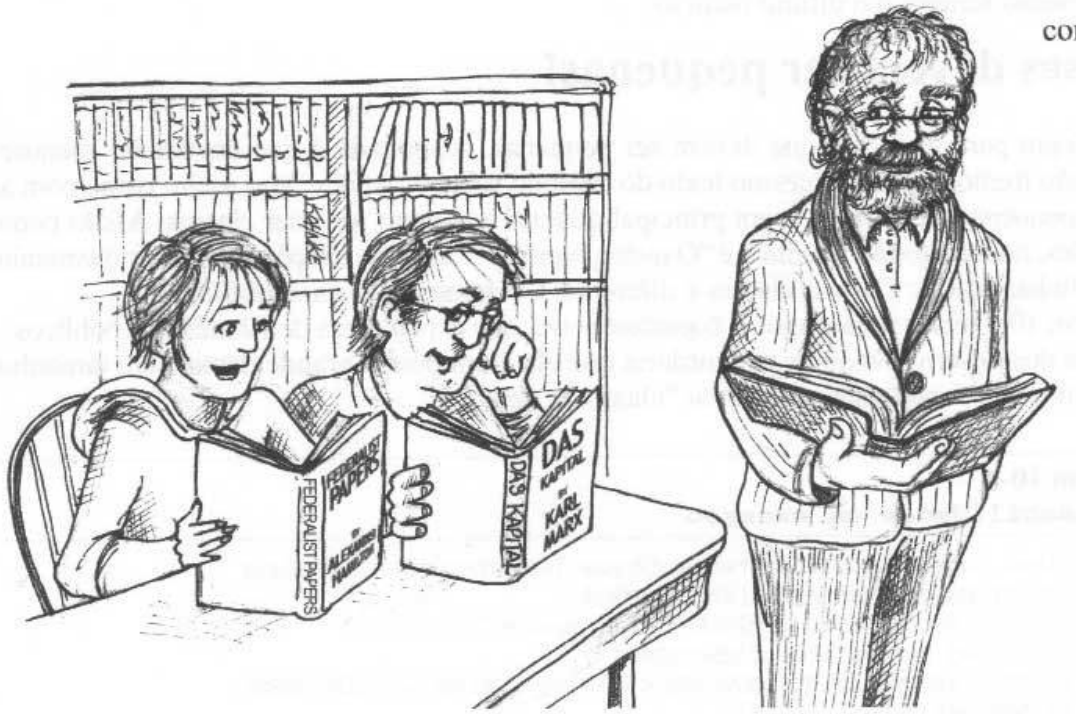
10

Classes

com Jeff Langr



Até agora, este livro se focou em como escrever bem linhas e blocos de código. Mergulhamos na composição apropriada para funções e como elas se interrelacionam. Mas apesar de termos falado bastante sobre a expressividade das instruções do código e as funções que o compõem, ainda não teremos um código limpo até discutirmos sobre os níveis mais altos da organização do código. Vamos fala agora sobre classes limpas.

Organização das classes

Seguindo uma convenção padrão Java, uma classe deve começar com uma lista de variáveis. As públicas (*public*), estáticas (*static*) e constantes (*constants*), se existirem, devem vir primeiro. Depois vêm as variáveis estáticas privadas (*private static*), seguidas pelas instâncias privadas. Raramente há uma boa razão para se ter uma variável pública (*public*).

As funções públicas devem vir após a lista de variáveis. Gostamos de colocar as tarefas privadas chamadas por uma função pública logo depois desta. Isso segue a regra de cima para baixo e ajuda o programa a ser lido como um artigo de jornal.

Encapsulamento

Gostaríamos que nossas variáveis e funções fossem privadas, mas não somos radicais.

Às vezes, precisamos tornar uma variável ou função protegida (*protected*) de modo que possa ser acessada para testes. Para nós, o teste tem prioridade. Se um teste no mesmo pacote precisa chamar uma função ou acessar uma variável, a tornaremos protegida ou apenas no escopo do pacote. Entretanto, primeiro buscaremos uma forma de manter a privacidade. Perder o encapsulamento sempre é o último recurso.

As classes devem ser pequenas!

A primeira regra para classes é que devem ser pequenas. A segunda é que devem ser menores ainda. Não, não iremos repetir o mesmo texto do capítulo sobre *Funções*. Mas assim como com as funções, ser pequena também é a regra principal quando o assunto for criar classes. Assim como com as funções, nossa questão imediata é "O quão pequena?". Com as funções medimos o tamanho contando as linhas físicas. Com as classes é diferente. Contamos as *responsabilidades*[1].

A Listagem 10.1 realça uma classe, a SuperDashboard, que expõe cerca de 70 métodos públicos.

A maioria dos desenvolvedores concordaria que ela é um pouco grande demais em tamanho. Outros diriam que a SuperDashboard é uma "classe de deus".

Listagem 10-1
Responsabilidades em excesso

```
public class SuperDashboard extends JFrame implements MetaDataUser
   public String getCustomizerLanguagePath()
   public void setSystemConfigPath(String systemConfigPath)
   public String getSystemConfigDocument()
   public void setSystemConfigDocument(String systemConfigDocument)
   public boolean getGuruState()
   public boolean getNoviceState()
   public boolean getOpenSourceState()
   public void showObject(MetaObject object)
   public void showProgress(String s)
```

1. [RDD].

Listagem 10-1 (continuação)
Responsabilidades em excesso

```
public boolean isMetadataDirty()
public void setIsMetadataDirty(boolean isMetadataDirty)
public Component getLastFocusedComponent()
public void setLastFocused(Component lastFocused)
public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
   MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPages(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
public void processProjectChangedState(boolean hasProjectChanged)
public void processObjectNameChanged(MetaObject object)
public void runProject()
```

Listagem 10-1 (continuação)
Responsabilidades em excesso

```
    public void setAçowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showHelper(MetaObject metaObject, String propertyName)
    // ... many non-public methods follow ...
}
```

Mas e se `SuperDashboard` possuísse apenas os métodos da Listagem 10.2?

Listagem 10-2
Pequena o suficiente?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Cinco não é muito, é? Neste caso é, pois apesar da pequena quantidade de métodos, a `SuperDashboard` possui muitas *responsabilidades*.

O nome de uma classe deve descrever quais responsabilidades ela faz. Na verdade, selecionar um nome é a primeira forma de ajudar a determinar o tamanho da classe. Se não derivarmos um nome conciso para ela, então provavelmente ela ficará grande. Quanto mais ambíguo for o nome da classe, maiores as chances de ela ficar com muitas responsabilidades. Por exemplo, nomes de classes que possuam palavras de vários sentidos, como `Processador` ou `Gerenciador` ou `Super`, geralmente indicam um acúmulo lastimável de responsabilidades.

Devemos também poder escrever com cerca de 25 palavras uma breve descrição da classe, sem usar as palavras "se", "e", "ou" ou "mas". Como descreveríamos a `SuperDashboard`? "A `SuperDashboard` oferece acesso ao componente que foi o último utilizado e também nos permite acompanhar os números da versão e da compilação".O primeiro "e" é uma dica de que a classe possui responsabilidades em excesso.

O Principio da Responsabilidade Única

O Principio da Responsabilidade Única (SRP, sigla em inglês)[2] afirma que uma classe ou módulo deve ter um, e apenas um, motivo para mudar. Este princípio nos dá uma definição de responsabilidade e uma orientação para o tamanho da classe. Estas devem ter apenas uma responsabilidade e um motivo para mudar.

Uma classe pequena como essa, a `SuperDashboard`, na Listagem 10.2, possui duas razões para ser alterada.

Primeiro, ela acompanha a informação sobre a versão, que precisa ser atualizada sempre que surgir uma nova distribuição do software. Segundo, ela gerencia os componentes Swing do Java

2. Pode-se ler muito mais sobre este princípio em [PPP].

(é um derivado do `JFrame`, a representação do Swing de uma janela GUI de alto nível). Sem dúvida iremos querer atualizar o número da versão se alterarmos qualquer código do Swing, mas o oposto não é necessariamente verdadeiro: podemos mudar as informações da versão baseando-nos nas modificações em outros códigos no sistema.

Tentar identificar as responsabilidades (motivos para alteração) costuma nos ajudar a reconhecer e criar abstrações melhores em nosso código. Podemos facilmente extrair todos os três métodos `SuperDashboard` que lidam com as informações de versão em uma classe separada chamada `Version`. (Veja a Listagem 10.3). A classe `Version` é um construtor que possui um alto potencial para reutilização em outros aplicativos!

Listagem 10-3
Classe com responsabilidade única

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

O SRP é um dos conceitos mais importantes no desenvolvimento OO. É também um dos mais simples para se entender e aprender. Mesmo assim, estranhamente, o SRP geralmente é o princípio mais ignorado na criação de classes. Frequentemente, encontramos classes que faz muitas coisas. Por quê?

Fazer um software funcionar e torná-lo limpo são duas coisas bem diferentes.

A maioria de nós tem uma mente limitada, por isso, nós tentamos fazer com que nosso código possua mais do que organização e clareza. Isso é totalmente apropriado. Manter uma separação de questões é tão importante em nossas *atividades* de programação como em nossos programas.

O problema é que muitos de nós achamos que já terminamos se o programa funciona.

Esquecemo-nos da *outra* questão de organização e de clareza. Seguimos para o próximo problema em vez de voltar e dividir as classes muito cheias em outras com responsabilidades únicas.

Ao mesmo tempo, muitos desenvolvedores temem que um grande número de classes pequenas e de propósito único dificulte mais o entendimento geral do código. Eles ficam apreensivos em ter de navegar de classe em classe para descobrir como é realizada uma parte maior da tarefa.

Entretanto, um sistema com muitas classes pequenas não possui tantas partes separadas a mais como um com classes grandes. Há também bastante a se aprender num sistema com poucas classes grandes. Portanto, a questão é: você quer suas ferramentas organizadas em caixas de ferramentas com muitas gavetas pequenas, cada um com objetos bem classificados e rotulados? Ou poucas gavetas nas quais você coloca tudo?

Todo sistema expansível poderá conter uma grande quantidade de lógica e complexidade. O objetivo principal no gerenciamento de tal complexidade é organizá-la de modo que o desenvolvedor saiba onde buscar o que ele deseja e que precise entender apenas a complexidade que afeta diretamente um dado momento. Em contrapartida, um sistema com classes maiores de vários propósitos sempre nos atrasa insistindo que percorramos por diversas coisas que não precisamos saber no momento.

Reafirmando os pontos anteriores: desejamos que nossos sistemas sejam compostos por muitas classes pequenas, e não poucas classes grandes. Cada classe pequena encapsula uma única

responsabilidade, possui um único motivo para ser alterada e contribui com poucas outras para obter os comportamentos desejados no sistema.

Coesão

As classes devem ter um pequeno número de instâncias de variáveis. Cada método de uma classe deve manipular uma ou mais dessas variáveis. De modo geral, quanto mais variáveis um método manipular, mais coeso o método é para sua classe. Uma classe na qual cada variável é utilizada por um método é totalmente coesa.

De modo geral, não é aconselhável e nem possível criar tais classes totalmente coesas; por outro lado, gostaríamos de obter uma alta coesão. Quando conseguimos, os métodos e as variáveis da classe são co-dependentes e ficam juntas como um todo lógico.

Considere a implementação de uma `Stack` (pilha) na Listagem 10.4. A classe é bastante coesa. Dos três métodos, apenas `size()` não usa ambas as variáveis.

Listagem 10-4
`Stack.java` - uma classe coesa.

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

A estratégia para manter funções pequenas e listas de parâmetros curtas às vezes pode levar à proliferação de instâncias de variáveis que são usadas por uma sequência de métodos. Quando isso ocorre, quase sempre significa que há pelo menos uma outra classe tentando sair da classe maior na qual ela está. Você sempre deve tentar separar as variáveis e os métodos em duas ou mais classes de modo que as novas classes sejam mais coesas.

Manutenção de resultados coesos em muitas classes pequenas

Só o ato de dividir funções grandes em menores causa a proliferação de classes. Imagine uma função grande com muitas variáveis declaradas. Digamos que você deseje extrair uma pequena parte dela para uma outra função. Entretanto, o código a ser extraído usa quatro das variáveis declaradas na função. Você deve passar todas as quatro para a nova função como parâmetros?

Absolutamente não! Se convertêssemos aquelas quatro variáveis em instâncias de variáveis da classe, então poderíamos extrair o código sem passar *qualquer* variável. Seria fácil dividir a função em partes menores.

Infelizmente, isso também significa que nossas classes perderiam coesão, pois acumulariam mais e mais instâncias de variáveis que existiriam somente para permitir que as poucas funções as compartilhassem. Mas, espere! Se há poucas funções que desejam compartilhar certas variáveis, isso não as torna uma cada uma classe? Claro que sim. Quando as classes perdem coesão, divida-as!

Portanto, separar uma função grande em muitas pequenas geralmente nos permite dividir várias classes também. Isso dá ao nosso programa uma melhor organização e uma estrutura mais transparente.

Como exemplo do que quero dizer. usemos um exemplo respeitado há anos, extraído do maravilhoso livro *Literate Programming*[3], de Knuth. A Listagem 10.5 mostra uma tradução em Java do programa `PrintPrimes` de Knuth. Para ser justo com Knuth, este não é o programa que ele escreveu, mas o que foi produzido pela sua ferramenta WEB. Uso-o por ser um grande ponto de partida para dividir uma função grande em muitas funções e classes menores.

Listagem 10-5
`PrintPrimes.java`

```
package literatePrimes;

public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
```

3. [Knuth92].

Listagem 10-5 (continuação)
PrintPrimes.java

```
    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];

    J = 1;
    K = 1;
    P[1] = 2;
    ORD = 2;
    SQUARE = 9;

    while (K < M) {
      do {
        J = J + 2;
        if (J == SQUARE) {
          ORD = ORD + 1;
          SQUARE = P[ORD] * P[ORD];
          MULT[ORD - 1] = J;
        }
        N = 2;
        JPRIME = true;
        while (N < ORD && JPRIME) {
          while (MULT[N] < J)
            MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
          N = N + 1;
        }
      } while (!JPRIME);
      K = K + 1;
      P[K] = J;
    }
    {
      PAGENUMBER = 1;
      PAGEOFFSET = 1;
      while (PAGEOFFSET <= M) {
        System.out.println("The First " + M +
                           " Prime Numbers --- Page " + PAGENUMBER);
        System.out.println("");
        for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++){
          for (C = 0; C < CC;C++)
            if (ROWOFFSET + C * RR <= M)
              System.out.format("%10d", P[ROWOFFSET + C * RR]);
          System.out.println("");
        }
        System.out.println("\f");
        PAGENUMBER = PAGENUMBER + 1;
        PAGEOFFSET = PAGEOFFSET + RR * CC;
      }
    }
  }
}
```

Este programa, escrito como uma única função, é uma zona; possui uma estrutura com muitas endentações, uma abundância de variáveis estranhas e uma estrutura fortemente acoplada. No mínimo essa grande função deve ser dividida em algumas menores.
Da Listagem 10.6 a 10.8 mostra o resultado da divisão do código na Listagem 10.5 em classes e funções menores, e seleciona nomes significativos para as classes, funções e variáveis.

Listagem 10-6
PrimePrinter.java (refatorado)

```
package literatePrimes;

public class PrimePrinter {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

    final int ROWS_PER_PAGE = 50;
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
                               COLUMNS_PER_PAGE,
                               "The First " + NUMBER_OF_PRIMES +
                                 " Prime Numbers");

    tablePrinter.print(primes);
  }

}
```

Listagem 10-7
RowColumnPagePrinter.java

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;

  public RowColumnPagePrinter(int rowsPerPage,
                              int columnsPerPage,
                              String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
    numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
  }
```

Listagem 10-7 (continuação)
RowColumnPagePrinter.java

```
  public void print(int data[]) {
    int pageNumber = 1;
    for (int firstIndexOnPage = 0;
         firstIndexOnPage < data.length;
         firstIndexOnPage += numbersPerPage) {
      int lastIndexOnPage =
        Math.min(firstIndexOnPage + numbersPerPage - 1,
                 data.length - 1);
      printPageHeader(pageHeader, pageNumber);
      printPage(firstIndexOnPage, lastIndexOnPage, data);
      printStream.println("\f");
      pageNumber++;
    }
  }

  private void printPage(int firstIndexOnPage,
                         int lastIndexOnPage,
                         int[] data) {
    int firstIndexOfLastRowOnPage =
      firstIndexOnPage + rowsPerPage - 1;
    for (int firstIndexInRow = firstIndexOnPage;
         firstIndexInRow <= firstIndexOfLastRowOnPage;
         firstIndexInRow++) {
      printRow(firstIndexInRow, lastIndexOnPage, data);
      printStream.println("");
    }
  }

  private void printRow(int firstIndexInRow,
                        int lastIndexOnPage,
                        int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
      int index = firstIndexInRow + column * rowsPerPage;
      if (index <= lastIndexOnPage)
        printStream.format("%10d", data[index]);
    }
  }

  private void printPageHeader(String pageHeader,
                               int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
  }

  public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
  }
}
```

Listagem 10-8
PrimeGenerator.java

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;

  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
    checkOddNumbersForSubsequentPrimes();
    return primes;
  }

  private static void set2AsFirstPrime() {
    primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }

  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
         primeIndex < primes.length;
         candidate += 2) {
      if (isPrime(candidate))
        primes[primeIndex++] = candidate;
    }
  }

  private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
      multiplesOfPrimeFactors.add(candidate);
      return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }

  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
  }

  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
      if (isMultipleOfNthPrimeFactor(candidate, n))
        return false;
    }
```

Listagem 10-8 (continuação)
PrimeGenerator.java

```
      return true;
    }

    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
      return
        candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }

    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
      int multiple = multiplesOfPrimeFactors.get(n);
      while (multiple < candidate)
        multiple += 2 * primes[n];
      multiplesOfPrimeFactors.set(n, multiple);
      return multiple;
    }
}
```

A primeira coisa que você deve perceber é que o programa ficou bem maior, indo de uma página para quase três. Há diversas razões para esse aumento. Primeiro, o programa refatorado usa nomes de variáveis mais longos e descritivos.
Segundo, ele usa declarações de funções e classes como uma forma de adicionar comentários ao código. Terceiro, usamos espaços em branco e técnicas de formatação para manter a legibilidade.

Note como o programa foi dividido em três responsabilidades principais. O programa principal está sozinho na classe `PrimePrinter` cuja responsabilidade é lidar com o ambiente de execução. Ela será modificada se o método de chamada também for. Por exemplo, se esse programa fosse convertido para um serviço SOAP, esta seria a classe afetada.

O `RowColumnPagePrinter` sabe tudo sobre como formatar uma lista de números em páginas com uma certa quantidade de linhas e colunas. Se for preciso modificar a formatação da saída, então essa seria a classe afetada.

A classe `PrimeGenerator` sabe como gerar uma lista de números primos. Note que ela não foi feita para ser instanciada como um objeto. A classe é apenas um escopo prático no qual suas variáveis podem ser declaradas e mantidas ocultas. Se o algoritmo para o cálculo de números primos mudar, essa classe também irá.

Não a reescrevemos! Não começamos do zero e criamos um programa novamente. De fato, se olhar os dois programas mais de perto, verá que usam o mesmo algoritmo e lógica para efetuar as tarefas.

A alteração foi feita criando-se uma coleção de testes que verificou o comportamento *preciso* do primeiro programa. Então, foram feitas umas pequenas mudanças, uma de cada vez. Após cada alteração, executava-se o programa para garantir que o comportamento não havia mudado. Um pequeno passo após o outro e o primeiro programa foi limpo e transformado no segundo.

Como organizar para alterar

Para a maioria dos sistemas, a mudança é constante. A cada uma, corremos o risco de o sistema não funcionar mais como o esperado. Em um sistema limpo, organizamos nossas classes de modo a reduzir os riscos nas alterações.

A classe `Sql` na Listagem 10.9 gera strings no formato SQL adequado dado um metadados apropriado. É um trabalho contínuo e, como tal, ainda não suporta funcionalidade SQL, como as instruções `update`. Quando chegar a hora da classe SQL suportar uma instrução update, teremos de "abrir" essa classe e fazer as alterações. Qualquer modificação na classe tem a possibilidade de estragar outro código na classe. É preciso testar tudo novamente.

Listagem 10-9
Classe que precisa ser aberta para alteração

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
   public String select(Criteria criteria)
   public String preparedInsert()
   private String columnList(Column[] columns)
   private String valuesList(Object[] fields, final Column[] columns)
   private String selectWithCriteria(String criteria)
   private String placeholderList(Column[] columns)
}
```

A classe `Sql` deve ser alterada quando adicionamos um novo tipo de instrução ou quando mudarmos os detalhes de um único tipo de instrução – por exemplo, se precisarmos modificar a funcionalidade do `select` para suportar "sub-selects". Esses dois motivos para alteração significam que a classe `Sql` viola o SRP.

Podemos ver essa quebra da regra num simples ponto horizontal. O método realçado da `SQL` mostra que há métodos privados, como o `selectWithCriteria`, que parecem se relacionar apenas às instruções `select`.

O comportamento do método privado aplicado apenas a um pequeno subconjunto de uma classe por ser uma heurística útil para visualizar possíveis áreas para aperfeiçoamento. Entretanto, o indício principal para se tomar uma ação deve ser a alteração do sistema em si. Se a classe `Sql` for considerada logicamente completa, então não temos de nos preocupar em separar as responsabilidades. Se não precisamos da funcionalidade `update` num futuro próximo, então devemos deixar a `Sql` em paz. Mas assim que tivermos de "abrir" uma classe, devemos considerar consertar nosso projeto.

E se considerássemos uma solução como a da Listagem 10.10? Cada método de interface pública definido na `Sql` anterior na Listagem 10.9 foi refatorado para sua própria classe derivada `Sql`. Note que os métodos privados, como `valuesList`, vão diretamente para onde são necessários. O comportamento privado comum foi isolado para um par de classes utilitárias, `Where` e `ColumnList`.

Listagem 10-10
Várias classes fechadas

```
abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}

public class CreateSql extends Sql {
   public CreateSql(String table, Column[] columns)
   @Override public String generate()
}

public class SelectSql extends Sql {
   public SelectSql(String table, Column[] columns)
   @Override public String generate()
}

public class InsertSql extends Sql {
   public InsertSql(String table, Column[] columns, Object[] fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
   public SelectWithCriteriaSql(
      String table, Column[] columns, Criteria criteria)
   @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
   public SelectWithMatchSql(
      String table, Column[] columns, Column column, String pattern)
   @Override public String generate()
}

public class FindByKeySql extends Sql
   public FindByKeySql(
      String table, Column[] columns, String keyColumn, String keyValue)
   @Override public String generate()
}

public class PreparedInsertSql extends Sql {
   public PreparedInsertSql(String table, Column[] columns)
   @Override public String generate() {
   private String placeholderList(Column[] columns)
}

public class Where {
   public Where(String criteria)
   public String generate()
}

public class ColumnList {
   public ColumnList(Column[] columns)
   public String generate()
}
```

O código em cada classe se torna absurdamente simples. O tempo necessário para entendermos qualquer classe caiu para quase nenhum. O risco de que uma função possa prejudicar outra se torna ínfima. Do ponto de vista de testes, virou uma tarefa mais fácil testar todos os pontos lógicos nesta solução, já que as classes estão isoladas umas das outras.

Tão importante quanto é quando chega a hora de adicionarmos as instruções `update`, nenhuma das classes existentes precisam ser alteradas! Programamos a lógica para construir instruções `update` numa nova subclasse de `Sql` chamada `UpdateSql`. Nenhum outro código no sistema sofrerá com essa mudança.

Nossa lógica reestruturada da `Sql` representa o melhor possível. Ela suporta o SRP e outro princípio-chave de projeto de classe OO, conhecido como Princípio de Aberto-Fechado OCP[4] (sigla em inglês). As classes devem ser abertas para expansão, mas fechadas para alteração. Nossa classe `Sql` reestruturada está aberta para permitir novas funcionalidades através da criação de subclasses, mas podemos fazer essa modificação ao mesmo tempo em que mantemos as outras classes fechadas. Simplesmente colocamos nossa classe `UpdateSql` em seu devido lugar. Desejamos estruturar nossos sistemas de modo que baguncemos o mínimo possível quando os atualizarmos. Num sistema ideal, incorporaríamos novos recursos através da expansão do sistema, e não alterando o código existente.

Como isolar das alterações

As necessidades mudarão, portanto o código também. Aprendemos na introdução à OO que há classes concretas, que contêm detalhes de implementação (código), e classes abstratas, que representam apenas conceitos. Uma classe do cliente dependente de detalhes concretos corre perigo quando tais detalhes são modificados. Podemos oferecer interfaces e classes abstratas para ajudar a isolar o impacto desses detalhes.

Depender de detalhes concretos gera desafios para nosso sistema de teste. Se estivermos construindo uma classe `Portfolio` e ela depender de uma API `TokyoStockExchange` externa para derivar o valor do portfolio, nossos casos de testes são afetados pela volatilidade dessa consulta. É difícil criar um teste quando obtemos uma resposta diferente a cada cinco minutos! Em vez de criar a *Portfolio* de modo que ela dependa diretamente de TokyoStockExchange, podemos criar uma interface StockExchange que declare um único método:

Desenvolvemos `TokyoStockExchange` para implementar essa interface. Também nos certificamos se o construtor de `Portfolio` recebe uma referência a `StockExchange` como parâmetro:

```
public Portfolio {
    private StockExchange exchange;
    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    // ...
}
```

Agora nosso teste pode criar uma implementação para testes da interface `StockExchange` que simula a `TokyoStockExchange`. Essa implementação fixará o valor para qualquer símbolo

4. [PPP]

que testarmos. Se nosso teste demonstrar a compra de cinco ações da Microsoft para nosso `portfolio`, programamos a implementação do teste para sempre retornar U$ 100 dólares por ação. Nossa implementação de teste da interface `StockExchange` se reduz a uma simples tabela de consulta. Podemos, então, criar um teste que espere U$ 500 dólares como o valor total do `portfolio`.

```
public class PortfolioTest {
    private FixedStockExchangeStub exchange;
    private Portfolio portfolio;

    @Before
    protected void setUp() throws Exception {
        exchange = new FixedStockExchangeStub();
        exchange.fix("MSFT", 100);
        portfolio = new Portfolio(exchange);
    }

    @Test
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {
        portfolio.add(5, "MSFT");
        Assert.assertEquals(500, portfolio.value());
    }
}
```

Se o um sistema estiver desacoplado o bastante para ser testado dessa forma, ele também será mais flexível e terá maior capacidade de reutilização. A falta de acoplamento significa que os elementos de nosso sistema ficam melhores quando isolados uns dos outros e das alterações, facilitando o entendimento de cada elemento.

Ao minimizar o acoplamento dessa forma, nossas classes aderem a outro princípio de projeto de classes conhecido como Princípio da Inversão da Independência ,DIP[5] (sigla em inglês). Basicamente, o DIP diz que nossas classes devem depender de abstrações, não de detalhes concretos.

Em vez de depender da implementação de detalhes da classe `TokyoStockExchange`, nossa classe `Portfolio` agora é dependente da interface `StockExchange`, que representa o conceito abstrato de pedir o preço atual de um símbolo. Essa isola todos os detalhes específicos da obtenção de tal preço, incluindo sua origem.

Bibliografia

[RDD]: *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca WirfsBrock et al., Addison-Wesley, 2002.

[PPP]: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[Knuth92]: *Literate Programming*, Donald E. Knuth, *Center for the Study of language and Information*, Leland Stanford Junior University, 1992.

5. [PPP]