

Introdução ao Código Limpo

Código Limpo

- Os programas de computador são modelos matemáticos realizados a partir de necessidades humanas.
- As necessidades humanas, porém, são descritas de maneira complexa e abstrata através da linguagem humana.
- Já os modelos matemáticos são formais e, portanto, não podem ser gerados através de sentimentos (*feelings*) vagos.

Código Limpo

- A materialização dos modelos matemáticos em programa de computador é o código.
- Por isso, o código é indispensável.
- O código é o que torna concreto o requisito abstrato.

Código Limpo

- A qualidade do código importa.
- Código de má qualidade, também chamado “código ruim”, impacta negativamente na produtividade das equipes.
- A maior parte do ciclo de vida do *software* é a sua manutenção.
- Além disso, o código não é de um desenvolvedor, mas de uma equipe.
- Isto significa que espera-se que a maior parte da vida de um *software* será preenchido por pessoas que não escreveram um pedaço específico de código o lendo.

Código Limpo

- É comum a justificativa da pressão dos prazos para a criação “código ruim”.
- Entretanto, o “código ruim” lentifica ainda mais o desenvolvimento e incentiva o descumprimento dos prazos.

Código Limpo

- Escrever “código bom”, porém, não é algo simples ou natural.
- É uma disciplina que envolve a prática de diversas técnicas.
- A palavra prática, neste caso, é muito importante porque não basta saber diferenciar o “código bom” do “código ruim”, é preciso praticar a escrita de código bom.

Código Limpo

Um código limpo tem, segundo alguns grandes nomes da indústria, as seguintes características:

- alta legibilidade;
- facilidade para ser alterado;
- mínimo de dependências;
- possui testes unitários e de aceitação;
- sem duplicações;

As características não se limitam a esta lista, mas estas são as mais importantes.

Nomes

- Na programação, todas as entidades (classes, métodos (ou funções), atributos (ou variáveis), módulos (ou pacotes)) acabam recebendo nomes únicos.
- A correta nomeação destes é fator importante para a escrita de código limpo.

Nomes devem revelar intenção

- Os nomes das entidades devem deixar claro a todos a função destas.
- Um bom nome é um nome que diz: o que é; o que faz; e como é usado.

Nomes devem revelar intenção

Tenham-se o código abaixo:¹

```
int d; \\ elapsed time in days
```

Ele poderia ser reescrito para:

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;
```

Observe que as três variáveis do código embaixo informam muito mais do que o código em cima.

¹Exemplo retirado de Martin, 2009.

Nomes devem revelar intenção

- O código abaixo faz parte de um jogo de campo minado.
- O que ele faz?²

```
public List<int []> getThem() {  
    List<int []> list1 = new ArrayList<int []>();  
    for (int [] x : theList) {  
        if (x[0] == 4) {  
            list1.add(x);  
        }  
    }  
    return list1;  
}
```

²Exemplo retirado de Martin, 2009.

Nomes devem revelar intenção

- Apesar de ser bastante simples, o código anterior é bastante difícil de ler.
- A começar pelo nome do método, que não explica quem é *them*.
- O que é `theList`?
- Por que fazer um teste do valor 4 contra o índice 0 do item de `theList`?

Nomes devem revelar intenção

Tendo o código antigo e o refatorado lado a lado, fica clara a diferença.

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList) {  
        if (x[0] == 4) {  
            list1.add(x);  
        }  
    }  
    return list1;  
}
```

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells =  
        new ArrayList<int[]>();  
    for (int[] cell : gameBoard) {  
        if (cell[STATUS.VALUE] == FLAGGED) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

Agora é possível perceber que o método retorna as células que foram marcadas com bandeira.

Nomes devem revelar intenção

E o código pode ser refatorado ainda mais uma vez.

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells =  
        new ArrayList<int[]>();  
    for (int[] cell : gameBoard) {  
        if (cell[STATUS_VALUE] == FLAGGED) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells =  
        new ArrayList<Cell>();  
    for (Cell cell : gameBoard) {  
        if (cell.isFlagged()) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

- Ao invés de usar uma vetor de inteiros, pode-se agrupar os dados da célula em um objeto célula.
- O método `isFlagged` torna a leitura do código ainda mais fluida.

Evitar a desinformação

- Poucas coisas podem ser piores que um nome que induz ao erro.
- Colocar o sufixo *List* em uma variável que não é uma lista pode causar muitos problemas.
- Deve-se evitar também nomes de métodos e/ou variáveis que são quase idênticos e podem levar o programador a se confundir.

Não usar palavras sem significado

- Variáveis, parâmetros e atributos como `aux` e `a1` simplesmente informam nada.
- Soma-se a isso os sufixos *Data* e *Info* que, na maioria das vezes, agregam valor nenhum.

Não usar palavras sem significado

- Um ponto um tanto controverso é sobre usar sufixos que expressem o tipo da variável ou atributo.
- Em alguns casos, especialmente em linguagens de tipagem dinâmica, esses sufixos podem até ser úteis.

Usar nomes pronunciáveis

- Nomes impronunciáveis são péssimas escolhas.
- Tenha-se o exemplo abaixo e sua refatoração:³

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = "102";  
};
```

Ele poderia ser reescrito para:

```
class Customer {  
    private Date generationTimestamp;  
    private Date modificationTimestamp;  
    private final String recordId = "102";  
    /* ... */  
};
```

³Exemplo adaptado de Martin, 2009.

Verbos e substantivos

- Métodos e funções devem ser nomeados com verbos, pois significam ações.
- Os atributos e variáveis devem ser nomeados com substantivos, já que significam coisas ou conceitos.

Uma palavra para cada conceito

- Cada conceito deve ser representado por uma única palavra.
- Se adicionar um item a uma estrutura de dados é através do método `add`, não faz sentido utilizar em outras partes do código palavras como **`insert`** ou **`append`**.
- Claro que, se há diferença semântica entre dois métodos que adicionam um item, pode-se utilizar duas palavras diferentes.
- Por exemplo, tenha-se dois métodos `insert` e `append`. O primeiro pode significar uma função que adiciona um item à estrutura de dados e o segundo indica que a adição é feita ao final da estrutura de dados.

Uma palavra para cada conceito

- Esta regra também se aplica a nome de classes e atributos.
- Os sufixos *Manager* e *Controller* geralmente significam a mesma coisa.
- Intercambiar estes termos no código pode gerar mais dificuldade de entendê-lo.

Contexto

- Em alguns casos, pode ser interessante inserir contexto, especialmente aos nomes de variáveis e atributos.
- O atributo `state` pode significar tanto uma unidade da federação quanto a situação corrente de alguma coisa.
- Porém, `addrState` deixa claro que é uma unidade da federação vinculada a um endereço.

Contexto

- Por outro lado, adicionar contexto, em alguns casos, pode ser ruim.
- `accountAddress` ou `costumerAddress` podem ser bons nomes para instâncias, mas péssimos para nomes de classes.
- `Address` parece um nome bastante melhor.

Funções

- A função é uma das entidades fundamentais de um programa de computador.
- Ela age como um subprograma que, a partir de um conjunto finito de entrada, calcula um conjunto finito de saída.

Funções

Tenha seguinte código de uma ferramenta de testes (FitNesse) apresentado em Clean Code:

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

Funções

Continua:

```

buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
    WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
        WikiPagePath tearDownPath = wikiPage.getPageCrawler().getFullPath(teardown);
        String tearDownPathName = PathParser.render(tearDownPath);
        buffer.append("\n").append("!include -teardown .")
            .append(tearDownPathName).append("\n");
    }
    if (includeSuiteSetup) {
        WikiPage suiteTeardown = PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE.TEARDOWN_NAME, wikiPage);
        if (suiteTeardown != null) {
            WikiPagePath pagePath = suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
            String pagePathName = PathParser.render(pagePath);
            buffer.append("!include -teardown .").append(pagePathName).append("\n");
        }
    }
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}

```

Funções

- É esperado que alguém que não conhece o contexto da ferramenta FitNesse tenha dificuldades em sequer conceber a ideia por trás da função mostrada.
- Entretanto, esse não é o maior dos problemas.
- A função `testableHtml` nesta forma apresenta inúmeros problemas de codificação.

Funções

- O primeiro problema é o nome da função e de algumas variáveis.
- `testableHtml` é um substantivo e não um verbo. Fica difícil dizer o que a função fará.
- Lendo com calma a função, percebe-se que ela tem como objetivo renderizar uma página de teste incluindo os *setups* e *teardowns*⁴
- Então, `renderPageWithSetupsAndTeardowns` seria um nome melhor.

⁴Em ferramentas de testes automatizados, é possível realizar ações antes (*setup* e após (*teardown*) a execução do teste.

Funções

- A variável `buffer` é um exemplo de variável mal nomeada.
- Ela não é um simples *buffer*, mas o conteúdo da página adicionados *setups* e *teardowns*;
- Martin sugere renomeá-la para `newPageContent`.

Funções devem ser pequenas

- Outro problema da função de exemplo é que ela é muito grande e com fluxo tão complexo a ponto de tornar difícil entender seu funcionamento.
- Uma função deveria ter pouquíssimas linhas.
- Robert Martin sugere até que os conteúdos dentro de estruturas de decisão e laços de repetição deveriam ser de apenas uma linha, possivelmente sendo uma chamada de função.
- Mais do que isso, os próprios condicionais destas estruturas podem ser funções.

Funções devem fazer apenas uma coisa

- O terceiro problema da função de exemplo está ligado também ao seu tamanho, mesmo que de maneira menos direta.
- `testableHtml` faz três ações por si mesmo.
- Ela verifica se é uma *suite* de testes, adiciona *setups* e *teardowns* e renderiza a página.
- O ideal é que as três primeiras atividades fossem funções e que a função renomeada para `renderPageWithSetupsAndTeardowns` só as chamasse.

Funções devem fazer apenas uma coisa

- Entretanto, se uma função chama várias funções que realizam outras atividades, ela estaria ainda assim fazendo estas atividades?
- A resposta rápida é que não.
- A questão aqui são os níveis de abstração.

Funções devem fazer apenas uma coisa

- Robert Martin sugere que se escreva as funções como parágrafos *PARA*.
- Assim, tem-se a função `renderPageWithSetupsAndTeardowns`:
- ① PARA renderizar a página incluindo *setups* e *teardowns*, nós incluimos os *setups*, então o conteúdo de teste da página, e, por fim, os *teardowns*.
 - ① PARA incluir *setups*, nós incluimos *setup* da *suite* de testes se for uma *suite*, então incluimos o *setup* normal.
 - ① PARA incluir *setup* da *suite* de testes. . .
- Cada parágrafo corresponde a um nível de abstração.
- Uma função deve-se restringir a um único nível de abstração.

Argumentos das funções

- Funções com poucos (ou nenhum) argumento são mais fáceis de ler.
- Entretanto, nem sempre será possível criar funções/métodos sem argumentos.
- Apesar disso, há técnicas para tentar reduzir a quantidade de argumentos ao mínimo.

Argumentos das funções

As funções com um argumento assumem geralmente três formas:

- ❶ `boolean fileExists(String filename)`
 - ❷ `InputStream fileOpen(String filename)`
 - ❸ `void recordNumberOfFailedAttempts(int nAttempts)`
- Nas duas primeiras formas, a função recebe um argumento, realiza um cálculo, e devolve um valor lógico.
 - Na terceira forma, a função é um evento que transforma o estado do sistema.

Argumentos das funções

- Há, porém, uma quarta forma que é bastante problemática.
- A função `void includeSetupPageInto(StringBuffer pageText)` usa como saída o próprio argumento de entrada.
- Note que exige que o leitor pare para pensar até chegar a esta conclusão.

Argumentos das funções

- Para reduzir a quantidade de argumentos de uma função, uma técnica simples é tentar livrar-se dos argumentos *flag*.
- *Flags* são passadas para funções para alterar o fluxo dessas.
- Faz mais sentido fazer mais de uma função para lidar com esse problema.
- Por exemplo, `render(boolean isSuite)` pode se transformar em `renderForSuite()` e `renderForSingleTest()`

Argumentos das funções

- Um ponto importante sobre argumentos é que, no caso de funções com mais de um argumento, deve-se observar a coesão e a ordem natural entre estes.
- Uma função que cria um ponto cartesiano ter dois argumentos (x e y) faz completo sentido.
- Até mesmo a ordem dos dois é a convencionada.
- Estivessem inversamente colocados, o código ficaria muito mais difícil de ler.

Argumentos das funções

- Outra técnica para reduzir a quantidade de argumentos é agrupar dois ou mais argumentos em um objeto.
- Tenha-se uma função que cria um círculo.
- Ela deve exigir as coordenadas do ponto central do círculo e o raio (ou diâmetro).
- As coordenadas do ponto central podem ser agrupadas em um argumento só através de um objeto que represente um ponto.

Não realizar efeitos colaterais

- As funções não devem realizar efeitos colaterais.
- Eles acabam gerando resultados não previstos para o programador e podem criar *bugs* desnecessários.

Não realizar efeitos colaterais

Tenha-se a função abaixo⁵:

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

- A função promete verificar se a senha é válida para um dado nome de usuário, porém ela faz mais que isso.
- Ela inicia uma nova seção.
- Em alguns casos isso significa encerrar uma seção ativa.

⁵Retirado de Martin, 2009.

Separação entre busca e comando

- Uma função serve ou para informar ou para realizar uma ação, nunca para ambas.
- Tenha-se o seguinte exemplo⁶:

```
public boolean set(String attribute , String value );  
if ( set("username", "unclebob" ) )...
```

- set atribui um valor a um atributo.
- Porém, como retorna um valor lógico, pode ser inserido dentro de um condicional.
- Para o leitor, a segunda linha pode o falso significado de dizer se username tem o valor unclebob.

⁶Retirado de Martin, 2009.

Separação entre busca e comando

- Outro hábito ruim, porém bastante comum é o uso de retorno de códigos de erro.
- Isto é muito comum em linguagens onde não há exceções.
- Em linguagens que implementam exceções, deve-se eliminar os códigos de erro nos retornos de função e utilizar as exceções.
- Códigos de erro obrigam quem recebeu o código de erro a tratá-lo imediatamente.
- Já as exceções dão mais flexibilidade ao programador.

Tratamento de erros

- Martin sugere que o tratamento de erros (geralmente cláusulas `catch` e `finally`) sejam realizadas em funções em separado.
- Isto serve para manter a função tendo responsabilidade única, pois tratar erro é uma responsabilidade.
- Por isso, faz sentido que haja uma função com esse objetivo.

Don't Repeat Yourself (DRY)

- Não apenas Martin, mas muitos outros autores falam sobre a não repetição de código.
- Em termos gerais, este é um bom conselho.
- Entretanto, deve-se tomar cuidado com a criação de abstrações (classes e métodos) desnecessárias.
- As repetições, às vezes, são momentâneas, então criar novas classes e métodos pode gerar complexidade desnecessária ao código.

Classes

- Assim como as funções (e métodos), as classes também devem ser pequenas.
- E isso tem a ver com o Princípio da Responsabilidade Única (*Single Responsibility Principle* - SRP)⁷.
- Um alerta para a violação deste princípio ocorre quando há uma classe com um número muito grande de métodos públicos.

⁷Uma classe só deve ter uma razão para ser mudada

Classes

- Robert Martin sugere que toda classe deve ser descrita em 25 palavras.
- Termos como "e", "ou", "mas" e "se" são indicadores de que a classe possui mais de uma responsabilidade.

Classes

- Uma maneira de demonstrar como o SRP funciona é através do exemplo mostrado em Martin (2009).
- Temos uma classe `Sql` abaixo:

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


Classes

- A partir da observação da classe `Sql`, nota-se que ela apenas recebe a tabela e um vetor de colunas.
- Então os métodos geram o código SQL requerido.
- O problema nesta classe é que ela pode ser modificada por dois motivos:
 - 1 para modificar um comando SQL;
 - 2 para adicionar um comando SQL.

Classes

A maneira adotada por Martin para resolver o problema é transformando Sql em uma classe abstrata e cada comando se tornar uma subclasse como mostrado (em parte) abaixo:

```
abstract public class Sql {  
    public Sql(String table, Column[] columns)  
    abstract public String generate();  
}  
  
public class CreateSql extends Sql {  
    public CreateSql(String table, Column[] columns)  
    @Override public String generate()  
}  
  
public class SelectSql extends Sql {  
    public SelectSql(String table, Column[] columns)  
    @Override public String generate()  
}  
  
public class InsertSql extends Sql {  
    public InsertSql(String table, Column[] columns, Object[] fields)  
    @Override public String generate()  
    private String valuesList(Object[] fields, final Column[] columns)  
}  
  
public class SelectWithCriteriaSql extends Sql {  
    public SelectWithCriteriaSql(String table, Column[] columns, Criteria criteria)  
    @Override public String generate()  
}
```

Classes

- Agora, para implementar um novo comando SQL, basta criar uma nova classe.
- Caso seja necessário modificar um comando, apenas uma classe será modificada.

Comentários

- É comum professores e livros de programação dizerem que códigos bem comentados são melhores que códigos sem comentários.
- Mas, de maneira geral, pra que servem os comentários?

Comentários

- A resposta mais comum é que os comentários tornam o código mais claro.
- Mas se o seu código é bem escrito e expressivo, então por que precisa de comentários?

Comentários

- Os comentários não devem ser vistos como a ambrosia⁸, mas como um mal necessário a ser utilizado em situações específicas.
- Eles adicionam mais um elemento dentro do código que deve ter manutenção.
- Pior do que código sujo e sem comentários é código (limpo ou sujo) com comentários errados ou mal colocados.

⁸Alimento dos deuses gregos.

Comentários

- A justificativa para os comentários surge, na maioria das vezes, para tentar resolver situações onde o programador não conseguiu escrever o código de maneira expressiva.
- Por isso que todo código limpo tende a ter poucos comentários.

Comentários

Tenha-se o seguinte exemplo⁹:

```
// Check to see if the employee is eligible for full benefits  
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

O comentário acima do código pode ser simplesmente retirado se o código for mais expressivo:

```
if (employee.isEligibleForFullBenefits())
```

⁹Retirado de Martin, 2009.

Bons comentários

- Nem todo comentário é ruim.
- Pelo contrário, alguns comentários agregam de maneira positiva ao código.

Informações legais

- O primeiro tipo de comentário útil é que trás questões legais sobre o código.
- Em alguns repositórios de código é comum, na parte superior do código, existir dados de *copyright* de marcas e a licença do código em questão.
- A questão é que alguns repositórios colocam a licença toda no arquivo.
- Isto é bastante desnecessário, bastando apenas ter referência à licença.

Explicação de intenções

- Outro tipo útil de comentário é quando, por mais que o código seja claro, o porquê de ter tomado a decisão pode não ser.
- Abaixo há um exemplo de explicação de intenções¹⁰:

```
public int compareTo(Object o)
{
    if(o instanceof WikiPagePath) {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, "");
        String compressedArgumentName = StringUtil.join(p.names, "")
        return compressedName.compareTo(compressedArgumentName);
    }
    return 1; // we are greater because we are the right type.
}
```

O programador explica que quando feita uma comparação, todos os objetos das outras classes devem ser considerados menores.

¹⁰Retirado de Martin, 2009.

Esclarecimentos

- Em alguns casos (bastante) específicos os comentários podem ajudar a partes do código em que não foi possível torná-lo mais claro.
- O exemplo abaixo mostra uma situação destas¹¹:

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");

    assertTrue(a.compareTo(a) == 0); // a == a
    assertTrue(a.compareTo(b) != 0); // a != b
    assertTrue(ab.compareTo(ab) == 0); // ab == ab
    assertTrue(a.compareTo(b) == -1); // a < b
    assertTrue(b.compareTo(a) == 1); // b > a
}
```

¹¹ Adaptado de Martin, 2009.

Aviso de consequências

- Algumas partes do código podem ser perigosas se executadas sem o devido cuidado.
- Comentários podem ser bastante úteis nesse caso.
- Pode-se observar no exemplo abaixo¹²:

```
// Don't run unless you have some time to kill.  
public void _testWithReallyBigFile()  
{  
    writeLinesToFile(10000000);  
    response.setBody(testFile);  
    response.readyToSend(this);  
    String responseString = output.toString();  
    assertSubString("Content-Length: 1000000000", responseString)  
    assertTrue(bytesSent > 1000000000);  
}
```

¹² Adaptado de Martin, 2009.

Bons comentários

- Comentários do tipo **TODO** também são úteis porque lembram ao programador o que ele deve fazer.
- Há também casos onde o comentário serve para enfatizar a importância de um pedaço do código. Isto evita que outro programador mude aquela parte do código sem levar em contato seus impactos.
- Por último, mas não menos importantes, há os comentários que geram documentação. Estes devem ser utilizados em APIs públicas, não fazendo tanto sentido em métodos e classes privados.

Maus comentários

- Além de saber os comentários que são úteis, é interessante saber quais tipos de comentários são ruins.
- Ao fim, deverá ser percebido que boa parte dos comentários ruins surgem da prática de comentar somente por comentar.

Comentários enigmáticos

- O primeiro tipo de comentário inútil são os comentários que mais parecem enigmas do que comentários em si.
- Tenha-se o exemplo abaixo¹³, onde o comentário é um enigma:

```
public void loadProperties() {  
    try {  
        String propertiesPath = propertiesLocation + "/" +  
            PROPERTIES.FILE;  
        FileInputStream propertiesStream =  
            new FileInputStream(propertiesPath);  
        loadedProperties.load(propertiesStream);  
    }  
    catch(IOException e){  
        // No properties files means all defaults are loaded  
    }  
}
```

- A exceção ocorre quando todos os arquivos padrão são carregados. Mas não diz quando e quem faz isso.
- Comentários que fazem o leitor ter de olhar em outros arquivos, não são bons comentários.

¹³ Adaptado de Martin, 2009.

Comentários óbvios

- Comentários que explicam o óbvio são como criar urubus pra comer os olhos do programador.
- Além de agregarem nada, ainda ocupam espaço e obrigam os programadores a modificá-los toda vez que o código for modificado.

Comentários redundantes e enganosos

Abaixo, há um exemplo de comentário que revela-se como o mais puro exercício do óbvio¹⁴:

```
// Utility method that returns when this.closed is true. Throws an ex
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
    throws Exception
{
    if (!closed){
        wait(timeoutMillis);
        if (!closed)
            throw new Exception(
                "MockResponseSender could not be closed");
    }
}
```

- E fica pior ainda porque o comentário não diz a verdade.
- A função não retorna **quando** `closed` é verdadeiro, mas **se** `closed` é verdadeiro.
- Apesar de ser algo sutil, pode fazer toda a diferença.

¹⁴ Adaptado de Martin, 2009.

Comentários

- Robert Martin trás outros diversos exemplos de comentários ruins.
- O que todos eles têm em comum é que ou são gerados pela má qualidade do código ou são completamente inúteis.
- Como linha geral, antes e após escrever um comentário, se pergunte se ele é realmente necessário e porquê ele é necessário.

Formatação

- A formatação geral dos arquivos também é bastante importante para a legibilidade do projeto como um todo.
- Arquivos pequenos tendem a ser lidos com mais facilidade.

Densidade vertical

- Um conceito importante é a densidade vertical.
- Quanto mais verticalmente denso, menos linhas em branco um arquivo terá.
- As linhas em branco exercem o importante papel de separar coisas que não são semanticamente ligadas.
- Colocar uma linha em branco após a declaração do pacote ou a importação de classes e/ou funções é interessante para facilitar a visualização.
- Linha em branco entre o bloco de declaração dos atributos e os métodos também é bom para legibilidade.

Densidade vertical

- Não existe uma regra definida e estrita.
- A inserção de linhas em branco deve ser também uma questão de bom senso e, se possível, um padrão para todo projeto.

Distância vertical

- Em linguagens como Pascal e C, as variáveis devem ser declaradas no início das funções.
- Isso não é verdade em diversas outras linguagens.
- Neste caso, é sugerido colocar a declaração da variável próximo de onde ela será utilizada na função.
- Por exemplo, se uma variável será utilizada dentro de um laço de repetição, ela deve ser declarada logo antes do laço ou dentro deste.

Distância vertical

- Exceto em linguagens como Pascal, C e C++, onde as funções/métodos devem ser declarados antes de serem invocados, Robert Martin sugere colocar as funções que serão invocadas logo após a função que as invoca.
- Segundo ele, isso faz com que o código siga a lógica de uma matéria de jornal, onde os conceitos mais gerais são apresentados antes dos conceitos mais específicos.
- Métodos com o mesmo nome sobrecarregados devem estar próximos também para evitar que o leitor tenha que procurar as demais versões do método pelo código.

Densidade horizontal

- Assim como a quantidade de linhas de um arquivo é importante para sua leitura, o tamanho dessas linhas também o é.
- Linhas muito longas são mais difíceis de ler e não se deve confiar na quebra de linhas feitas pelos editores.
- É difícil precisar a quantidade de caracteres por linha, mas geralmente recomenda-se algo entre 100 e 120 linhas.

Formatação

- Equipes mais profissionais geralmente definem um padrão de formatação dos arquivos.
- Empresas, a exemplo do Google, usam programas automatizados que impedem que códigos fora do padrão de formatação sejam adicionados aos seus repositórios.

Formatação

- O Google publica seu padrão de formatação para diversas linguagens. Estes padrões são vistos pela indústria como práticas bastante racionais e servem como base para muitas equipes.
- Extensões para IDEs, como a extensão da RedHat para Java no VS Code, também ajudam os programadores a manter um bom padrão de formatação de código.

Código limpo

- Escrever código limpo exige não apenas bom senso, mas grande dose de disciplina.
- Mais do que isso, exige responsabilidade do desenvolvedor. Ele deve se importar com o código e vê-lo como fruto do seu trabalho.

Referências

- Martin, Robert Cecil. Clean Code: A Handbook of Agile Software Craftsmanship. 2009. Prentice Hall.