Questão B15: Clean Code

Vasco Ramos [88931], 30/01/2021

À luz dos dados disponibilizados por um relatório do *GitHub*, é percetível que, tanto em equipas presenciais como em remotas, a indústria de *software* cresce desmesuradamente. Isto e a crescente proliferação de *self-taught developers*, presenteia-nos com um momento decisivo para olhar com maior detalhe para a qualidade e *cleanliness* do código desenvolvido. Tal como Robert Martin explora no livro *Clean Code* [1], cada linha de código é importante e deve ser devidamente ponderada, pois, eventualmente será lida por outro *developer*, ou nós próprios, no futuro. Ora, se o código não for claro, então será mais difícil corrigir problemas ou integrá-lo com novas funcionalidades.

Para fundamentar e demonstrar esta abordagem, apresentam-se os seguintes exemplos.

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

(a) Original

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

(b) Refactored

Figura 1: Nomes com contexto

Como é percetível na figura 1a, existem vários problemas ao nível da compreensão e contexto como: que informação está contida em *theList* ou qual o significado da entrada 0 de cada *x* ou, mesmo, como usar o valor de retorno?

Estas questões aparecem, pois, toda a nomenclatura da função e respetivas variáveis não transmitem clareza. Assumindo que a função serve para identificar e retornar as células que estão *flagged*, de um dado jogo de tabuleiro, a figura 1b apresenta uma alternativa mais clara. Para isto fez-se alterações como o nome da função que agora exprime claramente o seu objetivo e a utilização de uma classe *Cell* que substitui a estrutura original e encapsula a lógica para verifcar se uma célula está, ou não, *flagged*.

```java
public static String renderPageWithSetupAndTeardowns(PageData pageData,
                                                      boolean isSuite) throws Exception {

  boolean isTestPage = pageData.hasAttribute("Test");

  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

(a) Original

```java
public static String renderPageWithSetupAndTeardowns(PageData pageData,
                                                      boolean isSuite) throws Exception {

  if (isTestPage(pageData)) {
    includeSetupAndTeardownPages(pageData, isSuite);
  }

  return pageData.getHtml();
}
```

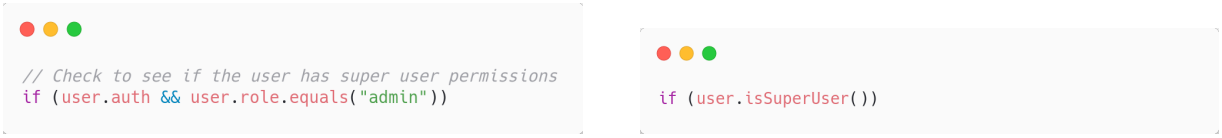
(b) Refactored

Figura 2: Tamanho e responsabilidade de uma função ([1], pp. 32-35)

O segundo exemplo aborda o problema de ter funções demasiado extensas e que apesar de terem um único objetivo, fazem mais do que uma coisa. Como é visível na figura 2a, esta função tem alguma complexidade e um conjunto de ações associadas demasiado extenso. Uma forma de simplificar este método é aplicar o que se chama de extração de métodos. Isto permite desacopolar a função em sub-funções, distribuindo responsabilidades e tornando as funções mais curtas e simples de ler, tal como se pode ver na figura 2b.

O terceiro e último exemplo aborda o problema dos comentários. Regra geral, comentários é algo que acrescenta valor ao código, contudo comentários a mais e desnecessários são um indicativo de que o

código não é claro. Um exemplo disso é usar um comentário para explicar uma porção de código que devia ser curta e auto-explicativa. A figura 3 mostra como a distribuição de responsabilidades e correta nomenclatura pode remover a necessidade de comentários que nunca deviam ter existido.



(a) Original (b) Refactored

Figura 3: Comentários desnecessários

Para lidar com o problema de assegurar que o código desenvolvido segue padrões de qualidade, tem vindo a ser aplicado um conjunto de práticas que se baseiam em 3 principais pilares: ***Code Style***, **Análise Estática** e ***Code Review***.

Atualmente é comum projetos seguirem um *Code Style*, ou seja, uma série de regras e padrões com o propósito de tornar o código mais coeso e uniforme (começa a ser cada vez mais frequente projetos de grande dimensão criarem o seu próprio *Code Style Guide*, tal como a *Airbnb*).

Contudo, apenas definir e utilizar um *Code Style* num projeto não é suficiente para garantir a sua correta aplicação. Para isso existem processos de análise estática e/ou *linting*, que permitem fazer uma análise do código, procurando por vulnerabilidades de segurança, duplicação de código, *code smells* (p.ex: funções demasiado extensas), tendo a vantagem de muitas ferramentas permitirem definir padrões específicos a um projeto, sendo possível integrar a validação de regras de *code style* próprias.

O último pilar relaciona-se com *Code Review* por pares. É importante perceber que a interação com outros *developers* é essencial para promover uma cultura de constante aprendizagem. Nesta perspetiva, as práticas de *code review* são ideais para fomentar essa cultura e facilitam a partilha de conhecimento através do *feedback* direto em pedaços de código, referindo possíveis melhorias de forma localizada.

References

[1] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 1 ed., 2008.