

expô-la através de uma função, como a `julianDateOfLastDayOfMonth`. Parece que ninguém precisa de uma função como essa. Ademais, pode-se facilmente colocar a tabela de volta na classe `DayDate` se qualquer implementação nova desta precisar daquela.

O mesmo vale para `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Agora, veremos as três séries de constantes que podem ser convertidas em enum (linhas 162-205).

A primeira das três seleciona uma semana dentro de um mês. Transformei-a em um enum chamado `WeekInMonth`.

```
public enum WeekInMonth {  
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);  
    public final int index;  
  
    WeekInMonth(int index) {  
        this.index = index;  
    }  
}
```

A segunda série de constantes (linhas 177-187) é um pouco mais confusa. Usam-se as constantes `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND` e `INCLUDE_BOTH` para descrever se as datas nas extremidades de um intervalo devam ser incluídas nele. Matematicamente, usam-se os termos “intervalo aberto”, “intervalo meio-aberto” e “intervalo fechado”.

Acho que fica mais claro se usarmos a nomenclatura matemática [N3], então transformei essa segunda série de constantes em um enum chamado `DateInterval` com enumeradores `CLOSED` (fechado), `CLOSED_LEFT` (esquerda\_fechado), `CLOSED_RIGHT` (direita\_fechado) e `OPEN` (aberto). A terceira série de constantes (linhas 18-205) descreve se uma busca por um dia particular da semana deve resultar na última, na próxima ou na mais próxima instância. Decidir o nome disso é no mínimo difícil. No final, optei por `WeekdayRange` com enumeradores `LAST` (último), `NEXT` (próximo) e `NEAREST` (mais próximo).

Talvez você não concorde com os nomes que escolhi. Para mim eles fazem sentido, pode não fazer para você. A questão é que agora eles estão num formato que facilita sua alteração [J3]. Eles não são mais passados como inteiros, mas como símbolos. Posso usar a função de “nome alterado” de minha IDE para mudar os nomes, ou os tipos, sem me preocupar se deixei passar algum -1 ou 2 em algum lugar do código ou se alguma declaração de um parâmetro do tipo int está mal descrito.

Parece que ninguém usa o campo de descrição da linha 208. Portanto, eu a exclui juntamente com seu método de acesso e de alteração [G9].

Também deletei o danoso construtor padrão da linha 213 [G12]. O compilador o criará para nós. Podemos pular o método `isValidWeekdayCode` (linhas 216-238), pois o excluímos quando criamos a enumeração de `Day`.

Isso nos leva ao método `stringToWeekdayCode` (linhas 242-270). Os Javadocs que não contribui muito à assinatura do método são apenas restos [C3],[G12]. O único valor que esse Javadoc adiciona é a descrição do valor retornado -1. Entretanto, como mudamos para a enumeração de `Day`, o comentário está de fato errado [C2]. Agora o método lança uma `IllegalArgumentException`. Sendo assim, deletei o Javadoc.

Também exclui a palavra reservada `final` das declarações de parâmetros e variáveis. Até onde

pude entender, ela não adicionava nenhum valor real, só adiciona mais coisas aos restos inúteis [G12]. Eliminar essas final contraria alguns conhecimentos convencionais. Por exemplo, Robert Simons<sup>6</sup> nos aconselha a “...colocar final em todo o seu código”. Obviamente, eu discordo. Acho que há alguns poucos usos para o final, como a constante final, mas fora isso, as palavras reservadas acrescentam pouca coisa e criam muito lixo. Talvez eu me sinta dessa forma porque os tipos de erros que o final possa capturar, os testes de unidade que escrevi já o fazem.

Não me importo com as estruturas `if` duplicadas [G5] dentro do loop `for` (linhas 259 e 263), portanto, eu os uni em um único `if` usando o operador `||`. Também usei a enumeração de `Day` para direcionar o loop `for` e fiz outros retoques.

Ocorreu-me que este método não pertence à `DayDate`. Ele é a função de análise sintática de `Day`. Então o movi para dentro da enumeração de `Day`, a qual ficou muito grande. Como `Day` não depende de `DayDate`, retirei a enumeração de `Day` da classe `DayDate` e coloquei em seu próprio arquivo fonte [G13].

Também movi a próxima função, `weekdayCodeToString` (linhas 272–286) para dentro da enumeração de `Day` e a chamei de `toString`.

```
public enum Day {
    MONDAY(Calendar.MONDAY),
    TUESDAY(Calendar.TUESDAY),
    WEDNESDAY(Calendar.WEDNESDAY),
    THURSDAY(Calendar.THURSDAY),
    FRIDAY(Calendar.FRIDAY),
    SATURDAY(Calendar.SATURDAY),
    SUNDAY(Calendar.SUNDAY);

    public final int index;
    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

    Day(int day) {
        index = day;
    }

    public static Day make(int index) throws IllegalArgumentException {
        for (Day d : Day.values())
            if (d.index == index)
                return d;
        throw new IllegalArgumentException(
            String.format("Illegal day index: %d.", index));
    }

    public static Day parse(String s) throws IllegalArgumentException {
        String[] shortWeekdayNames =
            dateSymbols.getShortWeekdays();
        String[] weekDayNames =
            dateSymbols.getWeekdays();

        s = s.trim();
        for (Day day : Day.values()) {
            if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
                s.equalsIgnoreCase(weekDayNames[day.index])) {
                return day;
            }
        }
    }
}
```

```
        }
    }
    throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
}

public String toString() {
    return dateSymbols.getWeekdays()[index];
}
}
```

Há duas funções `getMonths` (linhas 288-316). A primeira chama a segunda. Esta só é chamada por aquela. Sendo assim, juntei as duas numa única só e as simplifiquei consideravelmente [G9],[G12],[F4]. Por fim, troquei o nome para ficar um pouco mais descritivo [N1].

```
public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
```

A função `isValidMonthCode` (linhas 326-346) se tornou irrelevante devido ao enum `Month`. Portanto, deletei-a [G9]. A função `monthCodeToQuarter` (linhas 356–375) parece a FEATURE ENVY<sup>7</sup> [G14] e, provavelmente, pertence a enum `Month` como um método chamado `quarter`. Portanto, excluí-a.

```
public int quarter() {
    return 1 + (index-1)/3;
}
```

Isso deixou o enum `Month` grande o suficiente para ter sua própria classe. Sendo assim, retirei-o de `DayDate` para ficar mais consistente com o enum `Day` [G11],[G13].

Os dois métodos seguintes chamam-se `monthCodeToString` (linhas 377–426). Novamente, vemos um padrão de um método chamando sua réplica com uma flag. Costuma ser uma péssima idéia passar uma flag como parâmetro de uma função, especialmente qual a flag simplesmente seleciona o formato da saída [G15]. Renomeei, simplifiquei e reestruturei essas funções e as movi para o enum `Month` [N1],[N3],[C3],[G14].

```
public String toString() {
    return dateFormatSymbols.getMonths()[index - 1];
}

public String toShortString() {
    return dateFormatSymbols.getShortMonths()[index - 1];
}
```

O próximo método é o `stringToMonthCode` (linhas 428–472). Renomeei-o, movi-o para enum `Month` e o simplifiquei [N1],[N3],[C3],[G14],[G12].

```

public static Month parse(String s) {
    s = s.trim();
    for (Month m : Month.values())
        if (m.matches(s))
            return m;

    try {
        return make(Integer.parseInt(s));
    }
    catch (NumberFormatException e) {}
    throw new IllegalArgumentException("Invalid month " + s);
}

private boolean matches(String s) {
    return s.equalsIgnoreCase(toString()) ||
           s.equalsIgnoreCase(toShortString());
}

```

É possível tornar o método `isLeapYear` (linhas 495–517) um pouco mais expressivo [G16].

```

public static boolean isLeapYear(int year) {
    boolean fourth = year % 4 == 0;
    boolean hundredth = year % 100 == 0;
    boolean fourHundredth = year % 400 == 0;
    return fourth && (!hundredth || fourHundredth);
}

```

A próxima função, `leapYearCount` (linhas 519–536), realmente não pertence a `DayDate`. Ninguém a chama, exceto pelos dois métodos em `SpreadsheetDate`. Portanto, a movi para baixo [G6].

A função `lastDayOfMonth` (linhas 538–560) usa o array `LAST_DAY_OF_MONTH`, que pertence a enum `Month` [G17]. Portanto, movi-a para lá e também a simplifiquei e a tornei um pouco mais expressiva [G16].

```

public static int lastDayOfMonth(Month month, int year) {
    if (month == Month.FEBRUARY && isLeapYear(year))
        return month.lastDay() + 1;
    else
        return month.lastDay();
}

```

Agora as coisas estão ficando mais interessantes. A função seguinte é a `addDays` (linhas 562–576). Primeiramente, como ela opera nas variáveis de `DayDate`, ela não pode ser estática [G18]. Sendo assim, a transformei na instância de um método. Segundo, ela chama a função `toSerial`, que deve ser renomeada para `toOrdinal` [N1]. Por fim, é possível simplificar o método.

```

public DayDate addDays(int days) {
    return DayDateFactory.makeDate(toOrdinal() + days);
}

```

O mesmo vale para `addMonths` (linhas 578–602), que deverá ser a instância de um método. O algoritmo está um pouco mais complicado, então usei VARIÁVEIS TEMPORÁRIAS EXPLICATIVAS (EXPLAINING TEMPORARY VARIABLES<sup>8</sup>) para ficar mais transparente. Também renomeei o método `getYYY` para `getYear` [N1].

```
public DayDate addMonths(int months) {
    int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
    int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
    int resultYear = resultMonthAsOrdinal / 12;
    Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);

    int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

A função `addYears` (linhas 604–626) não possui nada de extraordinário em relação às outras.

```
public DayDate plusYears(int years) {
    int resultYear = getYear() + years;
    int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

Estou com uma pulga atrás da orelha sobre alterar esses métodos de estáticos para instâncias. A expressão `date.addDays(5)` deixa claro que o objeto `date` não é alterado e que é retornada uma nova instância de `DayDate`?

Ou indica erroneamente que estamos adicionando cinco dias ao objeto `date`? Você pode achar que isso não seja um grande problema, mas um pedaço de código parecido com o abaixo pode enganar bastante [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // pula a data em uma semana.
```

Alguém lendo esse código muito provavelmente entenderia apenas que `addDays` está alterando o objeto `date`. Portanto, precisamos de um nome que acabe com essa ambiguidade [N4]. Sendo assim, troquei os nomes de `plusDays` e `plusMonths`. Parece-me que a expressão abaixo indica bem o objetivo do método:

```
DayDate date = oldDate.plusDays(5);
```

Por outro lado, a instrução abaixo não é o suficiente para que o leitor deduza que o objeto `date` fora modificado:

```
date.plusDays(5);
```

Os algoritmos continuam a ficar mais interessantes. `getPreviousDayOfWeek` (linhas 628–660) funciona, mas está um pouco complicado. Após pensar um pouco sobre o que realmente está acontecendo [G21], fui capaz de simplificá-lo e usar as VARIÁVEIS TEMPORÁRIAS

EXPLICATIVAS [G19] para esclarecê-lo. Também a passei de um método estático para a instância de um método [G18], e me livrei das instâncias duplicadas [G5] (linhas 997–1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget >= 0)
        offsetToTarget -= 7;
    return plusDays(offsetToTarget);
}
```

Exatamente o mesmo ocorreu com `getFollowingDayOfWeek` (linhas 662–693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget <= 0)

        offsetToTarget += 7;
    return plusDays(offsetToTarget);
}
```

Consertamos a função seguinte, `getNearestDayOfWeek` (linhas 695–726), na página 270. Mas as alterações que fiz naquela hora não eram consistentes com o padrão atual nas duas últimas funções [G11]. Sendo assim, tornei-a consistente e usei algumas VARIÁVEIS TEMPORÁRIAS EXPLICATIVAS [G19] para esclarecer o algoritmo.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
    int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
    int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
    int offsetToPreviousTarget = offsetToFutureTarget - 7;

    if (offsetToFutureTarget > 3)
        return plusDays(offsetToPreviousTarget);
    else
        return plusDays(offsetToFutureTarget);
}
```

O método `getEndOfCurrentMonth` (linhas 728–740) está um pouco estranho por ser a instância de um método que inveja [G14] sua própria classe ao receber um parâmetro `DayDate`. Tornei-o a instância de um método real e esclareci alguns nomes.

```
public DayDate getEndOfMonth() {
    Month month = getMonth();
    int year = getYear();
    int lastDay = lastDayOfMonth(month, year);
    return DayDateFactory.makeDate(lastDay, month, year);
}
```

De fato, refatorar `weekInMonthToString` (lines 742–761) acabou sendo bem interessante. Usando as ferramentas de refatoração de minha IDE, primeiro movi o método para o enum `WeekInMonth` que criei na página 275. Então, renomeei o método para `toString`. Em seguida, alterei-o de um método estático para a instância de um método. Todos os testes ainda passavam com êxito (Já sabe o que vou fazer?).

Depois excluí o método inteiro! Cinco testes de confirmação falharam (linhas 411–415, Listagem B.4, página 374). Alterei essas linhas para usarem os nomes dos enumeradores (`FIRST`, `SECOND`, ...). Todos os testes passaram. Consegue ver por quê? Consegue ver também por que foi preciso cada um desses passos? A ferramenta de refatoração garantiu que todos os chamadores de `weekInMonthToString` agora invocassem `toString` no enum `weekInMonth`, pois todos os enumeradores implementam `toString` para simplesmente retornarem seus nomes...

Infelizmente, fui esperto demais. Por mais elegante que estivesse aquela linda sequência de refatorações, finalmente percebi que os únicos usuários dessa função eram os testes que eu acabara de modificar, portanto os exclui.

Enganou-me de novo, tenha vergonha na cara! Enganou-me duas vezes, eu é quem preciso ter vergonha na cara! Então, depois de determinar que ninguém além dos testes chamava `relativeToString` (linhas 765–781), simplesmente deletei a função e seus testes.

Finalmente chegamos aos métodos abstratos desta classe abstrata. E o primeiro não poderia ser mais apropriado: `toSerial` (linhas 838–844). Lá na página 279, troquei o nome para `toOrdinal`. Analisando isso com o contexto atual, decidi que o nome deve ser `getOrdinalDay`.

O próximo método abstrato é o `toDate` (linhas 838–844). Ele converte uma `DayDate` para uma `java.util.Date`. Por que o método é abstrato? Se olharmos sua implementação na `SpreadsheetDate` (linhas 198–207, Listagem B-5, página 382), vemos que ele não depende de nada na implementação daquela classe [G6]. Portanto, o movi para cima.

Os métodos `getYYYY`, `getMonth` e `getDayOfMonth` estão bem como abstratos. Entretanto, o `getDayOfWeek` é outro que deve ser retirado de `SpreadSheetDate`, pois ele não depende de nada que esteja em `DayDate` [G6]. Ou depende?

Se olhar com atenção (linha 247, Listagem B-5, página 382), verá que o algoritmo implicitamente depende da origem do dia ordinal (ou seja, do dia da semana do dia 0). Portanto, mesmo que essa função não tenha dependências físicas que possam ser movidas para `DayDate`, ela possui uma dependência lógica.

Dependências lógicas como essa me incomodam [G22]. Se algo lógico depende da implementação, então algo físico também depende. Ademais, parece-me que o próprio algoritmo poderia ser genérico com uma parte muito menor de si dependendo da implementação [G6].

Sendo assim, criei um método abstrato `getDayOfWeekForOrdinalZero` em `DayDate` e o implementei em `SpreadsheetDate` para retornar `Day.SATURDAY`. Então, subi o método `getDayOfWeek` para a `DayDate` e o alterei para chamar `getOrdinalDay` e `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {
    Day startingDay = getDayOfWeekForOrdinalZero();
    int startingOffset = startingDay.index - Day.SUNDAY.index;
    return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Como uma observação, olhe atentamente o comentário da linha 895 até a 899. Essa repetição era realmente necessária? Como de costume, exclui esse comentário juntamente com todos os outros. O próximo método é o `compare` (linhas 902–913). Novamente, ele não está adequadamente abstrato [G6], de modo que subi sua implementação para `DayDate`. Ademais, o nome não diz muito [N1]. Esse método realmente retorna a diferença em dias desde o passado por parâmetro.

Sendo assim, mudei seu nome para `daysSince`. Também notei que não havia testes para este método, então os escrevi.

As seis funções seguintes (linhas 915–980) são todas métodos abstratos que devem ser implementados em `DayDate`, onde os coloquei após retirá-los de `SpreadsheetDate`.

A última função, `isInRange` (linhas 982–995), também precisa ser movida para cima e refatorada.

A estrutura `switch` está um pouco feia [G23] e pode-se substitui-la movendo os `case` para o enum `DateInterval`.

```
public enum DateInterval {
    OPEN {
        public boolean isIn(int d, int left, int right) {
            return d > left && d < right;
        }
    },
    CLOSED_LEFT {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d < right;
        }
    },
    CLOSED_RIGHT {
        public boolean isIn(int d, int left, int right) {
            return d > left && d <= right;
        }
    },
    CLOSED {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d <= right;
        }
    }
};

public abstract boolean isIn(int d, int left, int right);
}
```

```
public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
    int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
    int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
    return interval.isIn(getOrdinalDay(), left, right);
}
```

Isso nos leva ao final de `DayDate`. Então, agora vamos dar mais uma passada por toda a classe para ver como ela flui.

Primeiramente, o comentário de abertura está desatualizado, então o reduzi e o melhorei [C2].

Depois, movi todos os enum restantes para seus próprios arquivos [G12].

Em seguida, movia variáveis estáticas `dateFormatSymbols` e três métodos estáticos (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) para uma nova classe chamada `DateUtil` [G6].

Subi os métodos abstratos para o topo, onde eles pertencem [G24].

Alterei de `Month.make` para `Month.fromInt` [N1] e fiz o mesmo com todos os outros enums.

Também criei um método acessor `toInt` para todos os enum e tornei privado o campo `index`. Havia umas duplicações interessantes [G5] em `plusYears` e em `plusMonths` que fui capaz de

eliminar extraíndo um novo método chamado `correctLastDayOfMonth`, o que deixou todos os três métodos muito mais claros.

Livrei-me do número mágico 1 [G25], substituindo-o por `Month.JANUARY.toInt()` ou `Day.SUNDAY.toInt()`, conforme apropriado. Gastei um tempo limpando um pouco os algoritmos de `SpreadsheetDate`. O resultado final vai da Listagem B.7 (p. 394) até a Listagem B.16 (p. 405). É interessante como a cobertura dos testes no código de `DayDate` caiu para 84.9%! Isso não se deve à menor quantidade de funcionalidade testada, mas à classe que foi tão reduzida que as poucas linhas que não eram testadas eram o maior problema. Agora a `DayDate` possui 45 de 53 instruções executáveis cobertas pelos testes. As linhas que não são cobertas são tão triviais que não vale a pena testá-las.

## Conclusão

Então, mais uma vez seguimos a Regra de Escoteiro. Deixamos o código um pouco mais limpo do que antes. Levou um tempo, mas vale a pena. Aumentamos a cobertura dos testes, consertamos alguns bugs e esclarecemos e reduzimos o código. Espero que a próxima pessoa que leia este código ache mais fácil lidar com ele do que nós achamos. Aquela pessoa provavelmente também será capaz de limpá-lo ainda mais do que nós.

## Bibliografia

[GOF]: Padrões de Projeto, Soluções Reutilizáveis de Software Orientado a Objetos, Gamma et al., Addison-Wesley, 1996.

[Simmons04]: Hardcore Java, Robert Simmons, Jr., O'Reilly, 2004.

[Refatoração]: Refatoração - Aperfeiçoando o Projeto de Código Existente, Martin Fowler et al., Addison-Wesley, 1999.

[Beck97]: Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997.

17

## Odores e Heurísticas



Em seu magnífico livro Refatoração<sup>1</sup>, Martin Fowler identificou muitos “odores diferentes de código”. A lista seguinte possui muitos desses odores de Martin e muitos outros meus. Há também outras pérolas e heurísticas que uso para praticar meu ofício.

Compilei essa lista ao analisar e refatorar diferentes programas. Conforme os alterava, eu me perguntava por que fiz aquela modificação e, então, escrevia o motivo aqui. O resultado é uma extensa lista de coisas que cheiram ruim para mim quando leio um código.

Esta lista é para ser lida de cima para baixo e também se deve usá-la como referência.

Há uma referência cruzada para cada heurística que lhe mostra onde está o que é referenciado no resto do texto no Apêndice C na página 409.

## Comentários

### C1: *Informações inapropriadas*

Não é apropriado para um comentário deter informações que ficariam melhores em um outro tipo diferente de sistema, como o seu sistema de controle de seu código-fonte, seu sistema de rastreamento de problemas ou qualquer outro sistema que mantenha registros. Alterar históricos, por exemplo, apenas amontoa os arquivos fonte com volumes de textos passados e desinteressantes. De modo geral, metadados, como autores, data da última atualização, número SRP e assim por diante, não deve ficar nos comentários. Estes devem conter apenas dados técnicas sobre o código e o projeto.

### C2: *Comentário obsoleto*

Um comentário que ficou velho, irrelevante e incorreto é obsoleto. Comentários ficam velhos muito rápido, logo é melhor não escrever um que se tornará obsoleto. Caso você encontre um, é melhor atualizá-lo ou se livrar dele o quanto antes. Comentários obsoletos tendem a se desviar do código que descreviam. Eles se tornam ilhas flutuantes de irrelevância no código e passam informações erradas.

### C3: *Comentários redundantes*

Um comentário é redundante se ele descreve algo que já descreve a si mesmo. Por exemplo:

```
i++; // incrementa i
```

Outro exemplo é um Javadoc que nada diz além da assinatura da função:

```
/**  
 * @param sellRequest  
 * @return  
 * @throws ManagedComponentException  
 */  
public SellResponse beginSellItem(SellRequest sellRequest)  
    throws ManagedComponentException
```

Os comentários devem informar o que o código não consegue por si só.

## C4: Comentário mal escrito

Um comentário que valha ser escrito deve ser bem escrito. Se for criar um, não tenha pressa e certifique-se de que seja o melhor comentário que você já escreveu. Selecione bem suas palavras. Use corretamente a gramática e a pontuação. Não erre. Não diga o óbvio.

Seja breve.

## C5: Código como comentário

Fico louco ao ver partes de código como comentários. Quem sabe a época em que foi escrito? Quem sabe se é ou não significativo? Mesmo assim, ninguém o exclui porque todos assumem que outra pessoa precisa dele ou tem planos para ele.

O código permanece lá e apodrece, ficando cada vez menos relevante a cada dia que passa. Ele chama funções que não existem mais; usa variáveis cujos nomes foram alterados; segue convenções que há muito se tornaram obsoletas; polui os módulos que o contêm e distrai as pessoas que tentam lê-lo. Colocar códigos em comentários é uma abominação.

Quando você vir um código como comentário, exclua-o! Não se preocupe, o sistema de controle de código fonte ainda se lembrará dele. Se alguém precisar dele, poderá verificar a versão anterior. Não deixe que códigos como comentários existam.

## Ambiente

### E1: Construir requer mais de uma etapa

Construir um projeto deve ser uma operação simples e única. Você não deve: verificar muitos pedacinhos do controle de código-fonte; precisar de uma sequência de comandos arcaicos ou scripts dependentes de contexto de modo a construir elementos individuais; ter de buscar perto e longe vários JARs extras, arquivos XML e outros componentes que o sistema precise. E você deve ser capaz de verificar o sistema com um único comando e, então, dar outro comando simples para construí-lo.

```
svn get mySystem  
cd mySystem  
ant all
```

### E2: Testes requerem mais de uma etapa

Você deve ser capaz de rodar todos os testes de unidade com apenas um comando. No melhor dos casos, você pode executar todos ao clicar em um botão em sua IDE. No pior, você deve ser capaz de dar um único comando simples numa shell. Poder rodar todos os testes é tão essencial e importante que deve ser rápido, fácil e óbvio de se fazer.

## Funções

### F1: Parâmetros em excesso

As funções devem ter um número pequeno de parâmetros. Ter nenhum é melhor. Depois vem um, dois e três. Mais do que isso é questionável e deve-se evitar com preconceito. (Consulte Parâmetros de funções na página 40.)

### F2: Parâmetros de saída

Os parâmetros de saída são inesperados. Os leitores esperam que parâmetros sejam de entrada, e não de saída. Se sua função deve alterar o estado de algo, faça-a mudar o do objeto no qual ela é chamada. (Consulte Parâmetros de saída na página 45.)

### F3: Parâmetros lógicos

Parâmetros booleanos explicitamente declaram que a função faz mais de uma coisa. Eles são confusos e se devem eliminá-los (Consulte Parâmetros lógicos na página 41).

### F4: Função morta

Devem-se descartar os métodos que nunca são chamados. Manter pedaços de código mortos é devastador. Não tenha receio de excluir a função. Lembre-se de que seu o sistema de controle de código fonte ainda se lembrará dela.

## Geral

### G1: Múltiplas linguagens em um arquivo fonte

Os ambientes modernos de programação atuais possibilitam colocar muitas linguagens distintas em um único arquivo fonte. Por exemplo, um arquivo-fonte Java pode conter blocos em XML, HTML, YAML, JavaDoc, português, JavaScript, etc. Outro exemplo seria adicionar ao HTML um arquivo JSP que contenha Java, uma sintaxe de biblioteca de tags, comentários em português, Javadoc, etc. Na melhor das hipóteses, isso é confuso, e na pior, negligentemente desleixado. O ideal para um arquivo-fonte é ter uma, apenas uma, linguagem. Mas na vida real, provavelmente teremos de usar mais de uma. Devido a isso, devemos minimizar tanto a quantidade como o uso de linguagens extras em nossos arquivos-fonte.

### G2: Comportamento óbvio não é implementado

Seguindo o “Princípio da Menor Supresa”<sup>2</sup>, qualquer função ou classe deve implementar os comportamentos que outro programador possivelmente esperaria. Por exemplo, considere uma função que traduza o nome de um dia em um enum que represente o dia.

```
Day day = DayDate.StringToDate(String dayName);
```

<sup>2</sup> Ou “O Princípio da Surpresa Mínima”: [http://en.wikipedia.org/wiki/Principle\\_of\\_least\\_astonishment](http://en.wikipedia.org/wiki/Principle_of_least_astonishment)

Esperamos que a string “Segunda” seja traduzido para Dia.SEGUNDA. Também esperamos que as abreviações comuns sejam traduzidas, e que a função não faça a distinção entre letras maiúsculas e minúsculas.

Quando um comportamento não é implementado, os leitores e usuários do código não podem mais depender de suas intuições sobre o que indica o nome das funções. Aquelas pessoas perdem a confiança no autor original e devem voltar e ler todos os detalhes do código.

### G3: *Comportamento incorreto nos limites*

Parece óbvio dizer que o código deva se comportar corretamente. O problema é que raramente percebemos como é complicado um comportamento correto. Desenvolvedores geralmente criam funções as quais eles acham que funcionarão, e, então, confiam em suas intuições em vez de se esforçar para provar que o código funciona em todos os lugares e limites.

Não existe substituição para uma dedicação minuciosa. Cada condição de limite, cada canto do código, cada trato e exceção representa algo que pode estragar um algoritmo elegante e intuitivo. Não dependa de sua intuição. Cuide de cada condição de limite e crie testes para cada.

### G4: *Seguranças anuladas*

Chernobyl derreteu porque o gerente da planta anulou cada um dos mecanismos de segurança, um a um. Os dispositivos de segurança estavam tornando inconveniente a execução de um experimento. O resultado era que o experimento não rodava, e o mundo viu a maior catástrofe civil nuclear.

É arriscado anular assegurâncias. Talvez seja necessário forçar o controle manual em serialVersionUID, mas há sempre um risco. Desabilitar certos avisos (ou todos!) do compilador talvez ajude a fazer a compilação funcionar com êxito, mas com o risco de infundáveis sessões de depuração. Desabilitar os testes de falhas e dizer a si mesmo que os aplicará depois é tão ruim quanto fingir que seus cartões de crédito sejam dinheiro gratuito.

### G5: *Duplicação*

Essa é uma das regras mais importantes neste livro, e você deve levá-la muito a sério. Praticamente, todo autor que escreve sobre projetos de software a mencionam. Dave Thomas e Andy Hunt a chamaram de princípio de DRY<sup>3</sup> (Princípio do Não Se Repita), o qual Kent Beck tornou o centro dos princípios da eXtreme Programming (XP) e o chamou de “Uma vez, e apenas uma”.

Ron Jeffries colocou essa como a segunda regra, sendo a primeira aquela em que se deve fazer todos os testes passarem com êxito.

Sempre que você vir duplicação em código, isso significa que você perdeu uma chance para . Aquela duplicação provavelmente poderia se tornar uma sub-rotina ou talvez outra classe completa. Ao transformar a duplicação em tal , você aumenta o vocabulário da linguagem de seu projeto. Outros programadores podem usar os recursos de que você criar, E a programação se torna mais rápida e menos propensa a erros devido a você ter elevado o nível de .

A forma mais óbvia de duplicação é quando você possui blocos de código idênticos, como se alguns programadores tivessem saído copiando e colando o mesmo código várias vezes. Aqui se deve substituir por métodos simples. Uma forma mais simples seriam as estruturas aninhadas de

`switch/case` e `if/else` que aparecem repetidas vezes em diversos módulos, sempre testando as mesmas condições. Nesse caso, deve-se substituir pelo polimorfismo.

Formas ainda mais sutis seriam os módulos que possuem algoritmos parecidos, mas que não possuem as mesmas linhas de código. Isso ainda é duplicação e deveria-se resolvê-la através do padrão TEMPLATE METHOD<sup>4</sup> ou STRATEGY<sup>5</sup>.

Na verdade, a maioria dos padrões de projeto que têm surgido nos últimos 15 anos são simplesmente maneiras bem conhecidas para eliminar a duplicação. Assim como as regras de normalização (Normal Forms) de Codd são uma estratégia para eliminar a duplicação em bancos de dados. A OO em si – e também a programação estruturada – é uma tática para organizar módulos e eliminar a duplicação.

Acho que a mensagem foi passada: encontre e elimine duplicações sempre que puder.

## G6: Códigos no nível errado de abstração

É importante criar abstrações que separem conceitos gerais de níveis mais altos dos conceitos detalhados de níveis mais baixos. Às vezes, fazemos isso criando classes abstratas que contenham os conceitos de níveis mais altos e gerando derivadas que possuam os conceitos de níveis mais baixos. Com isso, garantimos uma divisão completa. Queremos que todos os conceitos de níveis mais altos fiquem na classe base e que todos os de níveis mais baixos fiquem em suas derivadas.

Por exemplo, constantes, variáveis ou funções que possuam apenas a implementação detalhada não devem ficar na classe base. Essa não deve enxergar o resto.

Essa regra também se aplica aos arquivos-fonte, componentes e módulos. Um bom projeto de software exige que separemos os conceitos em níveis diferentes e que os coloquemos em contêineres distintos. De vez em quando, esses contêineres são classes base ou derivadas, e, às vezes, arquivos-fonte, módulos ou componentes. Seja qual for o caso, a separação deve ser total. Não queremos que os conceitos de baixo e alto níveis se misturem.

Considere o código seguinte:

```
public interface Stack {
    Object pop() throws EmptyException;
    void push(Object o) throws FullException;
    double percentFull();
    class EmptyException extends Exception {}
    class FullException extends Exception {}
}
```

A função `percentFull` está no nível errado de . Embora haja muitas implementações de `Stack` onde o conceito de plenitude seja razoável, existem outras implementações que simplesmente não poderiam enxergar quão completas elas são. Sendo assim, seria melhor colocar a função em uma interface derivada, como a `BoundedStack`.

Talvez você esteja pensando que a implementação poderia simplesmente retornar zero se a pilha (stack) fosse ilimitada. O problema com isso é que nenhuma pilha é realmente infinita. Você não consegue evitar uma `OutOfMemoryException` ao testar

```
stack.percentFull() < 50.0.
```

Implementar a função para retornar zero seria mentir.

A questão é que você não pode mentir ou falsificar para consertar uma mal posicionada. Isolar as abstrações é uma das coisas mais difíceis para os desenvolvedores de software, e não há uma solução rápida quando você erra.

## G7: As classes base dependem de suas derivadas

A razão mais comum para separar os conceitos em classes base e derivadas é para que os conceitos de níveis mais altos das classes base possam ficar independentes dos conceitos de níveis mais baixos das classes derivadas. Portanto, quando virmos classes base mencionando os nomes de suas derivadas, suspeitaremos de um problema. De modo geral, as classes base não deveriam enxergar nada em suas derivadas.

Essa regra possui exceções, é claro. De vez em quando, o número de derivadas é fixado, e a classe base tem códigos que consultam suas derivadas. Vemos isso em muitas implementações de máquinas com configuração finita. Porém, neste caso, as classes base e derivadas estão fortemente acopladas e são sempre implementadas juntas no mesmo arquivo jar. De modo geral, queremos poder implementar as classes base e derivadas em arquivos jar diferentes.

Conseguir isso e garantir que os arquivos base jar não enxerguem o conteúdo dos arquivos derivados jar, nos permite implementar nossos sistemas em componentes independentes e distintos. Ao serem modificados, podem-se implementar novamente esses componentes sem ter de fazer o mesmo com os componentes base. Isso significa que o impacto de uma alteração é consideravelmente reduzido, e fazer a manutenção dos sistemas no local se torna muito mais simples.

## G8: Informações excessivas

Módulos bem definidos possuem interfaces pequenas que lhe permite fazer muito com pouco. Já os mal definidos possuem interfaces grandes e longas que lhe obriga a usar muitas formas diferentes para efetuar coisas simples. Uma interface bem definida não depende de muitas funções, portanto, o acoplamento é fraco. E uma mal definida depende de diversas funções que devem ser chamadas, gerando um forte acoplamento.

Bons desenvolvedores de software aprendem a limitar o que expõem nas interfaces de suas classes e módulos. Quanto menos métodos tiver uma classe, melhor. Quanto menos variáveis uma função usar, melhor. Quanto menos variáveis tiver uma classe, melhor.

Esconda seus dados. Esconda suas funções utilitárias. Esconda suas constantes e suas variáveis temporárias.

Não crie classes com muitos métodos ou instâncias de variáveis. Não crie muitas variáveis e funções protegidas para suas subclasses. Concentre-se em manter as interfaces curtas e muito pequenas. Limite as informações para ajudar a manter um acoplamento fraco.

## G9: Código morto

Um código morto é aquele não executado. Pode-se encontrá-lo: no corpo de uma estrutura `if` que verifica uma condição que não pode acontecer; no bloco `catch` de um `try` que nunca ocorre; em pequenos métodos utilitários que nunca são chamados ou em condições da estrutura

switch/case que nunca ocorrem.

O problema com códigos mortos é que após um tempo ele começa a “cheirar”. Quanto mais antigo ele for, mais forte e desagradável o odor se torna. Isso porque um código morto não é atualizado completamente quando um projeto muda. Ele ainda compila, mas não segue as novas convenções ou regras. Ele foi escrito numa época quando o sistema era diferente. Quando encontrar um código morto, faça a coisa certa. Dê a ele um funeral decente. Exclua-o do sistema.

## G10: Separação vertical

Devem-se declarar as variáveis e funções próximas de onde são usadas. Devem-se declarar as variáveis locais imediatamente acima de seu primeiro uso, e o escopo deve ser vertical. Não queremos que variáveis locais sejam declaradas centenas de linhas afastadas de onde são utilizadas.

Devem-se declarar as funções provadas imediatamente abaixo de seu primeiro uso. Elas pertencem ao escopo de toda a classe. Mesmo assim, ainda desejamos limitar a distância vertical entre as chamadas e as declarações. Encontrar uma função privada deve ser uma questão de buscar para baixo a partir de seu primeiro uso.

## G11: Inconsistência

Se você fizer algo de uma determinada maneira, faça da mesma forma todas as outras coisas similares. Isso retoma o princípio da surpresa mínima. Atenção ao escolher suas convenções. Uma vez escolhidas, atente para continuar seguindo-as.

Se dentro de uma determinada função você usar uma variável de nome `response` para armazenar uma `HttpServletResponse`, então use o mesmo nome da variável de nome consistente nas outras funções que usem os objetos `HttpServletResponse`. Se chamar um método de `processVerificationRequest`, então use um nome semelhante, como `processDeletionRequest`, para métodos que processem outros tipos de pedidos (`request`). Uma simples consistência como essa, quando aplicada corretamente, pode facilitar muito mais a leitura e a modificação do código.

## G12: Entulho

De que serve um construtor sem implementação alguma? Só serve para amontoar o código com pedaços inúteis. Variáveis que não são usadas, funções que jamais são chamadas, comentários que não acrescentam informações e assim por diante, são tudo entulhos e devem ser removidos. Mantenha seus arquivos-fonte limpos, bem organizados e livres de entulhos.

## G13: Acoplamento artificial

Coisas que não dependem uma da outra não devem ser acopladas artificialmente. Por exemplo, enums genéricos não devem ficar dentro de classes mais específicas, pois isso obriga todo o aplicativo a enxergar mais essas classes. O mesmo vale para funções estáticas de propósito geral declaradas em classes específicas.

De modo geral, um acoplamento artificial é um acoplamento entre dois módulos que não possuem

um propósito direto. Isso ocorre quando se colocar uma variável, uma constante ou uma função em um local temporariamente conveniente, porém inapropriado. Isso é descuido e preguiça. Tome seu tempo para descobrir onde devem ser declaradas as funções, as constantes e as variáveis. Não as jogue no local mais conveniente e fácil e as deixe lá.

## G14: Feature Envy

Esse é um dos smels<sup>6</sup> (odores) de código de Martin Fowler. Os métodos de uma classe devem ficar interessados nas variáveis e funções da classe a qual eles pertencem, e não nas de outras classes. Quando um método usa métodos de acesso e de alteração de algum outro objeto para manipular os dados dentro deste objeto, o método inveja o escopo da classe daquele outro objeto. Ele queria estar dentro daquela outra classe de modo que pudesse ter acesso direto às variáveis que está manipulando. Por exemplo:

```
public class HourlyPayCalculator {  
    public Money calculateWeeklyPay(HourlyEmployee e) {  
        int tenthRate = e.getTenthRate().getPennies();  
        int tenthsWorked = e.getTenthsWorked();  
        int straightTime = Math.min(400, tenthsWorked);  
        int overTime = Math.max(0, tenthsWorked -  
            straightTime);  
        int straightPay = straightTime * tenthRate;  
        int overtimePay = (int) Math.  
            round(overTime*tenthRate*1.5);  
        return new Money(straightPay + overtimePay);  
    }  
}
```

O método `calculateWeeklyPay` consulta o objeto `HourlyEmployee` para obter os dados nos quais ele opera. Então, o método `HourlyEmployee` inveja o escopo de `HourlyEmployee`. Ele “queria” poder estar dentro de `HourlyEmployee`.

Sendo todo o resto igual, desejamos eliminar a Feature Envy (“inveja de funcionalidade”, tradução livre), pois ela expõe os componentes internos de uma classe à outra. De vez em quando, entretanto, esse é um mal necessário. Considere o seguinte:

```
public class HourlyEmployeeReport {  
    private HourlyEmployee employee ;  
  
    public HourlyEmployeeReport(HourlyEmployee e) {  
        this.employee = e;  
    }  
  
    String reportHours() {  
        return String.format(  
            "Name: %s\\tHours:%d.%ld\\n",  
            employee.getName(),  
            employee.getTenthsWorked()/10,  
            employee.getTenthsWorked()%10);  
    }  
}
```

} *... o código continua com mais comentários de código...*

Está claro que o método `reportHours` inveja a classe `HourlyEmployee`. Por outro lado, não queremos que `HourlyEmployee` tenha de enxergar o formato do relatório. Mover aquela string de formato para a classe `HourlyEmployee` violaria vários princípios do projeto orientada a objeto<sup>7</sup>, pois acoplaria `HourlyEmployee` ao formato do relatório, expondo as alterações feitas naquele formato.

## G15: Parâmetros seletores

Dificilmente há algo mais abominável do que um parâmetro `false` pendurado no final da chamada de uma função. O que ele significa? O que mudaria se ele fosse `true`? Não bastava ser difícil lembrar o propósito de um parâmetro seletor, cada um agrupa muitas funções em uma única. Os parâmetros seletores são uma maneira preguiçosa de não ter de dividir uma função grande em várias outras menores. Considere o seguinte:

```
public int calculateWeeklyPay(boolean overtime) {
    int tenthRate = getTenthRate();
    int tenthsWorked = getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
    int overtimePay = (int) Math.round(overTime * overtimeRate);
    return straightPay + overtimePay;
}
```

Você chama essa função com `true` se as horas extras forem pagas como uma hora e meia a mais, e `false` se forem como horas normais. Já é ruim o bastante ter de lembrar o que `calculateWeeklyPay(false)` significa sempre que você a vir. Mas o grande problema de uma função como essa está na oportunidade que o autor deixou passar de escrever o seguinte:

```
public int straightPay() {
    return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
    int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
    int overTimePay = overTimeBonus(overTimeTenths);
    return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
    double bonus = 0.5 * getTenthRate() * overTimeTenths;
    return (int) Math.round(bonus);
}
```

É claro que os seletores não precisam ser booleanos. Podem ser enums, inteiros ou outro tipo de parâmetro usado para selecionar o comportamento da função. De modo geral, é melhor ter

<sup>7</sup> Especificamente, o Princípio da Responsabilidade Única, o Princípio de Aberto-Fechado e o Princípio do Fecho Comum. Consulte [PPP].

muitas funções do que passar um código por parâmetro para selecionar o comportamento.

## G16: Propósito obscuro

Queremos que o código seja o mais expressivo possível. Expressões contínuas, notação húngara e números mágicos são elementos que obscurecem a intenção do autor. Por exemplo, abaixo está como poderia aparecer a função `overtimePay`:

```
public int overtimePay() {
    return iThsWkd * iThsRte +
           (int) Math.round(0.5 * iThsRte *
                           Math.max(0, iThsWkd - 400));
}
```

Pequena e concisa como pode parecer, ela também é praticamente impenetrável. Vale a pena separar um tempo para tornar visível o propósito de nosso código para nossos leitores.

## G17: Responsabilidade mal posicionada

Onde colocar o código é uma das decisões mais importantes que um desenvolvedor de software deve fazer. Por exemplo, onde colocar a constante PI? Na classe `Math`? Talvez na classe `Trigonometry`? Ou quem sabe na classe `Circle`?

O princípio da surpresa mínima entra aqui. Deve-se substituir o código onde um leitor geralmente espera. A constante PI deve ficar onde estão declaradas as funções de trigonometria. A constante OVERTIME\_RATE deve ser declarada na função `HourlyPayCalculator`.

Às vezes damos uma de “espertinhos” na hora de posicionar certa funcionalidade. Colocamo-la numa função que é conveniente para nós, mas não necessariamente intuitiva para o leitor. Por exemplo, talvez precisemos imprimir um relatório com o total de horas trabalhadas por um funcionário. Poderíamos somar todas aquelas horas no código que imprime o relatório, ou tentar criar um cálculo contínuo do total num código que aceite uma interação com os cartões de ponto. Uma maneira de tomar essa decisão é olhar o nome das funções. Digamos que nosso módulo de relatório possua uma função `getTotalHours`. Digamos também que esse módulo aceite uma interação com os cartões de ponto e tenha uma função `saveTimeCard`. Qual das duas funções, baseando-se no nome, indica que ela calcula o total? A resposta deve ser óvia.

Claramente, há, às vezes, questões de desempenho pelo qual se deva calcular o total usando-se os cartões de ponto em vez de fazê-lo na impressão do relatório. Tudo bem, mas os nomes das funções devem refletir isso. Por exemplo, deve existir uma função `computeRunningTotalOfHours` no módulo `timecard`.

## G18: Modo estático inadequado

`Math.max(double a, double b)` é um bom método estático. Ele não opera em só uma instância; de fato, seria tolo ter de usar `Math().max(a, b)` ou mesmo `a.max(b)`.

Todos os dados que `max` usa vêm de seus dois parâmetros, e não de qualquer objeto “pertencente”

a ele. Sendo mais específico, há quase nenhuma chance de querermos que `Math.max` seja polifórmico.

Às vezes, contudo, criamos funções estáticas que não deveriam ser. Por exemplo, considere a função

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Novamente, pode parecer uma função estática lógica. Ela não opera em nenhum objeto em particular e obtém todos os seus dados a partir de seus parâmetros. Entretanto, há uma chance razoável de desejarmos que essa função seja polifórmica. Talvez desejemos implementar diversos algoritmos diferentes para calcular o pagamento por hora, por exemplo, `OvertimeHourlyPayCalculator` e `StraightTimeHourlyPayCalculator`. Portanto, neste caso, a função não deve ser estática, e sim uma função membro não estática de `Employee`.

Em geral, deve-se dar preferência a métodos não estáticos. Na dúvida, torne a função não estática. Se você realmente quiser uma função estática, certifique-se de que não há possibilidades de você mais tarde desejar que ela se comporte de maneira polifórmica.

## G19: Use variáveis descritivas

Kent Beck escreveu sobre isso em seu ótimo livro chamado *Smalltalk Best Practice Patterns*<sup>8</sup>, e mais recentemente em outro livro também ótimo chamado *Implementation Patterns*<sup>9</sup>. Uma das formas mais poderosas de tornar um programa legível é separar os cálculos em valores intermediários armazenados em variáveis com nomes descritivos.

Considere o exemplo seguinte do FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
    String key = match.group(1);
    String value = match.group(2);
    headers.put(key.toLowerCase(), value);
}
```

O simples uso de variáveis descritivas esclarece que o primeiro grupo de comparação (`match group`) é a chave (`key`), e que o segundo é o valor (`value`).

É difícil fazer mais do que isso. Mais variáveis explicativas geralmente são melhores do que menos. É impressionante como um módulo opaco pode repentinamente se tornar transparente simplesmente ao separar os cálculos em valores intermediários bem nomeados.

## G20: Nomes de funções devem dizer o que elas fazem

Veja este código:

```
Date newDate = date.add(5);
```

Você acha que ele adiciona cinco dias à data? Ou seria a semanas, ou horas? A instância `date` mudou ou a função simplesmente retornou uma nova `Date` sem alterar a antiga?

8. [Beck97], p. 108.

9. [Beck07].

Não dá para saber a partir da chamada o que a função faz.

Se a função adiciona cinco dias à data e a altera, então ela deveria se chamar `addDaysTo` ou `increaseByDays`. Se, por outro lado, ela retorna uma nova data acrescida de cinco dias, mas não altera a instância date, ela deveria se chamar `daysLater` ou `daysSince`.

Se você tiver de olhar a implementação (ou a documentação) da função para saber o que ela faz, então é melhor selecionar um nome melhor ou reorganizar a funcionalidade de modo que esta possa ser colocada em funções com nomes melhores.

## G21: Entenda o algoritmo

Criam-se muitos códigos estranhos porque as pessoas não gastam tempo para entender o algoritmo. Elas fazem algo funcionar jogando estruturas `if` e flags, sem parar e pensar no que realmente está acontecendo.

Programa geralmente é uma análise. Você acha que conhece o algoritmo certo para algo e, então, acaba perdendo tempo com ele e pincelando aqui e ali até fazê-lo “funcionar”. Como você sabe que ele “funciona”? Porque ele passa nos casos de teste nos quais você conseguiu pensar.

Não há muito de errado com esta abordagem. De fato, costuma ser a única forma de fazer uma função funcionar como você acha que ela deva. Entretanto, deixar a palavra “funcionar” entre aspas não é o suficiente.

Antes de achar que já terminou com uma função, certifique-se de que você entenda como ela funciona. Ter passado em todos os testes não basta. Você deve compreender<sup>10</sup> que a solução está correta. Geralmente, a melhor forma de obter esse conhecimento e entendimento é refatorar a função em algo que seja tão limpo e expressivo que fique óbvio que ela funciona.

## G22: Torne dependências lógicas em físicas

Se um módulo depende de outro, essa dependência deve ser física, e não apenas lógica. O módulo dependente não deve fazer suposições (em outras palavras, dependências lógicas) sobre o módulo no qual ele depende. Em vez disso, ele deve pedir explicitamente àquele módulo todas as informações das quais ele depende.

Por exemplo, imagine que você esteja criando uma função que imprima um relatório de texto simples das horas trabalhadas pelos funcionários. Uma classe chamada `HourlyReporter` junta todos os dados em um formulário e, então, o passa para `HourlyReportFormatter` imprimi-lo. Não estar certo se um algoritmo é adequado costuma ser um fato da vida. Não estar certo do que faz o seu código é simplesmente preguiça.

**Listagem 17-1****HourlyReporter.java**

```

public class HourlyReporter {
    private HourlyReportFormatter formatter;
    private List<LineItem> page;
    private final int PAGE_SIZE = 55;

    public HourlyReporter(HourlyReportFormatter formatter) {
        this.formatter = formatter;
        page = new ArrayList<LineItem>();
    }

    public void generateReport(List<HourlyEmployee> employees) {
        for (HourlyEmployee e : employees) {
            addLineItemToPage(e);
            if (page.size() == PAGE_SIZE)
                printAndClearItemList();
        }
        if (page.size() > 0)
            printAndClearItemList();
    }

    private void printAndClearItemList() {
        formatter.format(page);
        page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
        LineItem item = new LineItem();
        item.name = e.getName();
        item.hours = e.getTenthsWorked() / 10;
        item.tenths = e.getTenthsWorked() % 10;
        page.add(item);
    }

    public class LineItem {
        public String name;
        public int hours;
        public int tenths;
    }
}

```

Esse código possui uma dependência que não foi transformada em física. Você consegue enxergá-la? É a constante PAGE\_SIZE. Por que HourlyReporter deveria saber o tamanho da página? Isso deveria ser responsabilidade de HourlyReportFormatter.

O fato de PAGE\_SIZE estar declarada em HourlyReporter representa uma responsabilidade mal posicionada [G17] que faz HourlyReporter assumir que ele sabe qual deve ser o tamanho da página. Tal suposição é uma dependência lógica. HourlyReporter depende do fato de que HourlyReportFormatter pode lidar com os tamanhos 55 de página. Se alguma implementação de HourlyReportFormatter não puder lidar com tais tamanhos, então haverá um erro. Podemos tornar essa dependência física criando um novo método chamado `getMaxPageSize()` em HourlyReportFormatter. Então, HourlyReporter chamará a função em vez de usar a constante PAGE\_SIZE.

## G23: Prefira polimorfismo a if...else ou switch...case

Essa pode parecer uma sugestão estranha devido ao assunto do Capítulo 6. Afinal, lá eu disse que as estruturas switch possivelmente são adequadas nas partes do sistema nas quais a adição de novas funções seja mais provável do que a de novos tipos.

Primeiro, a maioria das pessoas usa os switch por ser a solução por força bruta óbvia, e não por ser a correta para a situação. Portanto, essa heurística está aqui para nos lembrar de considerar o polimorfismo antes de usar um switch.

Segundo, são relativamente raros os casos nos quais as funções são mais voláteis do que os tipos. Sendo assim, cada estrutura switch deve ser um suspeito.

Eu uso a seguinte regra do “UM SWITCH”: Não pode existir mais de uma estrutura switch para um dado tipo de seleção. Os casos nos quais o switch deva criar objetos polifôrmicos que substituam outras estruturas switch no resto do sistema.

## G24: Siga as convenções padrões

Cada equipe deve seguir um padrão de programação baseando-se nas normas comuns do mercado. Esse padrão deve especificar coisas como onde declarar instâncias de variáveis; como nomear classes, métodos e variáveis; onde colocar as chaves; e assim por diante. A equipe não precisa de um documento que descreva essas convenções, pois seus códigos fornecem os exemplos.

Cada membro da equipe deve seguir essas convenções. Isso significa que cada um deve ser maduro o suficiente para entender que não importa onde você coloque suas chaves contanto que todos concordem onde colocá-las.

Se quiser saber quais convenções eu sigo, veja o código refatorado da Listagem B.7 (p. 394) até a B.14.

## G25: Substitua os números mágicos por constantes com nomes

Essa é provavelmente uma das regras mais antigas em desenvolvimento de software. Lembro-me de tê-la lido no final da década de 1960 nos manuais de introdução de COBOL, FORTRAN e PL/1. De modo geral, é uma péssima ideia ter números soltos em seu código. Deve-se escondê-los em constantes com nomes bem selecionados.

Por exemplo, o número 86.400 deve ficar escondido na constante SECONDS\_PER\_DAY. Se você for imprimir 55 linhas por página, então a constante 55 deva ficar na constante LINES\_PER\_PAGE.

Algumas constantes são tão fáceis de reconhecer que nem sempre precisam de um nome para armazená-las, contanto que sejam usadas juntamente com um código bastante auto-explicativo. Por exemplo:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Realmente precisamos das constantes FEET\_PER\_MILE, WORK\_HOURS\_PER\_DAY e TWO no

exemplo acima? Está óbvio que o último caso é um absurdo. Há algumas fórmulas nas quais fica melhor escrever as constantes simplesmente como números. Talvez você reclame do caso de `WORK_HOURS_PER_DAY`, pois as leis e convenções podem mudar. Por outro lado, aquela fórmula é tão fácil de ler com o 8 nela que eu ficaria relutante em adicionar 17 caracteres extras para o leitor. E no caso de `FEET_PER_MILE`, o número 5280 é tão conhecido e exclusivo que os leitores reconheceriam mesmo se estivesse numa página sem contexto ao redor.

Constantes como 3.141592653589793 também são tão conhecidas e facilmente reconhecíveis. Entretanto, a chance de erros é muito grande para deixá-las como números. Sempre que alguém vir 3.1415927535890793, saberão que é pi, e, portanto, não olharão com atenção. (Percebeu que um número está errado?) Também não queremos pessoas usando 3,14, 3,14159, 3,142, e assim por diante. Mas é uma boa coisa que `Math.PI` já tenha sido definida para nós.

O termo “Número Mágico” não se aplica apenas a números, mas também a qualquer token (simbolos, termos, expressões, números, etc.) que possua um valor que não seja auto-explicativo. Por exemplo:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

Há dois números mágicos nessa confirmação. Obviamente o primeiro é 7777, embora seu significado possa não ser óbvio. O segundo é “John Doe”, e, novamente, o propósito não está claro.

Acabou que “John Doe” é o nome do funcionário #7777 em um banco de dados de testes criado por nossa equipe. Todos na equipe sabem que ao se conectar a este banco de dados, ele já possuirá diversos funcionários incluídos com valores e atributos conhecidos. Também acabou que “John Doe” representa o único funcionário horista naquele banco de dados. Sendo assim, esse teste deve ser:

```
assertEquals(
    HOURLY_EMPLOYEE_ID,
    Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

## G26: Seja preciso

Esperar que a primeira comparação seja a única feita em uma consulta é ser ingênuo. Usar números de ponto flutuante para representar moedas é quase criminoso. Evitar gerenciamento de bloqueios e/ou transações porque você não acha que a atualização concorrente seja provável, é no mínimo desleixo. Declarar uma variável para ser uma `ArrayList` quando uma `List` é o suficiente, é totalmente constrangedor. Tornar protegidas por padrão todas as variáveis não é constrangedor o suficiente.

Quando você toma uma decisão em seu código, certifique-se de fazê-la precisamente. Saiba por que a tomou e como você lidará com quaisquer exceções. Não seja desleixado com a precisão de suas decisões. Se decidir chamar uma função que retorne `null`, certifique-se de verificar por `null`. Se for consultar o que você acha ser o único registro no banco de dados, garanta que seu código verifique se não há outros. Se precisar lidar com concorrência, use inteiros<sup>11</sup> e lide apropriadamente com o arredondamento. Se houver a possibilidade de atualização concorrente, certifique-se de implementar algum tipo de mecanismo de bloqueio.

Ambiguidades e imprecisão em códigos são resultado de desacordos ou desleixos. Seja qual for o caso, elas devem ser eliminadas.

<sup>11</sup> Ou, melhor ainda, uma classe `Money` que use inteiros.

## G27: Estrutura acima de convenção

Insista para que as decisões do projeto baseiem-se em estrutura acima de convenção. Convenções de nomenclatura são boas, mas são inferiores às estruturas, que forçam um certo cumprimento. Por exemplo, switch...cases com enumerações bem nomeadas são inferiores a classes base com métodos abstratos. Ninguém é obrigado a implementar a estrutura switch...case da mesma forma o tempo todo; mas as classes base obrigam a implementação de todos os métodos abstratos das classes concretas.

## G28: Encapsule as condicionais

A lógica booleana já é difícil o bastante de entender sem precisar vê-la no contexto de um if ou um while. Extraia funções que expliquem o propósito da estrutura condicional.

Por exemplo:

```
if (shouldBeDeleted(timer))
```

é melhor do que

```
if (timer.hasExpired() && !timer.isRecurrent())
```

## G29: Evite condicionais negativas

É um pouco mais difícil entender condições negativas do que afirmativas. Portanto, sempre que possível, use condicionais afirmativas. Por exemplo:

```
if (buffer.shouldCompact())
```

é melhor do que

```
if (!buffer.shouldNotCompact())
```

## G30: As funções devem fazer uma coisa só

Costuma ser tentador criar funções que tenham várias seções que efetuam uma série de operações. Funções desse tipo fazem mais de uma coisa, e devem ser divididas em funções melhores, cada um fazendo apenas uma coisa.

Por exemplo:

```
public void pay() {
    for (Employee e : employees) {
        if (e.isPayday()) {
            Money pay = e.calculatePay();
            e.deliverPay(pay);
        }
    }
}
```

Esse pedaço de código faz três coisas. Ele itera sobre todos os funcionários, verifica se cada um deve ser pago e, então, paga o funcionário. Esse código ficaria melhor assim:

```

public void pay() {
    for (Employee e : employees)
        payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
    if (e.isPayday())
        calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}

```

Cada uma dessas funções faz apenas uma coisa. (Consulte Faça apenas uma coisa na página 35.)

## G31: Acoplamentos temporários ocultos

Acoplamentos temporários costumam ser necessários, mas não se deve ocultá-los. Organize os parâmetros de suas funções de modo que a ordem na qual são chamadas fique óbvia.

Considere o seguinte:

```

public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
        saturateGradient();
        reticulateSplines();
        diveForMoog(reason);
    }
    ...
}

```

A ordem das três funções é importante. Você deve saturar o gradiente antes de poder dispor em formato de redes (reticulate) as ranhuras (splines) e só então você pode seguir para o Moog (dive for moog). Infelizmente, o código não exige esse acoplamento temporário. Outro programador poderia chamar `reticulateSplines` antes de `saturateGradient`, gerando uma `UnsaturatedGradientException`.

Uma solução melhor seria:

```

public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
        Gradient gradient = saturateGradient();
        List<Spline> splines = reticulateSplines(gradient);
        diveForMoog(reason);
    }
}

```

```
    diveForMoog(splines, reason);
}

...
}
```

Isso expõe o acoplamento temporário ao criar um “bucket brigade”\*1. Cada função produz um resultado que a próxima precisa, portanto não há uma forma lógica de chamá-los fora de ordem. Talvez você reclame que isso aumente a complexidade das funções, e você está certo. Mas aquela complexidade sintática extra expõe a complexidade temporal verdadeira da situação. Note que deixei as instâncias de variáveis. Presumo que elas sejam necessárias aos métodos privados na classe. Mesmo assim, mantive os parâmetros para tornar explícito o acoplamento temporário.

## G32: *Não seja arbitrário*

Tenha um motivo pelo qual você estruture seu código e certifique-se de que tal motivo seja informado na estrutura. Se esta parece arbitrária, as outras pessoas se sentirão no direito de alterá-la. Mas se uma estrutura parece consistente por todo o sistema, as outras pessoas irão usá-la e preservar a convenção utilizada. Por exemplo, recentemente eu fazia alterações ao FitNesse quando descobri que um de nossos colaboradores havia feito isso:

```
public class AliasLinkWidget extends ParentWidget
{
    public static class VariableExpandingWidgetRoot {
        ...
    }
}
```

O problema era que `VariableExpandingWidgetRoot` não tinha necessidade para estar dentro do escopo de `AliasLinkWidget`. Ademais, outras classes sem relação usavam a `AliasLinkWidget.VariableExpandingWidgetRoot`. Essas classes não precisavam enxergar a `AliasLinkWidget`.

Talvez o programador tenha jogado a `VariableExpandingWidgetRoot` dentro de `AliasWidget` por questão de conveniência. Ou talvez ele pensara que ela realmente precisava ficar dentro do escopo de `AliasWidget`. Seja qual for a razão, o resultado acabou sendo arbitrário. Classes públicas que não são usadas por outras classes não podem ficar dentro de outra classe. A convenção é torná-las públicas no nível de seus pacotes.

## G33: *Encapsule as condições de limites*

Condições de limite são difíceis de acompanhar. Coloque o processamento para elas em um único lugar.

Não as deixe espalhadas pelo código. Não queremos um enxame de `+1s` e `-1s` aparecendo aqui e acolá. Considere o exemplo abaixo do FIT:

```
if(level + 1 < tags.length)
{
```

```

        parts = new Parse(body, tags, level + 1, offset + endTag);
        body = null;
    }
}

```

Note que `level+1` aparece duas vezes. Essa é uma condição de limite que deveria estar encapsulada dentro de uma variável com um nome ou algo como `nextLevel`.

```

int nextLevel = level + 1;
if(nextLevel < tags.length)
{
    parts = new Parse(body, tags, nextLevel, offset + endTag);
    body = null;
}

```

### **G34: Funções devem descer apenas um nível de**

As instruções dentro de uma função devem ficar todas no mesmo nível de , o qual deve ser um nível abaixo da operação descrita pelo nome da função. Isso pode ser o mais difícil dessas heurísticas para se interpretar e seguir. Embora a ideia seja simples o bastante, os seres humanos são de longe ótimos em misturar os níveis de . Considere, por exemplo, o código seguinte retirado do FitNesse:

```

public String render() throws Exception
{
    StringBuffer html = new StringBuffer("<hr");
    if(size > 0)
        html.append(" size='").append(size + 1).append("'");
    html.append(">");
    return html.toString();
}

```

Basta analisar um pouco e você verá o que está acontecendo. Essa função constrói a tag HTML que desenha uma régua horizontal ao longo da página. A altura da régua é especificada na variável `size`.

Leia o método novamente. Ele está misturando pelo menos dois níveis de . O primeiro é a noção de que uma régua horizontal (horizontal rule, daí a tag hr) possui um tamanho. O segundo é a própria sintaxe da tag HR.

Esse código vem do módulo HruleWidget no FitNesse. Ele detecta uma sequência de quatro ou mais traços horizontais e a converte em uma tag HR apropriada. Quanto mais traços, maior o tamanho. Eu refatorei abaixo esse pedaço de código. Note que troquei o nome do campo `size` para refletir seu propósito real. Ele armazena o número de traços extras.

```

public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (extraDashes > 0)
        hr.addAttribute("size", hrSize(extraDashes));
    return hr.html();
}

```

```
private String hrSize(int height)
{
    int hrSize = height + 1;
    return String.format("%d", hrSize);
}
```

Essa mudança separa bem os dois níveis de . A função render simplesmente constrói uma tag HR, sem ter de saber a sintaxe HTML da tag.

O módulo HtmlTag trata de todas as questões complicadas da sintaxe.

Na verdade, ao fazer essa alteração, notei um pequeno erro. O código original não colocava uma barra na tag HR de fechamento, como o faria o padrão XHTML. (Em outras palavras, foi gerado `<hr>` em vez de `<hr />`.) O módulo HtmlTag foi alterado para seguir o XHTML há muito tempo. Separar os níveis de é uma das funções mais importantes da refatoração, e uma das mais difíceis também. Como um exemplo, veja o código abaixo. Ele foi a minha primeira tentativa em separar os níveis de no HruleWidget.render method.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (size > 0) {
        hr.addAttribute("size", ""+(size+1));
    }
    return hr.html();
}
```

Meu objetivo, naquele momento, era criar a separação necessária para fazer os testes passarem. Isso foi fácil, mas fiquei com uma função que ainda tinha níveis de misturados. Neste caso, eles estavam na construção da tag HR e na interpretação e formatação da variável size. Isso indica ao dividir uma função em linhas de , você geralmente descobre novas linhas de que estavam ofuscadas pela estrutura anterior.

## G35: Mantenha os dados configuráveis em níveis altos

Se você tiver uma constante, como um valor padrão ou de configuração, que seja conhecida e esperada em um nível alto de , não a coloque numa função de nível baixo. Exponha a constante como parâmetro para tal função, que será chamada por outra de nível mais alto. Considere o código seguinte do FitNesse:

```
public static void main(String[] args) throws Exception
{
    Arguments arguments = parseCommandLine(args);
    ...
}

public class Arguments
{
    public static final String DEFAULT_PATH = ".";
```

```

public static final String DEFAULT_ROOT = "FitNesseRoot";
public static final int DEFAULT_PORT = 80;
public static final int DEFAULT_VERSION_DAYS = 14;
...
}

```

Os parâmetros na linha de comando são analisados sintaticamente na primeira linha executável do FitNesse. O valor padrão deles é especificado no topo da classe Argument. Não é preciso sair procurando nos níveis mais baixos do sistema por instruções como a seguinte:

```
if (arguments.port == 0) // use 80 por padrao
```

As constantes de configuração ficam em um nível muito alto e são fáceis de alterar. Elas são passadas abaixo para o resto do aplicativo. Os níveis inferiores não detêm os valores de tais constantes.

## G36: Evite a navegação transitiva

De modo geral, não queremos que um único módulo saiba muito sobre seus colaboradores. Mais especificamente, se A colabora com B e B colabora com C, não queremos que os módulos que usem A enxerguem C. (Por exemplo, não queremos a.getB().getC().doSomething();.) Isso às vezes se chama de Lei de Demeter. Os programadores pragmáticos chamam de “criar um código tímido”<sup>12</sup>. Em ambos os casos, resume-se a se garantir que os módulos enxerguem apenas seus colaboradores imediatos, e não o mapa de navegação de todo o sistema.

Se muitos módulos usam algum tipo de instrução a.getB().getC(), então seria difícil alterar o projeto e a arquitetura para introduzir um Q entre B e C. Seria preciso encontrar cada instância de a.getB().getC() e converter para a.getB().getQ().getC().

É assim que as estruturas se tornam rígidas. Módulos em excesso enxergam demais sobre a arquitetura.

Em vez disso, queremos que nossos colaboradores imediatos ofereçam todos os serviços de que precisamos. Não devemos ter de percorrer a planta do sistema em busca do método que desejamos chamar, mas simplesmente ser capaz de dizer:

```
meuColaborador.facaAlgo()
```

## Java

### J1: Evite longas listas de importação usando wildcards (caracteres curinga)

Se você usa duas ou mais classes de um pacote, então importe o pacote todo usando

```
import package.*;
```

Listas longas de import intimidam o leitor. Não queremos amontoar o topo de nossos módulos com 80 linhas de import. Em vez disso, desejamos que os import sejam uma instrução concisa

sobre os pacotes que usamos.

Importações específicas são dependências fixas, enquanto importações com caracteres curingas não são. Se você não importar uma classe especificamente, então ela deve existir. Mas se você importa um pacote com um caractere curinga (wildcards), uma determinada classe não precisa existir. A instrução import simplesmente adiciona o pacote ao caminho de busca na procura por nomes. Portanto, os import não criam uma dependência real, e, portanto, servem para manter os módulos menos acoplados.

Há vezes nas quais a longa lista de import específicos pode ser útil. Por exemplo, se estiver lidando com código legado e deseja descobrir para quais classes você precisa para construir stubs (objetos que criam simulam um ambiente real para fins de testes) ou mocks (objetos similares que também analisam chamadas feitas a eles e avaliam a precisão delas), percorrer pela lista e encontrar os nomes verdadeiros de todas aquelas classes e, então, criar os stubs adequados. Entretanto, esse uso de import específicos é muito raro. Ademais, a maioria das IDEs modernas lhe permitem converter instruções import com caracteres curinga em um único comando. Portanto, mesmo no caso do código legado, é melhor importar usando caracteres curinga.

Import com caracteres curinga pode às vezes causar conflitos de nomes e ambiguidades. Duas classes com o mesmo nome, mas em pacotes diferentes, precisam ser importadas especificamente, ou pelo menos limitada especificamente quando usada. Isso pode ser chato, mas é raro o bastante para que o uso de import com caracteres curinga ainda seja melhor do que importações específicas.

## J2: Não herde as constantes

Já vi isso várias vezes e sempre faço uma careta quando vejo. Um programador coloca algumas constantes numa interface e, então, ganha acesso a elas herdando daquela interface. Observe o código seguinte:

```
public class HourlyEmployee extends Employee {  
    private int tenthsWorked;  
    private double hourlyRate;  
    public Money calculatePay() {  
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_  
WEEK);  
        int overtime = tenthsWorked - straightTime;  
        return new Money(  
            hourlyRate * (tenthsWorked + OVERTIME_RATE *  
overTime)  
        );  
    }  
    ...  
}
```

De onde vieram as constantes TENTHS\_PER\_WEEK e OVERTIME\_RATE? Devem ter vindo da classe Employee; então, vamos dar uma olhada:

```
public abstract class Employee implements PayrollConstants {  
    public abstract boolean isPayday();
```

```

    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}

```

Não, não está lá. Mas onde então? Leia atentamente a classe Employee. Ela implementa PayrollConstants.

```

public interface PayrollConstants {
    public static final int TENTHS_PER_WEEK = 400;
    public static final double OVERTIME_RATE = 1.5;
}

```

Essa é uma prática horrível! As constantes estão escondidas no topo da hierarquia de herança. Eca! Não use a herança como um meio para burlar as regras de escopo da linguagem. Em vez disso, use um import estático.

```

import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    private double hourlyRate;
    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_
WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            hourlyRate * (tenthsWorked + OVERTIME_RATE *
overTime)
        );
    }
    ...
}

```

### J3: Constantes versus enum

Agora que os enum foram adicionados à linguagem (Java 5), use-os! Não fique usando o truque antigo de public static final int. Pode-se perder o significado dos int, mas não dos enum, pois eles pertencem a uma enumeração que possui nomes.

Além do mais, estude cuidadosamente a sintaxe para os enum. Eles podem ter métodos e campos, o que os torna ferramentas muito poderosas que permitem muito mais expressividade e flexibilidade do que os int. Considere a variação abaixo do código da folha de pagamento (payroll).

```

public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    HourlyPayGrade grade;

    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_
WEEK);
        int overTime = tenthsWorked - straightTime;

```

```
        return new Money(
            grade.rate() * (tenthsWorked + OVERTIME_RATE *
overTime)
        );
    }
    ...
}

public enum HourlyPayGrade {
    APPRENTICE {
        public double rate() {
            return 1.0;
        }
    },
    LEUTENANT_JOURNEYMAN {
        public double rate() {
            return 1.2;
        }
    },
    JOURNEYMAN {
        public double rate() {
            return 1.5;
        }
    },
    MASTER {
        public double rate() {
            return 2.0;
        }
    };
}

public abstract double rate();
}
```

## Nomes

### N1: *Escolha nomes descritivos*

Não se apresse ao escolher um nome. Certifique-se de que ele seja descritivo. Lembre-se de que os sentidos tendem a se perder conforme o software evolui. Portanto, reavalie frequentemente a adequação dos nomes que você escolher.

Essa não é apenas uma recomendação para lhe “satisfazer”. Nomes em softwares são 90% responsáveis pela legibilidade do software. Você precisa tomar seu tempo para escolhê-los sabiamente e mantê-los relevantes. Nomes são muito importantes para serem tratados de qualquer jeito.

Considere o código abaixo. O que ele faz? Se eu lhe mostrasse o código com nomes bem escolhidos, ele faria sentido para você, mas como está abaixo, é apenas um emaranhado de símbolos e números mágicos.

```
public int x() {
    int q = 0;
    int z = 0;
```

```

for (int kk = 0; kk < 10; kk++) {
    if (l[z] == 10)
    {
        q += 10 + (l[z + 1] + l[z + 2]);
        z += 1;
    }
    else if (l[z] + l[z + 1] == 10)
    {
        q += 10 + l[z + 2];
        z += 2;
    } else {
        q += l[z] + l[z + 1];
        z += 2;
    }
}
return q;
}

```

Aqui está o código da forma como deveria ser. Na verdade, esse pedaço está menos completo do que o acima. Mesmo assim você pode entender imediatamente o que ele tenta fazer, e muito provavelmente até mesmo criar as funções que faltam baseando-se no que você compreendeu. Os números não são mais mágicos, e a estrutura do algoritmo está claramente descritiva.

```

public int score() {
    int score = 0;
    int frame = 0;
    for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
        if (isStrike(frame)) {
            score += 10 + nextTwoBallsForStrike(frame);
            frame += 1;
        } else if (isSpare(frame)) {
            score += 10 + nextBallForSpare(frame);
            frame += 2;
        } else {
            score += twoBallsInFrame(frame);
            frame += 2;
        }
    }
    return score;
}

```

O poder de selecionar cuidadosamente os nomes é que eles enchem a estrutura do código com descrições. Isso faz com que os leitores saibam o que esperar das outras funções no módulo. Só de olhar o código acima, você pode inferir a implementação de `isStrike()`. Quando você ler o método `isStrike`, ele será “basicamente o que você esperava”<sup>13</sup>.

```

private boolean isStrike(int frame) {
    return rolls[frame] == 10;
}

```

## N2: Escolha nomes no nível apropriado de abstração

Não escolha nomes que indiquem a implementação, mas nomes que refletem o nível de da classe ou função na qual você está trabalhando. Essa tarefa é árdua. Novamente, as pessoas são muito boas em misturar níveis de . Cada vez que você analisa seu código, provavelmente encontrará alguma variável nomeada baseando-se em um nível muito baixo. Você deve aproveitar a chance e trocar aqueles nomes quando os encontrar. Tornar o código legível requer dedicação a um aperfeiçoamento constante. Considere a interface Modem abaixo:

```
public interface Modem {
    boolean dial(String phoneNumber);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedPhoneNumber();
}
```

À primeira vista, tudo parece bem. As funções parecem adequadas. Na verdade, para muitos aplicativos elas são. Mas, agora, considere um aplicativo com alguns modens que não se conectem por discagem. Em vez disso, eles ficam sempre conectados juntos por meio de fios (pense em modens a cabo que oferecem acesso à Internet à maioria das casas atualmente). Talvez alguns se conectem enviando o número de uma porta para um switch em uma conexão USB. Obviamente, a noção de números de telefones está no nível errado de . Uma melhor estratégia de nomenclatura para este cenário seria:

```
public interface Modem {
    boolean connect(String connectionLocator);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedLocator();
}
```

Agora, os nomes não se restringem aos números de telefones. Eles ainda podem ser desse tipo, mas também podem usar outro tipo de conexão.

## N3: Use uma nomenclatura padrão onde for possível

Nomes são mais fáceis de entender se baseados numa convenção ou uso já existente. Por exemplo, se estiver usando o padrão DECORATOR, você deve usar a palavra DECORATOR nos nomes das classes que o usam. Por exemplo, AutoHangupModemDecorator deve ser o nome de uma classe que “decora” um Modem com a capacidade de desligar automaticamente ao fim da sessão.

Os padrões são apenas um tipo de padrão. Em Java, por exemplo, as funções que convertem objetos em representações de string costumam se chamar `toString`. É melhor seguir convenções como essas do que inventar a sua própria.

Equipes frequentemente inventarão seus próprios sistemas de padrões de nomes para um projeto em particular.

Eric Evans se refere a isso como uma linguagem generalizada para o projeto<sup>14</sup>. Seu código deve usar abundantemente os termos a partir dessa linguagem. Em suma, quanto mais você puder usar nomes que indiquem significados especiais relevantes ao seu projeto, mais fácil ficará para os leitores saberem sobre o que se trata seu código.

## N4: *Nomes não ambíguos*

Escolha nomes que não deixem as tarefas de uma função ou variável ambíguas. Considere o exemplo seguinte do FitNesse:

```
private String doRename() throws Exception
{
    if(refactorReferences)
        renameReferences();
    renamePage();
    pathToRename.removeNameFromEnd();
    pathToRename.addNameToEnd(newName);
    return PathParser.render(pathToRename);
}
```

O nome dessa função não diz o que a função faz, exceto em termos vagos. Isso foi enfatizado pelo fato de que há uma função chamada renamePage dentro da função chamada doRename! O que os nomes lhe dizem sobre a diferença entre as duas funções? Nada.

Um nome melhor para aquela função seria renamePageAndOptionallyAllReferences. Pode parecer longo, e é, mas ela só é chamada a partir de um local no módulo, portanto é um valor descritivo que compensa o comprimento.

## N5: *Use nomes longos para escopos grandes*

O comprimento de um nome deve estar relacionado com o do escopo. Você pode usar nomes de variáveis muito curtos para escopos minúsculos. Mas para escopos grandes, devem-se usar nomes extensos.

Nomes de variáveis como i e j são bons se seu escopo tiver cinco linhas apenas. Considere o pedaço de código abaixo do antigo padrão “Bowling Game” (jogo de boliche).

```
private void rollMany(int n, int pins)
{
    for (int i=0; i<n; i++)
        g.roll(pins);
}
```

Está perfeitamente claro e ficaria ofuscado se a variável i fosse substituída por algo importuno, como rollCount. Por outro lado, variáveis e funções com nomes curtos perdem seus sentidos em distâncias longas. Portanto, quanto maior o escopo do nome, maior e mais preciso o nome deve ser.

## N6: *Evite codificações*

Não se devem codificar nomes com informações sobre o tipo ou o escopo. Prefixos, como `m_` ou `f`, são inúteis nos ambientes de hoje em dia. E codificações em projetos e/ou subsistemas, como `siv_` (para sistema de imagem visual), são redundantes e distrativos. Novamente, os ambientes atuais fornecem todas essas informações sem precisar distorcer os nomes. Mantenha seus nomes livres da poluição húngara.

## **N7: Nomes devem descrever os efeitos colaterais**

Nomes devem descrever tudo o que uma função, variável ou classe é ou faz. Não oculte os efeitos colaterais com um nome. Não use um simples verbo para descrever uma função que faça mais do que uma mera ação. Por exemplo, considere o código abaixo do TestNG:

```
public ObjectOutputStream getOos() throws IOException {  
    if (m_oos == null) {  
        m_oos = new ObjectOutputStream(m_socket.getOutputStream());  
    }  
    return m_oos;  
}
```

Essa função faz um pouco mais além de pegar um “oos”; ela cria o “oos” se ele não tiver sido criado ainda. Logo, um nome melhor seria `createOrReturnOos`.

# Testes

## **T1: Testes insuficientes**

Uma coleção de testes deve ter quantos testes? Infelizmente, a medida que muitos programadores usam é “Parece que já está bom”. Uma coleção de testes deve testar tudo que pode vir a falhar. Os testes são insuficientes enquanto houver condições que não tenham sido exploradas pelos testes ou cálculos que não tenham sido validados.

## **T2: Use uma ferramenta de cobertura!**

Ferramentas de cobertura informam lacunas em sua estratégia de testes. Elas facilitam o encontro de módulos, classes e funções que são testados de modo insuficiente. A maioria das IDEs lhe dão uma indicação visual, marcando com verde as linhas que são cobertas pelos testes e de vermelho as que não são. Isso agiliza e facilita encontrar instruções `if` e `catch` cujos corpos não foram verificados.

## **T3: Não pule testes triviais**

Eles são fáceis de criar e seu valor de documentação é maior do que o custo de produzi-los.

## **T4: Um teste ignorado é uma questão sobre uma ambiguidade**

Nós, às vezes, não estamos certos sobre um detalhe de comportamento devido à falta de clareza

dos requisitos. Podemos expressar nossa questão sobre os requisitos como um teste que é posto como comentário que é anotado com um `@Ignore`. O que você escolher dependerá se a ambiguidade é sobre algo que deva ou não compilar.

## T5: *Teste as condições de limites*

Dê atenção especial aos testes de condições de limites. Geralmente, entendemos a parte central de um algoritmo, mas erramos sobre seus limites.

## T6: *Teste abundantemente bugs próximos*

Bugs tendem a se reunir. Quando encontrar um bug numa função, é sábio fazer um teste exaustivo nela. Provavelmente você verá que o bug não estava só.

## T7: *Padrões de falhas são reveladores*

De vez em quando, você pode diagnosticar um problema ao encontrar padrões na forma pela qual os casos de teste falharam.

Esse é outro argumento para tornar os casos de teste os mais completos possíveis. Eles, quando ordenados de uma maneira lógica, expõem os padrões.

Como um exemplo simples, suponha que você percebeu que todos os testes com uma entrada maior do que cinco caracteres tenham falhado? Ou e se um teste que passasse um número negativo para o segundo parâmetro de uma função falhasse? Às vezes, apenas ver o padrão de linhas vermelhas e verdes no relatório dos testes é o suficiente para soltar um “Arrá!” que leva à solução. Volte à página 267 para ver um exemplo interessante sobre isso no exemplo do `SerialDate`.

## T8: *Padrões de cobertura de testes podem ser reveladores*

Analizar o código que é ou não executado pelos testes efetuados dá dicas do porquê de os testes que falharam estão falhando.

## T9: *Testes devem ser rápidos*

Um teste lento é um que não será rodado. Quando as coisas ficam apertadas, são os testes lentos que serão descartados da coleção. Portanto, faça o que puder para manter seus testes rápidos.

# Conclusão

Mal poderíamos dizer que esta lista de heurísticas e odores esteja completa. De fato, não estou certo se ela jamais estará. Mas, talvez, a plenitude não deva ser o objetivo, pois o que a lista realmente faz é dar um valor ao sistema.

Na verdade, o sistema de valores tem sido o objetivo, e o assunto deste livro. Não se cria um código limpo seguindo uma série de regras. Você não se torna um especialista na arte de softwares

que é o que fazemos quando criamos um novo processo para executar o comando. No entanto, se seu script não estiver no caminho de trabalho, ou se o nome do script não estiver correto, o resultado será o mesmo.

---

# Apêndice A

---

## Concorrência II

por Brett L. Schuchert

Este apêndice é uma extensão do capítulo Concorrência da página 177. E foi escrito com uma série de tópicos independentes e, possivelmente, você poderá lê-lo em qualquer ordem devido a alguns assuntos repetidos entre as seções.

### Exemplo de cliente/servidor

Imagine um aplicativo do tipo cliente/servidor. Um servidor fica escutando um socket à espera que um cliente se conecte. Este se conecta e envia um pedido.

### O servidor

A seguir está uma versão simplificada de um aplicativo servidor. O código completo para este exemplo começa na página 343, Cliente/sevidor sem threads.

```
ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Este simples aplicativo espera por uma conexão, processa uma mensagem que chega e, então, espera novamente pelo pedido do próximo cliente. Abaixo está o código do cliente que se conecta a esse servidor:

```
private void connectSendReceive(int i) {  
    try {  
        Socket socket = new Socket("localhost", PORT);  
        MessageUtils.sendMessage(socket, Integer.toString(i));  
        MessageUtils.getMessage(socket);  
        socket.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Qual o nível de desempenho entre esse cliente/servidor? Podemos descrevê-lo formalmente? A seguir está um teste que confirma se o desempenho é “aceitável”:

```
@Test(timeout = 10000)  
  
public void shouldRunInUnder10Seconds() throws Exception {  
    Thread[] threads = createThreads();  
    startAllThreads(threads);  
    waitForAllThreadsToFinish(threads);  
}
```

A fim de manter o exemplo simples, deixamos configuração de fora (veja o ClienteTest.java, na página 344). Esse teste confirma se ele deve completar dentro de 10.000 milissegundos.

Esse é um exemplo clássico de validação da taxa de transferência de dados de um sistema. Esse deve completar uma série de pedidos do cliente em dez segundos. Enquanto o servidor puder processar cada pedido individualmente a tempo, o teste passará.

O que acontece se ele falhar? O limite de desenvolver um tipo de loop de transmissão por solicitação é que não há muito o que fazer dentro de uma única thread para agilizar este código. Usar múltiplas threads resolverá o problema? Talvez, mas precisamos saber onde está sendo desperdiçado tempo. Há duas possibilidades:

- E/S—usar um socket, conectar-se a um banco de dados, esperar pela troca com a memória virtual e assim por diante.
- Processador—cálculos numéricos, processamento de expressões regulares, coleta de lixo e assim por diante.

Os sistemas tipicamente possuem um pouco de cada, mas para uma dada operação tende a se usar apenas uma das duas. Se o código for baseado no processador, mais hardwares de processamento podem melhorar a taxa de transferência de dados, fazendo nossos testes passarem. Mas só que há tantos ciclos de CPU disponíveis de modo que adicionar threads a um problema baseando-se o processador não o tornará mais rápido. Por outro lado, se o processo for baseado em E/S, então

a concorrência pode aumentar com eficiência.

Quando uma parte do sistema está esperando por uma E/S, outra parte pode usar esse tempo de espera para processar outra coisa, tornando o uso da CPU disponível mais eficiente.

## Adição de threads

Agora vamos prever que o teste de desempenho falhe. Como podemos melhorar a taxa de transferência de dados de modo que ele passe? Se o método do processo do servidor for baseado em E/S, então há uma maneira de fazer o servidor usar threads (apenas mude a processMessage):

```
void process(final Socket socket) {
    if (socket == null)
        return;
    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.
getMessage(socket);
                MessageUtils.sendMessage(socket, "Processed:
" + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Assuma que essa mudança faça o teste passar<sup>1</sup>; o código está completo, certo?

## Observações do servidor

O servidor atualizado completa o teste com êxito em um segundo. Infelizmente, essa solução é um pouco ingênua e adiciona alguns problemas novos.

Quantas threads nosso servidor poderia criar? O código não estabelece limites, portanto poderíamos normalmente alcançar o limite imposto pela Java Virtual Machine (JVM). Para muitos sistemas simples, isso talvez satisfaça. Mas e se o sistema suportar muitos usuários numa rede pública? Se muitos se conectarem ao mesmo tempo, o sistema pode travar.

Mas deixe de lado o problema de comportamento por agora. A solução apresentada possui problemas de limpeza e estrutura. Quantas responsabilidades o código do servidor pode ter?

- Gerenciamento de conexão com o socket
- Processamento do cliente
- Diretrizes para uso de threads
- Diretrizes para desligamento do servidor

Infelizmente, todas essas responsabilidades ficam na função `process`. Ademais, o código mistura tantos níveis diferentes de. Sendo assim, por menor que esteja a função, ela precisa ser dividida.

O servidor possui diversas razões para ser alterado; entretanto, isso violaria o Princípio da Responsabilidade Única. Para manter limpos sistemas concorrentes, o gerenciamento de threads deve ser mantido em poucos locais bem controlados. Além do mais, qualquer código que gerencie threads só deva fazer essa tarefa. Por quê? Só pelo fato de que rastrear questões de concorrência é árduo o bastante sem ter de lidar ao mesmo tempo com outras questões não relacionadas à concorrência.

Se criarmos uma classe separada para cada responsabilidade listada acima, incluindo uma para o gerenciamento de threads, então quando mudarmos a estratégia para tal gerenciamento, a alteração afetará menos o código geral e não poluirá as outras responsabilidades. Isso também facilita muito testar todas as outras responsabilidades sem ter de se preocupar com o uso de threads. Abaixo está uma versão atualizada que faz justamente isso:

```
public void run() {
    while (keepProcessing) {
        try {
            ClientConnection clientConnection
                = connectionManager.awaitClient();
            ClientRequestProcessor requestProcessor
                = new ClientRequestProcessor(clientConnectio
n);
            clientScheduler.schedule(requestProcessor);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    connectionManager.shutdown();
}
```

Agora, se houver problemas de concorrência, só haverá um lugar para olhar, pois tudo relacionado a threads está num único local, em `clientScheduler`.

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

É fácil implementar a diretriz atual:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor
requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

Ter isolado todo o gerenciamento de threads num único lugar facilitou bastante a alteração do modo como controlamos as threads. Por exemplo, mover o framework Executor do Java 5 envolve criar uma nova classe e inseri-la no código (Listagem A.1).

## Conclusão

Apresentar concorrência neste exemplo em particular demonstra uma forma de melhorar a taxa de transferência de dados de um sistema e de validar aquela taxa por meio de um framework de teste. Centralizar todo o código de concorrência em um pequeno número de classes é um exemplo da aplicação do Princípio da Responsabilidade Única. Neste caso de programação concorrente, isso se torna especialmente importante devido à sua complexidade.

## Caminhos possíveis de execução

Revise o método `incrementValue` – método em Java de apenas uma linha sem iteração ou ramificação.

```
public class IdGenerator {  
    int lastIdUsed;  
    public int incrementValue() {  
        return ++lastIdUsed;  
    }  
}
```

Ignore o excesso de inteiros e assuma que apenas uma thread possua acesso a uma única instância de `IdGenerator`. Neste caso, só há um caminho de execução e um resultado garantido:

- O valor retornado é igual ao valor de `lastIdUsed`, e ambos estão uma unidade maior do que estavam antes de chamar o método.

### Listagem A-1

#### **ExecutorClientScheduler.java**

```
import java.util.concurrent.Executor;  
import java.util.concurrent.Executors;  
  
public class ExecutorClientScheduler implements ClientScheduler {  
    Executor executor;  
  
    public ExecutorClientScheduler(int availableThreads) {  
        executor = Executors.newFixedThreadPool(availableThreads);  
    }  
  
    public void schedule(final ClientRequestProcessor requestProcessor) {  
        Runnable runnable = new Runnable() {  
            public void run() {  
                requestProcessor.process();  
            }  
        };  
        executor.execute(runnable);  
    }  
}
```

O que acontece se usarmos duas threads e deixar o método inalterado? Quais os possíveis resultados se cada thread chamar `incrementValue` uma vez? Haverá quantos caminhos de execução possíveis? Primeiro, os resultados (assuma que `lastIdUsed` comece com o valor 93):

- Thread 1 recebe 94, thread 2 recebe 95 e `lastIdUsed` agora é 95.
- Thread 1 recebe 95, thread 2 recebe 94 e `lastIdUsed` agora é 95.
- Thread 1 recebe 94, thread 2 recebe 94 e `lastIdUsed` agora é 94.

O resultado final, mesmo que surpreendente, é possível. A fim de vermos como, precisamos entender a quantidade de caminhos possíveis de execução e como a Java Virtual Machine os executa.

## Quantidade de caminhos

Para cálculo do número de caminhos possíveis de execução, começaremos com o Bytecode gerado. A única linha em Java (`return ++lastIdUsed;`) retorna oito instruções em bytecode. É possível para as duas threads intercalarem a execução das oito instruções do modo como um embaralhador de cartas faz na hora de embaralhar<sup>2</sup>. Mesmo com apenas oito cartas em cada mão, há uma quantidade considerável de resultados distintos.

Para este caso simples de  $N$  instruções numa sequência, sem loop ou condicionais e sem  $T$  threads, a quantidade total de caminhos possíveis de execução é igual a

$$\frac{(NT)!}{NT}$$

### Como calcular as possíveis combinações

O trecho a seguir é de um e-mail do Uncle Bob para Brett:

Com  $N$  passos e  $T$  threads, há  $T^N$  passos no total. Antes de cada um, há uma troca de contexto que seleciona entre as  $T$  threads. Pode-se então representar cada caminho como uma string de dígitos denotando as trocas de contexto.

Dados os passos A e B e as threads 1 e 2, os seis caminhos possíveis 1122, 1212, 1221, 2112, 2121 e 2211. Ou, em termos de passos, é A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 e A2B2A1B1. Para três threads, a sequência é 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123...

Uma característica dessas strings é que deve sempre existir  $N$  instâncias de cada  $T$ . Portanto, a string 111111 é inválida, pois possui seis instâncias de 1 e nenhuma de 2 e 3.

Sendo assim, queremos que as permutações de  $N$  1's,  $N$  2's,... e  $N$  T's. Na verdade, essas são apenas as permutações de coisas  $N * T$  retiradas de uma de cada vez de  $N * T$ , que é  $(N * T)!$ , mas com todas as duplicações removidas. Portanto, o truque é contar as duplicações e subtrair de  $(N * T)!$ .

Dado dois passos e duas threads, há quantas duplicações? Cada string de quatro dígitos possui dois 1 e dois 2. Poderiam alternar cada par desses sem alterar o sentido da string. Você poderia trocar os 1 ou os 2, ambos ou nenhum. Portanto, há quatro isomorfos para cada string, isto é, há três duplicações. Sendo assim, três em cada quatro das opções são duplicações; por outro lado, uma das quatro permutações NÃO são duplicações.  $4! * .25 = 6$ . Dessa forma, este raciocínio parece funcionar.

Há quantas duplicações? No caso, em que  $N = 2$  e  $T = 2$ , eu poderia trocar os 1, os 2 ou ambos. Se  $N = 2$  e  $T = 3$ , eu poderia trocar os 1, os 2, os 3, os 1 e os 2, os 1 e os 3 ou os 2 e os 3. Trocar são apenas as permutações de  $N$ . Digamos que haja  $P$  permutações de  $N$ . A quantidade de formas diferentes para ordená-las seria  $P^{**}T$ .

Sendo assim, o número de isomorfos possíveis é de  $N!^{**}T$ . E, portanto, a quantidade de caminhos é de  $(T^N)!/(N!^{**}T)$ . Novamente, em nosso caso  $T = 2$  e  $N = 2$  obtemos 6 ( $24/4$ ).

Para  $N = 2$  e  $T = 3$ , obtemos  $720/8 = 90$ .

Para  $N = 3$  e  $T = 3$  obtemos  $9!/6^3 = 1680$ .

Para nosso simples caso de uma linha de código Java, que equivale a oito linhas de bytecode e duas threads, o número total de caminhos possíveis de execução é de 12.870. Se `lastIdUsed` for do tipo `long`, então cada leitura/escrita se torna duas operações em vez de uma, e o número possível de combinações se torna 2.704.156.

O que acontece se fizermos uma alteração neste método?

```
public synchronized void incrementValue() {
    ++lastIdUsed;
}
```

A quantidade de caminhos possíveis de execução se torna dois para duas threads e  $N!$  no caso geral.

## Indo mais a fundo

E o resultado surpreendente de que duas threads podiam ambas chamar o método uma vez (antes de adicionarmos `synchronized`) e obter o mesmo resultado numérico? Como isso pode ser possível? Calma, uma coisa de cada vez!

O que é uma operação atômica? Podemos definir uma operação atômica como qualquer operação que seja interrompível. Por exemplo, no código seguinte, linha 5, em que `lastId` recebe 0, é considerada uma operação atômica, pois segundo o modelo Java Memory, a atribuição de 0 a um valor de 32 bits é interrompível.

```
01: public class Example {
02:     int lastId;
03:
04:     public void resetId() {
05:         value = 0;
```

```

06: }
07:
08: public int getNextId() {
09:     ++value;
10: }
11:}

```

O que acontece se mudarmos o tipo de `lastId` de `int` para `long`? A linha 5 ainda será atômica? Não de acordo com a especificação da JVM. Poderia ser atômica em um processador em particular, mas segundo a especificação da JVM, a atribuição a qualquer valor de 64 bits requer duas de 32 bits. Isso significa que entre a primeira atribuição de 32 bits e a segunda também de 32 bits, alguma outra thread poderia se infiltrar e alterar um dos valores.

E o operador de pré-incremento, `++`, na linha 9? Ele pode ser interrompido, logo, ele não é atômico. Para entender, vamos revisar detalhadamente o bytecode de ambos os métodos. Antes de prosseguirmos, abaixo há três definições que serão importantes:

- **Quadro (frame)**—toda chamada de método querer um quadro, que inclui o endereço de retorno, qualquer parâmetro passado ao método e as variáveis locais definidas no método. Essa é uma técnica padrão usada para declarar uma pilha de chamadas usada pelas linguagens atuais, permitindo chamadas recursivas e funções/métodos básicos.
- **Variável local**—qualquer variável declarada no escopo do método. Todos os métodos não estáticos têm pelo menos uma variável `this`, que representa o objeto em questão – aquele que recebeu a mensagem mais recente (na thread em operação) – que gerou a chamada do método.
- **Pilha de operadores**—muitas das instruções na JVM recebem parâmetros, que ficam armazenados na pilha de operadores. A pilha é uma estrutura de dados do tipo LIFO (Last In, First Out, ou último a entrar, primeiro a sair).

A seguir está o bytecode gerado para `resetId()`:

Mnemônica	Descrição	Pilha de operadores depois
<code>ALOAD 0</code>	Coloca a <code>0o</code> variável na pilha de operadores. O que é a <code>0o</code> variável? É a <code>this</code> , o objeto atual. Quando o método é invocado, o receptor da mensagem – uma instância de <code>Example</code> – foi inserido no array de variáveis locais do quadro (frame) criado para a chamada do método. Essa é sempre a primeira variável colocada em cada instância do método.	<code>This</code>
<code>ICONST_0</code>	Coloca o valor constante 0 na pilha de operadores. <code>this, 0</code>	<code>this, 0</code>

PUTFIELD lastId	Armazena o valor no topo da pilha (que é 0) no valor do campo do objeto referido por referência do objeto um a partir do topo da pilha, this.	<empty>
--------------------	---	---------

Essas três instruções são certamente atômicas, pois, embora a thread que as executa pudesse ser interrompida após qualquer uma delas, outras threads não poderão tocar nas informações para a instrução PUTFIELD (o valor constante 0 no topo da pilha e a referência para este um abaixo do topo, juntamente com o valor do campo).

Portanto, quando ocorrer a atribuição, garantimos que o valor 0 seja armazenado no valor do campo. A operação é atômica. Todos os operadores lidam com informações locais ao método, logo não há interferência entre múltiplas threads.

Sendo assim, se essas três instruções forem executadas por dez threads, haverá  $4.38679733629e+24$  combinações. Mas como só existe um resultado possível as diferentes combinações são irrelevantes. Neste caso, acontece que o mesmo resultado é certo para o tipo long também. Por quê? Todas as dez threads atribuem um valor constante. Mesmo se intercalassem entre si, o resultado final seria o mesmo.

Com a operação ++ no método `getNextId`, haverá problemas. Suponha que `lastId` seja 42 no início deste método. A seguir está o bytecode gerado para este novo método:

Mnemônica	Descrição	Pilha de operadores depois
ALOAD 0	Coloca this na pilha de operadores	this
DUP	Copia o topo da pilha. Agora temos duas cópias de this na pilha de operadores.	this, this
GETFIELD lastId	Recupera o valor de <code>lastId</code> a partir do objeto apontado no topo da pilha (this) e armazena tal valor novamente na pilha.	this, 42
ICONST_1	Insere a constante inteira 1 na pilha.	this, 42, 1
IADD	Adiciona os dois valores inteiros no topo da pilha de operadores e armazena o resultado de volta na pilha de operadores.	this, 43
DUP_X1	Duplica o valor 43 e o coloca antes de this.	43, this, 43
PUTFIELD value	Armazena o valor do topo da pilha de operadores, 43, no valor do campo do objeto atual, representado pelo valor consecutivo ao topo na pilha de operadores, this.	43

IRETURN	retorna apenas o valor do topo da pilha.	<empty>
---------	--	---------

Imagine o caso no qual a primeira thread finaliza as três primeiras instruções, chega até GETFIELD e o adiciona e, então, é interrompida. Uma segunda thread assume o controle e executa todo o método, incrementando `lastId` em 1; ela recebe 43 de volta. Então, a primeira thread continua de onde havia parado; 42 ainda está na pilha de operadores, pois esse era o valor de `lastId` quando ele executou a GETFIELD. Ele adiciona 1 para obter 43 novamente e armazena o resultado. O valor 43 é retornado à primeira thread também. O resultado é que aquele 1 do incremento se perde, pois a primeira thread passou por cima da segunda depois desta ter interrompido aquela. Sincronizar o método `getNextId()` resolve esse problema.

## Conclusão

Não é necessário saber muito sobre bytecode para entender como as threads podem passar uma por cima da outra. Se puder entender este exemplo, ele demonstra a possibilidade de múltiplas threads passando uma sobre a outra, o que já é conhecimento suficiente.

Dito isso, o que esse simples exemplo mostra é a necessidade de entender o modelo de memória suficientemente para saber o que é e o que não é seguro. É comum pensarem erroneamente que o operador `++` (seja de pré ou pós-incremento) seja atômico, pois ele obviamente não é. Isso significa que você precisa saber:

- Onde há valores/objetos compartilhados
- Qual código pode causar questões de leitura/atualização concorrentes
- Como evitar ocorrência de tais questões de concorrência

## Conheça sua biblioteca

### Framework Executor

Como mostramos no `ExecutorClientScheduler.java` (p. 321), o framework Executor surgido no Java 5 permite uma execução sofisticada usando-se uma coleção de threads. Essa classe está no pacote `java.util.concurrent`.

Se estiver criando threads, e não usando uma coleção delas, ou estiver usando uma criada manualmente, considere usar o `Executor`. Ele tornará seu código mais limpo, fácil de acompanhar e menor.

O framework Executor unirá threads, mudará automaticamente o tamanho e recriará threads se necessário. Ele também suporta futures, uma construção comum de programação concorrente. Este framework trabalha com classes que implementam a `Runnable` e também com classes que implementem a interface `Callable`. Esta se parece com uma `Runnable`, mas ela pode retornar um resultado, que é uma necessidade comum em soluções multithread.

Um future é útil quando o código precisa executar múltiplas operações independentes e esperar que ambas finalizem:

```
public String processRequest(String message) throws Exception {
```

```
Callable<String> makeExternalCall = new Callable<String>() {
    public String call() throws Exception {
        String result = "";
        // faz um pedido externo
        return result;
    }
};

Future<String> result = executorService.
submit(makeExternalCall);
String partialResult = doSomeLocalProcessing();
return result.get() + partialResult;
}
```

Neste exemplo, o método começa executando o objeto `makeExternalCall`. O método continua com outro processamento. A linha final chama `result.get()`, que o bloqueia até que o future finalize.

## Soluções sem bloqueio

A Virtual Machine do Java 5 tira proveito do projeto dos processadores modernos, que suportam atualizações sem bloqueio e confiáveis. Considere, por exemplo, uma classe que use sincronização (e, portanto, bloqueio) para proporcionar uma atualização segura para threads de um valor:

```
public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

O Java 5 possui uma série de novas classes para situações como essa: `AtomicBoolean`, `AtomicInteger` e `AtomicReference` são três exemplos; há diversas outras. Podemos reescrever o código acima para usar uma abordagem sem bloqueios, como segue:

```
public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);
    public void incrementValue() {
        value.incrementAndGet();
    }
    public int getValue() {
        return value.get();
    }
}
```

Mesmo que esteja usando um objeto em vez de um tipo primitivo e enviando mensagens, como `incrementAndGet()` em vez de `++`, o desempenho dessa classe quase sempre superará o da versão anterior. Em alguns casos, ela será só levemente mais rápida, mas casos em que ela fique mais lenta praticamente não existem.

Como isso é possível? Processadores modernos têm uma operação tipicamente chamada de Comparar e Swap (CAS, sigla em inglês). Essa operação é análoga ao bloqueio otimista de

bancos de dados, enquanto a versão sincronizada é análoga ao bloqueio pessimista.

A palavra reservada `synchronized` sempre requer um bloqueio, mesmo quando uma segunda thread não esteja tentando atualizar o mesmo valor. Mesmo que o desempenho de bloqueios intrínsecos tenha melhorado de versão para versão, eles ainda saem caro.

A versão sem bloqueio assume que múltiplas threads não costumam modificar o mesmo valor frequentemente o bastante para criar um problema. Em vez disso, ela detecta de forma eficiente se tal situação ocorreu e tenta novamente até que a atualização ocorra com sucesso.

Essa detecção quase sempre sai menos cara do que adquirir um bloqueio, mesmo em situações que vão de moderadas a de alta contenção.

Como a Virtual Machine consegue isso? A operação CAS é atômica. Logicamente, ela se parece com o seguinte:

```
int variableBeingSet;
void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
        variableBeingSet = newValue;
        return currentValue;
    }
    return variableBeingSet;
}
```

Quando um método tenta atualizar uma variável compartilhada, a operação CAS verifica se a variável sendo configurada ainda possui o último valor conhecido. Caso possua, ela é, então, alterada. Caso contrário, ela não é configurada porque outra thread está usando-a. O método tentando alterá-la (usando a operação CAS) percebe que mudança não foi feita e tenta de novo.

## Classes não seguras para threads

Há algumas classes que, por natureza, não são seguras para threads. Aqui estão alguns exemplos:

- `SimpleDateFormat`
- Conexões a banco de dados
- Contêineres em `java.util`
- Servlets

Note que algumas classes de coleção possuem métodos individuais que são seguros para threads. Entretanto, qualquer operação que envolva chamar mais de um método não é segura. Por exemplo, se você não quiser substituir algo numa `HashTable` porque ele já está lá, você poderia escrever o seguinte:

```
if(!hashTable.containsKey(someKey)) {  
    hashTable.put(someKey, new SomeValue());  
}
```

Cada método é seguro para threads. Contudo, outra thread poderia adicionar um valor entre a `containsKey` e as chamadas a `put`.

- Bloqueie primeiro a `HashTable` e certifique-se de que todos os outros usuários dela façam o mesmo – bloqueio baseando-se no cliente:

```
synchronized(map) {  
    if(!map.containsKey(key))  
        map.put(key,value);  
}
```

- Coloque a `HashTable` em seu próprio objeto e use uma API diferente – bloqueio baseando-se no servidor usando um ADAPTER:

```
public class WrappedHashtable<K, V> {  
    private Map<K, V> map = new Hashtable<K, V>();  
    public synchronized void putIfAbsent(K key, V value) {  
        if (map.containsKey(key))  
            map.put(key, value);  
    }  
}
```

- Use as coleções seguras para threads.

```
ConcurrentHashMap<Integer, String> map = new  
ConcurrentHashMap<Integer,  
String>();  
map.putIfAbsent(key, value);
```

A coleção `java.util.concurrent` possui funções como a `putIfAbsent()` para acomodar tais operações.

## Dependências entre métodos podem danificar o código concorrente

Abaixo está um exemplo simples de uma maneira de inserir dependências entre métodos:

```
public class IntegerIterator implements Iterator<Integer>  
    private Integer nextValue = 0;  
  
    public synchronized boolean hasNext() {  
        return nextValue < 100000;  
    }  
  
    public synchronized Integer next() {
```

```
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }

    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

A seguir está um código para usar este IntegerIterator:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    // faz algo com nextValue
}
```

Se uma thread executar este código, não haverá problema. Mas o que acontece se duas threads tentarem compartilhar uma única instância de IntegerIterator considerando que cada uma processará o valor que receber, mas cada elemento da lista só é processado uma vez? Na maioria das vezes, nada de ruim ocorre; as threads compartilham a lista, processam os elementos que receberem do iterator e param quando este finaliza. Entretanto, há uma pequena chance de que, no final da iteração, as duas threads interajam entre si e façam com que uma vá além do iterator e lance uma exceção.

O problema é o seguinte: A Thread 1 faz a pergunta `hasNext()`, que retorna `true`. Então ela é bloqueada e a Thread 2 faz a mesma pergunta, que ainda retorna `true`. Então, a Thread 2 chama `next()`, que retorna um valor como esperado, mas com um efeito colateral de fazer `hasNext()` retornar `false`. A Thread 1 reinicia, pensando `hasNext()` ainda é `true`, e, então, chama `next()`. Mesmo que os métodos estejam sincronizados, o cliente usa dois.

Esse é o problema real e um exemplo dos tipos que surgem no código concorrente. Neste caso em particular, este problema é consideravelmente sutil, pois a única vez em que ele ocorre é durante a iteração final do iterator.

Se acontecer das threads darem erro na hora certa, então uma delas poderia ir além do final do iterator. Esse tipo de bug que ocorre bem depois de um sistema já está em produção, e fica difícil encontrá-lo.

Você tem três opções:

- Aceite a falha.
- Resolva o problema alterando o cliente: bloqueio baseando-se no cliente.
- Resolva o problema alterando o servidor, que adiciona alterações ao cliente: bloqueio baseando-se no servidor.

## Aceite a falha

Às vezes você pode configurar as coisas de tal forma que as falhas não causam mal algum. Por exemplo, o cliente acima capture uma exceção e a ignora. Francamente, isso é um pouco desleixado. É como reiniciar à meia-noite de modo a liberar a memória.

## Bloqueio baseando-se no cliente

A fim de fazer o `IntegerIterator` funcionar corretamente com múltiplas threads, altere o cliente abaixo (e cada outro cliente), como segue:

```
IntegerIterator iterator = new IntegerIterator();
while (true) {
    int nextValue;
    synchronized (iterator) {
        if (!iterator.hasNext())
            break;
        nextValue = iterator.next();
    }
    doSomethingWith(nextValue);
}
```

Cada cliente adiciona um bloqueio via a palavra reservada `synchronized`. Essa duplicação viola o Princípio do Não Se Repita (DRY, sigla em inglês), mas talvez seja necessário, caso o código use ferramentas de terceiros não-seguras para threads.

Essa estratégia é arriscada, pois todos os programadores que usarem o servidor deverão se lembrar de bloqueá-lo antes de usá-lo e, quando terminar, desbloqueá-lo. Há muitos, muitíssimos anos atrás, trabalhei num sistema que usava num recurso compartilhado o bloqueio baseando-se no cliente. Usavam-se o recurso em centenas de lugares diferentes ao longo do código. Um pobre programador esqueceu de bloquear tal recurso em um desses lugares.

O sistema era software de contabilidade executando um sistema com compartilhamento de tempo e diversos terminais para a Local 705 do sindicato dos caminhoneiros. O computador estava num nível elevado, em uma sala de ambiente controlado a 80 km ao norte da sede da Local 705. Na sede, havia dezenas de funcionários digitando no terminal os dados dos relatórios de impostos sindicais. Os terminais estavam conectados ao computador através de linhas telefônicas exclusivas e modens semiduplex de 600 bps. (Isso foi há muito, muito tempo atrás).

Cerca de uma vez por dia, um dos terminais “travava”. Não havia motivo para isso. O “travamento” não mostrava referência para nenhum terminal ou horário em particular. Era como se alguém jogasse dados para saber a hora e o terminal a bloquear.

De vez em quando, mais de um terminal travava. Às vezes, passavam-se dias sem um travamento.

À primeira vista, a única solução era uma reinicialização. Mas isso era difícil de coordenar. Tínhamos de ligar para a sede e pedir que em todos os terminais todos finalizassem o que estivessem fazendo.

Então, poderíamos desligar e reiniciar. Se alguém estivesse fazendo algo importante que levasse

uma ou duas horas, o terminal bloqueado simplesmente teria de permanecer assim.

Após poucas semanas de depuração, descobrimos que a causa era um contador de buffer circular que havia saído da sincronia com seu ponteiro. Esse buffer controlava a saída para o terminal. O valor do ponteiro indicava que o buffer estava vazio, mas o contador dizia que estava cheio. Como ele estava vazio, não havia nada a exibir; mas como também estava cheio, não se podia adicionar nada ao buffer para que fosse exibido na tela.

Portanto, sabíamos que os terminais estavam travando, mas não o porquê do buffer circular estar saindo de sincronia. Então, adicionamos um paliativo para contornar com o problema. Era possível ler os interruptores do painel central no computador (isso foi há muito, muito, muito tempo).

Criamos uma pequena função como armadilha para quando um detector desses interruptores fosse alterado e, então, buscávamos por um buffer circular que estivesse vazio e cheio. Se encontrasse um, o buffer seria configurado para vazio. Voilá! O(s) terminal(ais) travado(s) começaria(ram) a funcionar novamente.

Portanto, agora não tínhamos de reiniciar o sistema quando um terminal travasse. A Local 705 simplesmente nos ligaria e diria que tínhamos um travamento, e, então, bastaria irmos até a sala do computador e mexer no interruptor.

É claro que, às vezes, eles da Local 605 trabalhavam nos finais de semana, mas nós não. Então, adicionamos uma função ao programador que verificava todos os buffers circulares uma vez a cada minuto e zerava os que estavam vazios e cheios ao mesmo tempo. Com isso as telas abriam antes mesmo da Local 705 pegar o telefone.

Levou mais algumas semanas de leitura minuciosa, página após página, do gigantesco código em linguagem assembly para descobrirmos o verdadeiro culpado. Havíamos calculado que a frequência dos bloqueios era consistente com um único caso desprotegido do buffer circular. Então, tudo o que tínhamos a fazer era encontrar aquele uso falho. Infelizmente, isso foi há muito tempo e não tínhamos as ferramentas de busca ou de referência cruzada ou qualquer outro tipo de ajuda automatizada.

Simplesmente tivemos de ler os códigos.

Aprendi uma lição importante naquele gélido inverno de Chicago, em 1971. O bloqueio baseando-se no cliente era realmente uma desgraça.

## Bloqueio baseando-se no servidor

Pode-se remover a duplicação através das seguintes alterações ao IntegerIterator:

```
public class IntegerIteratorServerLocked {  
    private Integer nextValue = 0;  
  
    public synchronized Integer getNextOrNull() {  
        if (nextValue < 100000)  
            return nextValue++;  
        else  
            return null;  
    }  
}
```

E o código do cliente também muda:

```
while (true) {
```

```
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // faz algo com nextValue
}
```

Neste caso, realmente alteramos a API de nossa classe para ser multithread<sup>3</sup>. O cliente precisa efetuar uma verificação de null em vez de checar hasNext().

Em geral, deve-se dar preferência ao bloqueio baseando-se no servidor, pois:

- Ele reduz códigos repetidos – o bloqueio baseando-se no cliente obriga cada cliente a bloquear o servidor adequadamente. Ao colocar o código de bloqueio no servidor, os clientes ficam livres para usar o objeto e não ter de criar códigos de bloqueio extras.
- Permite melhor desempenho – você pode trocar um servidor seguro para threads por um não-seguro, no caso de uma implementação de thread única, evitando todos os trabalhos extras.
- Reduz a possibilidade de erros – basta o programador esquecer de bloquear devidamente.
- Força a uma única diretriz – um local, o servidor, em vez de muitos lugares, cada cliente.
- Reduz o escopo das variáveis compartilhadas – o cliente não as enxerga ou não sabe como estão bloqueadas. Tudo fica escondido no servidor. Quando ocorre um erro, diminui a quantidade dos locais nos quais onde procurar.
- E se você não for o dono do código do servidor?
- Use um ADAPTER para alterar a API e adicionar o bloqueio

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();

    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- Ou, melhor ainda, use coleções seguras para threads com interfaces estendidas.

## Como aumentar a taxa de transferência de dados

Assumamos que desejamos entrar na net e ler o conteúdo de uma série de páginas de uma lista de URLs. Conforme cada página é lida, analisaremos sua sintaxe para reunir algumas estatísticas. Após ter lido todas as páginas, imprimiremos um relatório resumido.

A classe seguinte retorna o conteúdo de uma página dado um URL.

```
public class PageReader {
    //...
```

<sup>3</sup> Na verdade, a interface Iterator é, por natureza, segura para threads. Ela nunca foi projetada para ser usada por múltiplas threads, logo isso não deveria ser

```

public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);
    try {
        httpClient.executeMethod(method);
        String response = method.getResponseBodyAsString();
        return response;
    } catch (Exception e) {
        handle(e);
    } finally {
        method.releaseConnection();
    }
}
}

```

A próxima classe é o iterator que fornece o conteúdo das páginas baseando-se num iterator de URLs:

```

public class PageIterator {
    private PageReader reader;
    private URLIterator urls;

    public PageIterator(PageReader reader, URLIterator urls) {
        this.urls = urls;
        this.reader = reader;
    }

    public synchronized String getNextPageOrNull() {
        if (urls.hasNext())
            getPageFor(urls.next());
        else
            return null;
    }

    public String getPageFor(String url) {
        return reader.getPageFor(url);
    }
}

```

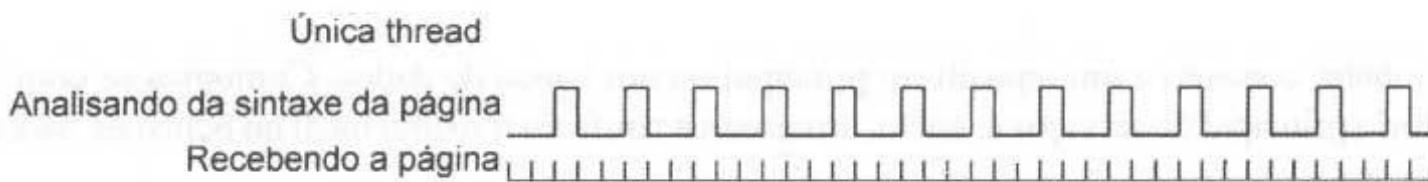
Pode-se compartilhar uma instância de PageIterator entre muitas threads, cada uma usando sua própria instância para ler e analisar a sintaxe das páginas que receber do iterator. Note que mantivemos o bloco synchronized bem pequeno. Ele contém apenas a seção crítica dentro do PageIterator. É sempre melhor sincronizar o menos possível, e não o contrário.

## Cálculo da taxa de transferência de dados com uma única thread

Agora, façamos alguns cálculos simples. Em relação aos parâmetros, assuma o seguinte:

- Tempo de E/S recupera uma página (média): 1 segundo

- Tempo de processamento para analisar a sintaxe da página (média): .5 segundos
- A E/S requer 0% do uso da CPU enquanto o processamento exige 100%.

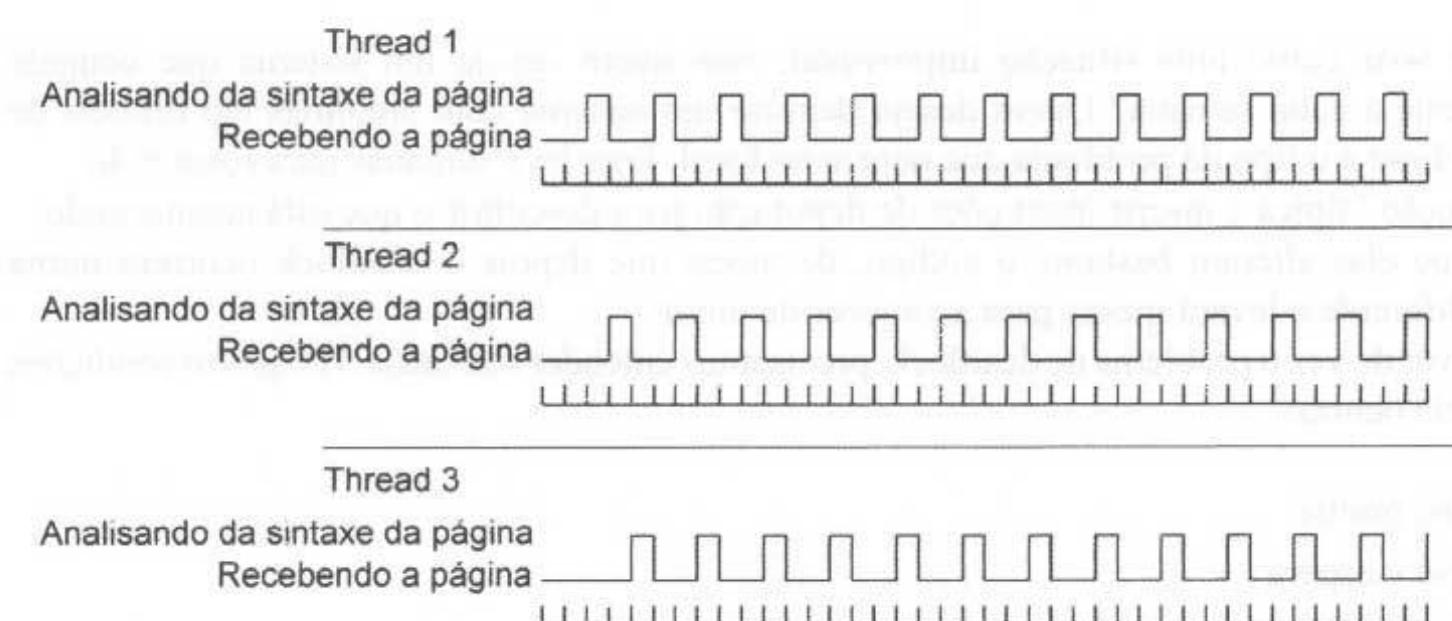


Para N páginas sendo processadas por uma única thread, o tempo total de execução é de 1,5 segundo  
 \* N. A Figura A.1 demonstra o processo com 13 páginas ou por volta de 19,5 segundos.

## Cálculo da taxa de transferência de dados com múltiplas threads

Se for possível recuperar as páginas em qualquer ordem e processá-las independentemente, então é possível usar múltiplas threads para aumentar a taxa de transferência de dados. O que acontece se usarmos múltiplas threads? Quantas páginas poderemos obter ao mesmo tempo?

Como pode ver na Figura A.2, a solução multithread permite que o processo análise da sintaxe das páginas (uso do processador) se sobreponha com a leitura delas (E/S). Num mundo ideal, isso significa que o processador está sendo totalmente utilizado. Cada leitura de um segundo por página se sobrepõe com duas análises de sintaxe. Logo, podemos processar duas páginas por segundo, o que é três vezes maior que a taxa de transferência de dados da solução com uma única thread.



**Figura A.2 - A solução multithread**

## Deadlock

Imagine um aplicativo Web com dois conjuntos de recursos compartilhados de um tamanho finito.

- Um conjunto de conexões a banco de dados para trabalho local no armazenamento do processo

- Um conjunto de conexões MQ para um repositório principal

Assuma que haja duas operações neste aplicativo, criar e atualizar:

- Criar – obtém conexão a um repositório principal ou um banco de dados. Comunica-se com o repositório principal de serviços e, então, armazena a tarefa no trabalho local no banco de dados do processo.
- Atualizar – obtém conexão a um banco de dados e, então a um repositório principal. Lê a partir da tarefa no banco de dados do processo e, então, envia ao repositório principal.

O que acontece quando o número de usuários é maior que o do conjunto de recursos? Considere que cada conjunto tenha um tamanho 10.

- Dez usuários tentam usar o “criar”, então todas as dez conexões ao banco de dados ficam ocupadas, e cada thread é interrompida após obter tal conexão, porém antes de conseguir uma com o repositório principal.
- Dez usuários tentam usar o “atualizar”, então todas as dez conexões ao repositório principal ficam ocupadas, e cada thread é interrompida após obter tal conexão, porém antes de conseguir uma com o banco de dados.
- Agora, as dez threads “criar” devem esperar para conseguir uma conexão com o repositório principal, mas as dez threads “atualizar” devem esperar para conseguir uma conexão com o banco de dados.
- Deadlock (bloqueio infinito). O sistema fica preso.

Isso pode soar como uma situação improvável, mas quem deseja um sistema que congele infinitamente a cada semana? Quem deseja depurar um sistema com sintomas tão difíceis de produzir? Esse é o tipo de problema que ocorre no local, logo leva semanas para resolvê-lo. Uma “solução” típica é inserir instruções de depuração para descobrir o que está acontecendo. É claro que elas alteram bastante o código, de modo que depois o deadlock ocorrerá numa situação diferente e levará meses para acontecer de novo<sup>4</sup>.

Para resolver de vez o problema de deadlock, precisamos entender sua causa. Há quatro condições para que ele ocorra:

- Exclusão mútua
- Bloqueio e espera
- Sem preempção
- Espera circular

## Exclusão mútua

Isso ocorre quando múltiplas threads precisam usar os mesmos recursos e eles

4. For example, someone adds some debugging output and the problem “disappears.” The debugging code “fixes” the problem so it remains in the system.

- Não possam ser usados por múltiplas threads ao mesmo tempo.
- Possuem quantidade limitada.

Um exemplo de tal recurso é uma conexão a um banco de dados, um arquivo aberto para escrita, um bloqueio para registro ou um semáforo.

## Bloqueio e espera

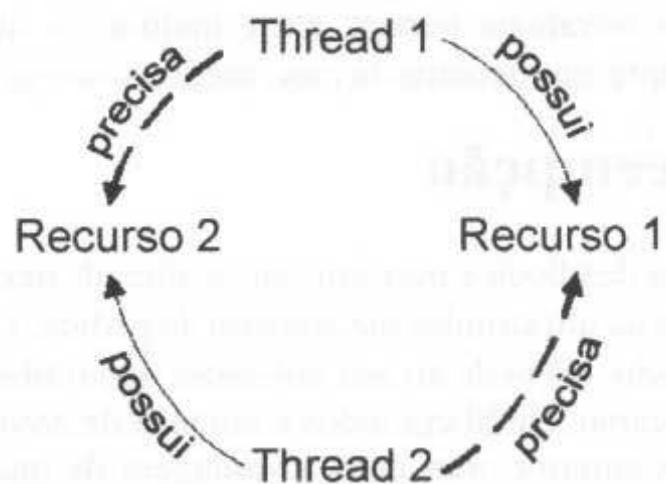
Uma vez que uma thread pega um recurso, ela não o liberará até que tenha obtido todos os outros necessários para que ela complete sua tarefa.

## Sem preempção

Uma thread não pode tomar os recursos de outra. Uma vez que uma thread pega um recurso, a única forma de outra pegá-lo também é que a outra o libere.

## Espera circular

Também chamado de “abraço mortal”. Imagine duas threads, T1 e T2, e dois recursos, R1 e R2. T1 possui R1, T2 possui R2. T1 também precisa de R2, e T2 de R1.  
Esse esquema se parece com a Figura A.3:



Todas essas quatro condições devem existir para que ocorra um deadlock. Se apenas uma não estiver presente, o deadlock não será possível.

## Como evitar a exclusão mútua

Uma estratégia para evitar um deadlock é impedir exclusão mútua. Talvez seja capaz de fazer isso se você:

- Usar recursos que permitam uso simultâneo, por exemplo, o `AtomicInteger`.
- Aumentar a quantidade de recursos de modo que ele se iguale ou exceda o número de threads que competem por eles.

- Verificar se todos os seus recursos estão livres antes de usar um.

Infelizmente, a maioria dos recursos possui número limitado e não permite uso simultâneo. E não é comum que segundo recurso dependa dos resultados obtidos ao usar o primeiro. Mas não desanime; ainda há três condições.

## Como evitar o bloqueio e espera

Você também pode eliminar o deadlock se recusar-se a esperar. Verifique cada recurso antes de usá-lo, e libere todos os recursos e inicie novamente se tentar usar um que esteja ocupado. Essa abordagem adiciona diversos possíveis problemas:

- Espera indefinida (Starvation)—uma thread fica impossibilitada de obter os recursos dos quais ela precisa (talvez seja uma combinação exclusiva de recursos os quais raramente ficam disponíveis).
- Livelock—diversas threads podem ficar num entrave e todas obterem um recurso e, então, liberar um recurso, e assim indefinidamente. Isso ocorre especial e provavelmente com simples algoritmos de agendamento da CPU (pense em dispositivos embutidos ou threads simples criadas manualmente equilibrando os algoritmos).

Ambos os problemas acima podem causar taxas de transferência de dados ruins. O primeiro resulta num uso baixo da CPU, enquanto o segundo causa um uso desnecessário e alto da CPU. Por mais ineficiente que essa estratégia pareça, ela é melhor do que nada. Isso porque há a vantagem de poder quase sempre implementá-la caso todas as outras falhem.

## Como evitar a preempção

Outra estratégia para evitar um deadlock é permitir que as threads peguem os recursos de outras. Geralmente isso é feito através de um simples mecanismo de pedido. Quando uma thread descore que um recurso está sendo usado, ela pede ao seu utilizador que o libere. Se este também estiver esperando por algum outro recurso, ele libera todos e começa de novo.

Essa técnica se parece com a anterior, mas com a vantagem de que se permite a uma thread esperar por um recurso. Isso reduz a quantidade de reinício de tarefas. Entretanto, tenha em mente que gerenciar todos aqueles pedidos pode ser traiçoeiro.

## Como evitar a espera circular

Essa é o método mais comum para evitar um deadlock. Para a maioria dos sistemas é preciso nada mais do que uma simples convenção combinada por todos os colaboradores.

No exemplo acima com a Thread 1 requisitando tanto o Recurso 1 como o Recurso 2, e a Thread 2 esperando o Recurso 2 e, então o recurso 1, simplesmente força as Threads 1 e 2 a alocarem recursos na mesma ordem impossibilita a espera circular.

De modo mais geral, se todas as threads puderem usar uma ordenação de recursos global e se todas os alocarem naquela ordem, então o deadlock se torna impossível. Como todas as outras estratégias, essa pode gerar problemas:

- A ordem de aquisição pode não corresponder com a de uso; embora um recurso obtido no início possa não ser utilizado até o final. Isso pode bloquear os recursos por mais tempo do que o necessário.
- De vez em quando, você não tem como ordenar a aquisição de recursos. Se a ID do segundo recurso vier de uma operação efetuada no primeiro, então a ordenação não é viável.

Sendo assim, há tantas formas de evitar um deadlock. Algumas levam à espera infinita (starvation), enquanto outras aumentam em muito o uso da CPU e reduz a rapidez de resposta.

Isolar a parte relacionada à thread de sua solução para permitir a sincronização e as experiências é uma maneira poderosa de obter o conhecimento necessário para escolher as melhores estratégias.

## Teste de código multithread

Como criar um teste que mostre que o código seguinte está errado?

```
01: public class ClassWithThreadingProblem {  
02:     int nextId;  
03:  
04:     public int takeNextId() {  
05:         return nextId++;  
06:     }  
07: }
```

A seguir está uma descrição de um teste que provará que o código está errado:

- Lembre-se do valor atual de nextId.
- Crie duas threads, cada uma chamando takeNextId() uma vez.
- Verifique se nextId é maior duas vezes mais do que quando começamos.
- Execute até expor que nextId só é incrementado em uma unidade em vez de duas.

A Listagem A.2 mostra esse teste:

### Listagem A-2

#### **ClassWithThreadingProblemTest.java**

```
01: package example;  
02:  
03: import static org.junit.Assert.fail;  
04:  
05: import org.junit.Test;  
06:  
07: public class ClassWithThreadingProblemTest {  
08:     @Test  
09:     public void twoThreadsShouldFailEventually() throws Exception {  
10:         final ClassWithThreadingProblem classWithThreadingProblem  
11:             = new ClassWithThreadingProblem();
```

**Listagem A-2 (continuação)****ClassWithThreadingProblemTest.java**

```

12:     Runnable runnable = new Runnable() {
13:         public void run() {
14:             classWithThreadingProblem.takeNextId();
15:         }
16:     };
17:
18:     for (int i = 0; i < 50000; ++i) {
19:         int startingId = classWithThreadingProblem.lastId;
20:         int expectedResult = 2 + startingId;
21:
22:         Thread t1 = new Thread(runnable);
23:         Thread t2 = new Thread(runnable);
24:         t1.start();
25:         t2.start();
26:         t1.join();
27:         t2.join();
28:
29:         int endingId = classWithThreadingProblem.lastId;
30:
31:         if (endingId != expectedResult)
32:             return;
33:     }
34:
35:     fail("Should have exposed a threading issue but it did not.");
36: }
37: }
```

Linha	Descrição
10	Cria uma única instância de ClassWithThreadingProblem. Note que devemos usar a palavra reservada final, pois a usamos embaixo de uma classe anônima interna.
12-16	Crie uma classe anônima interna que use uma única instância de ClassWithThreadingProblem
18	Execute esse código quantas vezes for necessário para mostrar que o código falhou, mas não demasiadamente para que o teste “demore muito”. Este é um ato balanceado; não queremos esperar muito para expor uma falha. Pegar este número é difícil – embora mais adiante veremos que podemos reduzi-lo consideravelmente.
19	Lembre-se do valor inicial. Esse teste tenta provar que o código em na ClassWithThreadingProblem está errado. Se este teste passar, então ficou provado que o código é falho. Se não passar, o teste não foi capaz de provar que o código está errado.
20	Esperamos que o valor final seja duas vezes mais do que o valor atual.

22-23	Crie duas threads, ambas usando o objeto que criamos nas linhas 12-16. Isso nos dá as duas threads possíveis tentando usar nossa única instância de ClassWithThreadingProblem e que estão interferindo uma na outra.
24-25	Torne executável nossas duas threads.
26-27	Espere ambas as threads finalizarem antes de verificar os resultados.
29	Registre o valor final atual.
31-32	Nosso endingId difere do que esperávamos? Caso seja positivo, retorne e termine o teste – provamos que o código está errado. Caso contrário, tente novamente.
35	Se chegarmos aqui, nosso teste foi incapaz de provar em tempo “razoável” que o código de produção estava errado; nosso código falhou. Ou o código não está quebrado ou não pudemos efetuar iterações suficientes para fazer a condição de erro acontecer.

Este teste certamente configura as condições para um problema de atualização concorrente. Entretanto, ele ocorre tão raramente que na grande maioria das vezes este teste não o detectará. Na verdade, para realmente detectar o problema, precisamos configurar a quantidade de iterações para mais de um milhão. Mesmo assim, em dez execuções com um contador de loop a 1.000.000, o problema só acontecerá uma vez. Isso significa que provavelmente devemos colocar o contador da iteração bem acima de 100 milhões, de modo a obter falhas confiáveis. Quanto tempo estamos dispostos a esperar?

Mesmo se direcionássemos o teste para obter falhas confiáveis em uma máquina, provavelmente teríamos de redirecioná-lo com valores diferentes para expor as falhas em outra máquina, sistema operacional ou versão da JVM.

E esse é um problema simples. Se não pudermos mostrar facilmente com este problema que um código está errado, então como poderemos detectar problemas realmente mais complexos? Portanto, que abordagens podemos tomar para expor essa simples falha? E, o mais importante, como criar testes que demonstrem as falhas num código mais complexo? Como seremos capazes de descobrir se nosso código possui falhas se não sabemos onde procurar?

Aqui estão algumas ideias:

- **Teste de Monte Carlo.** Faça testes flexíveis, de modo que possam ser otimizados. Então, execute-os repetidas vezes – digamos, num servidor – aleatoriamente alterando os valores de otimização. Se os testes falharem, o código está errado. Certifique-se de começar a criação desses testes o quanto antes. Assim, um servidor de integração constante os inicia logo. A propósito, certifique-se de cuidadosamente registrar as condições sob as quais o teste falhou.
- Execute o teste em cada uma das plataformas de implementação finais. Repetidamente. Constantemente. Quanto mais os testes executarem sem falha, mais provavelmente:
  - O código de produção está correto ou
  - Os testes não estão adequados para expor os problemas.
- Rode os testes numa máquina com processos variados. Se você puder simular os processos parecidos a um ambiente de produção, faça.

Ainda assim, mesmo que você faça tudo isso, as chances de encontrar problemas com threads em seu código não são muito boas. Os problemas mais traiçoeiros são aqueles que possuem uma amostragem tão pequena que só ocorrem uma vez em um bilhão de chances. Eles são o terror de sistemas complexos.

## Ferramentas de suporte para testar códigos com threads

A IBM criou uma ferramenta chamada ConTest<sup>6</sup>. Ela altera classes de modo a tornar menos provável que código não seguro para threads falhe.

Não temos nenhum relacionamento direto com a IBM ou a equipe que desenvolveu o ConTest. Foi um colega nosso que o indicou a nós. Notamos uma grande melhoria em nossa capacidade para encontrar questões relacionadas a threads minutos após usar a ferramenta.

A seguir está o resumo de como usar o ConTest:

- Crie testes e código de produção, certifique-se de que há testes desenvolvidos especificamente para simular usuários múltiplos sob cargas variadas, como mencionado anteriormente.
- Altere o teste e o código de produção com o ConTest.
- Execute os testes.

Quando alteramos o código com o ConTest, nossa taxa de êxito caiu bruscamente de uma falha em dez milhões de iterações para uma falha em trinta iterações. Os valores do loop para diversas execuções do teste após a alteração são: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Portanto, claramente as classes alteradas falhavam muito antes e com confiabilidade muito maior.

## Conclusão

Este capítulo foi uma parada muito breve pelo território amplo e traiçoeiro da programação concorrente. Abordamos muito pouco aqui. Focamo-nos nas técnicas para ajudar a manter o código concorrente limpo, mas há muito mais a se aprender se quiser criar sistemas concorrentes. Recomendamos que comece pelo maravilhoso livro de Doug Lea chamado *Concurrent Programming in Java: Design Principles and Patterns*<sup>7</sup>.

Neste capítulo falamos sobre atualização concorrente e as técnicas de sincronização limpa e bloqueios para evitá-la. Discutimos sobre como as threads podem aumentar a taxa de transferência de dados de um sistema baseado em E/S e mostramos técnicas limpas para alcançar tais melhorias. Falamos também sobre deadlock e as técnicas para evitá-lo de uma forma limpa. Por fim, discutimos sobre estratégias para expor os problemas concorrentes através da alteração de seu código.

<sup>6</sup> <http://www.hoifa.ibm.com/projekte/verification/contest/index.html>

## Tutorial: Exemplos com códigos completos

### Cliente/servidor sem threads

#### Listagem A-3

##### Server.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }
}
```

**Listagem A-3 (continuação)****Server.java**

```

void process(Socket socket) {
    if (socket == null)
        return;

    try {
        System.out.printf("Server: getting message\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Server: got message: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Server: sending reply: %s\n", message);
        MessageUtils.sendMessage(socket, "Processed: " + message);
        System.out.printf("Server: sent\n");
        closeIgnoringException(socket);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void closeIgnoringException(Socket socket) {
    if (socket != null)
        try {
            socket.close();
        } catch (IOException ignore) {
        }
}

private void closeIgnoringException(ServerSocket serverSocket) {
    if (serverSocket != null)
        try {
            serverSocket.close();
        } catch (IOException ignore) {
        }
}
}

```

**Listagem A-4****ClientTest.java**

```

package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;
}

```

**Listagem A-4 (continuação)****ClientTest.java**

```
public Server(int port, int millisecondsTimeout) throws IOException {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(millisecondsTimeout);
}

public void run() {
    System.out.printf("Server Starting\n");

    while (keepProcessing) {
        try {
            System.out.printf("accepting client\n");
            Socket socket = serverSocket.accept();
            System.out.printf("got client\n");
            process(socket);
        } catch (Exception e) {
            handle(e);
        }
    }
}

private void handle(Exception e) {
    if (!(e instanceof SocketException)) {
        e.printStackTrace();
    }
}

public void stopProcessing() {
    keepProcessing = false;
    closeIgnoringException(serverSocket);
}

void process(Socket socket) {
    if (socket == null)
        return;

    try {
        System.out.printf("Server: getting message\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Server: got message: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Server: sending reply: %s\n", message);
        MessageUtils.sendMessage(socket, "Processed: " + message);
        System.out.printf("Server: sent\n");
        closeIgnoringException(socket);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void closeIgnoringException(Socket socket) {
    if (socket != null)
        try {
            socket.close();
        }
```

**Listagem A-4 (continuação)**

### ClientTest.java

```
        } catch (IOException ignore) {
    }
}

private void closeIgnoringException(ServerSocket serverSocket) {
    if (serverSocket != null)
        try {
            serverSocket.close();
        } catch (IOException ignore) {
    }
}
```

### Listagem A-5

## MessageUtils.java

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
    public static void sendMessage(Socket socket, String message)
        throws IOException {
        OutputStream stream = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(stream);
        oos.writeUTF(message);
        oos.flush();
    }

    public static String getMessage(Socket socket) throws IOException {
        InputStream stream = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(stream);
        return ois.readUTF();
    }
}
```

## Cliente/servidor usando threads

Alterar o servidor para usar threads requer simplesmente uma mudança no processamento da mensagem (novas linhas foram realçadas para dar ênfase):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                System.out.printf("Servidor: recebendo
mensagem\n");
                String message = MessageUtils.
                getMessage(socket);
                System.out.printf("Servidor: mensagem
recebida: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Servidor: enviando
resposta: %s\n", message);
                MessageUtils.sendMessage(socket,
"Processado: " + message);
                System.out.printf("Servidor: enviado\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

# Apêndice B

## org.jfree.date.SerialDate

**Listing B-1**  
**SerialDate.java**

```
1  /* =====
2   * JCommon : a free general purpose class library for the Java(tm) platform
3   * =====
4   *
5   * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6   *
7   * Project Info: http://www.jfree.org/jcommon/index.html
8   *
9   * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * -----
28  * SerialDate.java
29  * -----
30  * (C) Copyright 2001-2005, by Object Refinery Limited.
31  *
32  * Original Author: David Gilbert (for Object Refinery Limited);
33  * Contributor(s):  -;
34  *
35  * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
36  *
37  * Changes (from 11-Oct-2001)
```

**Listing B-1 (continuação)****SerialDate.java**

```
38 * -----
39 * 11-Oct-2001 : Re-organised the class and moved it to new package
40 *           com.jrefinery.date (DG);
41 * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
42 *           class (DG);
43 * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
44 *           class is gone (DG); Changed getPreviousDayOfWeek(),
45 *           getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
46 *           bugs (DG);
47 * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
48 * 29-May-2002 : Moved the month constants into a separate interface
49 *           (MonthConstants) (DG);
50 * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
51 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
52 * 13-Mar-2003 : Implemented Serializable (DG);
53 * 29-May-2003 : Fixed bug in addMonths method (DG);
54 * 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
55 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
56 *
57 */
58
59 package org.jfree.date;
60
61 import java.io.Serializable;
62 import java.text.DateFormatSymbols;
63 import java.text.SimpleDateFormat;
64 import java.util.Calendar;
65 import java.util.GregorianCalendar;
66
67 /**
68 * An abstract class that defines our requirements for manipulating dates,
69 * without tying down a particular implementation.
70 * <P>
71 * Requirement 1 : match at least what Excel does for dates;
72 * Requirement 2 : class is immutable;
73 * <P>
74 * Why not just use java.util.Date? We will, when it makes sense. At times,
75 * java.util.Date can be *too* precise - it represents an instant in time,
76 * accurate to 1/1000th of a second (with the date itself depending on the
77 * time-zone). Sometimes we just want to represent a particular day (e.g. 21
78 * January 2015) without concerning ourselves about the time of day, or the
79 * time-zone, or anything else. That's what we've defined SerialDate for.
80 * <P>
81 * You can call getInstance() to get a concrete subclass of SerialDate,
82 * without worrying about the exact implementation.
83 *
84 * @author David Gilbert
85 */
86 public abstract class SerialDate implements Comparable,
87                                     Serializable,
88                                     MonthConstants {
89
90     /** For serialization. */
91     private static final long serialVersionUID = -293716040467423637L;
92
93     /** Date format symbols. */
94     public static final DateFormatSymbols
95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
96
97     /** The serial number for 1 January 1900. */
98     public static final int SERIAL_LOWER_BOUND = 2;
99
100    /** The serial number for 31 December 9999. */
101    public static final int SERIAL_UPPER_BOUND = 2958465;
```

**Listing B-1 (continuação)****SerialDate.java**

```
103     /** The lowest year value supported by this date format. */
104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106     /** The highest year value supported by this date format. */
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113      * Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY.
114      */
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118      * Useful constant for Wednesday. Equivalent to
119      * java.util.Calendar.WEDNESDAY.
120      */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124      * Useful constant for Thursday. Equivalent to java.util.Calendar.THURSDAY.
125      */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132      * Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY.
133      */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
136     /** Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** The number of days in each month in non leap years. */
140     static final int[] LAST_DAY_OF_MONTH =
141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31};
142
143     /** The number of days in a (non-leap) year up to the end of each month. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** The number of days in a year up to the end of the preceding month. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151     /** The number of days in a leap year up to the end of each month. */
152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155     /**
156      * The number of days in a leap year up to the end of the preceding month.
157      */
158     static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162     /** A useful constant for referring to the first week in a month. */
163     public static final int FIRST_WEEK_IN_MONTH = 1;
```

**Listing B-1 (continuação)****SerialDate.java**

```
165     /** A useful constant for referring to the second week in a month. */
166     public static final int SECOND_WEEK_IN_MONTH = 2;
167
168     /** A useful constant for referring to the third week in a month. */
169     public static final int THIRD_WEEK_IN_MONTH = 3;
170
171     /** A useful constant for referring to the fourth week in a month. */
172     public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174     /** A useful constant for referring to the last week in a month. */
175     public static final int LAST_WEEK_IN_MONTH = 0;
176
177     /** Useful range constant. */
178     public static final int INCLUDE_NONE = 0;
179
180     /** Useful range constant. */
181     public static final int INCLUDE_FIRST = 1;
182
183     /** Useful range constant. */
184     public static final int INCLUDE_SECOND = 2;
185
186     /** Useful range constant. */
187     public static final int INCLUDE_BOTH = 3;
188
189     /**
190      * Useful constant for specifying a day of the week relative to a fixed
191      * date.
192      */
193     public static final int PRECEDING = -1;
194
195     /**
196      * Useful constant for specifying a day of the week relative to a fixed
197      * date.
198      */
199     public static final int NEAREST = 0;
200
201     /**
202      * Useful constant for specifying a day of the week relative to a fixed
203      * date.
204      */
205     public static final int FOLLOWING = 1;
206
207     /** A description for the date. */
208     private String description;
209
210     /**
211      * Default constructor.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Returns <code>true</code> if the supplied integer code represents a
218      * valid day-of-the-week, and <code>false</code> otherwise.
219      *
220      * @param code the code being checked for validity.
221      *
222      * @return <code>true</code> if the supplied integer code represents a
223      *         valid day-of-the-week, and <code>false</code> otherwise.
224      */
225     public static boolean isValidWeekdayCode(final int code) {
226
```

**Listing B-1 (continuação)****SerialDate.java**

```
227     switch(code) {
228         case SUNDAY:
229         case MONDAY:
230         case TUESDAY:
231         case WEDNESDAY:
232         case THURSDAY:
233         case FRIDAY:
234         case SATURDAY:
235             return true;
236         default:
237             return false;
238     }
239 }
240 /**
241 * Converts the supplied string to a day of the week.
242 *
243 * @param s a string representing the day of the week.
244 *
245 * @return <code>-1</code> if the string is not convertable, the day of
246 *         the week otherwise.
247 */
248 public static int stringToWeekdayCode(String s) {
249
250     final String[] shortWeekdayNames
251         = DATE_FORMAT_SYMBOLS.getShortWeekdays();
252     final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
253
254     int result = -1;
255     s = s.trim();
256     for (int i = 0; i < weekDayNames.length; i++) {
257         if (s.equals(shortWeekdayNames[i])) {
258             result = i;
259             break;
260         }
261         if (s.equals(weekDayNames[i])) {
262             result = i;
263             break;
264         }
265     }
266     return result;
267 }
268 /**
269 * Returns a string representing the supplied day-of-the-week.
270 * <P>
271 * Need to find a better approach.
272 *
273 * @param weekday the day of the week.
274 *
275 * @return a string representing the supplied day-of-the-week.
276 */
277 public static String weekdayCodeToString(final int weekday) {
278
279     final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
280     return weekdays[weekday];
281
282 }
283 /**
284 *
```

**Listing B-1 (continuação)****SerialDate.java**

```
289     * Returns an array of month names.  
290     *  
291     * @return an array of month names.  
292     */  
293     public static String[] getMonths() {  
294  
295         return getMonths(false);  
296     }  
297  
298     /**  
299     * Returns an array of month names.  
300     *  
301     * @param shortened a flag indicating that shortened month names should  
302     *                   be returned.  
303     *  
304     * @return an array of month names.  
305     */  
306     public static String[] getMonths(final boolean shortened) {  
307  
308         if (shortened) {  
309             return DATE_FORMAT_SYMBOLS.getShortMonths();  
310         }  
311         else {  
312             return DATE_FORMAT_SYMBOLS.getMonths();  
313         }  
314     }  
315  
316 }  
317  
318     /**  
319     * Returns true if the supplied integer code represents a valid month.  
320     *  
321     * @param code the code being checked for validity.  
322     *  
323     * @return <code>true</code> if the supplied integer code represents a  
324     *         valid month.  
325     */  
326     public static boolean isValidMonthCode(final int code) {  
327  
328         switch(code) {  
329             case JANUARY:  
330             case FEBRUARY:  
331             case MARCH:  
332             case APRIL:  
333             case MAY:  
334             case JUNE:  
335             case JULY:  
336             case AUGUST:  
337             case SEPTEMBER:  
338             case OCTOBER:  
339             case NOVEMBER:  
340             case DECEMBER:  
341                 return true;  
342             default:  
343                 return false;  
344         }  
345     }  
346  
347     /**  
348     * Returns the quarter for the specified month.  
349     *350 }
```

**Listing B-1 (continuação)****SerialDate.java**

```
351     * @param code the month code (1-12).
352     *
353     * @return the quarter that the month belongs to.
354     * @throws java.lang.IllegalArgumentException
355     */
356    public static int monthCodeToQuarter(final int code) {
357
358        switch(code) {
359            case JANUARY:
360            case FEBRUARY:
361            case MARCH: return 1;
362            case APRIL:
363            case MAY:
364            case JUNE: return 2;
365            case JULY:
366            case AUGUST:
367            case SEPTEMBER: return 3;
368            case OCTOBER:
369            case NOVEMBER:
370            case DECEMBER: return 4;
371            default: throw new IllegalArgumentException(
372                "SerialDate.monthCodeToQuarter: invalid month code.");
373        }
374    }
375
376
377    /**
378     * Returns a string representing the supplied month.
379     * <P>
380     * The string returned is the long form of the month name taken from the
381     * default locale.
382     *
383     * @param month the month.
384     *
385     * @return a string representing the supplied month.
386     */
387    public static String monthCodeToString(final int month) {
388
389        return monthCodeToString(month, false);
390    }
391
392
393    /**
394     * Returns a string representing the supplied month.
395     * <P>
396     * The string returned is the long or short form of the month name taken
397     * from the default locale.
398     *
399     * @param month the month.
400     * @param shortened if <code>true</code> return the abbreviation of the
401     *                  month.
402     *
403     * @return a string representing the supplied month.
404     * @throws java.lang.IllegalArgumentException
405     */
406    public static String monthCodeToString(final int month,
407                                         final boolean shortened) {
408
409        // check arguments...
410        if (!isValidMonthCode(month)) {
411            throw new IllegalArgumentException(
412                "SerialDate.monthCodeToString: month outside valid range.");
413    }
```

**Listing B-1 (continuação)****SerialDate.java**

```
413     }
414
415     final String[] months;
416
417     if (shortened) {
418         months = DATE_FORMAT_SYMBOLS.getShortMonths();
419     }
420     else {
421         months = DATE_FORMAT_SYMBOLS.getMonths();
422     }
423
424     return months[month - 1];
425
426 }
427
428 /**
429 * Converts a string to a month code.
430 * <P>
431 * This method will return one of the constants JANUARY, FEBRUARY, ..., DECEMBER that corresponds to the string. If the string is not recognised, this method returns -1.
432 *
433 * @param s the string to parse.
434 *
435 * @return <code>-1</code> if the string is not parseable, the month of the year otherwise.
436 *
437 */
438 public static int stringToMonthCode(String s) {
439
440     final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
441     final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
442
443     int result = -1;
444     s = s.trim();
445
446     // first try parsing the string as an integer (1-12)...
447     try {
448         result = Integer.parseInt(s);
449     }
450     catch (NumberFormatException e) {
451         // suppress
452     }
453
454     // now search through the month names...
455     if ((result < 1) || (result > 12)) {
456         for (int i = 0; i < monthNames.length; i++) {
457             if (s.equals(shortMonthNames[i])) {
458                 result = i + 1;
459                 break;
460             }
461             if (s.equals(monthNames[i])) {
462                 result = i + 1;
463                 break;
464             }
465         }
466     }
467
468     !
469
470     return result;
471
472 }
473
474 /**

```

**Listing B-1 (continuação)****SerialDate.java**

```
475     * Returns true if the supplied integer code represents a valid
476     * week-in-the-month, and false otherwise.
477     *
478     * @param code the code being checked for validity.
479     * @return <code>true</code> if the supplied integer code represents a
480     *         valid week-in-the-month.
481     */
482    public static boolean isValidWeekInMonthCode(final int code) {
483
484        switch(code) {
485            case FIRST_WEEK_IN_MONTH:
486            case SECOND_WEEK_IN_MONTH:
487            case THIRD_WEEK_IN_MONTH:
488            case FOURTH_WEEK_IN_MONTH:
489            case LAST_WEEK_IN_MONTH: return true;
490            default: return false;
491        }
492    }
493
494
495    /**
496     * Determines whether or not the specified year is a leap year.
497     *
498     * @param yyyy the year (in the range 1900 to 9999).
499     *
500     * @return <code>true</code> if the specified year is a leap year.
501     */
502    public static boolean isLeapYear(final int yyyy) {
503
504        if ((yyyy % 4) != 0) {
505            return false;
506        }
507        else if ((yyyy % 400) == 0) {
508            return true;
509        }
510        else if ((yyyy % 100) == 0) {
511            return false;
512        }
513        else {
514            return true;
515        }
516    }
517
518
519    /**
520     * Returns the number of leap years from 1900 to the specified year
521     * INCLUSIVE.
522     * <p>
523     * Note that 1900 is not a leap year.
524     *
525     * @param yyyy the year (in the range 1900 to 9999).
526     *
527     * @return the number of leap years from 1900 to the specified year.
528     */
529    public static int leapYearCount(final int yyyy) {
530
531        final int leap4 = (yyyy - 1896) / 4;
532        final int leap100 = (yyyy - 1800) / 100;
533        final int leap400 = (yyyy - 1600) / 400;
534        return leap4 - leap100 + leap400;
535    }
536 }
```

**Listing B-1 (continuação)****SerialDate.java**

```

537
538     /**
539      * Returns the number of the last day of the month, taking into account
540      * leap years.
541      *
542      * @param month  the month.
543      * @param yyyy  the year (in the range 1900 to 9999).
544      *
545      * @return the number of the last day of the month.
546      */
547     public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
551             return result;
552         }
553         else if (isLeapYear(yyyy)) {
554             return result + 1;
555         }
556         else {
557             return result;
558         }
559     }
560
561     /**
562      * Creates a new date by adding the specified number of days to the base
563      * date.
564      *
565      * @param days  the number of days to add (can be negative).
566      * @param base  the base date.
567      *
568      * @return a new date.
569      */
570     public static SerialDate addDays(final int days, final SerialDate base) {
571
572         final int serialDayNumber = base.toSerial() + days;
573         return SerialDate.createInstance(serialDayNumber);
574
575     }
576
577     /**
578      * Creates a new date by adding the specified number of months to the base
579      * date.
580      * <P>
581      * If the base date is close to the end of the month, the day on the result
582      * may be adjusted slightly: 31 May + 1 month = 30 June.
583      *
584      * @param months  the number of months to add (can be negative).
585      * @param base  the base date.
586      *
587      * @return a new date.
588      */
589     public static SerialDate addMonths(final int months,
590                                         final SerialDate base) {
591
592         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
593                     / 12;
594         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
595                     % 12 + 1;
596         final int dd = Math.min(
597             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy))

```

**Listing B-1 (continuação)****SerialDate.java**

```
599      );
600      return SerialDate.createInstance(dd, mm, yy);
601
602 }
603
604 /**
605 * Creates a new date by adding the specified number of years to the base
606 * date.
607 *
608 * @param years  the number of years to add (can be negative).
609 * @param base   the base date.
610 *
611 * @return A new date.
612 */
613 public static SerialDate addYears(final int years, final SerialDate base) {
614
615     final int baseY = base.getYYYY();
616     final int baseM = base.getMonth();
617     final int baseD = base.getDayOfMonth();
618
619     final int targetY = baseY + years;
620     final int targetD = Math.min(
621         baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622     );
623
624     return SerialDate.createInstance(targetD, baseM, targetY);
625
626 }
627
628 /**
629 * Returns the latest date that falls on the specified day-of-the-week and
630 * is BEFORE the base date.
631 *
632 * @param targetWeekday a code for the target day-of-the-week.
633 * @param base   the base date.
634 *
635 * @return the latest date that falls on the specified day-of-the-week and
636 *         is BEFORE the base date.
637 */
638 public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                              final SerialDate base) {
640
641     // check arguments...
642     if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643         throw new IllegalArgumentException(
644             "Invalid day-of-the-week code."
645         );
646     }
647
648     // find the date...
649     final int adjust;
650     final int baseDOW = base.getDayOfWeek();
651     if (baseDOW > targetWeekday) {
652         adjust = Math.min(0, targetWeekday - baseDOW);
653     }
654     else {
655         adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656     }
657
658     return SerialDate.addDays(adjust, base);
659
660 }
```

**Listing B-1 (continuação)****SerialDate.java**

```
661
662     /**
663      * Returns the earliest date that falls on the specified day-of-the-week
664      * and is AFTER the base date.
665      *
666      * @param targetWeekday a code for the target day-of-the-week.
667      * @param base the base date.
668      *
669      * @return the earliest date that falls on the specified day-of-the-week
670      *         and is AFTER the base date.
671      */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                                 final SerialDate base) {
674
675         // check arguments...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code.");
679         }
680     }
681
682     // find the date...
683     final int adjust;
684     final int baseDOW = base.getDayOfWeek();
685     if (baseDOW > targetWeekday) {
686         adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687     }
688     else {
689         adjust = Math.max(0, targetWeekday - baseDOW);
690     }
691
692     return SerialDate.addDays(adjust, base);
693 }
694
695 /**
696  * Returns the date that falls on the specified day-of-the-week and is
697  * CLOSEST to the base date.
698  *
699  * @param targetDOW a code for the target day-of-the-week.
700  * @param base the base date.
701  *
702  * @return the date that falls on the specified day-of-the-week and is
703  *         CLOSEST to the base date.
704  */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                               final SerialDate base) {
707
708         // check arguments...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code.");
712         }
713     }
714
715     // find the date...
716     final int baseDOW = base.getDayOfWeek();
717     int adjust = -Math.abs(targetDOW - baseDOW);
718     if (adjust >= 4) {
719         adjust = 7 - adjust;
720     }
721     if (adjust <= -4) {
722         adjust = 7 + adjust;
```

**Listing B-1 (continuação)****SerialDate.java**

```
723     }
724     return SerialDate.addDays(adjust, base);
725   }
726 
727   /**
728   * Rolls the date forward to the last day of the month.
729   *
730   * @param base  the base date.
731   *
732   * @return a new serial date.
733   */
734   public SerialDate getEndOfCurrentMonth(final SerialDate base) {
735     final int last = SerialDate.lastDayOfMonth(
736       base.getMonth(), base.getYYYY());
737     return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
738   }
739 
740 /**
741 * Returns a string corresponding to the week-in-the-month code.
742 * <P>
743 * Need to find a better approach.
744 *
745 * @param count  an integer code representing the week-in-the-month.
746 *
747 * @return a string corresponding to the week-in-the-month code.
748 *
749 */
750 public static String weekInMonthToString(final int count) {
751 
752   switch (count) {
753     case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
754     case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
755     case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
756     case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
757     case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
758     default :
759       return "SerialDate.weekInMonthToString(): invalid code.";
760   }
761 }
762 
763 /**
764 * Returns a string representing the supplied 'relative'.
765 * <P>
766 * Need to find a better approach.
767 *
768 * @param relative  a constant representing the 'relative'.
769 *
770 * @return a string representing the supplied 'relative'.
771 *
772 */
773 public static String relativeToString(final int relative) {
774 
775   switch (relative) {
776     case SerialDate.PRECEDING : return "Preceding";
777     case SerialDate.NEAREST : return "Nearest";
778     case SerialDate.FOLLOWING : return "Following";
779     default : return "ERROR : Relative To String";
780   }
781 }
782 
783 }
784 }
```

**Listing B-1 (continuação)****SerialDate.java**

```
785     /**
786      * Factory method that returns an instance of some concrete subclass of
787      * {@link SerialDate}.
788      *
789      * @param day the day (1-31).
790      * @param month the month (1-12).
791      * @param yyyy the year (in the range 1900 to 9999).
792      *
793      * @return An instance of {@link SerialDate}.
794      */
795     public static SerialDate createInstance(final int day, final int month,
796                                             final int yyyy) {
797         return new SpreadsheetDate(day, month, yyyy);
798     }
799
800     /**
801      * Factory method that returns an instance of some concrete subclass of
802      * {@link SerialDate}.
803      *
804      * @param serial the serial number for the day (! January 1900 = 2).
805      *
806      * @return a instance of SerialDate.
807      */
808     public static SerialDate createInstance(final int serial) {
809         return new SpreadsheetDate(serial);
810     }
811
812     /**
813      * Factory method that returns an instance of a subclass of SerialDate.
814      *
815      * @param date A Java date object.
816      *
817      * @return a instance of SerialDate.
818      */
819     public static SerialDate createInstance(final java.util.Date date) {
820
821         final GregorianCalendar calendar = new GregorianCalendar();
822         calendar.setTime(date);
823         return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                   calendar.get(Calendar.MONTH) + 1,
825                                   calendar.get(Calendar.YEAR));
826     }
827
828
829     /**
830      * Returns the serial number for the date, where 1 January 1900 = 2 (this
831      * corresponds, almost, to the numbering system used in Microsoft Excel for
832      * Windows and Lotus 1-2-3!).
833      *
834      * @return the serial number for the date.
835      */
836     public abstract int toSerial();
837
838     /**
839      * Returns a java.util.Date. Since java.util.Date has more precision than
840      * SerialDate, we need to define a convention for the 'time of day'.
841      *
842      * @return this as <code>java.util.Date</code>.
843      */
844     public abstract java.util.Date toDate();
845
846     /**
```

**Listing B-1 (continuação)****SerialDate.java**

```
847     * Returns a description of the date.  
848     *  
849     * @return a description of the date.  
850     */  
851     public String getDescription() {  
852         return this.description;  
853     }  
854  
855     /**  
856     * Sets the description for the date.  
857     *  
858     * @param description the new description for the date.  
859     */  
860     public void setDescription(final String description) {  
861         this.description = description;  
862     }  
863  
864     /**  
865     * Converts the date to a string.  
866     *  
867     * @return a string representation of the date.  
868     */  
869     public String toString() {  
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())  
871             + "-" + getYYYY();  
872     }  
873  
874     /**  
875     * Returns the year (assume a valid range of 1900 to 9999).  
876     *  
877     * @return the year.  
878     */  
879     public abstract int getYYYY();  
880  
881     /**  
882     * Returns the month (January = 1, February = 2, March = 3).  
883     *  
884     * @return the month of the year.  
885     */  
886     public abstract int getMonth();  
887  
888     /**  
889     * Returns the day of the month.  
890     *  
891     * @return the day of the month.  
892     */  
893     public abstract int getDayOfMonth();  
894  
895     /**  
896     * Returns the day of the week.  
897     *  
898     * @return the day of the week.  
899     */  
900     public abstract int getDayOfWeek();  
901  
902     /**  
903     * Returns the difference (in days) between this date and the specified  
904     * 'other' date.  
905     * <P>  
906     * The result is positive if this date is after the 'other' date and  
907     * negative if it is before the 'other' date.  
908     */
```

**Listing B-1 (continuação)****SerialDate.java**

```
909     * @param other the date being compared to.  
910     *  
911     * @return the difference between this and the other date.  
912     */  
913     public abstract int compare(SerialDate other);  
914  
915     /**  
916      * Returns true if this SerialDate represents the same date as the  
917      * specified SerialDate.  
918      *  
919      * @param other the date being compared to.  
920      *  
921      * @return <code>true</code> if this SerialDate represents the same date as  
922      *          the specified SerialDate.  
923      */  
924     public abstract boolean isOn(SerialDate other);  
925  
926     /**  
927      * Returns true if this SerialDate represents an earlier date compared to  
928      *          the specified SerialDate.  
929      *  
930      * @param other The date being compared to.  
931      *  
932      * @return <code>true</code> if this SerialDate represents an earlier date  
933      *          compared to the specified SerialDate.  
934      */  
935     public abstract boolean isBefore(SerialDate other);  
936  
937     /**  
938      * Returns true if this SerialDate represents the same date as the  
939      * specified SerialDate.  
940      *  
941      * @param other the date being compared to.  
942      *  
943      * @return <code>true</code> if this SerialDate represents the same date  
944      *          as the specified SerialDate.  
945      */  
946     public abstract boolean isOnOrBefore(SerialDate other);  
947  
948     /**  
949      * Returns true if this SerialDate represents the same date as the  
950      * specified SerialDate.  
951      *  
952      * @param other the date being compared to.  
953      *  
954      * @return <code>true</code> if this SerialDate represents the same date  
955      *          as the specified SerialDate.  
956      */  
957     public abstract boolean isAfter(SerialDate other);  
958  
959     /**  
960      * Returns true if this SerialDate represents the same date as the  
961      * specified SerialDate.  
962      *  
963      * @param other the date being compared to.  
964      *  
965      * @return <code>true</code> if this SerialDate represents the same date  
966      *          as the specified SerialDate.  
967      */  
968     public abstract boolean isOnOrAfter(SerialDate other);  
969  
970     /**  
971      * Returns <code>true</code> if this {@link SerialDate} is within the
```

**Listing B-1 (continuação)****SerialDate.java**

```
972     * specified range (INCLUSIVE). The date order of d1 and d2 is not
973     * important.
974     *
975     * @param d1 a boundary date for the range.
976     * @param d2 the other boundary date for the range.
977     *
978     * @return A boolean.
979     */
980    public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982    /**
983     * Returns <code>true</code> if this (@link SerialDate) is within the
984     * specified range (caller specifies whether or not the end-points are
985     * included). The date order of d1 and d2 is not important.
986     *
987     * @param d1 a boundary date for the range.
988     * @param d2 the other boundary date for the range.
989     * @param include a code that controls whether or not the start and end
990     *                 dates are included in the range.
991     *
992     * @return A boolean.
993     */
994    public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995                                     int include);
996
997    /**
998     * Returns the latest date that falls on the specified day-of-the-week and
999     * is BEFORE this date.
1000    *
1001    * @param targetDOW a code for the target day-of-the-week.
1002    *
1003    * @return the latest date that falls on the specified day-of-the-week and
1004    *         is BEFORE this date.
1005    */
1006    public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007        return getPreviousDayOfWeek(targetDOW, this);
1008    }
1009
1010    /**
1011     * Returns the earliest date that falls on the specified day-of-the-week
1012     * and is AFTER this date.
1013     *
1014     * @param targetDOW a code for the target day-of-the-week.
1015     *
1016     * @return the earliest date that falls on the specified day-of-the-week
1017     *         and is AFTER this date.
1018     */
1019    public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020        return getFollowingDayOfWeek(targetDOW, this);
1021    }
1022
1023    /**
1024     * Returns the nearest date that falls on the specified day-of-the-week.
1025     *
1026     * @param targetDOW a code for the target day-of-the-week.
1027     *
1028     * @return the nearest date that falls on the specified day-of-the-week.
1029     */
1030    public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031        return getNearestDayOfWeek(targetDOW, this);
1032    }
1033
1034 }
```

**Listagem B-2****SerialDateTest.java**

```
1  /*
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * -----
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * SerialDateTests.java
29 * -----
30 * (C) Copyright 2001-2005, by Object Refinery Limited.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s): -;
34 *
35 * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Changes
38 * -----
39 * 15-Nov-2001 : Version 1 (DG);
40 * 25-Jun-2002 : Removed unnecessary import (DG);
41 * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
42 * 13-Mar-2003 : Added serialization test (DG);
43 * 05-Jan-2005 : Added test for bug report 1096282 (DG);
44 *
45 */
46
47 package org.jfree.date.junit;
48
49 import java.io.ByteArrayInputStream;
50 import java.io.ByteArrayOutputStream;
51 import java.io.ObjectInput;
52 import java.io.ObjectInputStream;
53 import java.io.ObjectOutput;
54 import java.io.ObjectOutputStream;
55
56 import junit.framework.Test;
57 import junit.framework.TestCase;
58 import junit.framework.TestSuite;
59
60 import org.jfree.date.MonthConstants;
61 import org.jfree.date.SerialDate;
62
```

**Listagem B-2 (continuação)****SerialDateTest.java**

```
63 /**
64  * Some JUnit tests for the {@link SerialDate} class.
65 */
66 public class SerialDateTests extends TestCase {
67
68     /** Date representing November 9. */
69     private SerialDate nov9Y2001;
70
71     /**
72      * Creates a new test case.
73      *
74      * @param name the name.
75      */
76     public SerialDateTests(final String name) {
77         super(name);
78     }
79
80     /**
81      * Returns a test suite for the JUnit test runner.
82      *
83      * @return The test suite.
84      */
85     public static Test suite() {
86         return new TestSuite(SerialDateTests.class);
87     }
88
89     /**
90      * Problem set up.
91      */
92     protected void setUp() {
93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
94     }
95
96     /**
97      * 9 Nov 2001 plus two months should be 9 Jan 2002.
98      */
99     public void testAddMonthsTo9Nov2001() {
100         final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101         final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102         assertEquals(answer, jan9Y2002);
103     }
104
105    /**
106     * A test case for a reported bug, now fixed.
107     */
108    public void testAddMonthsTo5Oct2003() {
109        final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110        final SerialDate d2 = SerialDate.addMonths(2, d1);
111        assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112    }
113
114    /**
115     * A test case for a reported bug, now fixed.
116     */
117    public void testAddMonthsTo1Jan2003() {
118        final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119        final SerialDate d2 = SerialDate.addMonths(0, d1);
120        assertEquals(d2, d1);
121    }
122
123    /**
124     * Monday preceding Friday 9 November 2001 should be 5 November.
```

**Listagem B-2 (continuação)****SerialDateTest.java**

```
125  /*
126  public void testMondayPrecedingFriday9Nov2001() {
127      SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128          SerialDate.MONDAY, this.nov9Y2001
129      );
130      assertEquals(5, mondayBefore.getDayOfMonth());
131  }
132
133 /**
134 * Monday following Friday 9 November 2001 should be 12 November.
135 */
136 public void testMondayFollowingFriday9Nov2001() {
137     SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138         SerialDate.MONDAY, this.nov9Y2001
139     );
140     assertEquals(12, mondayAfter.getDayOfMonth());
141 }
142
143 /**
144 * Monday nearest Friday 9 November 2001 should be 12 November.
145 */
146 public void testMondayNearestFriday9Nov2001() {
147     SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148         SerialDate.MONDAY, this.nov9Y2001
149     );
150     assertEquals(12, mondayNearest.getDayOfMonth());
151 }
152
153 /**
154 * The Monday nearest to 22nd January 1970 falls on the 19th.
155 */
156 public void testMondayNearest22Jan1970() {
157     SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970);
158     SerialDate mondayNearest=SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970);
159     assertEquals(19, mondayNearest.getDayOfMonth());
160 }
161
162 /**
163 * Problem that the conversion of days to strings returns the right result. Actually, this
164 * result depends on the Locale so this test needs to be modified.
165 */
166 public void testWeekdayCodeToString() {
167
168     final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169     assertEquals("Saturday", test);
170 }
171
172 /**
173 * Test the conversion of a string to a weekday. Note that this test will fail if the
174 * default locale doesn't use English weekday names...devise a better test!
175 */
176 public void testStringToWeekday() {
177
178     int weekday = SerialDate.stringToWeekdayCode("Wednesday");
179     assertEquals(SerialDate.WEDNESDAY, weekday);
180
181     weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
182     assertEquals(SerialDate.WEDNESDAY, weekday);
183
184 }
```

**Listagem B-2 (continuação)****SerialDateTest.java**

```
185     weekday = SerialDate.stringToWeekdayCode("Wed");
186     assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     /*
189      * Test the conversion of a string to a month. Note that this test will fail if the
190      * default locale doesn't use English month names...devise a better test!
191      */
192     public void testStringToMonthCode() {
193
194         int m = SerialDate.stringToMonthCode("January");
195         assertEquals(MonthConstants.JANUARY, m);
196
197         m = SerialDate.stringToMonthCode(" January ");
198         assertEquals(MonthConstants.JANUARY, m);
199
200         m = SerialDate.stringToMonthCode("Jan");
201         assertEquals(MonthConstants.JANUARY, m);
202
203     }
204
205     /*
206      * Tests the conversion of a month code to a string.
207      */
208     public void testMonthCodeToStringCode() {
209
210         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
211         assertEquals("December", test);
212
213     }
214
215     /*
216      * 1900 is not a leap year.
217      */
218     public void testIsNotLeapYear1900() {
219         assertTrue(!SerialDate.isLeapYear(1900));
220     }
221
222     /*
223      * 2000 is a leap year.
224      */
225     public void testIsLeapYear2000() {
226         assertTrue(SerialDate.isLeapYear(2000));
227     }
228
229     /*
230      * The number of leap years from 1900 up-to-and-including 1899 is 0.
231      */
232     public void testLeapYearCount1899() {
233         assertEquals(SerialDate.leapYearCount(1899), 0);
234     }
235
236     /*
237      * The number of leap years from 1900 up-to-and-including 1903 is 0.
238      */
239     public void testLeapYearCount1903() {
240         assertEquals(SerialDate.leapYearCount(1903), 0);
241     }
242
243     /*
244      * The number of leap years from 1900 up-to-and-including 1904 is 1.
245      */
246
```

**Listagem B-2 (continuação)****SerialDateTest.java**

```
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253      * The number of leap years from 1900 up-to-and-including 1999 is 24.
254      */
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * The number of leap years from 1900 up-to-and-including 2000 is 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Serialize an instance, restore it, and check for equality.
268      */
269     public void testSerialization() {
270
271         SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272         SerialDate d2 = null;
273
274         try {
275             ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276             ObjectOutputStream out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInputStream in = new ObjectInputStream(
281                     new ByteArrayInputStream(buffer.toByteArray()));
282             d2 = (SerialDate) in.readObject();
283             in.close();
284         } catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288     }
289
290     /**
291      * A test for bug report 1096282 (now fixed).
292      */
293     public void test1096282() {
294         SerialDate d = SerialDate.createInstance(29, 2, 2004);
295         d = SerialDate.addYears(1, d);
296         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
297         assertTrue(d.isOn(expected));
298     }
299
300     /**
301      * Miscellaneous tests for the addMonths() method.
302      */
303     public void testAddMonths() {
304         SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
```

**Listagem B-2 (continuação)****SerialDateTest.java**

```
307     SerialDate d2 = SerialDate.addMonths(1, d1);
308     assertEquals(30, d2.getDayOfMonth());
309     assertEquals(6, d2.getMonth());
310     assertEquals(2004, d2.getYYYY());
311
312     SerialDate d3 = SerialDate.addMonths(2, d1);
313     assertEquals(51, d3.getDayOfMonth());
314     assertEquals(7, d3.getMonth());
315     assertEquals(2004, d3.getYYYY());
316
317     SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318     assertEquals(30, d4.getDayOfMonth());
319     assertEquals(7, d4.getMonth());
320     assertEquals(2004, d4.getYYYY());
321 }
322 }
```

**Listagem B-3 (continuação)****MonthConstants.java**

```
1  /*
2  * =====
3  * JCommon : a free general purpose class library for the Java(tm) platform
4  * =====
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301.
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * MonthConstants.java
29 * -----
30 * (C) Copyright 2002, 2003, by Object Refinery Limited.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s): -;
34 *
35 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Changes
38 * -----
```

**Listagem B-3 (continuação)****MonthConstants.java**

```

39 * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
40 *
41 */
42
43 package org.jfree.date;
44
45 /**
46 * Useful constants for months. Note that these are NOT equivalent to the
47 * constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
48 * <P>
49 * Used by the SerialDate and RegularTimePeriod classes.
50 *
51 * @author David Gilbert
52 */
53 public interface MonthConstants {
54
55     /** Constant for January. */
56     public static final int JANUARY = 1;
57
58     /** Constant for February. */
59     public static final int FEBRUARY = 2;
60

```

**Listagem B-4 (continuação)****BobsSerialDateTest.java**

```

1 package org.jfree.date.junit;
2
3 import junit.framework.TestCase;
4 import org.jfree.date.*;
5 import static org.jfree.date.SerialDate.*;
6
7 import java.util.*;
8
9 public class BobsSerialDateTest extends TestCase {
10
11     public void testIsValidWeekdayCode() throws Exception {
12         for (int day = 1; day <= 7; day++) {
13             assertTrue(isValidWeekdayCode(day));
14             assertFalse(isValidWeekdayCode(0));
15             assertFalse(isValidWeekdayCode(8));
16         }
17
18         public void testStringToWeekdayCode() throws Exception {
19
20             assertEquals(-1, stringToWeekdayCode("Hello"));
21             assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22             assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo             assertEquals(MONDAY, stringToWeekdayCode("monday"));
24 //             assertEquals(MONDAY, stringToWeekdayCode("MONDAY"));
25 //             assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27             assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28             assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 //             assertEquals(TUESDAY, stringToWeekdayCode("tuesday"));
30 //             assertEquals(TUESDAY, stringToWeekdayCode("TUESDAY"));
31 //             assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 //             assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34             assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35             assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));

```

**Listagem B-4 (continuação)****BobsSerialDateTest.java**

```
54     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
55     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
56 //    assertEquals(WEDNESDAY, stringToWeekdayCode("wednesday"));
57 //    assertEquals(WEDNESDAY, stringToWeekdayCode("WEDNESDAY"));
58 //    assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
59
60     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
61     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
62 //    assertEquals(THURSDAY, stringToWeekdayCode("thursday"));
63 //    assertEquals(THURSDAY, stringToWeekdayCode("THURSDAY"));
64 //    assertEquals(THURSDAY, stringToWeekdayCode("thu"));
65 //    assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
66
67     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
68     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
69 //    assertEquals(FRIDAY, stringToWeekdayCode("friday"));
70 //    assertEquals(FRIDAY, stringToWeekdayCode("FRIDAY"));
71 //    assertEquals(FRIDAY, stringToWeekdayCode("fri"));
72
73     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
74     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
75 //    assertEquals(SATURDAY, stringToWeekdayCode("saturday"));
76 //    assertEquals(SATURDAY, stringToWeekdayCode("SATURDAY"));
77 //    assertEquals(SATURDAY, stringToWeekdayCode("sat"));
78
79     assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
80     assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
81 //    assertEquals(SUNDAY, stringToWeekdayCode("sunday"));
82 //    assertEquals(SUNDAY, stringToWeekdayCode("SUNDAY"));
83 //    assertEquals(SUNDAY, stringToWeekdayCode("sun"));
84
85     public void testWeekdayCodeToString() throws Exception {
86         assertEquals("Sunday", weekdayCodeToString(SUNDAY));
87         assertEquals("Monday", weekdayCodeToString(MONDAY));
88         assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
89         assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
90         assertEquals("Thursday", weekdayCodeToString(THURSDAY));
91         assertEquals("Friday", weekdayCodeToString(FRIDAY));
92         assertEquals("Saturday", weekdayCodeToString(SATURDAY));
93     }
94
95     public void testIsValidMonthCode() throws Exception {
96         for (int i = 1; i <= 12; i++)
97             assertTrue(isValidMonthCode(i));
98         assertFalse(isValidMonthCode(0));
99         assertFalse(isValidMonthCode(13));
100    }
101
102    public void testMonthToQuarter() throws Exception {
103        assertEquals(1, monthCodeToQuarter(JANUARY));
104        assertEquals(1, monthCodeToQuarter(FEBRUARY));
105        assertEquals(1, monthCodeToQuarter(MARCH));
106        assertEquals(2, monthCodeToQuarter(APRIL));
107        assertEquals(2, monthCodeToQuarter(MAY));
108        assertEquals(2, monthCodeToQuarter(JUNE));
109        assertEquals(3, monthCodeToQuarter(JULY));
110        assertEquals(3, monthCodeToQuarter(AUGUST));
111        assertEquals(3, monthCodeToQuarter(SEPTMBER));
112        assertEquals(4, monthCodeToQuarter(OCTOBER));
113        assertEquals(4, monthCodeToQuarter(NOVEMBER));
```

**Listagem B-4 (continuação)****BobsSerialDateTest.java**

```
94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
95     assertEquals(4, monthCodeToQuarter(DECEMBER));
96
97     try {
98         monthCodeToQuarter(-1);
99         fail("Invalid Month Code should throw exception");
100    } catch (IllegalArgumentException e) {
101    }
102 }
103
104 public void testMonthCodeToString() throws Exception {
105     assertEquals("January", monthCodeToString(JANUARY));
106     assertEquals("February", monthCodeToString(FEBRUARY));
107     assertEquals("March", monthCodeToString(MARCH));
108     assertEquals("April", monthCodeToString(APRIL));
109     assertEquals("May", monthCodeToString(MAY));
110     assertEquals("June", monthCodeToString(JUNE));
111     assertEquals("July", monthCodeToString(JULY));
112     assertEquals("August", monthCodeToString(AUGUST));
113     assertEquals("September", monthCodeToString(SEPTEMBER));
114     assertEquals("October", monthCodeToString(OCTOBER));
115     assertEquals("November", monthCodeToString(NOVEMBER));
116     assertEquals("December", monthCodeToString(DECEMBER));
117
118     assertEquals("Jan", monthCodeToString(JANUARY, true));
119     assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120     assertEquals("Mar", monthCodeToString(MARCH, true));
121     assertEquals("Apr", monthCodeToString(APRIL, true));
122     assertEquals("May", monthCodeToString(MAY, true));
123     assertEquals("Jun", monthCodeToString(JUNE, true));
124     assertEquals("Jul", monthCodeToString(JULY, true));
125     assertEquals("Aug", monthCodeToString(AUGUST, true));
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
128
129     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
130     assertEquals("Dec", monthCodeToString(DECEMBER, true));
131
132     try {
133         monthCodeToString(-1);
134         fail("Invalid month code should throw exception");
135     } catch (IllegalArgumentException e) {
136     }
137 }
138
139 public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY, stringToMonthCode("1"));
141     assertEquals(FEBRUARY, stringToMonthCode("2"));
142     assertEquals(MARCH, stringToMonthCode("3"));
143     assertEquals(APRIL, stringToMonthCode("4"));
144     assertEquals(MAY, stringToMonthCode("5"));
145     assertEquals(JUNE, stringToMonthCode("6"));
146     assertEquals(JULY, stringToMonthCode("7"));
147     assertEquals(AUGUST, stringToMonthCode("8"));
148     assertEquals(SEPTEMBER, stringToMonthCode("9"));
149     assertEquals(OCTOBER, stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER, stringToMonthCode("12"));
152
153 //todo     assertEquals(-1, stringToMonthCode("0"));
154 //      assertEquals(-1, stringToMonthCode("13"));
```

**Listagem B-4 (continuação)**  
**BobSerialDateTest.java**

```
155     assertEquals(-1, stringToMonthCode("Hello"));
156
157     for (int m = 1; m <= 12; m++) {
158         assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
159         assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
160     }
161
162 //    assertEquals(1, stringToMonthCode("jan"));
163 //    assertEquals(2, stringToMonthCode("feb"));
164 //    assertEquals(3, stringToMonthCode("mar"));
165 //    assertEquals(4, stringToMonthCode("apr"));
166 //    assertEquals(5, stringToMonthCode("may"));
167 //    assertEquals(6, stringToMonthCode("jun"));
168 //    assertEquals(7, stringToMonthCode("jul"));
169 //    assertEquals(8, stringToMonthCode("aug"));
170 //    assertEquals(9, stringToMonthCode("sep"));
171 //    assertEquals(10, stringToMonthCode("oct"));
172 //    assertEquals(11, stringToMonthCode("nov"));
173 //    assertEquals(12, stringToMonthCode("dec"));
174
175 //    assertEquals(1, stringToMonthCode("JAN"));
176 //    assertEquals(2, stringToMonthCode("FEB"));
177 //    assertEquals(3, stringToMonthCode("MAR"));
178 //    assertEquals(4, stringToMonthCode("APR"));
179 //    assertEquals(5, stringToMonthCode("MAY"));
180 //    assertEquals(6, stringToMonthCode("JUN"));
181 //    assertEquals(7, stringToMonthCode("JUL"));
182 //    assertEquals(8, stringToMonthCode("AUG"));
183 //    assertEquals(9, stringToMonthCode("SEP"));
184 //    assertEquals(10, stringToMonthCode("OCT"));
185 //    assertEquals(11, stringToMonthCode("NOV"));
186 //    assertEquals(12, stringToMonthCode("DEC"));
187
188 //    assertEquals(1, stringToMonthCode("january"));
189 //    assertEquals(2, stringToMonthCode("february"));
190
191 //    assertEquals(3, stringToMonthCode("march"));
192 //    assertEquals(4, stringToMonthCode("april"));
193 //    assertEquals(5, stringToMonthCode("may"));
194 //    assertEquals(6, stringToMonthCode("june"));
195 //    assertEquals(7, stringToMonthCode("july"));
196 //    assertEquals(8, stringToMonthCode("august"));
197 //    assertEquals(9, stringToMonthCode("september"));
198 //    assertEquals(10, stringToMonthCode("october"));
199 //    assertEquals(11, stringToMonthCode("november"));
200 //    assertEquals(12, stringToMonthCode("december"));
201
202 //    assertEquals(1, stringToMonthCode("JANUARY"));
203 //    assertEquals(2, stringToMonthCode("FEBRUARY"));
204 //    assertEquals(3, stringToMonthCode("MAR"));
205 //    assertEquals(4, stringToMonthCode("APRIL"));
206 //    assertEquals(5, stringToMonthCode("MAY"));
207 //    assertEquals(6, stringToMonthCode("JUNE"));
208 //    assertEquals(7, stringToMonthCode("JULY"));
209 //    assertEquals(8, stringToMonthCode("AUGUST"));
210 //    assertEquals(9, stringToMonthCode("SEPTEMBER"));
211 //    assertEquals(10, stringToMonthCode("OCTOBER"));
212 //    assertEquals(11, stringToMonthCode("NOVEMBER"));
213 //    assertEquals(12, stringToMonthCode("DECEMBER"));
214 }
215
```

**Listagem B-4 (continuação)****BobSerialDateTest.java**

```
216 public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218         assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221 }
222
223 public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236 }
237
238 public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
252
253     assertEquals(97, leapYearCount(2301));
254     assertEquals(122, leapYearCount(2401));
255 }
256
257 public void testLastDayOfMonth() throws Exception {
258     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
259     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
260     assertEquals(31, lastDayOfMonth(MARCH, 1901));
261     assertEquals(30, lastDayOfMonth(APRIL, 1901));
262     assertEquals(31, lastDayOfMonth(MAY, 1901));
263     assertEquals(30, lastDayOfMonth(JUNE, 1901));
264     assertEquals(31, lastDayOfMonth(JULY, 1901));
265     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
266     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
267     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
268     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
269     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
270     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
271 }
272
273 public void testAddDays() throws Exception {
274     SerialDate newYears = d(1, JANUARY, 1900);
275     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
```

**Listagem B-4 (continuação)****BobsSerialDateTest.java**

```
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278 }
279
280 private static SpreadsheetDate d(int day, int month, int year) {return new
SpreadsheetDate(day, month, year);}
281
282 public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294 }
295
296
297 public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302 }
303
304 public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2005)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005)));
309
310     try {
311         getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312         fail("Invalid day of week code should throw exception");
313     } catch (IllegalArgumentException e) {
314     }
315 }
316
317 public void testGetFollowingDayOfWeek() throws Exception {
318 //    assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));
321
322     try {
323         getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324         fail("Invalid day of week code should throw exception");
325     } catch (IllegalArgumentException e) {
326     }
327 }
328
329 public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
```

**Listagem B-4 (continuação)****BobSerialDateTest.java**

```

334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(15, APRIL, 2006)));
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 //     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 //     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 //     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 //     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 //     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 //     assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 //     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 //     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 //     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370 //     assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 //     assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 //     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 //     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 //     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
377     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
378
379 //     assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
380 //     assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
381 //     assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
382 //     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
383 //     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
384 //     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
385     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
386
387     try {
388         getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
389         fail("Invalid day of week code should throw exception");
390     } catch (IllegalArgumentException e) {
391     }

```

**Listagem B-4 (continuação)****BobsSerialDateTest.java**

```

392
393     public void testEndOfCurrentMonth() throws Exception {
394         SerialDate d = SerialDate.createInstance(2);
395         assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396         assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397         assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398         assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399         assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400         assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401         assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402         assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403         assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404         assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405         assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406         assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407         assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408     }
409
410     public void testWeekInMonthToString() throws Exception {
411         assertEquals("First", weekInMonthToString(FIRST_WEEK_IN_MONTH));
412         assertEquals("Second", weekInMonthToString(SECOND_WEEK_IN_MONTH));
413         assertEquals("Third", weekInMonthToString(THIRD_WEEK_IN_MONTH));
414         assertEquals("Fourth", weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415         assertEquals("Last", weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417     //todo    try {
418     //        weekInMonthToString(-1);
419     //        fail("Invalid week code should throw exception");
420     //    } catch (IllegalArgumentException e) {
421     //    }
422     }
423
424     public void testRelativeToString() throws Exception {
425         assertEquals("Preceding", relativeToString(PRECEDING));
426         assertEquals("Nearest", relativeToString(NEAREST));
427         assertEquals("Following", relativeToString(FOLLOWING));
428
429     //todo    try {
430     //        relativeToString(-1000);
431     //        fail("Invalid relative code should throw exception");
432     //    } catch (IllegalArgumentException e) {
433     //    }
434     }
435
436     public void testCreateInstanceFromDDMMYY() throws Exception {
437         SerialDate date = createInstance(1, JANUARY, 1900);
438         assertEquals(1, date.getDayOfMonth());
439         assertEquals(JANUARY, date.getMonth());
440         assertEquals(1900, date.getYear());
441         assertEquals(2, date.toSerial());
442     }
443
444     public void testCreateInstanceFromSerial() throws Exception {
445         assertEquals(d(1, JANUARY, 1900), createInstance(2));
446         assertEquals(d(1, JANUARY, 1901), createInstance(367));
447     }
448
449     public void testCreateInstanceFromJavaDate() throws Exception {
450         assertEquals(d(1, JANUARY, 1900),
451                     createInstance(new GregorianCalendar(1900, 0, 1).getTime()));
452         assertEquals(d(1, JANUARY, 2006),
453                     createInstance(new GregorianCalendar(2006, 0, 1).getTime()));

```

**Listagem B-4 (continuação)****BobsSerialDateTest.java**

```

452 }
453
454 public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456 }
457 }
```

**Listing B-5****SpreadsheetDate.java**

```

1  /*
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * -----
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * SpreadsheetDate.java
29 * -----
30 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s):   -;
34 *
35 * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36 *
37 * Changes
38 * -----
39 * 11-Oct-2001 : Version 1 (DG);
40 * 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
41 * 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
42 *                 Fixed a bug in calculating day, month and year from serial
43 *                 number (DG);
44 * 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
45 *                 month and year. Thanks to Trevor Hills for the report (DG);
46 * 29-May-2002 : Added equals(Object) method (SourceForge ID 556850) (DG);
47 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
48 * 13-Mar-2003 : Implemented Serializable (DG);
49 * 04-Sep-2003 : Completed isInRange() methods (DG);
50 * 05-Sep-2003 : Implemented Comparable (DG);
51 * 21-Oct-2003 : Added hashCode() method (DG);
```

**Listing B-5 (continuação)**  
**SpreadsheetDate.java**

```
52  *
53  */
54
55 package org.jfree.date;
56
57 import java.util.Calendar;
58 import java.util.Date;
59
60 /**
61 * Represents a date using an integer, in a similar fashion to the
62 * implementation in Microsoft Excel. The range of dates supported is
63 * 1-Jan-1900 to 31-Dec-9999.
64 * <P>
65 * Be aware that there is a deliberate bug in Excel that recognises the year
66 * 1900 as a leap year when in fact it is not a leap year. You can find more
67 * information on the Microsoft website in article Q181370:
68 * <P>
69 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70 * <P>
71 * Excel uses the convention that 1-Jan-1900 = 1. This class uses the
72 * convention 1-Jan-1900 = 2.
73 * The result is that the day number in this class will be different to the
74 * Excel figure for January and February 1900...but then Excel adds in an extra
75 * day (29-Feb-1900 which does not actually exist!) and from that point forward
76 * the day numbers will match.
77 *
78 * @author David Gilbert
79 */
80 public class SpreadsheetDate extends SerialDate {
81
82     /** For serialization. */
83     private static final long serialVersionUID = -2039586705374454461L;
84
85     /**
86      * The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
87      * 2958465).
88      */
89     private int serial;
90
91     /** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
92     private int day;
93
94     /** The month of the year (1 to 12). */
95     private int month;
96
97     /** The year (1900 to 9999). */
98     private int year;
99
100    /** An optional description for the date. */
101    private String description;
102
103    /**
104     * Creates a new date instance.
105     *
106     * @param day the day (in the range 1 to 28/29/30/31).
107     * @param month the month (in the range 1 to 12).
108     * @param year the year (in the range 1900 to 9999).
109     */
110    public SpreadsheetDate(final int day, final int month, final int year) {
111
```

**Listing B-5 (continuação)****SpreadsheetDate.java**

```

112     if ((year >= 1900) && (year <= 9999)) {
113         this.year = year;
114     }
115     else {
116         throw new IllegalArgumentException(
117             "The 'year' argument must be in range 1900 to 9999.");
118     }
119 }
120
121     if ((month >= MonthConstants.JANUARY)
122         && (month <= MonthConstants.DECEMBER)) {
123         this.month = month;
124     }
125
126     else {
127         throw new IllegalArgumentException(
128             "The 'month' argument must be in the range 1 to 12.");
129     }
130
131     if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132         this.day = day;
133     }
134     else {
135         throw new IllegalArgumentException("Invalid 'day' argument.");
136     }
137
138 // the serial number needs to be synchronised with the day-month-year...
139 this.serial = calcSerial(day, month, year);
140
141 this.description = null;
142 }
143
144 /**
145 * Standard constructor - creates a new date object representing the
146 * specified day number (which should be in the range 2 to 2958465).
147 *
148 * @param serial the serial number for the day (range: 2 to 2958465).
149 */
150
151 public SpreadsheetDate(final int serial) {
152
153     if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154         this.serial = serial;
155     }
156     else {
157         throw new IllegalArgumentException(
158             "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159     }
160
161 // the day-month-year needs to be synchronised with the serial number...
162 calcDayMonthYear();
163 }
164
165 /**
166 * Returns the description that is attached to the date. It is not
167 * required that a date have a description, but for some applications it
168 * is useful.
169 *
170 * @return The description that is attached to the date.

```

**Listing B-5 (continuação)**  
**SpreadsheetDate.java**

```
172     */
173     public String getDescription() {
174         return this.description;
175     }
176
177     /**
178      * Sets the description for the date.
179      *
180      * @param description the description for this date (<code>null</code>
181      *                      permitted).
182      */
183     public void setDescription(final String description) {
184         this.description = description;
185     }
186
187     /**
188      * Returns the serial number for the date, where 1 January 1900 = 2
189      * (this corresponds, almost, to the numbering system used in Microsoft
190      * Excel for Windows and Lotus 1-2-3).
191      *
192      * @return The serial number of this date.
193      */
194     public int toSerial() {
195         return this.serial;
196     }
197
198     /**
199      * Returns a <code>java.util.Date</code> equivalent to this date.
200      *
201      * @return The date.
202      */
203     public Date toDate() {
204         final Calendar calendar = Calendar.getInstance();
205         calendar.set(getFullYear(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206         return calendar.getTime();
207     }
208
209     /**
210      * Returns the year (assume a valid range of 1900 to 9999).
211      *
212      * @return The year.
213      */
214     public int getFullYear() {
215         return this.year;
216     }
217
218     /**
219      * Returns the month (January = 1, February = 2, March = 3).
220      *
221      * @return The month of the year.
222      */
223     public int getMonth() {
224         return this.month;
225     }
226
227     /**
228      * Returns the day of the month.
229      *
230      * @return The day of the month.
```

**Listing B-5 (continuação)**  
**SpreadsheetDate.java**

```
231     */
232     public int getDayOfMonth() {
233         return this.day;
234     }
235
236     /**
237      * Returns a code representing the day of the week.
238      * <P>
239      * The codes are defined in the {@link SerialDate} class as:
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
242      * <code>SATURDAY</code>.
243      *
244      * @return A code representing the day of the week.
245      */
246     public int getDayOfWeek() {
247         return (this.serial + 6) % 7 + 1;
248     }
249
250     /**
251      * Tests the equality of this date with an arbitrary object.
252      * <P>
253      * This method will return true ONLY if the object is an instance of the
254      * {@link SerialDate} base class, and it represents the same day as this
255      * {@link SpreadsheetDate}.
256      *
257      * @param object the object to compare (<code>null</code> permitted).
258      *
259      * @return A boolean.
260      */
261     public boolean equals(final Object object) {
262
263         if (object instanceof SerialDate) {
264             final SerialDate s = (SerialDate) object;
265             return (s.toSerial() == this.toSerial());
266         }
267         else {
268             return false;
269         }
270     }
271
272     /**
273      * Returns a hash code for this object instance.
274      *
275      * @return A hash code.
276      */
277     public int hashCode() {
278         return toSerial();
279     }
280
281     /**
282      * Returns the difference (in days) between this date and the specified
283      * 'other' date.
284      *
285      * @param other the date being compared to.
286      *
287      * @return The difference (in days) between this date and the specified
288      *         'other' date.
289      */
290 
```

**Listing B-5 (continuação)**  
**SpreadsheetDate.java**

```
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Implements the method required by the Comparable interface.
297      *
298      * @param other the other object (usually another SerialDate).
299      *
300      * @return A negative integer, zero, or a positive integer as this object
301      *         is less than, equal to, or greater than the specified object.
302      */
303     public int compareTo(final Object other) {
304         return compare((SerialDate) other);
305     }
306
307     /**
308      * Returns true if this SerialDate represents the same date as the
309      * specified SerialDate.
310      *
311      * @param other the date being compared to.
312      *
313      * @return <code>true</code> if this SerialDate represents the same date as
314      *         the specified SerialDate.
315      */
316     public boolean isOn(final SerialDate other) {
317         return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Returns true if this SerialDate represents an earlier date compared to
322      * the specified SerialDate.
323      *
324      * @param other the date being compared to.
325      *
326      * @return <code>true</code> if this SerialDate represents an earlier date
327      *         compared to the specified SerialDate.
328      */
329     public boolean isBefore(final SerialDate other) {
330         return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Returns true if this SerialDate represents the same date as the
335      * specified SerialDate.
336      *
337      * @param other the date being compared to.
338      *
339      * @return <code>true</code> if this SerialDate represents the same date
340      *         as the specified SerialDate.
341      */
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Returns true if this SerialDate represents the same date as the
348      * specified SerialDate.
349      *
350      * @param other the date being compared to.
351      *
```

**Listing B-5 (continuação)**  
**SpreadsheetDate.java**

```
352     * @return <code>true</code> if this SerialDate represents the same date
353     *         as the specified SerialDate.
354     */
355    public boolean isAfter(final SerialDate other) {
356        return (this.serial > other.toSerial());
357    }
358
359    /**
360     * Returns true if this SerialDate represents the same date as the
361     * specified SerialDate.
362     *
363     * @param other the date being compared to.
364     *
365     * @return <code>true</code> if this SerialDate represents the same date as
366     *         the specified SerialDate.
367     */
368    public boolean isOnOrAfter(final SerialDate other) {
369        return (this.serial >= other.toSerial());
370    }
371
372    /**
373     * Returns <code>true</code> if this {@link SerialDate} is within the
374     * specified range (INCLUSIVE). The date order of d1 and d2 is not
375     * important.
376     *
377     * @param d1 a boundary date for the range.
378     * @param d2 the other boundary date for the range.
379     *
380     * @return A boolean.
381     */
382    public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383        return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384    }
385
386    /**
387     * Returns true if this SerialDate is within the specified range (caller
388     * specifies whether or not the end-points are included). The order of d1
389     * and d2 is not important.
390     *
391     * @param d1 one boundary date for the range.
392     * @param d2 a second boundary date for the range.
393     * @param include a code that controls whether or not the start and end
394     *               dates are included in the range.
395     *
396     * @return <code>true</code> if this SerialDate is within the specified
397     *         range.
398     */
399    public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                           final int include) {
401        final int s1 = d1.toSerial();
402        final int s2 = d2.toSerial();
403        final int start = Math.min(s1, s2);
404        final int end = Math.max(s1, s2);
405
406        final int s = toSerial();
407        if (include == SerialDate.INCLUDE_BOTH) {
408            return (s >= start && s <= end);
409        }
410        else if (include == SerialDate.INCLUDE_FIRST) {
411            return (s >= start && s < end);
```

**Listing B-5 (continuação)****SpreadsheetDate.java**

```
412         }
413     else if (include == SerialDate.INCLUDE_SECOND) {
414         return (s > start && s <= end);
415     }
416     else {
417         return (s > start && s < end);
418     }
419 }
420 /**
421 * Calculate the serial number from the day, month and year.
422 * <P>
423 * 1-Jan-1900 = 2.
424 *
425 * @param d the day.
426 * @param m the month.
427 * @param y the year.
428 *
429 * @return the serial number from the day, month and year.
430 */
431 private int calcSerial(final int d, final int m, final int y) {
432     final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
433     int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
434     if (m > MonthConstants.FEBRUARY) {
435
436         if (SerialDate.isLeapYear(y)) {
437             mm = mm + 1;
438         }
439     }
440     final int dd = d;
441     return yy + mm + dd + 1;
442 }
443 /**
444 * Calculate the day, month and year from the serial number.
445 */
446 private void calcDayMonthYear() {
447
448     // get the year from the serial date
449     final int days = this.serial - SERIAL_LOWER_BOUND;
450     // overestimated because we ignored leap days
451     final int overestimatedYYYY = 1900 + (days / 365);
452     final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
453     final int nonleapdays = days - leaps;
454     // underestimated because we overestimated years
455     int underestimatedYYYY = 1900 + (nonleapdays / 365);
456
457     if (underestimatedYYYY == overestimatedYYYY) {
458         this.year = underestimatedYYYY;
459     }
460     else {
461         int ssl = calcSerial(1, 1, underestimatedYYYY);
462         while (ssl <= this.serial) {
463             underestimatedYYYY = underestimatedYYYY + 1;
464             ssl = calcSerial(1, 1, underestimatedYYYY);
465         }
466         this.year = underestimatedYYYY - 1;
467     }
468
469     final int ss2 = calcSerial(1, 1, this.year);
470     int[] daysToEndOfPrecedingMonth
```

### **Listing B-5 (continuação)**

#### **SpreadsheetDate.java**

```

473         = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475     if (isLeapYear(this.year)) {
476         daysToEndOfPrecedingMonth
477             = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478     }
479
480     // get the month from the serial date
481     int mm = 1;
482     int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483     while (sss < this.serial) {
484         mm = mm + 1;
485         sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486     }
487     this.month = mm - 1;
488
489     // what's left is d(+1);
490     this.day = this.serial - ss2
491         - daysToEndOfPrecedingMonth[this.month] + 1;
492
493 }
494
495 }
```

### **Listagem B-6**

#### **RelativeDayOfWeekRule.java**

```

1  /*
2  * =====
3  * JCommon : a free general purpose class library for the Java(tm) platform
4  * =====
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 */
28 * -----
29 * RelativeDayOfWeekRule.java
30 *
31 * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
32 *
33 * Original Author: David Gilbert (for Object Refinery Limited);
34 * Contributor(s): -;
```

**Listagem B-6 (continuação)****RelativeDayOfWeekRule.java**

```
35 * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $  
36 *  
37 * Changes (from 26-Oct-2001)  
38 *-----  
39 * 26-Oct-2001 : Changed package to com.jrefinery.date.*;  
40 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);  
41 *  
42 */  
43  
44 package org.jfree.date;  
45  
46 /**  
47 * An annual date rule that returns a date for each year based on (a) a  
48 * reference rule; (b) a day of the week; and (c) a selection parameter  
49 * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).  
50 * <P>  
51 * For example, Good Friday can be specified as 'the Friday PRECEDING Easter  
52 * Sunday'.  
53 *  
54 * @author David Gilbert  
55 */  
56 public class RelativeDayOfWeekRule extends AnnualDateRule {  
57  
58     /** A reference to the annual date rule on which this rule is based. */  
59     private AnnualDateRule subrule;  
60  
61     /**  
62      * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).  
63      */  
64     private int dayOfWeek;  
65  
66     /** Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). */  
67     private int relative;  
68  
69     /**  
70      * Default constructor - builds a rule for the Monday following 1 January.  
71      */  
72     public RelativeDayOfWeekRule() {  
73         this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);  
74     }  
75  
76     /**  
77      * Standard constructor - builds rule based on the supplied sub-rule.  
78      *  
79      * @param subrule the rule that determines the reference date.  
80      * @param dayOfWeek the day-of-the-week relative to the reference date.  
81      * @param relative indicates *which* day-of-the-week (preceding, nearest  
82      *                  or following).  
83      */  
84     public RelativeDayOfWeekRule(final AnnualDateRule subrule,  
85                                     final int dayOfWeek, final int relative) {  
86         this.subrule = subrule;  
87         this.dayOfWeek = dayOfWeek;  
88         this.relative = relative;  
89     }  
90  
91     /**  
92      * Returns the sub-rule (also called the reference rule).  
93      *  
94      * @return The annual date rule that determines the reference date for this  
95      *        rule.
```

**Listagem B-6 (continuação)****RelativeDayOfWeekRule.java**

```
91     /**
92      * Returns the sub-rule (also called the reference rule).
93      *
94      * @return The annual date rule that determines the reference date for this
95      *         rule.
96      */
97     public AnnualDateRule getSubrule() {
98         return this.subrule;
99     }
100
101    /**
102     * Sets the sub-rule.
103     *
104     * @param subrule the annual date rule that determines the reference date
105     *                 for this rule.
106     */
107    public void setSubrule(final AnnualDateRule subrule) {
108        this.subrule = subrule;
109    }
110
111    /**
112     * Returns the day-of-the-week for this rule.
113     *
114     * @return the day-of-the-week for this rule.
115     */
116    public int getDayOfWeek() {
117        return this.dayOfWeek;
118    }
119
120    /**
121     * Sets the day-of-the-week for this rule.
122     *
123     * @param dayOfWeek the day-of-the-week (SerialDate.MONDAY,
124     *                  SerialDate.TUESDAY, and so on).
125     */
126    public void setDayOfWeek(final int dayOfWeek) {
127        this.dayOfWeek = dayOfWeek;
128    }
129
130    /**
131     * Returns the 'relative' attribute, that determines *which*
132     * day-of-the-week we are interested in (SerialDate.PRECEDING,
133     * SerialDate.NEAREST or SerialDate.FOLLOWING).
134     *
135     * @return The 'relative' attribute.
136     */
137    public int getRelative() {
138        return this.relative;
139    }
140
141    /**
142     * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
143     * SerialDate.FOLLOWING).
144     *
145     * @param relative determines *which* day-of-the-week is selected by this
146     *                  rule.
147     */
148    public void setRelative(final int relative) {
149        this.relative = relative;
150    }
151 }
```

**Listagem B-6 (continuação)****RelativeDayOfWeekRule.java**

```
152     /**
153      * Creates a clone of this rule.
154      *
155      * @return a clone of this rule.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167      * Returns the date generated by this rule, for the specified year.
168      *
169      * @param year the year (1900 <= year <= 9999).
170      *
171      * @return The date generated by the rule for the given year (possibly
172      *         <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // check argument...
177         if !(year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // calculate the date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                     result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                         base);
192                     break;
193                 case(SerialDate.NEAREST):
194                     result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                         base);
196                     break;
197                 case(SerialDate.FOLLOWING):
198                     result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                         base);
200                     break;
201                 default:
202                     break;
203             }
204         }
205         return result;
206     }
207
208 }
```

**Listagem B-7 (continuação)****DayDate.java (Final)**

```
1  /*
2   * JCommon : a free general purpose class library for the Java(tm) platform
3   * -----
4   *
5   * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6   */
7  package org.jfree.date;
8
9  import java.io.Serializable;
10 import java.util.*;
11
12 /**
13  * An abstract class that represents immutable dates with a precision of
14  * one day. The implementation will map each date to an integer that
15  * represents an ordinal number of days from some fixed origin.
16  *
17  * Why not just use java.util.Date? We will, when it makes sense. At times,
18  * java.util.Date can be *too* precise - it represents an instant in time,
19  * accurate to 1/1000th of a second (with the date itself depending on the
20  * time-zone). Sometimes we just want to represent a particular day (e.g. 21
21  * January 2015) without concerning ourselves about the time of day, or the
22  * time-zone, or anything else. That's what we've defined DayDate for.
23  *
24  * Use DayDateFactory.makeText to create an instance.
25  *
26  * @author David Gilbert
27  * @author Robert C. Martin did a lot of refactoring.
28  */
29
30 public abstract class DayDate implements Comparable, Serializable {
31     public abstract int getOrdinalDay();
32     public abstract int getYear();
33     public abstract Month getMonth();
34     public abstract int getDayOfMonth();
35
36     protected abstract Day getDayOfWeekForOrdinalZero();
37
38     public DayDate plusDays(int days) {
39         return DayDateFactory.makeText(getOrdinalDay() + days);
40     }
41
42     public DayDate plusMonths(int months) {
43         int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
44         int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
45         int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
46         int resultYear = resultMonthAndYearAsOrdinal / 12;
47         int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
48         Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
49         int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
50         return DayDateFactory.makeText(resultDay, resultMonth, resultYear);
51     }
52
53     public DayDate plusYears(int years) {
54         int resultYear = getYear() + years;
55         int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
56         return DayDateFactory.makeText(resultDay, getMonth(), resultYear);
57     }
58
59     private int correctLastDayOfMonth(int day, Month month, int year) {
60         int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
61         if (day > lastDayOfMonth)
```

**Listagem B-7 (continuação)****DayDate.java (Final)**

```
92         day = lastDayOfMonth;
93         return day;
94     }
95
96     public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
97         int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
98         if (offsetToTarget >= 0)
99             offsetToTarget -= 7;
100        return plusDays(offsetToTarget);
101    }
102
103    public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104        int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105        if (offsetToTarget <= 0)
106            offsetToTarget += 7;
107        return plusDays(offsetToTarget);
108    }
109
110    public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111        int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112        int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113        int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115        if (offsetToFutureTarget > 3)
116            return plusDays(offsetToPreviousTarget);
117        else
118            return plusDays(offsetToFutureTarget);
119    }
120
121    public DayDate getEndOfMonth() {
122        Month month = getMonth();
123        int year = getYear();
124        int lastDay = DateUtil.lastDayOfMonth(month, year);
125        return DayDateFactory.makeDate(lastDay, month, year);
126    }
127
128    public Date toDate() {
129        final Calendar calendar = Calendar.getInstance();
130        int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131        calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132        return calendar.getTime();
133    }
134
135    public String toString() {
136        return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137    }
138
139    public Day getDayOfWeek() {
140        Day startingDay = getDayOfWeekForOrdinalZero();
141        int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142        int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143        return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144    }
145
146    public int daysSince(DayDate date) {
147        return getOrdinalDay() - date.getOrdinalDay();
148    }
149
150    public boolean isOn(DayDate other) {
151        return getOrdinalDay() == other.getOrdinalDay();
152    }
153
```

**Listagem B-7 (continuação)****DayDate.java (Final)**

```

154     public boolean isBefore(DayDate other) {
155         return getOrdinalDay() < other.getOrdinalDay();
156     }
157
158     public boolean isOnOrBefore(DayDate other) {
159         return getOrdinalDay() <= other.getOrdinalDay();
160     }
161
162     public boolean isAfter(DayDate other) {
163         return getOrdinalDay() > other.getOrdinalDay();
164     }
165
166     public boolean isOnOrAfter(DayDate other) {
167         return getOrdinalDay() >= other.getOrdinalDay();
168     }
169
170     public boolean isInRange(DayDate d1, DayDate d2) {
171         return isInRange(d1, d2, DateInterval.CLOSED);
172     }
173
174     public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175         int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176         int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177         return interval.isIn(getOrdinalDay(), left, right);
178     }
179 }
```

**Listagem B-8****Month.java (Final)**

```

1 package org.jfree.date;
2
3 import java.text.DateFormatSymbols;
4
5 public enum Month {
6     JANUARY(1), FEBRUARY(2), MARCH(3),
7     APRIL(4), MAY(5), JUNE(6),
8     JULY(7), AUGUST(8), SEPTEMBER(9),
9     OCTOBER(10), NOVEMBER(11), DECEMBER(12);
10    private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11    private static final int[] LAST_DAY_OF_MONTH =
12        {0, 31, 28, 31, 30, 31, 30, 31, 30, 31, 31};
13
14    private int index;
15
16    Month(int index) {
17        this.index = index;
18    }
19
20    public static Month fromInt(int monthIndex) {
21        for (Month m : Month.values()) {
22            if (m.index == monthIndex)
23                return m;
24        }
25        throw new IllegalArgumentException("Invalid month index " + monthIndex);
26    }
27
28    public int lastDay() {
29        return LAST_DAY_OF_MONTH[index];
30    }
31 }
```

**Listagem B-8 (continuação)****Month.java (Final)**

```
32 public int quarter() {
33     return 1 + (index - 1) / 3;
34 }
35
36 public String toString() {
37     return dateFormatSymbols.getMonths()[index - 1];
38 }
39
40 public String toShortString() {
41     return dateFormatSymbols.getShortMonths()[index - 1];
42 }
43
44 public static Month parse(String s) {
45     s = s.trim();
46     for (Month m : Month.values())
47         if (m.matches(s))
48             return m;
49
50     try {
51         return fromInt(Integer.parseInt(s));
52     }
53     catch (NumberFormatException e) {}
54     throw new IllegalArgumentException("Invalid month " + s);
55 }
56
57 private boolean matches(String s) {
58     return s.equalsIgnoreCase(toString()) ||
59            s.equalsIgnoreCase(toShortString());
60 }
61
62 public intToInt() {
63     return index;
64 }
65 }
```

**Listagem B-9****Day.java (Final)**

```
1 package org.jfree.date;
2
3 import java.util.Calendar;
4 import java.text.DateFormatSymbols;
5
6 public enum Day {
7     MONDAY(Calendar.MONDAY),
8     TUESDAY(Calendar.TUESDAY),
9     WEDNESDAY(Calendar.WEDNESDAY),
10    THURSDAY(Calendar.THURSDAY),
11    FRIDAY(Calendar.FRIDAY),
12    SATURDAY(Calendar.SATURDAY),
13    SUNDAY(Calendar.SUNDAY);
14
15    private final int index;
16    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18    Day(int day) {
19        index = day;
20    }
21}
```

**Listagem B-9 (continuação)****Day.java (Final)**

```

22     public static Day fromInt(int index) throws IllegalArgumentException {
23         for (Day d : Day.values())
24             if (d.index == index)
25                 return d;
26         throw new IllegalArgumentException(
27             String.format("Illegal day index: %d.", index));
28     }
29
30     public static Day parse(String s) throws IllegalArgumentException {
31         String[] shortWeekdayNames =
32             dateSymbols.getShortWeekdays();
33         String[] weekDayNames =
34             dateSymbols.getWeekdays();
35
36         s = s.trim();
37         for (Day day : Day.values()) {
38             if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39                 s.equalsIgnoreCase(weekDayNames[day.index])) {
40                 return day;
41             }
42         }
43         throw new IllegalArgumentException(
44             String.format("%s is not a valid weekday string", s));
45     }
46
47     public String toString() {
48         return dateSymbols.getWeekdays()[index];
49     }
50
51     public int toInt() {
52         return index;
53     }
54 }
```

**Listagem B-10****DateInterval.java (Final)**

```

1 package org.jfree.date;
2
3 public enum DateInterval {
4     OPEN {
5         public boolean isIn(int d, int left, int right) {
6             return d > left && d < right;
7         }
8     },
9     CLOSED_LEFT {
10        public boolean isIn(int d, int left, int right) {
11            return d >= left && d < right;
12        }
13    },
14    CLOSED_RIGHT {
15        public boolean isIn(int d, int left, int right) {
16            return d > left && d <= right;
17        }
18    },
19    CLOSED {
20        public boolean isIn(int d, int left, int right) {
21            return d >= left && d <= right;
22        }
23    };
24
25    public abstract boolean isIn(int d, int left, int right);
26 }
```

**Listagem B-11****WeekInMonth.java (Final)**

```
1 package org.jfree.date;
2
3 public enum WeekInMonth {
4     FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
5     private final int index;
6
7     WeekInMonth(int index) {
8         this.index = index;
9     }
10
11    public int toInt() {
12        return index;
13    }
14 }
```

**Listagem B-12****WeekdayRange.java (Final)**

```
1 package org.jfree.date;
2
3 public enum WeekdayRange {
4     LAST, NEAREST, NEXT
5 }
```

**Listagem B-13****DateUtil.java (Final)**

```
1 package org.jfree.date;
2
3 import java.text.DateFormatSymbols;
4
5 public class DateUtil {
6     private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
7
8     public static String[] getMonthNames() {
9         return dateFormatSymbols.getMonths();
10    }
11
12    public static boolean isLeapYear(int year) {
13        boolean fourth = year % 4 == 0;
14        boolean hundredth = year % 100 == 0;
15        boolean fourHundredth = year % 400 == 0;
16        return fourth && (!hundredth || fourHundredth);
17    }
18
19    public static int lastDayOfMonth(Month month, int year) {
20        if (month == Month.FEBRUARY && isLeapYear(year))
21            return month.lastDay() + 1;
22        else
23            return month.lastDay();
24    }
25
26    public static int leapYearCount(int year) {
27        int leap4 = (year - 1896) / 4;
28        int leap100 = (year - 1800) / 100;
29        int leap400 = (year - 1600) / 400;
30        return leap4 - leap100 + leap400;
31    }
32 }
```

**Listagem B-14****DayDateFactory.java (Final)**

```
1 package org.jfree.date;
2
3 public abstract class DayDateFactory {
4     private static DayDateFactory factory = new SpreadsheetDateFactory();
5     public static void setInstance(DayDateFactory factory) {
6         DayDateFactory.factory = factory;
7     }
8
9     protected abstract DayDate _makeDate(int ordinal);
10    protected abstract DayDate _makeDate(int day, Month month, int year);
11    protected abstract DayDate _makeDate(int day, int month, int year);
12    protected abstract DayDate _makeDate(java.util.Date date);
13    protected abstract int _getMinimumYear();
14    protected abstract int _getMaximumYear();
15
16    public static DayDate makeDate(int ordinal) {
17        return factory._makeDate(ordinal);
18    }
19
20    public static DayDate makeDate(int day, Month month, int year) {
21        return factory._makeDate(day, month, year);
22    }
23
24    public static DayDate makeDate(int day, int month, int year) {
25        return factory._makeDate(day, month, year);
26    }
27
28    public static DayDate makeDate(java.util.Date date) {
29        return factory._makeDate(date);
30    }
31
32    public static int getMinimumYear() {
33        return factory._getMinimumYear();
34    }
35
36    public static int getMaximumYear() {
37        return factory._getMaximumYear();
38    }
39 }
```

**Listagem B-15****SpreadsheetDateFactory.java (Final)**

```
1 package org.jfree.date;
2
3 import java.util.*;
4
5 public class SpreadsheetDateFactory extends DayDateFactory {
6     public DayDate _makeDate(int ordinal) {
7         return new SpreadsheetDate(ordinal);
8     }
9
10    public DayDate _makeDate(int day, Month month, int year) {
11        return new SpreadsheetDate(day, month, year);
12    }
13
14    public DayDate _makeDate(int day, int month, int year) {
15        return new SpreadsheetDate(day, month, year);
16    }
}
```

**Listagem B-15****SpreadsheetDateFactory.java (Final)**

```
17
18 public DayDate _makeDate(Date date) {
19     final GregorianCalendar calendar = new GregorianCalendar();
20     calendar.setTime(date);
21     return new SpreadsheetDate(
22         calendar.get(Calendar.DATE),
23         Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24         calendar.get(Calendar.YEAR));
25 }
26
27 protected int _getMinimumYear() {
28     return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29 }
30
31 protected int _getMaximumYear() {
32     return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33 }
34 }
```

**Listagem B-16****SpreadsheetDate.java (Final)**

```
1 /**
2 * =====
3 * JCommon : a free general purpose class library for the Java(tm) platform
4 *
5 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6 */
7
8 */
9
10 package org.jfree.date;
11
12 import static org.jfree.date.Month.FEBRUARY;
13
14 import java.util.*;
15
16 /**
17 * Represents a date using an integer, in a similar fashion to the
18 * implementation in Microsoft Excel. The range of dates supported is
19 * 1-Jan-1900 to 31-Dec-9999.
20 * <p>
21 * Be aware that there is a deliberate bug in Excel that recognises the year
22 * 1900 as a leap year when in fact it is not a leap year. You can find more
23 * information on the Microsoft website in article Q181370:
24 * <p>
25 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
26 * <p/>
27 * Excel uses the convention that 1-Jan-1900 = 1. This class uses the
28 * convention 1-Jan-1900 = 2.
29 * The result is that the day number in this class will be different to the
30 * Excel figure for January and February 1900...but then Excel adds in an extra
31 * day (29-Feb-1900 which does not actually exist!) and from that point forward
32 * the day numbers will match.
33 * <br/>
34 * @author David Gilbert
35 */
36
37 public class SpreadsheetDate extends DayDate {
38     public static final int EARLIEST_DATE_ORDINAL = 2;      // 1/1/1900
39     public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
40 }
```

**Listagem B-16 (continuação)****SpreadsheetDate.java (Final)**

```
83 public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
84 public static final int MINIMUM_YEAR_SUPPORTED = 1900;
85 public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
86 static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
88 static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
90
91 private int ordinalDay;
92 private int day;
93 private Month month;
94 private int year;
95
96 public SpreadsheetDate(int day, Month month, int year) {
97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
98         throw new IllegalArgumentException(
99             "The 'year' argument must be in range " +
100             MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101    if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102        throw new IllegalArgumentException("Invalid 'day' argument.");
103
104    this.year = year;
105    this.month = month;
106
107    this.day = day;
108    ordinalDay = calcOrdinal(day, month, year);
109 }
110
111 public SpreadsheetDate(int day, int month, int year) {
112     this(day, Month.fromInt(month), year);
113 }
114
115 public SpreadsheetDate(int serial) {
116     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
117         throw new IllegalArgumentException(
118             "SpreadsheetDate: Serial must be in range 2 to 2958465.");
119
120     ordinalDay = serial;
121     calcDayMonthYear();
122 }
123
124 public int getOrdinalDay() {
125     return ordinalDay;
126 }
127
128 public int getYear() {
129     return year;
130 }
131
132 public Month getMonth() {
133     return month;
134 }
135
136 public int getDayOfMonth() {
137     return day;
138 }
139
140 protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
141
142 public boolean equals(Object object) {
143     if (!(object instanceof DayDate))
144         return false;
```

**Listagem B-16 (continuação)****SpreadsheetDate.java (Final)**

```
144     DayDate date = (DayDate) object;
145     return date.getOrdinalDay() == getOrdinalDay();
146   }
147 
148 
149   public int hashCode() {
150     return getOrdinalDay();
151   }
152 
153   public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155   }
156 
157   private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162       daysUpToMonth++;
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165   }
166 
167   private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172 
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177   }
178 
179   private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183       aMonth++;
184 
185     return Month.fromInt(aMonth - 1);
186   }
187 
188   private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190       return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192       return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193   }
194 
195   private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198       aYear++;
199 
200     return aYear - 1;
201   }
202 
203   private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
```

**Listagem B-16 (continuação)****SpreadsheetDate.java (Final)**

```
205  }
206
207 public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                               Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                               calendar.get(Calendar.YEAR));
213 }
214 }
215 }
```

---

# Apêndice C

---

## Referência Cruzada das Heurísticas

Referência cruzada de "odores" e heurísticas. Podem-se excluir todas as outras.

C1 .....	16-276, 16-279, 17-292
C2 .....	16-279, 16-285, 16-295, 17-292
C3 .....	16-283, 16-285, 16-288, 17-293
C4 .....	17-293
C5 .....	17-293
E1 .....	17-294
E2 .....	17-294
F1 .....	14-239, 17-295
F2 .....	17-295
F3 .....	17-295
F4 .....	14-289, 16-273, 16-285, 16-287, 16-288, 17-295
G1 .....	16-276, 17-295
G2 .....	16-273, 16-274, 17-296
G3 .....	16-274, 17-296
G4 .....	9-31, 16-279, 16-286, 16-291, 17-297
G5 .....	9-31, 16-279, 16-286, 16-291, 16-296, 17-297
G ...66-106, 16-280, 16-283, 16-284, 16-289, 16-293, 16-294, 16-296, 17-299	
G7 .....	16-281, 16-283, 17-300
G8 .....	16-283, 17-301
G9 .....	16-283, 16-285, 16-286, 16-287, 17-302
G10 .....	5-86, 15-264, 16-276, 16-284, 17-302
G11 .....	15-264, 16-284, 16-288, 16-292, 17-302
G12 .....	16-284, 16-285, 16-286, 16-287, 16-288, 16-295, 17-303

G13 .....	16-286, 16-288, 17-303
G14 .....	16-288, 16-292, 17-304
G15 .....	16-288, 17-305
G16 .....	16-289, 17-306
G17 .....	16-289, 17-307, 17-312
G18 .....	16-289, 16-290, 16-291, 17-308
G19 .....	16-290, 16-291, 16-292, 17-309
G20 .....	16-290, 17-309
G21 .....	16-291, 17-310
G22 .....	16-294, 17-322
G23 .....	?-44, 14-239, 16-295, 17-313
G24 .....	16-296, 17-313
G25 .....	16-296, 17-314
G26 .....	17-316
G27 .....	17-316
G28 .....	15-262, 17-317
G29 .....	15-262, 17-317
G30 .....	15-263, 17-317
G31 .....	15-264, 17-318
G32 .....	15-265, 17-319
G33 .....	15-265, 15-266, 17-320
G34 .....	1-40, 6-106, 17-321
G35 .....	5-90, 17-323
G36 .....	6-103, 17-324
J1 .....	16-276, 17-325
J2 .....	16-278, 16-285, 17-326
J3 .....	16-283, 16-285, 17-327
N1 .....	15-264, 16-277, 16-279, 16-282, 16-287, 16-288, 16-289, 16-290, 16-294, 16-296, 17-328
N2 .....	16-277, 17-330
N3 .....	16-284, 16-288, 17-331
N4 .....	15-263, 16-291, 17-332
N5 .....	2-26, 14-221, 15-262, 17-332
N6 .....	15-261, 17-333
N7 .....	15-263, 17-333
T1 .....	16-273, 16-274, 17-334
T2 .....	16-273, 17-334
T3 .....	16-274, 17-334
T4 .....	17-334
T5 .....	16-274, 16-275, 17-335
T6 .....	16-275, 17-335
T7 .....	16-275, 17-335
T8 .....	16-275, 17-335
T9 .....	17-336

---

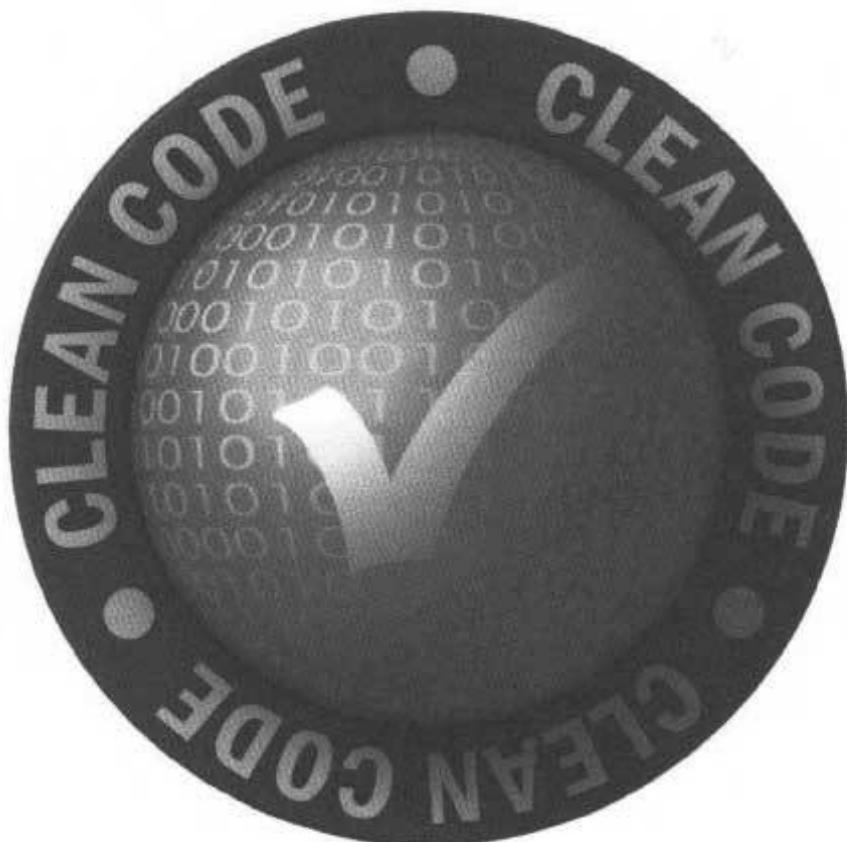
# Epílogo

---

Em 2005, durante a conferência sobre o Agile em Denver, EUA, Elisabeth Hendrickson<sup>1</sup> me ofereceu uma fitinha verde de punho parecida àquela que Lance Armstrong tornou tão popular. Nela estava escrito “Obcecado por testes”. Com satisfação, amarrei-a em meu punho e a usei com orgulho. Desde que Kent Beck me ensinou, em 1999, sobre o TDD, fiquei, de fato, obcecado pelo desenvolvimento dirigido a testes.

Porém, algo estranho aconteceu. Descobri que eu não poderia retirar minha fitinha do punho. Não por que ela estivesse presa a ele, mas por estar moralmente presa. A fita fazia uma afirmação evidente sobre minha ética profissional. Era visível meu comprometimento em criar o melhor código que eu pudesse. Tirá-la seria como trair tal ética e comprometimento.

Devido a isso, ela ainda está em meu punho. Quando escrevo um código, a vejo através de minha visão periférica. Ela é um lembrete constante da promessa que fiz a mim mesmo para criar códigos limpos.



Por outro lado, uma tríade que não é nem tão perigosa assim é a `assertEquals(1.0, amount, .001)`. Embora ainda seja preciso lê-la duas vezes, é uma que vale a pena. Sempre é bom uma dica de que a igualdade de valores do tipo ponto flutuante é relativa.

## Objetos como Parâmetros

Quando uma função parece precisar de mais de dois ou três parâmetros, é provável que alguns deles podem ser colocados em uma classe própria. Considere, por exemplo, a diferença entre as duas declarações seguintes:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Reducir o número de parâmetros através da criação de objetos a partir deles pode parecer uma trapaça, mas não é. Quando grupos de variáveis são passados juntos, como `x` e `y` no exemplo acima, é mais provável que sejam parte de um conceito que mereça um nome só para ele.

## Listas como Parâmetros

Às vezes, queremos passar um número variável de parâmetros para uma função. Considere, por exemplo, o método `String.format`:

```
String.format("%s worked %.2f hours.", name, hours);
```

Se os parâmetros variáveis forem todos tratados da mesma forma, como no exemplo acima, então eles serão equivalentes a um único parâmetro do tipo lista. Devido a isso, `String.format` é realmente díade. Na verdade, sua declaração, como mostra abaixo, é explicitamente díade.

```
public String format(String format, Object... args)
```

Portanto, todas as mesmas regras são aplicáveis. As funções que recebem argumentos variáveis podem ser mônades, díades ou mesmo tríades. Mas seria um erro passar mais parâmetros do que isso.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

## Verbos e Palavras-Chave

Escolher bons nomes para funções pode ir desde explicar o propósito da função à ordem e a finalidade dos parâmetros. No caso de uma mònade, a função e o parâmetro devem formar um belo par verbo/substantivo. Por exemplo, escrever (`nome`) é bastante claro. Seja o que for esse “`nome`”, ele será “escrito”. Um nome ainda melhor seria “`escreverCampo (nome)`”, que nos diz que “`nome`” é um “campo”.

Este último é um exemplo do formato *palavra-chave* do nome de uma função. Ao usar este