

**Listagem 14-9 (continuação)****Args.java (Booleano apenas)**

```
        return true;
    }

    private void parseSchemaElement(String element) {
        if (element.length() == 1) {
            parseBooleanSchemaElement(element);
        }
    }

    private void parseBooleanSchemaElement(String element) {
        char c = element.charAt(0);
        if (Character.isLetter(c)) {
            booleanArgs.put(c, false);
        }
    }

    private boolean parseArguments() {
        for (String arg : args)
            parseArgument(arg);
        return true;
    }

    private void parseArgument(String arg) {
        if (arg.startsWith("-"))
            parseElements(arg);
    }

    private void parseElements(String arg) {
        for (int i = 1; i < arg.length(); i++)
            parseElement(arg.charAt(i));
    }

    private void parseElement(char argChar) {
        if (isBoolean(argChar)) {
            numberOfArguments++;
            setBooleanArg(argChar, true);
        } else
            unexpectedArguments.add(argChar);
    }

    private void setBooleanArg(char argChar, boolean value) {
        booleanArgs.put(argChar, value);
    }

    private boolean isBoolean(char argChar) {
        return booleanArgs.containsKey(argChar);
    }

    public int cardinality() {
        return numberOfArguments;
    }

    public String usage() {
        if (schema.length() > 0)
            return "-["+schema+"]";
```

**Listagem 14-9 (continuação)****Args.java (Booleano apenas)**

```

        else
            return "";
    }

    public String errorMessage() {
        if (unexpectedArguments.size() > 0) {
            return unexpectedArgumentMessage();
        } else
            return "";
    }

    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer("Argument(s) -");
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(" unexpected.");
        return message.toString();
    }

    public boolean getBoolean(char arg) {
        return booleanArgs.get(arg);
    }
}

```

Embora você possa encontrar muitas formas de criticar esse código, ele não está tão ruim assim. Ele está compacto, simples e fácil de entender. Entretanto, dentro dele é fácil identificar as “sementes” que bagunçarão o código. Está bem claro como ele se tornará uma grande zona. Note que a bagunça futura possui apenas mais dois tipos de parâmetros do que estes: `String` e `integer`. Só essa adição tem um impacto negativo enorme no código. Ele o transformou de algo que seria razoavelmente passível de manutenção em algo confuso cheio de bugs.

Inseri esses dois tipos de parâmetro de modo gradual. Primeiro, adicionei o parâmetro do tipo `String`, que resultou no seguinte:

**Listagem 14-10****Args.java (Booleano e String)**

```

package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();

```

**Listagem 14-10 (continuação)****Args.java (Booleano e String)**

```
private Map<Character, String> stringArgs =
    new HashMap<Character, String>();
private Set<Character> argsFound = new HashSet<Character>();
private int currentArgument;
private char errorArgument = '\0';

enum ErrorCode {
    OK, MISSING_STRING}

private ErrorCode errorCode = ErrorCode.OK;

public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
}

private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
        return true;
    parseSchema();
    parseArguments();
    return valid;
}

private boolean parseSchema() throws ParseException {
    for (String element : schema.split(","))
        if (element.length() > 0) {
            String trimmedElement = element.trim();
            parseSchemaElement(trimmedElement);
        }
    return true;
}

private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElement(elementId);
    if (isBooleanSchemaElement(elementTail))
        parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
}

private void validateSchemaElement(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}

private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
}
```

**Listagem 14-10 (continuação)****Args.java (Booleano e String)**

```

private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("");
}

private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}

private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
}

private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

private void parseArgument(String arg) {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        valid = false;
    }
}

private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
        setBooleanArg(argChar, true);
    else if (isString(argChar))
        setStringArg(argChar, "");
    else
        set = false;

    return set;
}

private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {

```

**Listagem 14-10 (continuação)****Args.java (Booleano e String)**

```
    stringArgs.put(argChar, args[currentArgument]);
} catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgument = argChar;
    errorCode = ErrorCode.MISSING_STRING;
}
}

private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
    } else
        switch (errorCode) {
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.",
                                     errorArgument);
            case OK:
                throw new Exception("TILT: Should not get here.");
        }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
}
```

**Listagem 14-10 (continuação)****Args.java (Booleano e String)**

```
public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
}

private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
}

public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
}

private String blankIfNull(String s) {
    return s == null ? "" : s;
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public boolean isValid() {
    return valid;
}
}
```

Você pode ver que as coisas começaram a fugir ao controle. Ainda não está horrível, mas a bagunça certamente está crescendo. É um amontoado, mas ainda não está tão grande assim. Isso ocorreu com a adição do parâmetro do tipo integer.

## Portanto, eu parei

Eu tinha pelo menos mais dois tipos de parâmetros para adicionar, e eu poderia dizer que eles piorariam as coisas. Se eu forçasse a barra, provavelmente os faria funcionar também, mas deixaria um rastro de bagunça grande demais para consertar. Se a estrutura deste desse código algum dia for passível de manutenção, este é o momento para consertá-lo.

Portanto, parei de adicionar novos recursos e comecei a refatorar. Como só adicionei os parâmetros do tipo string e integer, eu saiba que cada tipo exigia um bloco de código novo em três locais principais. Primeiro, cada tipo requer uma forma de analisar a sintaxe de seu elemento de modo a selecionar o HashMap para aquele tipo. Depois, seria preciso passar cada tipo nas strings da linha de comando e convertê-lo para seu tipo verdadeiro. Por fim, cada tipo precisaria de um método getXXX de modo que pudesse ser retornado ao chamador já em seu tipo verdadeiro.

Muitos tipos diferentes, todos com métodos similares—isso parece uma classe para mim. E, então, surgiu o ArgumentMarshaler.

## Incrementalismo

Uma das melhores maneiras de arruinar um programa é fazer modificações excessivas em sua estrutura visando uma melhoria. Alguns programas jamais se recuperam de tais “melhorias”. O

problema é que é muito difícil fazer o programa funcionar como antes da “melhoria”. A fim de evitar isso, sigo o conceito do Desenvolvimento dirigido a testes (TDD, sigla em inglês). Uma das doutrinas centrais dessa abordagem é sempre manter o sistema operante. Em outras palavras, ao usar o TDD, não posso fazer alterações ao sistema que o danifiquem. Cada uma deve mantê-lo funcionando como antes.

Para conseguir isso, precisei de uma coleção de testes automatizados que eu pudesse rodar quando desejasse e que verificasse se o comportamento do sistema continua inalterado. Para a classe Args, criei uma coleção de testes de unidade e de aceitação enquanto eu bagunçava o código. Os testes de unidade estão em Java e são gerenciados pelo JUnit. Os de aceitação são como páginas wiki no FitNesse. Eu poderia rodar esses testes quantas vezes quisesse, e, se passassem, eu ficaria confiante de que o sistema estava funcionando como eu especificara.

Sendo assim, continuei e fiz muitas alterações minúsculas. Cada uma movia a estrutura do sistema em direção ao ArgumentMarshaller. E ainda assim, cada mudança mantinha o sistema funcionando. A primeira que fiz foi adicionar ao esqueleto do ArgumentMarshaller ao final da pilha amontoada de códigos (Listagem 14.11).

### **Listing 14-11**

#### **ArgumentMarshaller appended to Args.java**

```
private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() {return booleanValue;}
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {}

private class StringArgumentMarshaler extends ArgumentMarshaler {}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {}
}
```

Obviamente, isso não danificaria nada. Portanto, fiz a modificação mais simples que pude, uma que danificaria o mínimo possível. Troquei o `HashMap` dos parâmetros do tipo `booleano` para receber um `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
new HashMap<Character, ArgumentMarshaler>();

Isso danificou algumas instruções, que rapidamente consertei.

...
private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());
}
...
```

```

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).setBoolean(value);
}

...
public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg).getBoolean());
}

```

Note como essas mudanças estão exatamente nas áreas mencionadas anteriormente: o parse, o set e o get para o tipo do parâmetro. Infelizmente, por menor que tenha sido essa alteração, alguns testes começaram a falhar. Se olhar com atenção para o getBoolean, verá que se você o chamar com “y”, mas não há houver parâmetro y, booleanArgs.get('y') retornará null e a função lançará uma NullPointerException. Usou-se a função falseIfNull para evitar que isso ocorresse, mas, com a mudança que fiz, ela se tornou irrelevante.

O incrementalismo exige que eu conserte isso rapidamente antes de fazer qualquer outra alteração. De fato, a solução não foi muito difícil. Só tive de mover a verificação por null. Não era mais um booleano null que eu deveria verificar, mas o ArgumentMarshaller.

Primeiro, removi a chamada a falseIfNull na função getBoolean. Ela não fazia mais nada agora, portanto a eliminei. Os testes ainda falhavam de certa forma, então eu estava certo de que não havia gerado novos erros.

```

public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
}

```

Em seguida, dividi dividi a função em duas linhas e coloquei o ArgumentMarshaller em sua própria variável chamada argumentMarshaller. Não me preocupara com o tamanho do nome; ele era redundante e bagunçava a função. Portanto o reduzi para am [N5].

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am.getBoolean();
}

```

E, então, inseri a lógica de detecção de null.

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && am.getBoolean();
}

```

## Parâmetros do tipo string

Adicionar parâmetros do tipo String é muito semelhante à adição de parâmetros booleanos. Eu tive de mudar o HashMap e fazer as funções parse, get e set funcionarem. Não há muitas surpresas depois, talvez pareça que eu esteja colocando toda a implementação de disponibilização (marshalling) na classe base ArgumentMarshaller em vez de seus derivados.

```

private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();

```

```
...
    private void parseStringSchemaElement(char elementId) {
        stringArgs.put(elementId, new StringArgumentMarshaler());
    }
...
    private void setStringArg(char argChar) throws ArgsException {
        currentArgument++;
        try {
            stringArgs.get(argChar).setString(args[currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            valid = false;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }
...
    public String getString(char arg) {
        Args.ArgumentMarshaler am = stringArgs.get(arg);
        return am == null ? " " : am.getString();
    }
...
    private class ArgumentMarshaler {
        private boolean booleanValue = false;
        private String stringValue;

        public void setBoolean(boolean value) {
            booleanValue = value;
        }

        public boolean getBoolean() {
            return booleanValue;
        }

        public void setString(String s) {
            stringValue = s;
        }

        public String getString() {
            return stringValue == null ? " " : stringValue;
        }
    }
}
```

Novamente, essas alterações foram feitas uma de cada vez e de tal forma que os testes estavam sempre funcionando. Quando um falhava, eu me certificava de fazê-lo passar com êxito antes de fazer a próxima mudança.

Mas, a esta altura, você já deve saber o que pretendo. Após eu colocar todo o comportamento de disponibilização dentro da classe base `ArgumentMarshaler`, começarei a passá-lo hierarquia abaixo para os derivados. Isso me permitirá manter tudo operante enquanto eu modifiro gradualmente a forma deste programa.

O próximo, e óbvio, passo foi mover a funcionalidade do tipo int para ArgumentMarshaler. Novamente, não há nada de novo aqui.

```
private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
...
private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, new IntegerArgumentMarshaler());
}
...
private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.get(argChar).setInteger(Integer.
parseInt(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}
...
public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
}
...
private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    private int integerValue;
    public void setBoolean(boolean value) {
        booleanValue = value;
    }
    public boolean getBoolean() {
        return booleanValue;
    }
    public void setString(String s) {
        stringValue = s;
    }
}
```

```
public String getString() {
    return stringValue == null ? "" : stringValue;
}

public void setInteger(int i) {
    integerValue = i;
}

public int getInteger() {
    return integerValue;
}
}
```

Após mover toda a disponibilização para ArgumentMarshaler, comecei a passar a funcionalidade para os derivados. O primeiro passo foi mover a função setBoolean para BooleanArgumentMarshaler e me certificar de que fosse chamada corretamente. Portanto, criei um método set abstrato.

```
private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() {
        return booleanValue;
    }

    public void setString(String s) {
        stringValue = s;
    }

    public String getString() {
        return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
        integerValue = i;
    }

    public int getInteger() {
        return integerValue;
    }

    public abstract void set(String s);
}
```

Então, implementei o método set em BooleanArgumentMarshaler.

```

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
        booleanValue = true;
    }
}

```

E, finalmente, substitui a chamada a setBoolean pela chamada ao set.

```

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
}

```

Os testes ainda passavam. Como essa mudança causou a implementação do set em BooleanArgumentMarshaler, removi o método setBoolean da classe base ArgumentMarshaler.

Note que a função set abstrata recebe um parâmetro do tipo String, mas a implementação em BooleanArgumentMarshaller não o usa. Coloquei um parâmetro lá porque eu sabia que StringArgumentMarshaller e IntegerArgumentMarshaller o usariam.

Depois, eu queria implementar o método get em BooleanArgumentMarshaler. Mas implementar funções get sempre fica ruim, pois o tipo retornado tem de ser um Object, e neste caso teria de ser declarado como um booleano.

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am .get();
}

```

Só para que isso compile, adicionei a função get a ArgumentMarshaler.

```

private abstract class ArgumentMarshaler {
    ...
    public Object get() {
        return null;
    }
}

```

A compilação ocorreu e, obviamente, os testes falharam. Fazê-los funcionarem novamente era simplesmente uma questão de tornar o get abstrato e implementá-lo em BooleanArgumentMarshaler.

```

private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    ...
    public abstract Object get();
}

```

```

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
        booleanValue = true;
    }
}

```

```
    public Object get() {
        return booleanValue;
    }
}
```

Mais uma vez, os testes passaram. Portanto, as implementações de `get` e `set` ficam em `BooleanArgumentMarshaler`!

Isso me permitiu remover a função `getBoolean` antiga de `ArgumentMarshaler`, mover a variável protegida `booleanValue` abaixo para `BooleanArgumentMarshaler` e torná-la privada.

Fiz as mesmas alterações para os tipos `String`. Implementei `set` e `get` e exclui as funções não mais utilizadas e movi as variáveis.

```
private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.get(argChar).set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}
...
public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : (String) am.get();
}
...
private abstract class ArgumentMarshaler {
    private int integerValue;

    public void setInteger(int i) {
        integerValue = i;
    }

    public int getInteger() {
        return integerValue;
    }

    public abstract void set(String s);
    public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;
    public void set(String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}
```

```

}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";
    public void set(String s) {
        stringValue = s;
    }
    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
    }

    public Object get() {
        return null;
    }
}
}

```

Por fim, repeti o processo para os inteiros (integer). Só um pouco mais complicado, pois os inteiros precisam ser analisados sintaticamente, e esse processo pode lançar uma exceção. Mas o resultado fica melhor, pois toda a ação de NumberFormatException fica escondido em IntegerArgumentMarshaler.

```

private boolean isIntArg(char argChar) {return intArgs
containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setBooleanArg(char argChar) {

```

```
try {
    booleanArgs.get(argChar).set("true");
} catch (ArgsException e) {
}
}

...
public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am .get();
}

...
private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
}

...
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

É claro que os testes continuavam a passar. Depois, me livre dos três maps diferentes no início do algoritmo. Isso generalizou muito mais o sistema todo. Entretanto, não consegui me livrar dele apenas excluindo-os, pois isso danificaria o sistema.

Em vez disso, adicionei um novo Map a ArgumentMarshaler e, então, um a um, mudei os métodos e o usei este novo Map no lugar dos três originais.

```
public class Args {
    ...

    private Map<Character, ArgumentMarshaler> booleanArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> stringArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();

    ...
    private void parseBooleanSchemaElement(char elementId) {
        ArgumentMarshaler m = new BooleanArgumentMarshaler();
        booleanArgs.put(elementId, m);
        marshalers.put(elementId, m);
    }
}
```

```

}

private void parseIntegerSchemaElement(char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
    marshalers.put(elementId, m);
}

private void parseStringSchemaElement(char elementId) {
    ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
    marshalers.put(elementId, m);
}

```

É claro que todos os testes ainda passaram. Em seguida, modifiquei o isBooleanArg disso:

```

private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
}

```

para isso:

```

private boolean isBooleanArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof BooleanArgumentMarshaler;
}

```

Os testes ainda passaram. Portanto, fiz a mesma mudança em isIntArg e isStringArg.

```

private boolean isIntArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof StringArgumentMarshaler;
}

```

Os testes ainda passaram. Portanto, eliminei todas as chamadas repetidas a marshalers.get:

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (isBooleanArg(m))
        setBooleanArg(argChar);
    else if (isStringArg(m))
        setStringArg(argChar);
    else if (isIntArg(m))
        setIntArg(argChar);
    else
        return false;
    return true;
}

```

```
private boolean isIntArg(ArgumentMarshaler m) {
    return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(ArgumentMarshaler m) {
    return m instanceof StringArgumentMarshaler;
}

private boolean isBooleanArg(ArgumentMarshaler m) {
    return m instanceof BooleanArgumentMarshaler;
}
```

Isso não fez nada de mais para os três métodos `isxxxxArg`. Portanto, eu os encurtei:

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(argChar);
    else if (m instanceof StringArgumentMarshaler)
        setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(argChar);
    else
        return false;
    return true;
}
```

Depois, comecei a usar os map do `marshalers` nas funções `set`, danificando o uso dos outros três maps. Comecei com os booleanos.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
        setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(argChar);
    else
        return false;
    return true;
}

...
private void setBooleanArg(ArgumentMarshaler m) {
    try {
        m.set("true"); // was: booleanArgs.get(argChar).
    set("true");
    } catch (ArgsException e) {
    }
}
```

Os testes continuam passando, portanto, fiz o mesmo para strings e inteiros. Isso me permitiu integrar parte do código de gerenciamento de exceção à função setArgument.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
        m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

```

Quase consegui remover os três maps antigos. Primeiro, precisei alterar a função getBoolean disso:

```
public boolean getBoolean(char arg) {
```

```

    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
}

```

para isso:

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}

```

Você deve ter se surpreendido com essa última alteração. Por que decidi usar de repente a ClassCastException? O motivo é que tenho uma série de testes de unidade e uma série separada de testes de aceitação escritos no FitNesse. Acabou que os testes do FitNesse garantiram que, se você chamassem a getBoolean em um parâmetro não booleano, você recebia falso. Mas os testes de unidade não.

Até este momento, eu só havia rodado os testes de unidade<sup>2</sup>.

Essa última alteração me permitiu remover outro uso do map booleano:

```

private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
}

```

E agora podemos excluir o map booleano.

```

public class Args {
    ...
private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
    ...
}

```

Em seguida, migrei os parâmetros do tipo String e Integer da mesma maneira e fiz uma pequena limpeza nos booleanos.

```

private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
}

private void parseIntegerSchemaElement(char elementId) {
}

```

<sup>2</sup> A fim de evitar surpresas desse tipo mais tarde, adicionei um novo teste de unidade que invoca todos os testes do FitNesse.

```

        marshalers.put(elementId, new IntegerArgumentMarshaler());
    }

private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
}
...

public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}

public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}

public class Args {
    ...

    private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();

    ...
}

```

Em seguida, encurtei os métodos parse porque eles não fazem mais muita coisa.

```

private void parseSchemaElement(String element) throws
ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
        throw new ParseException(String.format(
            "Parâmetro: %c possui formato inválido: %s.", elementId,
elementTail), 0);
    }
}

```

Tudo bem, agora vejamos o todo novamente. A Listagem 14.12 mostra a forma atual da classe Args.

**Listagem 14-12****Args.java (Após primeira refatoração)**

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT
    }

    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }

    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                String trimmedElement = element.trim();
                parseSchemaElement(trimmedElement);
            }
        }
        return true;
    }

    private void parseSchemaElement(String element) throws ParseException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
        validateSchemaElement(elementId);
        if (isBooleanSchemaElement(elementTail))
            marshalers.put(elementId, new BooleanArgumentMarshaler());
        else if (isStringSchemaElement(elementTail))
            marshalers.put(elementId, new StringArgumentMarshaler());
    }

    private void validateSchemaElement(char elementId) {
        if (!Character.isLetter(elementId))
            unexpectedArguments.add(elementId);
    }

    private boolean isBooleanSchemaElement(String elementTail) {
        return elementTail.equals("true") || elementTail.equals("false");
    }

    private boolean isStringSchemaElement(String elementTail) {
        return elementTail.startsWith("\"") && elementTail.endsWith("\"");
    }
}
```

**Listagem 14-12 (continuação)****Args.java (Após primeira refatoração)**

```

else if (isIntegerSchemaElement(elementTail)) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
} else {
    throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}

private void validateSchemaElement(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}

private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}

private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

```

**Listagem 14-12 (continuação)****Args.java (Após primeira refatoração)**

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
        m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private void setBooleanArg(ArgumentMarshaler m) {
    try {
        m.set("true");
    } catch (ArgsException e) {
    }
}

public int cardinality() {
    return argsFound.size();
}
```

**Listagem 14-12 (continuação)****Args.java (Após primeira refatoração)**

```

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                errorArgumentId);
    }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
}

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}

public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}

```

**Listagem 14-12 (continuação)****Args.java (Após primeira refatoração)**

```
public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public boolean isValid() {
    return valid;
}

private class ArgsException extends Exception {
}

private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
        stringValue = s;
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (Exception e) {
            throw new ArgsException("Argumento inválido");
        }
    }
}
```

**Listagem 14-12 (continuação)**

Args.java (Após primeira refatoração)

```
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

Depois de todo esse trabalho, isso é um pouco frustrante. A estrutura ficou um pouco melhor, mas ainda temos todas aquelas variáveis no início; Ainda há uma estrutura case de tipos em setArgument; e todas aquelas funções set estão horríveis. Sem contar com todos os tratamentos de erros. Ainda temos muito trabalho pela frente.

Eu realmente gostaria de me livrar daquele case de tipos em `setArgument` [G23]. Eu preferia ter nele apenas uma única chamada, a `ArgumentMarshaler.set`. Isso significa que preciso empurrar `setIntArg`, `setStringArg` e `setBooleanArg` hierarquia abaixo para os derivados de `ArgumentMarshaler` apropriados. Mas há um problema.

Se observar `setIntArg` de perto, notará que ele usa duas instâncias de variáveis: `args` e `currentArg`. Para mover `setIntArg` para baixo até `BooleanArgumentMarshaler`, terei de passar passar ambas as variáveis como parâmetros da função. Isso suja o código [F1]. Prefiro passar um parâmetro em vez de dois. Felizmente, há uma solução simples. Podemos converter o array `args` em uma lista e passar um Iterador abaixo para as funções `set`. Levei dez passos no exemplo a seguir para passar todos os testes após cada passo. Mas só lhe mostrarei o resultado. Você deve ser capaz de descobrir o que era a maioria desses pequenos passos.

```
public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new
TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
    private List<String> argsList;

    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER,
UNEXPECTED_ARGUMENT}

    public Args(String schema, String[] args) throws ParseException
    {
        this.schema = schema;
        argsList = Arrays.asList(args);
```

```
        valid = parse();
    }

private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
        return true;
    parseSchema();
    try {
        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}
---

private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator();
currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
    }
    return true;
}
---

private void setIntArg(ArgumentMarshaler m) throws ArgsException
{
    String parameter = null;
    try {
        parameter = currentArgument.next();
        m.set(parameter);
    } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws
ArgsException {
    try {
        m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}
```

Essas foram simples modificações nas quais todos os testes passaram. Agora podemos começar a mover as funções set para os derivados apropriados. Primeiro, preciso fazer a seguinte alteração em setArgument:

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

```

Essa mudança é importante porque queremos eliminar completamente a cadeia if-else. Logo, precisamos retirar a condição de erro lá de dentro.

Agora podemos começar a mover as funções set. A setBooleanArg é trivial, então trabalharemos nela primeiro. Nosso objetivo é alterá-la simplesmente para repassar para BooleanArgumentMarshaler.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m, currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

private void setBooleanArg(ArgumentMarshaler m,
    Iterator<String> currentArgument)
throws ArgsException {
    try {
        m.set("true");
    } catch (ArgsException e) {
    }
}

```

Não acabamos de incluir aquele tratamento de exceção? É muito comum na refatoração inserir e remover as coisas novamente. A pequenezas dos passos e a necessidade de manter os testes funcionando significa que você move bastante as coisas. Refatorar é como resolver o cubo de Rubik. Há muitos passos pequenos necessários para alcançar um objetivo maior. Cada passo possibilita o próximo.

Por que passamos aquele iterator quando setBooleanArg certamente não precisa dele? Porque setIntArg e setStringArg precisarão! E porque queremos implementar todas essas três funções através de um método abstrato em ArgumentMarshaller e passá-lo para setBooleanArg. Dessa forma, setBooleanArg agora não serve para nada. Se houvesse uma função set em ArgumentMarshaller, poderíamos chamá-la diretamente. Portanto, é hora de criá-la! O primeiro passo é adicionar o novo método abstrato a ArgumentMarshaller.

```
private abstract class ArgumentMarshaller {  
    public abstract void set(Iterator<String> currentArgument)  
        throws ArgsException;  
    public abstract void set(String s) throws ArgsException;  
    public abstract Object get();  
}
```

É claro que isso separa todos os derivados. Portanto, vemos implementar o novo método em cada um.

```
private class BooleanArgumentMarshaller extends ArgumentMarshaller {  
    private boolean booleanValue = false;  
  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        booleanValue = true;  
    }  
  
    public void set(String s) {  
        booleanValue = true;  
    }  
  
    public Object get() {  
        return booleanValue;  
    }  
}  
  
private class StringArgumentMarshaller extends ArgumentMarshaller {  
    private String stringValue = "";  
  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
    }  
  
    public void set(String s) {  
        stringValue = s;  
    }  
  
    public Object get() {  
        return stringValue;  
    }  
}
```

```

}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws
ArgsException {
    }

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}

```

E, agora, podemos eliminar setBooleanArg!

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

```

Todos os testes passam e a função set é implementada para BooleanArgumentMarshaler! Agora podemos fazer o mesmo para Strings e Integers.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)

```

```
        m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
        m.set(currentArgument);
    else if (m instanceof IntegerArgumentMarshaler)
        m.set(currentArgument);
} catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
}
return true;
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";
    public void set(Iterator<String> currentArgument) throws
ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }

    public void set(String s) {
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;
    public void set(Iterator<String> currentArgument) throws
ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            set(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (ArgsException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw e;
        }
    }

    public void set(String s) throws ArgsException {
    }
}
```

```

        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}

```

E, agora o coup de grace: pode-se remover o caso do tipo! Touché!

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        m.set(currentArgument);
        return true;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
}

```

Agora podemos nos livrar de algumas funções inúteis em IntegerArgumentMarshaler e limpá-la um pouco.

```

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0
    public void set(Iterator<String> currentArgument) throws
ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw new ArgsException();
        }
    }
    public Object get() {
        return intValue;
    }
}

```

Também podemos transformar ArgumentMarshaler em uma interface.

```
private interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
    Object get();
}
```

Portanto, agora vejamos como fica fácil adicionar um novo tipo de parâmetro a à nossa estrutura. São necessárias poucas mudanças, e elas devem ser isoladas. Primeiro, adicionamos um novo teste de caso para verificar se o parâmetro double funciona corretamente.

```
public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "42.3"});
    assertTrue(args.isValid());
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
}
```

Agora, limpamos o código de análise da sintaxe e adicionamos o ## para detectar o parâmetro do tipo double.

```
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
        throw new ParseException(String.format(
            "Parâmetro: %c possui formato inválido: %s.", elementId, elementTail), 0);
}
```

Em seguida, criamos a classe DoubleArgumentMarshaler.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;
    public void set(Iterator<String> currentArgument) throws
ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_DOUBLE;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
        }
    }
}
```

```

        errorCode = ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
    }

    public Object get() {
        return doubleValue;
    }
}

```

Isso nos força a adicionar um novo ErrorCode.

```

private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_
ARGUMENT,
MISSING_DOUBLE, INVALID_DOUBLE}

```

E precisamos de uma função getDouble.

```

public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
        return 0.0;
    }
}

```

E todos os testes passaram! Não houve problemas. Portanto, agora vamos nos certificar de que todo o processamento de erros funcione corretamente. No próximo caso de teste, um erro é declarado se uma string cujo valor não corresponda ao tipo esperado é passada em um parâmetro ##.

```

public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Parâmetro -x espera um double mas recebeu 'Quarenta e
Dois'.",
    args.errorMessage());
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Não deve entrar aqui.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Não foi possível encontrar um
parâmetro string para -%c.",
errorArgumentId);
    }
}

```

```

        case INVALID_INTEGER:
            return String.format("Parâmetro -%c espera um integer
mas recebeu
                '%s'.", errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Não foi possível encontrar um
parâmetro integer para -%c.",

errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Parâmetro -%c espera um double mas
recebeu '%s'.",
                errorArgumentId, errorParameter);

        case MISSING_DOUBLE:
            return String.format("Não foi possível encontrar um parâmetro
double para -%c.", errorArgumentId);
    }
    return "";
}

```

O teste passa com êxito. O próximo garante que detectemos devidamente um parâmetro double que está faltando.

```

public void testMissingDouble() throws Exception {
Args args = new Args("x##", new String[]{"-x"});
assertFalse(args.isValid());
assertEquals(0, args.cardinality());
assertFalse(args.has('x'));
assertEquals(0.0, args.getDouble('x'), 0.01);
assertEquals("Não foi possível encontrar um parâmetro double para -x.",
args.errorMessage());
}

```

O teste passa normalmente. Fizemos isso apenas por questão de finalização.

O código de exceção está horrível e não pertence à classe Args. Também estamos lançando a ParseException, que não cabe a nós. Portanto, vamos unir todas as exceções em uma única classe, ArgsException, e colocá-la em seu próprio módulo.

```

public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    public ArgsException() {}

    public ArgsException(String message) {super(message);}

    public enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER,
UNEXPECTED_ARGUMENT,
        MISSING_DOUBLE, INVALID_DOUBLE}
}
---
```

```
public class Args {  
    ...  
    private char errorArgumentId = '\0';  
    private String errorParameter = "TILT";  
    private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.  
OK;  
    private List<String> argsList;  
  
    public Args(String schema, String[] args) throws ArgsException {  
        this.schema = schema;  
        argsList = Arrays.asList(args);  
        valid = parse();  
    }  
  
    private boolean parse() throws ArgsException {  
        if (schema.length() == 0 && argsList.size() == 0)  
            return true;  
        parseSchema();  
        try {  
            parseArguments();  
        } catch (ArgsException e) {  
        }  
        return valid;  
    }  
  
    private boolean parseSchema() throws ArgsException {  
        ...  
    }  
  
    private void parseSchemaElement(String element) throws ArgsException {  
        ...  
        else  
            throw new ArgsException(  
                String.format("Parâmetro: %c possui formato inválido:  
%s.",  
                elementId, elementTail));  
    }  
  
    private void validateSchemaElementId(char elementId) throws  
ArgsException {  
        if (!Character.isLetter(elementId)) {  
            throw new ArgsException(  
                "Caractere errado:" + elementId + " no  
formato d Args: " + schema);  
        }  
    }  
  
    ...  
  
    private void parseElement(char argChar) throws ArgsException {  
        if (setArgument(argChar))  
            argsFound.add(argChar);  
        else {  
            unexpectedArguments.add(argChar);  
            errorCode = ArgsException.ErrorCode.UNEXPECTED_  
ARGUMENT;  
            valid = false;  
        }  
    }
```

```
    }

    ...
}

private class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws
ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {
            errorCode = ArgsException.ErrorCode.MISSING_
STRING;
            throw new ArgsException();
        }
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws
ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ArgsException.ErrorCode.MISSING_
INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ArgsException.ErrorCode.INVALID_
INTEGER;
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}

private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws
ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
        }
```

```
        doubleValue = Double.parseDouble(parameter);
    } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_
DOUBLE;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_
DOUBLE;
        throw new ArgsException();
    }
}

public Object get() {
    return doubleValue;
}
}
```

Isso é bom. Agora Args lança apenas a exceção `ArgsException` que, ao ser movida para seu próprio módulo, agora podemos retirar de `Args` os variados códigos de suporte a erros e colocar naquele módulo. Isso oferece um local óbvio e natural para colocar todo aquele código e ainda nos ajuda a limpar o módulo `Args`.

Portanto, agora separamos completamente os códigos de exceção e de erro do módulo Args. (Veja da Listagem 14.13 à 14.16). Conseguimos isso apenas com pequenos 30 passos, fazendo com que os testes rodem entre cada um.

### Listagem 14-13

## ArgsTest.java

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
    public void testCreateWithNoSchemaOrArguments() throws Exception {
        Args args = new Args("", new String[0]);
        assertEquals(0, args.cardinality());
    }

    public void testWithNoSchemaButWithOneArgument() throws Exception {
        try {
            new Args("", new String[] {"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }

    public void testWithNoSchemaButWithMultipleArguments() throws Exception {
        try {
            new Args("", new String[] {"-x", "-y"});
            fail();
        }
    }
}
```

**Listagem 14-13 (continuação)****ArgsTest.java**

```
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }

    }

public void testNonLetterSchema() throws Exception {
    try {
        new Args("", new String[]{});
        fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {

        assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                    e.getErrorCode());
        assertEquals('*', e.getErrorArgumentId());
    }
}

public void testInvalidArgumentFormat() throws Exception {
    try {
        new Args("f~", new String[]{});
        fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
        assertEquals('f', e.getErrorArgumentId());
    }
}

public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args("x", new String[]{"-x"});
    assertEquals(1, args.cardinality());
    assertEquals(true, args.getBoolean('x'));
}

public void testSimpleStringPresent() throws Exception {
    Args args = new Args("x*", new String[]{"-x", "param"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals("param", args.getString('x'));
}

public void testMissingStringArgument() throws Exception {
    try {
        new Args("x*", new String[]{"-x"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }
}

public void testSpacesInFormat() throws Exception {
    Args args = new Args("x y", new String[]{"-xy"});
    assertEquals(2, args.cardinality());
```

**Listagem 14-13 (continuação)****ArgsTest.java**

```
assertTrue(args.has('x'));
assertTrue(args.has('y'));
}

public void testSimpleIntPresent() throws Exception {
    Args args = new Args("x#", new String[] {"-x", "42"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42, args.getInt('x'));
}

public void testInvalidInteger() throws Exception {
    try {
        new Args("x#", new String[] {"-x", "Forty two"});

        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", e.getErrorParameter());
    }
}

public void testMissingInteger() throws Exception {
    try {
        new Args("x#", new String[] {"-x"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }
}

public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "42.3"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
}

public void testInvalidDouble() throws Exception {
    try {
        new Args("x##", new String[] {"-x", "Forty two"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", e.getErrorParameter());
    }
}

public void testMissingDouble() throws Exception {
    try {
        new Args("x##", new String[] {"-x"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", e.getErrorParameter());
    }
}
```

**Listagem 14-13 (continuação)****ArgsTest.java**

```
    new Args("x##", new String[]{"-x", "Forty two"});  
    fail();  
} catch (ArgsException e) {  
    assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());  
    assertEquals('x', e.getErrorArgumentId());  
    assertEquals("Forty two", e.getErrorParameter());  
}  
}  
  
public void testMissingDouble() throws Exception {  
    try {  
        new Args("x##", new String[]{"-x"});  
        fail();  
    } catch (ArgsException e) {  
        assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());  
        assertEquals('x', e.getErrorArgumentId());  
    }  
}
```

**Listagem 14-14****ArgsExceptionTest.java**

```
public class ArgsExceptionTest extends TestCase {  
    public void testUnexpectedMessage() throws Exception {  
        ArgsException e =  
            new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,  
                'x', null);  
        assertEquals("Argument -x unexpected.", e.errorMessage());  
    }  
  
    public void testMissingStringMessage() throws Exception {  
        ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,  
            'x', null);  
        assertEquals("Could not find string parameter for -x.", e.errorMessage());  
    }  
  
    public void testInvalidIntegerMessage() throws Exception {  
        ArgsException e =  
            new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,  
                'x', "Forty two");  
        assertEquals("Argument -x expects an integer but was 'Forty two'.",  
            e.errorMessage());  
    }  
  
    public void testMissingIntegerMessage() throws Exception {  
        ArgsException e =  
            new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);  
        assertEquals("Could not find integer parameter for -x.", e.errorMessage());  
    }  
  
    public void testInvalidDoubleMessage() throws Exception {  
        ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
```

**Listagem 14-14 (continuação)**

## ArgsExceptionTest.java

```
        'x', "Forty two");
assertEquals("Argument -x expects a double but was 'Forty two'.",
            e.errorMessage());
}

public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
}
```

## Listagem 14-15

## ArgsException.java

```
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    public ArgsException() {}

    public ArgsException(String message) {super(message);}

    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }

    public ArgsException(ErrorCode errorCode, String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
    }

    public ArgsException(ErrorCode errorCode, char errorArgumentId,
                        String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
        this.errorArgumentId = errorArgumentId;
    }

    public char getErrorArgumentId() {
        return errorArgumentId;
    }

    public void setErrorArgumentId(char errorArgumentId) {
        this.errorArgumentId = errorArgumentId;
    }
}
```

**Listagem 14-15 (continuação)**

ArgsException.java

```
public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter() {
    return errorParameter;
}

public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode() {
    return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return String.format("Argument -%c unexpected.", errorArgumentId);
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                errorArgumentId, errorParameter);

        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                errorArgumentId);
    }
    return "";
}

public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

## Listagem 14-16

## Args.java

**Listagem 14-16 (continuação)****Args.java**

```
private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
    }
}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                               argChar, null);
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        m.set(currentArgument);
        return true;
    } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
    }
}
```

A maioria das mudanças da classe Args foiforam exclusões. Muito do código foi apenas removido de Args e colocado em ArgsException. Ótimo. Também movemos todos os ArgumentMarshaler para seus próprios arquivos. Melhor ainda.

Muitos dos bons projetos de software se resumem ao particionamento—criar locais apropriados para colocar diferentes tipos de código. Essa separação de preocupações torna o código muito mais simples para se manter e compreender.

De interesse especial temos o método errorMessage de ArgsException. Obviamente, é uma violação ao SRP colocar a formatação da mensagem de erro em Args. Este deve apenas processar os parâmetros, e não o formato de tais mensagens. Entretanto, realmente faz sentido colocar o código da formatação das mensagens de erro em ArgsException?

Francamente, é uma obrigação. Os usuários não gostam de ter que de criar eles mesmos as mensagens de erro fornecidas por ArgsException. Mas a conveniência de ter mensagens de erro prontas e já preparadas para você não é insignificante.

A esta altura já deve estar claro que estamos perto da solução final que surgiu no início deste capítulo. Deixarei como exercício para você as últimas modificações.

## Conclusão

Isso não basta para que um código funcione. Um código que funcione, geralmente possui bastantes erros. Os programadores que se satisfazem só de em verem um código funcionando não estão se comportando de maneira profissional. Talvez temam que não tenham tempo para melhorar a estrutura e o modelo de seus códigos, mas eu discordo. Nada tem um efeito mais intenso e degradante em longo termo sobre um projeto de desenvolvimento do que um código ruim. É possível refazer cronogramas e redefinir requisitos ruins.

Podem-se consertar as dinâmicas ruins de uma equipe. Mas um código ruim apodrece e degrada, tornando-se um peso que se leva a equipe consigo. Não me canso de ver equipes caírem num passo lentíssimo devido à pressa inicial que os levou a criar uma massa maligna de código que acabou selando seus destinos.

É claro que se pode-se limpar um código ruim. Mas isso sai caro. Conforme o código degrada, os módulos se perpetuam uns para os outros, criando muitas dependências ocultas e intrincadas. Encontrar e remover dependências antigas é uma tarefa árdua e longa. Por outro lado, manter o código limpo é relativamente fácil. Se você fizer uma bagunça em um módulo pela manhã, é mais fácil limpá-lo pela tarde. Melhor ainda, se fez a zona a cinco minutos atrás, é muito mais fácil limpá-la agora mesmo.

Sendo assim, a solução é constantemente manter seu código o mais limpo e simples possível. Jamais deixe que ele comece a se degradar.

1. Recentemente reescrevi este modulo em Ruby e ele ficou com 1/7 do tamanho e com uma estrutura levemente melhor.
2. A fim de evitar surpresas desse tipo mais tarde, adicionei um novo teste de unidade que invoca todos os testes do FitNesse.

## Características Internas do JUnit

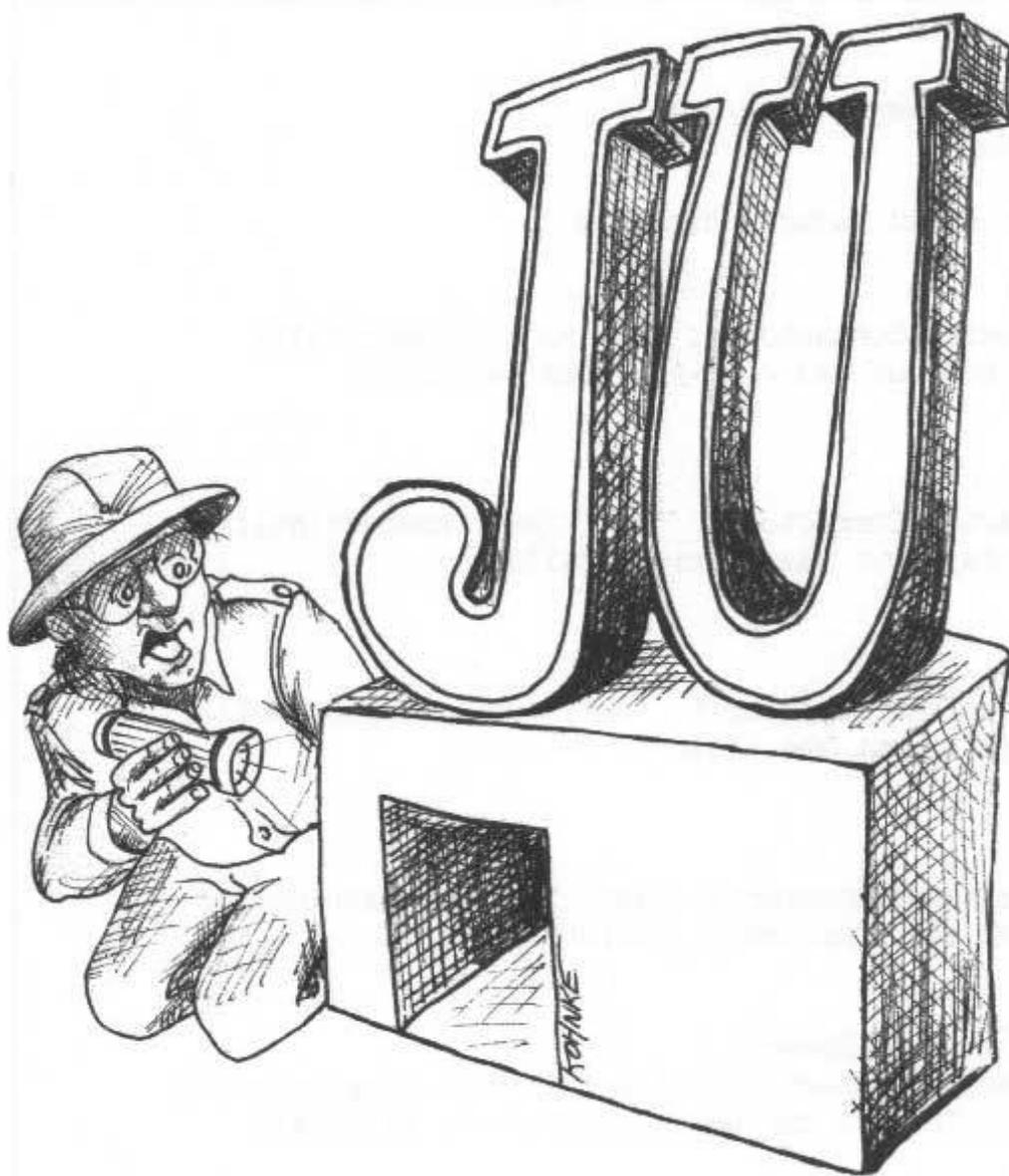
Resumindo, o JUnit é um framework que faz parte da família dos frameworks de teste unitário. Ele é usado para escrever testes automatizados de classes e métodos. O JUnit é uma biblioteca Java que fornece uma estrutura para escrever testes de unidade. Ele também fornece ferramentas para executar os testes e gerenciar dependências entre os testes.

---

O JUnit é um framework de teste unitário que fornece uma estrutura para escrever testes de unidade. Ele é usado para escrever testes automatizados de classes e métodos. O JUnit é uma biblioteca Java que fornece uma estrutura para escrever testes de unidade. Ele também fornece ferramentas para executar os testes e gerenciar dependências entre os testes.

15

## Características Internas do JUnit



O JUnit é um dos frameworks Java mais famosos de todos. Em relação a frameworks, ele é simples, preciso em definição e elegante na implementação. Mas como é o código? Neste capítulo, analisaremos um exemplo retirado do framework JUnit.

## O framework JUnit

O JUnit tem tido muitos autores, mas ele começou com Kent Beck e Eric Gamma dentro de um avião indo à Atlanta. Kent queria aprender Java; e Eric, sobre o framework Smalltalk de testes de Kent. “O que poderia ser mais natural para uma dupla de geeks num ambiente apertado do que abrirem seus notebooks e começarem a programar?”<sup>1</sup> Após três horas de programação em alta altitude, eles criaram os fundamentos básicos do JUnit.

O módulo que veremos, o ComparisonCompactor, é a parte engenhosa do código que ajuda a identificar erros de comparação entre strings. Dadas duas strings diferentes, como ABCDE e ABXDE, o módulo exibirá a diferença através da produção de uma string como <...B[X]D...>. Eu poderia explicar mais, porém, os casos de teste fazem um trabalho melhor. Sendo assim, veja a Listagem 15.1 e você entenderá os requisitos deste módulo em detalhes. Durante o processo, analise a estrutura dos testes. Elas poderiam ser mais simples ou óbvias?

### Listagem 15-1

#### `ComparisonCompactorTest.java`

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

    public void testMessage() {
        String failure= new ComparisonCompactor(0, "b", "c").compact("a");
        assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
    }

    public void testStartSame() {
        String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
        assertEquals("expected:<b[a]> but was:<b[c]>", failure);
    }

    public void testEndSame() {
        String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
        assertEquals("expected:<[a]b> but was:<[c]b>", failure);
    }

    public void testSame() {
        String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
        assertEquals("expected:<ab> but was:<ab>", failure);
    }

    public void testNoContextStartAndEndSame() {
        String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
        assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
    }
}
```

**Listagem 15-1 (continuação)****ComparisonCompactorTest.java**

```
public void testStartAndEndContext() {
    String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
    assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
}

public void testStartAndEndContextWithEllipses() {
    String failure=
        new ComparisonCompactor(1, "abcde", "abfde").compact(null);
    assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
}

public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
}

public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
}

public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
}

public void testComparisonErrorOverlapingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
    assertEquals("expected:<...[...]...> but was:<...[b]...>", failure);
}

public void testComparisonErrorOverlapingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
}

public void testComparisonErrorOverlapingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde",
"abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[...]...>", failure);
}

public void testComparisonErrorOverlapingMatches2Context() {
    String failure=
        new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
}

public void testComparisonErrorWithActualNull() {
    String failure= new ComparisonCompactor(0, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}

public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
```

Efetuei uma análise geral do código no ComparisonCompactor usando esses testes, que cobrem 100% do código: cada linha, cada estrutura if e loop for. Isso me dá um alto grau de confiança de que o código funciona e de respeito pela habilidade de seus autores.

O código do ComparisonCompactor está na Listagem 15.2. Passe um momento analisando-o. Penso que você o achará bem particionado, razoavelmente expressivo e de estrutura simples. Quando você terminar, entenderemos juntos os detalhes.

**Listing 15-2****ComparisonCompactor.java (Original)**

```
package junit.framework;

public class ComparisonCompactor {

    private static final String ELLIPSIS = "...";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";

    private int fContextLength;
    private String fExpected;
    private String fActual;
    private int fPrefix;
    private int fSuffix;

    public ComparisonCompactor(int contextLength,
                               String expected,
                               String actual) {
        fContextLength = contextLength;
        fExpected = expected;
        fActual = actual;
    }

    public String compact(String message) {
        if (fExpected == null || fActual == null || areStringsEqual())
            return Assert.format(message, fExpected, fActual);

        findCommonPrefix();
        findCommonSuffix();
        String expected = compactString(fExpected);
        String actual = compactString(fActual);
        return Assert.format(message, expected, actual);
    }

    private String compactString(String source) {
        String result = DELTA_START +
                        source.substring(fPrefix, source.length() -
                                         fSuffix + 1) + DELTA_END;
        if (fPrefix > 0)
            result = computeCommonPrefix() + result;
        if (fSuffix > 0)
            result = result + computeCommonSuffix();
        return result;
    }

    private void findCommonPrefix() {
        fPrefix = 0;
```

**Listing 15-2****ComparisonCompactor.java (Original)**

```

int end = Math.min(fExpected.length(), fActual.length());
for (; fPrefix < end; fPrefix++) {
    if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
        break;
}
}

private void findCommonSuffix() {
    int expectedSuffix = fExpected.length() - 1;
    int actualSuffix = fActual.length() - 1;
    for (; actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
          actualSuffix--, expectedSuffix--) {
        if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
            break;
    }
    fSuffix = fExpected.length() - expectedSuffix;
}

private String computeCommonPrefix() {
    return (fPrefix > fContextLength ? ELLIPSIS : "") +
           fExpected.substring(Math.max(0, fPrefix - fContextLength),
                               fPrefix);
}

private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
                       fExpected.length());
    return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
           (fExpected.length() - fSuffix + 1 < fExpected.length() -
            fContextLength ? ELLIPSIS : "");
}

```

Talvez você tenha algumas críticas em relação a este módulo. Há expressões muito extensas e uns estranhos `+1s` e por aí vai. Mas, de modo geral, este módulo está muito bom. Afinal de contas, ele deveria estar como na Listagem 15.3.

**Listagem 15-3****ComparisonCompactor.java (sem refatoração)**

```

package junit.framework;

public class ComparisonCompactor {
    private int ctxt;
    private String s1;
    private String s2;
    private int pfx;
    private int sfx;

    public ComparisonCompactor(int ctxt, String s1, String s2) {
        this.ctxt = ctxt;
        this.s1 = s1;
        this.s2 = s2;
    }
}

```

**Listagem 15-3****ComparisonCompator.java (sem refatoração)**

```

}

public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
        return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
        if (s1.charAt(pfx) != s2.charAt(pfx))
            break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
        if (s1.charAt(sfx1) != s2.charAt(sfx2))
            break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
}

private String compactString(String s) {
    String result =
        "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
        result = (pfx > ctxt ? "..." : "") +
            s1.substring(Math.max(0, pfx - ctxt), pfx) + result;

    if (sfx > 0) {
        int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
        result = result + (s1.substring(s1.length() - sfx + 1, end) +
            (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
    }
    return result;
}
}

```

Mesmo que os autores tenham deixado este módulo num formato muito bom, a Regra de Escoteiro<sup>2</sup> nos diz que devemos deixá-lo mais limpo do que como você o encontrou. Portanto, como podemos melhorar o código original na Listagem 15.2?

Primeiro, não deve se preocupar com o prefixo *f* nas variáveis membro [N6]. Os ambientes de hoje em dia tornam esse tipo de escopo redundante. Sendo assim, eliminemos todos os *f*.

```

private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;

```

Depois, temos uma estrutura condicional não encapsulada no início da função compact. [G28].

```
public String compact(String message) {
    if (expected == null || actual == null || areStringsEqual())
        return Assert.format(message, expected, actual);
    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
}
```

Devemos encapsular este if para deixar clara nossa intenção. Portanto, criemos um método que a explique.

```
public String compact(String message) {
    if (shouldNotCompact())
        return Assert.format(message, expected, actual);
    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {
    return expected == null || actual == null || areStringsEqual();
}
```

Não ligo muito para a notação this.expected e this.actual na função compact. Isso ocorreu quando alteramos o nome de fExpected para expected. Por que há variáveis nesta função que possuem os mesmos nomes das variáveis membro? Elas não representam outra coisa [N4]? Devemos desambigüizar os nomes.

```
String compactExpected = compactString(expected);
String compactActual = compactString(actual);
```

É um pouco mais difícil entender negativas do que afirmativas [G29]. Portanto, vamos inverter aquela estrutura if.

```
public String compact(String message) {
    if (canBeCompacted()) {
        findCommonPrefix();
        findCommonSuffix();
        String compactExpected = compactString(expected);
        String compactActual = compactString(actual);
        return Assert.format(message, compactExpected,
compactActual);
    } else {
        return Assert.format(message, expected, actual);
}
```

```

    }
    private boolean canBeCompacted() {
        return expected != null && actual != null && !areStringsEqual();
    }
}

```

O nome da função está estranho [N7]. Embora ela realmente compacte as strings, ela talvez não o faça se canBeCompacted retornar false. Portanto, nomear essa função como compact esconde o efeito colateral da verificação de erro. Note também que, além das strings compactadas, a função retorna uma mensagem formatada. Dessa forma, o nome da função deveria ser formatCompactedComparison. Assim fica muito melhor quando lido juntamente com o parâmetro da função.

```
public String formatCompactedComparison(String message) {
```

O corpo da estrutura if é onde ocorre a compactação real das strings. Deveríamos extrair isso para um método chamado compactExpectedAndActual. Entretanto, queremos que a função formatCompactedComparison faça toda a formatação. A função compact... não deve fazer nada além da compactação [G30]. Portanto, vamos dividi-la assim:

```

...
private String compactExpected;
private String compactActual;

...

public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
        compactExpectedAndActual();
        return Assert.format(message, compactExpected,
compactActual);
    } else {
        return Assert.format(message, expected, actual);
    }
}

private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

```

Note que isso nos força a transformar compactExpected e compactActual em variáveis-membro. Não gosto da forma de que as duas últimas linhas da nova função retorna as variáveis, mas duas primeiras não retornam. Elas não estão usando convenções consistentes [G11]. Portanto, devemos alterar findCommonPrefix e findCommonSuffix para retornar os valores do prefixo e do sufixo.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
```

```

suffixIndex = findCommonSuffix();
compactExpected = compactString(expected);
compactActual = compactString(actual);
}

private int findCommonPrefix() {
    int prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++) {
        if (expected.charAt(prefixIndex) != actual.
            charAt(prefixIndex))
            break;
    }
    return prefixIndex;
}

private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >=
prefixIndex;
        actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.
            charAt(actualSuffix))
            break;
    }
    return expected.length() - expectedSuffix;
}

```

Também deveríamos tornar os nomes das variáveis-membro um pouco mais precisos [N1]; apesar de tudo, ambas são índices.

Uma análise cuidadosa de `findCommonSuffix` expõe um acoplamento temporário escondido [G31]; ele depende do fato de `prefixIndex` ser calculado por `findCommonPrefix`. Se chamassem essas duas funções fora de ordem, a sessão de depuração depois ficaria difícil. Portanto, para expor esse acoplamento temporário, vamos fazer com que `findCommonSuffix` receba `prefixIndex` como um parâmetro.

```

private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix(prefixIndex);
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >=
prefixIndex;
        actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.
            charAt(actualSuffix))
            break;
    }
    return expected.length() - expectedSuffix;
}

```

```

        charAt(actualSuffix))
            break;
    }
    return expected.length() - expectedSuffix;
}

```

Não estou satisfeito com isso. A passagem de `prefixIndex` ficou um pouco arbitrária [G32]. Ela serve para estabelecer a ordenação, mas não explica essa necessidade. Outro programador talvez desfaça o que acabamos de fazer por não haver indicação de que o parâmetro é realmente necessário. Portanto, tentemos de outra maneira.

```

private void compactExpectedAndActual() {
    findCommonPrefixAndSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >=
prefixIndex;
          actualSuffix--, expectedSuffix--)
    ) {
        if (expected.charAt(expectedSuffix) != actual.
charAt(actualSuffix))
            break;
    }
    suffixIndex = expected.length() - expectedSuffix;
}

private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++)
        if (expected.charAt(prefixIndex) != actual.
charAt(prefixIndex))
            break;
}

```

Colocamos `findCommonPrefix` e `findCommonSuffix` como estavam antes, mudando o nome de `findCommonSuffix` para `findCommonPrefixAndSuffix` e fazendo-a chamar `findCommonPrefix` antes de qualquer outra coisa. Isso estabelece a natureza temporária das duas funções de uma maneira muito mais expressiva do que a solução anterior. E também indicar como `findCommonPrefixAndSuffix` é horrível. Vamos limpá-la agora.

```

private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int suffixLength = 1;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {

```

```

        if (charFromEnd(expected, suffixLength) !=
            charFromEnd(actual, suffixLength))
            break;
    }
    suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
    return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
}

```

Assim ficou muito melhor, pois mostra que suffixIndex é o comprimento (Length) do sufixo e que não está bem nomeado. O mesmo vale para prefixIndex, embora neste caso “índice” (Index) e “comprimento” (Length) sejam sinônimos. O problema é que a variável suffixIndex não começa com zero, mas por 1, e, portanto, não é um comprimento real. Esse também é o motivo pelo qual existem todos aqueles 1+s em computeCommonSuffix [G33]. Sendo assim, vamos consertar isso.

O resultado está na Listagem 15.4.

#### Listagem 15-4 **ComparisonCompactor.java (temporário)**

```

public class ComparisonCompactor {
    ...
    private int suffixLength;
    ...
    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength))
                break;
        }
    }

    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }

    private boolean suffixOverlapsPrefix(int suffixLength) {
        return actual.length() - suffixLength <= prefixLength ||
        expected.length() - suffixLength <= prefixLength;
    }

    ...
    private String compactString(String source) {
        String result =
            DELTA_START +
            source.substring(prefixLength, source.length() - suffixLength) +
            DELTA_END;
    }
}

```

**Listagem 15-4 (continuação)****ComparisonCompactor.java (temporário)**

```

    DELTA_END;
    if (prefixLength > 0)
        result = computeCommonPrefix() + result;

    if (suffixLength > 0)
        result = result + computeCommonSuffix();
    return result;
}

...
private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength +
        contextLength, expected.length());
    return
        expected.substring(expected.length() - suffixLength, end) +
        (expected.length() - suffixLength <
            expected.length() - contextLength ?
                ELLIPSIS : "");
}

```

Substituímos os `+1s` em `computeCommonSuffix` por um `-1` em `charFromEnd`, onde faz mais sentido, e dois operadores `<=` em `suffixOverlapsPrefix`, onde também fazem mais sentido. Isso nos permitiu alterar o nome de `suffixIndex` para `suffixLength`, aumentando consideravelmente a legibilidade do código.

Porém, há um problema. Enquanto eu eliminava os `+1s`, percebi a seguinte linha em `compactString`:

```
if (suffixLength > 0)
```

Observe-a na Listagem 15.4. Pela lógica, como agora `suffixLength` está uma unidade menor do que anteriormente, eu deveria alterar o operador `>` para  `$\geq$` . Mas isso não faz sentido. Agora faz! Isso significa que antes não fazia sentido e que provavelmente era um bug.

Bem, não um bug. Após análise mais detalhada, vemos que agora a estrutura `if` evita que um sufixo de comprimento zero seja anexado. Antes de fazermos a alteração, a estrutura `if` não era funcional, pois `suffixIndex` jamais poderia ser menor do que um.

Isso levanta a questão sobre ambas as estruturas `if` em `compactString`! Parece que poderiam ser eliminadas. Portanto, vamos colocá-las como comentários e rodar os testes. Eles passaram! Sendo assim, vamos reestruturar `compactString` para eliminar as estruturas `if` irrelevantes e tornar a função mais simples.

```

private String compactString(String source) {
    return
        computeCommonPrefix() +
        DELTA_START +
        source.substring(prefixLength, source.length() -
suffixLength) +
        DELTA_END +
        computeCommonSuffix();
}

```

Assim está muito melhor! Agora vemos que a função compactString está simplesmente unindo os fragmentos. Provavelmente, podemos tornar isso ainda mais claro. De fato, há várias e pequenas limpezas que poderíamos fazer. Mas em vez de lhe arrastar pelas modificações finais, mostrarei logo o resultado na Listagem 15.5.

**Listagem 15-5****ComparisonCompactor.java (final)**

```
package junit.framework;

public class ComparisonCompactor {

    private static final String ELLIPSIS = "...";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";

    private int contextLength;
    private String expected;
    private String actual;
    private int prefixLength;
    private int suffixLength;

    public ComparisonCompactor(
        int contextLength, String expected, String actual
    ) {
        this.contextLength = contextLength;
        this.expected = expected;
        this.actual = actual;
    }

    public String formatCompactedComparison(String message) {
        String compactExpected = expected;
        String compactActual = actual;
        if (shouldBeCompacted()) {
            findCommonPrefixAndSuffix();
            compactExpected = compact(expected);
            compactActual = compact(actual);
        }
        return Assert.format(message, compactExpected, compactActual);
    }

    private boolean shouldBeCompacted() {
        return !shouldNotBeCompacted();
    }

    private boolean shouldNotBeCompacted() {
        return expected == null ||
               actual == null ||
               expected.equals(actual);
    }

    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
    }
```

**Listagem 15-5 (continuação)****ComparisonCompactor.java (final)**

```
suffixLength = 0;
for (; !suffixOverlapsPrefix(); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
        charFromEnd(actual, suffixLength))
    )
        break;
}

private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
}

private boolean suffixOverlapsPrefix() {
    return actual.length() - suffixLength <= prefixLength ||
           expected.length() - suffixLength <= prefixLength;
}

private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
        if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
            break;
}

private String compact(String s) {
    return new StringBuilder()
        .append(startingEllipsis())
        .append(startingContext())
        .append(DELTA_START)
        .append(delta(s))
        .append(DELTA_END)
        .append(endingContext())
        .append(endingEllipsis())
        .toString();
}

private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
}

private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
}

private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
}
```

**Listagem 15-5 (continuação)****ComparisonCompactor.java (final)**

```
private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
        Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
}

private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
}
```

Ficou realmente belo. O módulo está separado em um grupo de funções de análise e em um de síntese. E estão topologicamente organizadas de modo que a declaração de cada uma aparece logo após seu uso. Todas as funções de análise aparecem primeiro, e as de síntese por último. Se olhar com atenção, verá que reverti diversas das decisões que eu havia feito anteriormente neste capítulo. Por exemplo, coloquei de volta alguns métodos extraídos em `formatCompactedComparison` e alterei o sentido da expressão `shouldNotBeCompacted`. Isso é típico. Geralmente, uma refatoração leva à outra, que leva ao desfazimento da primeira. Refatorar é um processo iterativo cheio de tentativas e erros, convergindo inevitavelmente em algo que consideramos digno de um profissional.

## Conclusão

E também cumprimos a Regra de Escoteiro. Deixamos este módulo um pouco mais limpo do que como o encontramos. Não que já não estivesse limpo. Seus autores fizeram um trabalho excelente com ele.

Mas nenhum módulo está imune a um aperfeiçoamento, e cada um de nós tem o dever de deixar o código um pouco melhor do que como o encontramos.

Esta página foi deixada intencionalmente em branco

16

16

# Refatorando o `SerialDate`

# Refatorando o SerialDate



Se você for ao site <http://www.jfree.org/jcommon/index.php>, encontrará a biblioteca JCommon, dentro da qual há um pacote chamado `org.jfree.date`, que possui uma classe chamada `SerialDate`. Iremos explorar essa classe.

David Gilbert é o criador da `SerialDate`. Obviamente, ele é um programador competente e experiente. Como veremos, ele exibe um nível considerável de profissionalismo e disciplina dentro do código. Para todos os efeitos, esse é um “bom código”. E eu irei desmembrá-lo em partes.

Não posso mais intenções aqui. Nem acho que eu seja tão melhor do que o David que, de alguma forma, eu teria o direito de julgar seu código. De fato, se visse alguns de meus códigos, estou certo de que você encontraria várias coisas para criticar.

Não, também não é uma questão de arrogância ou maldade. O que pretendo fazer aqui nada mais é do que uma revisão profissional. Algo com a qual todos nós deveríamos nos sentir confortáveis em fazer. E algo que deveríamos receber bem quando fizessem conosco. É só através de críticas como essas que aprenderemos. Os médicos fazem isso. Os pilotos também fazem. Os advogados também. E nós programadores precisamos aprender a fazer também.

E mais uma coisa sobre David Gilbert: ele é muito mais do que um bom programador.

David teve a coragem e a boa vontade de oferecer de graça seu código à comunidade em geral. Ele o colocou à disposição para que todos vissem, usassem e analisassem. Isso foi um bem feito! `SerialDate` (Listagem B.1, página 349) é uma classe que representa uma data em Java. Por que ter uma classe que represente uma data quando o Java já possui a `java.util.Date` e a `java.util.Calendar`, dentre outras? O autor criou essa classe em resposta a um incômodo que eu mesmo costumo sentir. O comentário em seu Javadoc de abertura (linha 67) explica bem isso. Poderíamos deduzir a intenção do autor, mas eu certamente já tive de lidar com essa questão e, então, sou grato a essa classe que trata de datas ao invés de horas.

## Primeiro, faça-a funcionar

Há alguns testes de unidade numa classe chamada `SerialDateTests` (Listagem B.2, página 366). Todos os testes passam. Infelizmente, uma rápida análise deles mostra que não testam tudo [T1]. Por exemplo, efetuar uma busca “Encontrar Tarefas” no método `MonthCodeToQuarter` (linha 334) indica que ele não é usado [F4]. Logo, os testes de unidade não o testam.

Então, rodei o Clover para ver o que os testes de unidade cobriam e não cobriam. O Clover informou que os testes de unidades só executavam 91 das 185 instruções na `SerialDate` (~50%) [T2]. O mapa do que era testado parecia uma colcha de retalhos, com grandes buracos de código não executado amontoados ao longo de toda a classe.

Meu objetivo era entender completamente e também refatorar essa classe. Eu não poderia fazer isso sem um teste de maior cobertura. Portanto, criei minha própria coleção de testes de unidade completamente independente (Listagem B.4, página 374).

Ao olhar esses testes, você verá que muitos foram colocados como comentários.

Esses testes falharam. Eles representam o comportamento que eu acho que a `SerialDate` deveria ter. Dessa forma, conforme refatoro a `SerialDate`, farei também com que esses testes passem com êxito.

Mesmo com alguns dos testes colocados como comentário, o Clover informa que os novos testes de unidade executam 170 (92%) das 185 instruções executáveis. Isso é muito bom, e acho que poderemos aumentar ainda mais esse número.

Os primeiros poucos testes postos como comentários (linhas 23-63) são um pouco de prepotência de minha parte. O programa não foi projetado para passar nesses testes, mas o comportamento parece óbvio [G2] para mim.

Não estou certo por que o método `testWeekdayCodeToString` foi criado, mas como ele está lá, parece óbvio que ele não deve diferir letras maiúsculas de minúsculas. Criar esses testes foi simples [T3]. Fazê-los passar foi mais simples ainda; apenas alterei as linhas 259 e 263 para usarem `equalsIgnoreCase`.

Coloquei os testes nas linhas 32 e 45 como comentários porque não está claro para mim se as abreviações “tues” (Tuesday, terça-feira em inglês) e “thurs” (Thursday, quinta-feira em inglês) devem ser suportadas.

Os testes nas linhas 153 e 154 falham. Obviamente, eles deveriam mesmo [G2]. Podemos facilmente consertá-los e, também, os testes das linhas 163 a 213, fazendo as seguintes modificações à função `stringToMonthCode`.

```
457     if ((result < 1) || (result > 12)) {
458         result = -1;
459         for (int i = 0; i < monthNames.length; i++) {
460             if (s.equalsIgnoreCase(shortMonthNames[i])) {
461                 result = i + 1;
462                 break;
463             if (s.equalsIgnoreCase(monthNames[i])) {
464                 result = i + 1;
465                 break;
466             }
467         }
468     }
```

O teste colocado como comentário na linha 318 expõe um bug no método `getFollowingDayOfWeek` (linha 672). A data 25 de dezembro de 2004 caiu num sábado. E o sábado seguinte em 1º de janeiro de 2005. Entretanto, quando efetuamos o teste, vemos que `getFollowingDayOfWeek` retorna 25 de dezembro como o sábado seguinte a 25 de dezembro. Isso está claramente errado [G3], [T1]. Vimos o problema na linha 685, que é um típico erro de condição de limite [T5]. Deveria estar escrito assim:

```
685     if (baseDOW >= targetWeekday) {
```

É interessante notar que essa função foi o alvo de um conserto anterior. O histórico de alterações (linha 43) mostra que os “bugs” em `getPreviousDayOfWeek`, `getFollowingDayOfWeek` e `getNearestDayOfWeek` [T6] foram consertados.

O teste de unidade `testGetNearestDayOfWeek` (linha 329), que testa o método `getNearestDayOfWeek` (linha 705), não rodou por muito tempo e não foi completo como está configurado para ser. Adicionei muitos casos de teste ao método, pois nem todos os meus iniciais passaram [T6]. Você pode notar o padrão de falhas ao verificar quais casos de teste estão como comentários [T7].

Isso mostra que o algoritmo falha se o dia mais próximo estiver no futuro. Obviamente, há um tipo de erro de condição de limite [T5].

O padrão do que o teste cobre informado pelo Clover também é interessante [T8]. A linha 719 nunca é executada! Isso significa que a estrutura `if` na linha 718 é sempre falsa. De fato, basta olhar o código para ver que isso é verdade. A variável `adjust` é sempre negativa e, portanto, não pode ser maior ou igual a 4. Sendo assim, este algoritmo está errado.

O algoritmo correto é disponibilizado abaixo:

```

int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
    adjust -= 7;

return SerialDate.addDays(adjust, base);

```

Por fim, podem-se fazer os testes na linha 417 e 429 obterem êxito simplesmente lançando uma `IllegalArgumentException` ao invés de uma string de erro a partir de `weekInMonthToString` e `relativeToString`.

Com essas alterações, todos os teste de unidade passam, e creio eu que agora a `SerialDate` funcione. Portanto, está na hora de torná-la “certa”.

## Então, torne-a certa

Iremos abordar a `SerialDate` de cima para baixo, aperfeiçoando-a no caminho. Embora você não veja esse processo, passarei todos os teste de unidade do `JCommon`, incluindo o meu melhorado para a `SerialDate`, após cada alteração que eu fizer. Portanto, pode ter certeza de que cada mudança que você vir aqui funciona com todos os testes do `JCommon`.

Começando pela linha 1, vemos uma grande quantidade de comentários sobre licença, direitos autorais, criadores e histórico de alterações. Reconheço que é preciso tratar de certos assuntos legais. Sendo assim, os direitos autorais e as informações sobre a licença devem permanecer. Por outro lado, o histórico de alterações é um resquício da década de 1960, e deve ser excluído [C1]. Poderia-se reduzir a lista de importações (`import`) na linha 61 usando `java.text.*` e `java.util.*`. [J1]

Não gostei da formatação HTML no Javadoc (linha 67), pois o que me preocupa é ter um arquivo fonte com mais de uma linguagem contida nele. E ele possui quatro: Java, português, Javadoc e html [G1]. Com tantas linguagens assim, fica difícil manter tudo em ordem. Por exemplo, a boa posição das linhas 71 e 72 ficam perdidas quando o Javadoc é criado, e, mesmo assim, quer deseja ver `<ul>` e `<li>` no código-fonte? Uma boa estratégia seria simplesmente envolver todo o comentário com `<pre>` de modo que a formatação visível no código-fonte seja preservada dentro do Javadoc<sup>1</sup>.

A linha 86 é a declaração da classe. Por que essa classe se chama `SerialDate`? Qual o significado de “serial”? Seria por que a classe é derivada de `Serializable`? Parece pouco provável.

Não vou deixar você tentando adivinhar. Eu sei o porquê (pelo menos acho que sei) da palavra “serial”. A dica está nas constantes `SERIAL_LOWER_BOUND` e `SERIAL_UPPER_BOUND` nas linhas 98 e 101. Uma dica ainda melhor está no comentário que começa na linha 830. A classe se chama `SerialDate` por ser implementada usando-se um “serial number” (número de série, em português), que é o número de dias desde 30 de dezembro de 1899.

Tenho dois problemas com isso. Primeiro, o termo “serial number” não está correto. Isso pode soar como uma crítica, mas a representação está mais para uma medida do que um número de série. O termo “serial number” tem mais a ver com a identificação de produtos do que com datas. Portanto, não acho este nome descriptivo [N1]. Um termo mais explicativo seria “ordinal”.

O segundo problema é mais significativo. O nome `SerialDate` implica uma implementação.

<sup>1</sup> Uma solução ainda melhor seria fazer o Javadoc apresentar todos os comentários de forma pré-formatada, de modo que tivessem o mesmo formato no código

Essa classe é abstrata, logo não há necessidade indicar coisa alguma sobre a implementação, a qual, na verdade, há uma boa razão para ser ocultada. Sendo assim, acho que esse nome esteja no nível errado de [N2]. Na minha opinião, o nome da classe deveria ser simplesmente Date.

Infelizmente, já existem muitas classes na biblioteca Java chamadas de Date. Portanto, essa, provavelmente, não é o melhor nome. Como essa classe é sobre dias ao invés de horas, pensei em chamá-la de Day (dia em inglês), mas também se usa demasiadamente esse nome em outros locais. No final, optei por DayDate (DiaData) como a melhor opção.

A partir de agora, usarei o termo DayDate. Mas espero que se lembre de que nas listagens as quais você vir, DayDate representará `Serializable`.

Entendo o porquê de DayDate herdar de Comparable e Serializable. Mas ela herda de MonthConstants? Esta classe (Listagem B.3, página 372) é um bando de constantes estáticas finais que definem os meses. Herdar dessas classes com constantes é um velho truque usado pelos programadores Java de modo que não precisassem usar expressões como MonthConstants.January – mas isso é uma péssima idéia [J2]. A MonthConstants deveria ser realmente um enum.

```
public abstract class DayDate implements Comparable,  
    Serializable {  
  
    public static enum Month {  
        JANUARY(1),  
        FEBRUARY(2),  
        MARCH(3),  
        APRIL(4),  
        MAY(5),  
        JUNE(6),  
        JULY(7),  
        AUGUST(8),  
        SEPTEMBER(9),  
        OCTOBER(10),  
        NOVEMBER(11),  
        DECEMBER(12);  
  
        Month(int index) {  
            this.index = index;  
        }  
  
        public static Month make(int monthIndex) {  
            for (Month m : Month.values()) {  
                if (m.index == monthIndex)  
                    return m;  
            }  
            throw new IllegalArgumentException("Invalid month index " + monthIndex);  
        }  
        public final int index;  
    }  
}
```

Alterar a MonthConstants para este enum exige algumas alterações na classe DayDate e para todos os usuários. Levei uma hora para fazer todas as modificações. Entretanto, qualquer função costumava pegar um int para um mês, agora pega um enum Month. Isso significa que podemos nos livrar do método isValidMonthCode (linha 326) e de toda verificação de erro no código dos Month, como aquela em monthCodeToQuarter (linha 356) [G5].

Em seguida, temos a linha 91, serialVersionUID – variável usada para controlar o “serializador”.

Se a alterarmos, então qualquer `DayDate` criado com uma versão antiga do software não será mais legível e produzirá uma `InvalidClassException`. Se você não declarar a variável `serialVersionUID`, o compilador gerará automaticamente uma para você, e ela será diferente toda vez que você alterar o módulo. Sei que todos os documentos recomendam controlar manualmente essa variável, mas me parece que o controle automático de serialização é bem mais seguro [G4]. Apesar de tudo, eu prefiro depurar uma `InvalidClassException` ao comportamento estranho que surgiria caso eu me esquecesse de alterar o `serialVersionUID`. Sendo assim, vou excluir a variável – pelo menos por agora<sup>2</sup>.

Acho o comentário da linha 93 redundante. Eles só servem para passar mentiras e informações erradas [C2]. Portanto, vou me livrar de todo esse tipo de comentários.

Os comentários das linhas 97 e 100 falam sobre os números de série (serial numbers) que mencionei anteriormente [C1]. As variáveis que eles representam são as datas mais antigas e atuais possíveis que a `DayDate` pode descrever. É possível esclarecer isso um pouco mais [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2;      // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

Não está claro para mim porque `EARLIEST_DATE_ORDINAL` é 2 em vez de 0. Há uma dica no comentário na linha 829 sugerindo que tem algo a ver com a forma de representação das datas no Microsoft Excel. A `SpreadsheetDate` (Listagem B.5, página 382), uma derivada de `DayDate`, possui uma explicação muito mais descritiva. O comentário na linha 71 descreve bem a questão. Meu problema com isso é que a questão parece estar relacionada à implementação de `SpreadsheetDate` e não tem nada a ver com `DayDate`. Cheguei à conclusão de que `EARLIEST_DATE_ORDINAL` e `LATEST_DATE_ORDINAL` realmente não pertencem à `DayDate` e devem ser movidos para `SpreadsheetDate` [G6].

Na verdade, uma busca pelo código mostra que essas variáveis são usadas apenas dentro de `SpreadsheetDate`. Nada em `DayDate` ou em qualquer outra classe no framework JCommon as usa. Sendo assim, moverei-as abaixo para `SpreadsheetDate`.

As variáveis seguintes, `MINIMUM_YEAR_SUPPORTED` e `MAXIMUM_YEAR_SUPPORTED` (linhas 104 e 107), geram um dilema. Parece claro que, se `DayDate` é uma classe abstrata que não dá nenhuma indicação de implementação, então ela não deveria nos informar sobre um ano mínimo ou máximo. Novamente, fico tentado a mover essas variáveis abaixo para `SpreadsheetDate` [G6]. Entretanto, uma busca rápida pelos usuários que as usam mostra que uma outra classe as usa: `RelativeDayOfWeekRule` (Listagem B.6, página 390). Vemos que, nas linhas 177 e 178 na função `getDate`, as variáveis são usadas para verificar se o parâmetro para `getDate` é um ano válido. O dilema é que um usuário de uma classe abstrata necessita de informações sobre sua implementação.

O que precisamos é fornecer essas informações sem poluir a `DayDate`. Geralmente, pegaríamos as informações da implementação a partir de uma instância de uma derivada. Entretanto, a função `getDate` não recebe uma instância de uma `DayDate`, mas retorna tal instância, o que significa que, em algum lugar, ela a deve estar criando. Da linha 187 a 205, dão a dica. A instância de `DayDate` é criada por uma dessas três funções: `getPreviousDayOfWeek`, `getNearestDayOfWeek` ou `getFollowingDayOfWeek`.

Olhando novamente a listagem de `DayDate`, vemos que todas essas funções (linhas 638-724)

<sup>2</sup> Diversas pessoas que revisaram novamente esse texto fizeram objeção a essa decisão. Elas argumentaram que em uma framework de código aberto é melhor manter a mesma versão em vez de variabilizar os IDs de modo que pequenas alterações no software não invalidem as datas antigas serializadas. É um

retornam uma data criada por addDays (linha 571), que chama createInstance (linha 808), que cria uma SpreadsheetDate! [G7].

Costuma ser uma péssima ideia para classes base enxergar seus derivados. Para consertar isso, devemos usar o padrão ABSTRACT FACTORY<sup>3</sup> e criar uma DayDateFactory. Essa factory criará as instâncias de DayDate que precisamos e também poderá responder às questões sobre a implementação, como as datas mínima e máxima.

```
public abstract class DayDateFactory {  
    private static DayDateFactory factory = new SpreadsheetDateFactory();  
    public static void setInstance(DayDateFactory factory) {  
        DayDateFactory.factory = factory;  
    }  
  
    protected abstract DayDate _makeDate(int ordinal);  
    protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);  
    protected abstract DayDate _makeDate(int day, int month, int year);  
    protected abstract DayDate _makeDate(java.util.Date date);  
    protected abstract int _getMinimumYear();  
    protected abstract int _getMaximumYear();  
  
    public static DayDate makeDate(int ordinal) {  
        return factory._makeDate(ordinal);  
    }  
  
    public static DayDate makeDate(int day, DayDate.Month month, int year) {  
        return factory._makeDate(day, month, year);  
    }  
  
    public static DayDate makeDate(int day, int month, int year) {  
        return factory._makeDate(day, month, year);  
    }  
  
    public static DayDate makeDate(java.util.Date date) {  
        return factory._makeDate(date);  
    }  
  
    public static int getMinimumYear() {  
        return factory._getMinimumYear();  
    }  
  
    public static int getMaximumYear() {  
        return factory._getMaximumYear();  
    }  
}
```

Essa classe factory substitui os métodos createInstance pelos makeDate, que melhora um pouco os nomes [N1]. O padrão se vira para SpreadsheetDateFactory, mas que pode ser modificado a qualquer hora para usar uma factory diferente. Os métodos estáticos que delegam responsabilidade aos métodos abstratos para usarem uma combinação dos padrões SINGLETON<sup>4</sup>, DECORATOR<sup>5</sup> e ABSTRACT FACTORY, que tenho achado úteis.

A SpreadsheetDateFactory se parece com isso:

```
public class SpreadsheetDateFactory extends DayDateFactory {  
    public DayDate _makeDate(int ordinal) {  
        return new SpreadsheetDate(ordinal);  
    }  
  
    public DayDate _makeDate(int day, DayDate.Month month, int year) {  
        return new SpreadsheetDate(day, month, year);  
    }  
  
    public DayDate _makeDate(int day, int month, int year) {  
        return new SpreadsheetDate(day, month, year);  
    }  
  
    public DayDate _makeDate(Date date) {  
        final GregorianCalendar calendar = new GregorianCalendar();  
        calendar.setTime(date);  
        return new SpreadsheetDate(  
            calendar.get(Calendar.DATE),  
            DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),  
            calendar.get(Calendar.YEAR));  
    }  
  
    protected int _getMinimumYear() {  
        return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;  
    }  
  
    protected int _getMaximumYear() {  
        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;  
    }  
}
```

Como pode ver, já movi as variáveis `MINIMUM_YEAR_SUPPORTED` e `MAXIMUM_YEAR_SUPPORTED` para onde elas pertencem [G6], em `SpreadsheetDate`.

A próxima questão na `DayDate` são as constantes `day` que começam na linha 109. Isso deveria ser outro enum [J3]. Já vimos esse padrão antes, portanto, não o repetirei aqui. Você o verá nas últimas listagens.

Em seguida, precisamos de uma série de tabelas começando com `LAST_DAY_OF_MONTH` na linha 140. Meu primeiro problema com essas tabelas é que os comentários que as descrevem são redundantes [C3]. Os nomes estão bons. Sendo assim, irei excluir os comentários.

Parece não haver uma boa razão para que essa tabela não seja privada [G8], pois existe uma função `lastDayOfMonth` estática que fornece os mesmos dados.

A próxima tabela, `AGGREGATE_DAYS_TO_END_OF_MONTH`, é um pouco mais enigmática, pois não é usada em lugar algum no framework JCommon [G9]. Portanto, excluí-a.

O mesmo vale para `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Apenas a `SpreadsheetDate` (linhas 434 e 473) usa a tabela seguinte, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`. Isso leva à questão se ela deve ser movida para `SpreadsheetDate`. O argumento para não fazê-lo é que a tabela não é específica a nenhuma implementação em particular [G6]. Por outro lado, não há implementação além da `SpreadsheetDate`, e, portanto, a tabela deve ser colocada próxima do local onde ela é usada.

Para mim, o que decide é que para ser consistente [G11], precisamos tornar a tabela privada e

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 infinidáveis 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 desenho e 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