas.

Ao retornar um código de erro, você cria um problema para o chamador, que deverá lidar imediatamente com o erro.

```
if (deletePage(page) == E_OK) {
    if (registry.deleteReference(page.name) == E_OK) {
            if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
                    logger.log("página excluída");
            } else {
                    logger.log("configKey não foi excluída");
            }
    } else {
            logger.log("deleteReference não foi excluído do
registro");
    }
} else {
    logger.log("a exclusão falhou");
    return E_ERROR;
}
```

Por outro lado, se você usar exceções em vez de retornar códigos de erros, então o código de tratamento de erro poderá ficar separado do código e ser simplificado:

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}
```

Extraia os blocos try/catch

Esses blocos não têm o direito de serem feios. Eles confundem a estrutura do código e misturam o tratamento de erro com o processamento normal do código. Portanto, é melhor colocar as estruturas `try` e `catch` em suas próprias funções.

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}
private void deletePageAndAllReferences(Page page) throws Exception
{
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

A função delete acima só faz tratamento de erro. E é fácil entendê-la e seguir adiante. A função `deletePageAndAllReferences` só trata de processos que excluem toda uma página. Pode-se ignorar o tratamento de erro. Isso oferece uma boa separação que facilita a compreensão e alteração do código.

Tratamento de erro é uma coisa só

As funções devem fazer uma coisa só. Tratamento de erro é uma coisa só. Portanto, uma função que trata de erros não deve fazer mais nada. Isso implica (como no exemplo acima) que a palavra try está dentro de uma função e deve ser a primeira instrução e nada mais deve vir após os blocos catch/finally.

`Error.java, o chamariz à dependência`

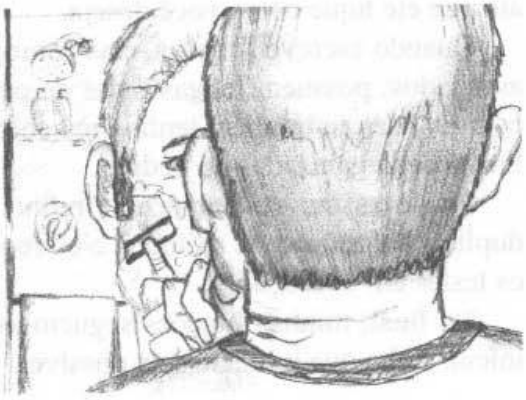
Retornar códigos de erro costuma implicar que há classes ou enum nos quais estão definidos todos os códigos de erro.

```
public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Classes como esta são *chamarizes à dependência*, muitas outras classes devem importá-las e usá-las. Portanto, quando o `enum` da classe `Error` `enum`, é preciso recompilar todas as outras classes e redistribuí-las[11]. Isso coloca uma pressão negativa na classe Error. Os programadores não querem adicionar novos erros porque senão eles teriam de compilar e distribuir tudo novamente. Por isso, eles reutilizam códigos de erros antigos em vez de adicionar novos.

Quando se usam exceções em vez de códigos de erro, as novas exceções são *derivadas* da classe de exceções. Podem-se adicioná-las sem ter de recompilar ou redistribuir[12].

Evite repetição[13]



Leia novamente com atenção a Listagem 3.1 e notará que há um algoritmo que se repete quatro vezes em quatro casos: `SetUp`, `SuiteSetUp`, `TearDown` e `SuiteTearDown`. Não é fácil perceber essa duplicação, pois as quatro instâncias estão misturadas com outros códigos e não estão uniformemente repetidas. Mesmo assim, a duplicação é um problema, pois ela amontoa o código e serão necessárias quatro modificações se o algoritmo mudar. Além de serem quatro oportunidades para a omissão de um erro.

Sanou-se essa duplicação através do método `include` na Listagem 3.7. Leia este código novamente e note como a legibilidade do módulo inteiro foi melhorada com a retirada de tais repetições.

A duplicação pode ser a raiz de todo o mal no software. Muitos princípios e práticas têm sido criados com a finalidade de controlá-la ou eliminá-la. Considere, por exemplo, que todas as regras de normalização de bando de dados de Ted Codd servem para eliminar duplicação de dados. Considere também como a programação orientada a objeto serve para centralizar o código em classes-base que seriam outrora redundantes. Programação estruturada, Programação Orientada a Aspecto e Programação Orientada a Componentes são todas, em parte, estratégias para eliminar duplicação de código. Parece que desde a invenção da sub-rotina, inovações no desenvolvimento de software têm sido uma tentativa contínua para eliminar a duplicação de nossos códigos-fonte.

Programação estruturada

Alguns programadores seguem as regras programação estruturada de Edsger Dijkstra[14], que disse que cada função e bloco dentro de uma função deve ter uma entrada e uma saída. Cumprir essas regras significa que deveria haver apenas uma instrução `return` na função, nenhum `break` ou `continue` num loop e jamais um `goto`.

Enquanto somos solidários com os objetivos e disciplinas da programação estruturada, tais regras oferecem pouca vantagem quando as funções são muito pequenas. Apenas em funções maiores tais regras proporcionam benefícios significativos.

Portanto, se você mantiver suas funções pequenas, então as várias instruções `return`, `break` ou `continue` casuais não trarão problemas e poderão ser até mesmo mais expressivas do que

11. Aqueles que pensaram que poderiam se livrar da recompilação e da redistribuição foram encontrados, e tomadas as devidas providências.

a simples regra de uma entrada e uma saída. Por outro lado, o `goto` só faz sentido em funções grandes, portanto ele deve-se evitá-lo.

Como escrever funções como essa?

Criar um software é como qualquer outro tipo de escrita. Ao escrever um artigo, você primeiro coloca seus pensamentos no papel e depois os organiza de modo que fiquem fáceis de ler. O primeiro rascunho pode ficar desastroso e desorganizado, então você o apare, reestrutura e refina até que ele fique como você deseja.

Quando escrevo funções, elas começam longas e complexas; há muitas endentações e loops aninhados; possuem longas listas de parâmetros; os nomes são arbitrários; e há duplicação de código. Mas eu também tenho uma coleção de testes de unidade que analisam cada uma dessas linhas desorganizadas do código.

Sendo assim, eu organizo e refino o código, divido funções, troco os nomes, elimino a duplicação, reduzo os métodos e os reorganizo. Às vezes, desmonto classes inteiras, tudo com os testes em execução.

No final, minhas funções seguem as regras que citei neste capítulo. Não as aplico desde o início. Acho que isso não seja possível.

Conclusão

Cada sistema é construído a partir de uma linguagem específica a um domínio desenvolvida por programadores para descrever o sistema. As funções são os verbos dessa linguagem, e classes os substantivos. Isso não é um tipo de retomada da antiga noção de que substantivos e verbos nos requerimentos de um documento sejam os primeiros palpites das classes e funções de um sistema. Mas sim uma verdade muito mais antiga. A arte de programar é, e sempre foi, a arte do projeto de linguagem.

Programadores experientes veem os sistemas como histórias a serem contadas em vez de programas a serem escritos. Eles usam os recursos da linguagem de programação que escolhem para construir uma linguagem muito mais rica e expressiva do que a usada para contar a estória. Parte da linguagem específica a um domínio é a hierarquia de funções que descreve todas as ações que ocorrem dentro daquele sistema. Em um ato engenhoso, escrevem-se essas funções para usar a mesma linguagem específica a um domínio que eles criaram para contar sua própria parte da história.

Este capítulo falou sobre a mecânica de se escrever bem funções. Se seguir as regras aqui descritas, suas funções serão curtas, bem nomeadas e bem organizadas. Mas jamais se esqueça de que seu objetivo verdadeiro é contar a história do sistema, e que as funções que você escrever precisam estar em perfeita sincronia e formar uma linguagem clara e precisa para lhe ajudar na narração.

SetupTeardownIncluder

Listagem 3-7
SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws Exception
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }
```

Listagem 3-7 (continuação):
`SetupTeardownIncluder.java`

```
private void includeSetupPages() throws Exception {
  if (isSuite)
    includeSuiteSetupPage();
  includeSetupPage();
}

private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) throws Exception {
  WikiPage inheritedPage = findInheritedPage(pageName);
  if (inheritedPage != null) {
    String pagePathName = getPathNameForPage(inheritedPage);
    buildIncludeDirective(pagePathName, arg);
  }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
  return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

private String getPathNameForPage(WikiPage page) throws Exception {
  WikiPagePath pagePath = pageCrawler.getFullPath(page);
  return PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {
  newPageContent
    .append("\n!include ")
```