Padrões e Desenho de Software

Professores:
José Luis Oliveira
Sérgio Matos

Clean Code

Hugo Paiva, 93195

DETI
Universidade de Aveiro
05-06-2020

Conteúdo

1 Clean Code

1.1 Introdução

Clean Code é o conceito de um código fácil de entender e susceptível a mudanças que ganhou relevância em 2008, quando *Robert Cecil Martin* o mostrou ao mundo através do seu livro *Clean Code: A Handbook of Agile Software Craftsmanship*. Neste livro, *Robert* apresenta ao detalhe as técnicas e princípio ideais para o desenvolvimento de software de fácil compreensão.

Um sistema é algo que nunca está terminado, que necessita sempre de ser atualizado, quer seja devido à implementação de novas funcionalidade, resolução de problemas ou, até, devido a se ter tornado obsoleto. Ao longo destes ciclos de desenvolvimento, de modo a reduzir os custos de manutenção, é imperial a utilização de código limpo. Uma fraca qualidade de código leva a uma grande carga cognitiva, sendo necessárias mais horas de trabalho para a resolução de problemas. O problema é tão relevante que o livro refere um rácio de 10 leituras de código até começar a escrita.

Um código limpo é, portanto, algo que leva tempo, atenção e dedicação mas que tem resultados visíveis.

1.2 Princípios e Boas Práticas

Estando profundamente associado à complexidade de um sistema, ou seja, quanto mais limpo está o código, menos complexidade o sistema tem, é natural que o estudo dos sintomas desta complexidade seja um fator relevante para a escrita de um código limpo.

Com isto em mente, *Robert* definiu alguns princípios e boas práticas para chegar ao conceito de *Clean Code*, tendo em conta toda a sua experiência em programação desde os anos 70. De entre os referidos no livro, destacaram-se os seguintes:

- **Nomes Significativos**
- **Organização de Funções**
- **Organização das Classes e Estruturas de Dados**
- **Comentários Expressivos**
- **Formatação de Código**
- **Tratamento de Erros**
- **Testes Limpos**

1.2.1 Nomes Significativos

Tal como o próprio princípio diz, os nomes devem ser significativos e de grande importância para manter um código compreensível. Independentemente do tipo de nome (funções, variáveis, métodos, etc.), segundo este princípio, os nomes devem seguir dois pontos principais:

- Ir diretos ao ponto, passando a sua ideia central
- Em caso de necessidade, utilizar nomes grandes sem preocupações, garantindo a sua compreensão

Dito isto, são de evitar exemplos deste tipo, onde é pouco perceptível qual o contexto do problema:

```
int[] f; // frutas

for (int l=0; l<50; l++) {
    if (f[l] == 1) {
        f[l] = 2;
    }
}
```

Ao invés, deve-se utilizar abordagens do género:

```
int fruitsAtHome;

final int NUMBER_OF_FRUITS = 50;
for (int l=0; l<NUMBER_OF_FRUITS; l++) {
    if (fruitsAtHome[l] == ROTTEN) {
        fruitsAtHome[l] = TRASH;
    }
}
```

Existem, no entanto, outros aspetos que devem ser tido em consideração. Evitar símbolos e emojis, utilizar nomes de facil pronunciação, utilizar verbos em métodos e nomes em classes e utilizar sempre as mesmas palavras para um determinado contexto (utilizar sempre get ao invés de fetch) são alguns destes aspetos.

1.2.2 Organização de Funções

Segundo o autor deste conceito, existem duas regras para a criação das funções:

- As funções devem ser pequenas
- As funções devem ser ainda mais pequenas

O objetivo com este trocadilho de regras é manter as funções com o mínimo de funcionalidades possíveis, permitindo uma menor complexidade ao longo do programa e, utilizando os nomes significativos, o código deverá estar organizado de maneira a que qualquer pessoa consiga ver todos os percursos ao longo da execução do programa facilmente. Aliás, durante o livro, é referido que as funções apenas devem fazer uma coisa.

Deve-se, também, utilizar menos argumentos, evitando, novamente, o aumento de complexidade:

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

A declaração desta função é, sem dúvida, mais clara que a seguinte:

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

Basicamente, continua-se a tendência de manter a menor complexidade possível, evitando efeitos secundários para além do objetivo das funções, repetições de código (DRY -Don't Repeat Yourself), *output* de vários argumentos mas utilizando tratamento de erros.

1.2.3 Organização das Classes e Estruturas de Dados

As classes devem manter-se pequenas e respeitar ao máximo princípios como o da *Single Responsibility Principle* e *Open-Closed Principle*, mantendo apenas uma responsabilidade e permitindo extensões mas não modificações, novamente, devido à complexidade.

Como consequência, todas as variáveis devem manter-se privadas ou, no máximo, *protected*, de modo a permitir testes. A abstração dos dados passa a ser uma prioridade com a criação de interfaces mais concisas, passando, por exemplo, de:

```
public interface Phone {
    double getBatteryCapacityInMah();
    double getMahOfEnergyRemaing();
}
```

Para:

```
public interface Phone {
    double getPercentBatteryRemaining();
}
```

Toda a comunicação entre classes deve também ser reduzida, comunicando apenas com as que se relacionam de alguma forma como por exemplo, com objetos criados em seus métodos, com outros objetos passados como argumentos, etc.

1.2.4 Comentários Expressivos

Apesar dos comentários serem uma parte importante da legibilidade do código, é também relevante verificar se estes se justificam:

```
// Verificar se o carro ja deixou de pagar IUC em Portugal
if (car.age>25 || (car.eletric == true))...
```

Neste caso, os comentários não são necessário. A abordagem mais correta seria algo deste género:

```
if (!car.paysIuc())...
```

A partir deste exemplo, é possível chegar à conclusão que um *Clean Code* não precisa realmente de comentário, o código deve ser auto explicativo. Ainda assim, caso seja necessário a utilização de comentários, estes devem ser claros, informativos e dar ênfase aos pontos importantes e nunca deixar código comentado em ambientes de produção.

1.2.5 Formatação de Código

Sem formatação de código, este passaria a ser praticamente impossível de ler pelos desenvolvedores. Deve-se ter em atenção a formatação na vertical como na horizontal no entanto, é necessário identificar os casos onde esta é necessário.

```
public class Student{
 private    int         number;
 private    String      name;

 public void    insertData(int   n,
                                  String    na){
  number = n;
  name =    na;
 }
```

Num caso destes, não faz sentido aplicar o alinhamento horizontal pois este dificulta a leitura dos tipos de dados bem como de outras informações relevantes.

```
public class Student {
    private int number;
    private String name;

    public void insertData(int r, String na) {
        number = n;
        name = na;
    }
```

Por outro lado, a formatação é muito vantajosa para para perceber em que contexto certas funções e variáveis estão, devido à profundidade no código, com recurso a espaços antes de si:

```
public class Dog {
   public Dog(String name) {
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      Dog myDog = new Dog( "Lazy" );
   }
}
```

1.2.6 Tratamento de Erros

Tendo em conta que os erros no desenvolvimento de software são recorrentes, é dever dos programadores preparar os seus programas para lidar com eles mas, de forma clara, de modo a manter a limpeza do código.

É aconselhada a utilização de exceções ao invés de *returns*, principalmente devido à sua lógica (*bug*) não ser escondida. Caso este seja utilizado, nunca se deve retornar *null* pois teremos que lidar com ele, o que gera ainda mais problemas.

Um código limpo tem, portanto, de apresentar robustez logo, deve ser garantida a consistência do programa mesmo quando os erros acontecem.

1.2.7 Testes Limpos

Um código só é considerado limpo após passar uma séries de testes, também estes limpos.

As principais regras que estes testes têm de seguir são de acordo com o acrónimo F.I.R.S.T:

- **Fast** - Devem ser rápidos, permitindo a sua execução múltiplas vezes vezes
- **Independent** - Devem ser independentes, evitando erros em cascata
- **Repeatable** - Devem permitir a repetição do teste em qualquer ambiente (num portátil, sem internet...)
- **Self-Validating** - Devem retornar um *boolean* para permitir saber eficazmente o resultado do teste
- **Timely** - Devem ser escritos antes do código, facilitando a sua execução no código e, também, pontuais

1.3 Clean Code aplicado no Mundo

Como já foi referido, um código limpo é algo que leva tempo, atenção e dedicação, algo que o sector empresarial, por vezes, não pretende despender. Como consequência, o código limpo acaba por ser a qualidade no triângulo da gestão de um projeto.

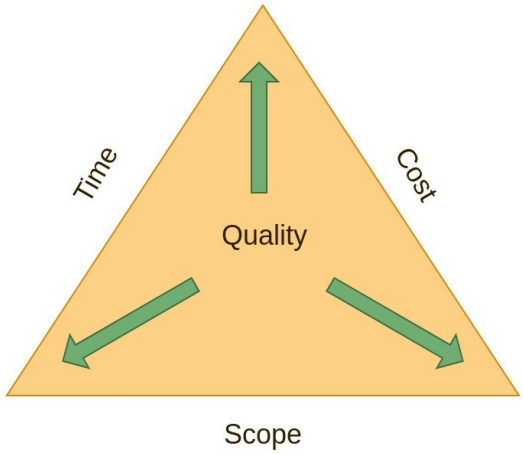

Figura 1: Representação do triângulo da gestão de um projeto

A partir da Figura 1, é possível observar as escolhas que um gestor de projeto tem em relação ao tempo que pretende gastar (***Time***), ao custo total para terminar o projeto (***Cost***) e às funcionalidade que este tem de ter implementadas para ser considerado como terminado (***Scope***). A qualidade do código, que neste contexto estamos a associar ao conceito de *Clean Code*, é uma relação direta destas 3 restrições.

No caso de um projeto com tempo limitado, este é o lado do triângulo que será fixo. Se pretendermos manter todas as funcionalidades, teremos de ter em consideração o custo para, por exemplo, contratar mais desenvolvedores de modo a cumprir os prazos estabelecidos no entanto, para isto acontecer, este lado teria de ser flexível, ou seja, o custo tem de suportar aumentos ou diminuições, de modo a manter a uniformidade no triângulo e assegurar o desenvolvimento do projeto.

É usual dizer-se que dentro destas 3 restrições, apenas é possível escolher 2. Logo, pegando no exemplo anterior, muito provavelmente a empresa não estaria interessada em aumentar os custos do projeto, deixando duas opções, ou diminui-se consideravelmente a qualidade do código, ou diminui-se as funcionalidades requeridas, algo que, devido à restrição do tempo, pode também não ser suficiente para assegurar a qualidade. Assim sendo, tendo em conta que muitos dos clientes de uma empresa de desenvolvimento de software apenas querem que um problema esteja resolvido, não estando interessados na qualidade do código, esta acaba por cair para segundo plano.

1.4 Conclusão

Concluindo, *Clean Code* é um conceito extremamente importante para reduzir os custos e tempo despendido na manutenção de um sistema a longo prazo mas, devido às prioridades do mundo empresarial face a um projeto, muitas das vezes estes princípios acabam por cair no esquecimento. Ainda assim, boas práticas como estas são fundamentais na aprendizagem de um desenvolvedor de software, garantido uma maior fiabilidade e robustez nos seus projetos futuros.

2 Bibliografia

[1] https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

[2] https://garywoodfine.com/what-is-clean-code/

[3] https://www.infoq.com/br/articles/clean-code-book-review/

[4] http://ceur-ws.org/Vol-2066/isee2018paper06.pdf

[5] https://simpleprogrammer.com/clean-code-principles-better-programmer/

[6] https://x-team.com/blog/principles-clean-code/

[7] https://codingsans.com/blog/clean-code

[8] https://www.hostgator.com.br/blog/clean-code-o-que-e/

[9] https://www.butterfly.com.au/blog/website-development/clean-high-quality-code-a-guide-on-how-to-b

[11] https://en.wikipedia.org/wiki/Project_management_triangle

[10] Martin, R. (2009). Clean code : a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall.