Universidade do Minho

Mestrado Integrado em Engenharia Informática

Gestão de Processo de Software 18/19

Caracterização de normas de codificação

João Pedro Ferreira Vieira A78468
Simão Paulo Leal Barbosa A77689

13 de Abril de 2019

Conteúdo

1 Introdução

O presente documento descreve o processo de estudo e desenvolvimento do trabalho prático da UC de *Gestão de Processo de Software*.

O tema do projeto passa pela caraterização de normas de codificação, no qual é obtido o acesso a uma lista de normas de codificação de uma dada organização, podendo ainda serem acrescentadas algumas outras normas que sejam consideradas relevantes e interessantes a considerar.

O objetivo, sendo assim, passa por estudar o conceito de normas de codificação e, de seguida, programar *scripts* que testem automaticamente a conformidade de código desenvolvido (neste caso em concreto, escrito em `C#`), com as normas estabelecidas.

2 *Static analysis*

As equipas de desenvolvimento de *software* estão por norma em alta pressão, tendo em conta os prazos de entrega de projetos e a necessidade de garantir a qualidade dos mesmos constantemente. A juntar a isto, os objetivos e restrições de desenvolvimento do produto tem que ser alcançados e os erros não são uma opção neste contexto. Sendo assim, surge a necessidade da utilização de ferramentas de *static analysis*.

O objectivo de *Static analysis* (ou análises estáticas em tradução para português) é encontrar defeitos no código fonte do software, sem a execução do mesmo.

Static code analysis é conseguido ao analisar código em comparação com regras de codificação definidas anteriormente.

Desta forma, este tipo de análise não envolve a execução dinâmica do *software* a ser testado e podem detetar possíveis defeitos numa fase inicial, sendo isto realizado depois de codificar e antes de executar testes unitários às partes construídas.

Static analysis é realizado com recurso a uma **máquina**, com o intuito de "percorrer"automaticamente o código-fonte e detetar regras que não estão a ser cumpridas. Normalmente, um exemplo mais utilizado para este tipo de processo é um compilador típico de uma dada linguagem. É também utilizado para forçar os programadores a não correrem riscos ou levar à criação de *bugs* associados a uma dada linguagem de programação, ao definirem regras que não devem ser usadas.

2.1 O que se procura alcançar?

Quando os programadores dão uso a este tipo de análises, algumas das métricas que procuram obter informação passam por:

- Linhas de código - por exemplo, uma dada empresa pode limitar a escrita de código de um dado projeto, podendo ter no máximo cada ficheiro 100 linhas de código.
- Frequência de comentários - com o intuito de perceber se o código está comentado o suficiente até para ser percetível por outros colegas do mesmo projeto.
- Aninhamento - relacionado com a complexidade do código.
- Número de chamadas de funções - perceber a utilzação dada às funções/métodos definidos e como as mesmas são executadas (complexidade).
- Complexidade ciclomática - de forma a evitar demasiada complexidade de ciclos definidos.
- entre outros ...

Alguns atributos de qualidade que podem ser o principal foco destas análises estáticas são:

- Manutenibilidade - facilidade com que se altera determinado *software* a fim de corrigir defeitos, adequar-se a novos requisitos, etc.
- Testabilidade - é o grau em que um determinado *software* suporta testes num determinado contexto de testes.

- Reutilização - o uso de *software* existente, ou do conhecimento de *software*, para a construção de um novo produto.
- Portabilidade - capacidade de um produto ser compilado ou executado em diferentes arquiteturas (seja de *hardware* ou de *software*).
- entre outros ...

2.2 Quais as vantagens da utilização destas análises?

- Pode encontrar pontos fracos no código analisado e retornar o seu local exato.
- Permite encontrar falhas numa fase inicial de desenvolvimento (*life cycle*), reduzindo o custo de resolver as mesmas.
- O código pode ser mais facilmente percebido por outros ou em desenvolvimentos futuros.
- Possibilidade de existirem menos defeitos em testes mais tarde realizados.
- Permite encontrar certos problemas que dificilmente (ou nunca) são encontrados utilizando testes dinâmicos, tais como:
  - *Unreachable code* - partes de código que nunca é executado.
  - Gestão de variáveis - não declaradas, não usadas, etc.
  - Funções não chamadas (não utilizadas).
  - entre outros ...

2.3 *Static analysis* no Visual Studio

Tendo em conta que o *Visual Studio* é o `IDE` que a empresa *F3M* utiliza nos seus projetos, empresa da qual recebemos algumas das regras de codificação a implementar neste projeto, consideramos relevante fazer um levantamento de estatísticas que o mesmo oferece acerca do código produzido.

O *Visual Studio* tem uma ferramenta de *Static analysis* tratada como *Code Analysis*, que avalia as seguintes métricas de código:

- *Maintainability Index*
  Calcula um valor de 0 a 100 que representa o nível de facilidade de manutenção do código.
- *Cyclomatic Complexity*
  Avalia a complexidade estrutural do código, calculando o número de diferentes caminhos na lógica do código.
- *Depth of Inheritance*
  Indica o número de definições de classes que se estendem desde a raiz da hierarquia de classes.
- *Class Coupling*
  Medida do aninhamento de classes únicas por parâmetros, variáveis locais, chamadas de funções, etc.
- *Lines of Code*
  Valor aproximado do número de linhas do código.

O *Visual Studio* ainda analisa outras métricas de código não relacionadas com a ferramenta *Code Analysis*, da mesma forma que a maior parte dos IDE's o fazem, como por exemplo variáveis/métodos não utilizados e sintaxe incorreta.

3 Regras de codificação definidas

No processo de levantamento das regras a implementar neste projeto, foram tidas em conta duas diferentes origens para as mesmas: algumas foram selecionadas das boas práticas que a empresa **F3M** implementa na sua organização no desenvolvimento de *software*, enquanto outras têm origem do **nosso conhecimento** e aprendizagem obtidas também do estudo de ***refactoring*** **e** ***bad smells***.

3.1 F3M

3.1.1 Regras de criação de tabelas

Regras associadas a ficheiros `C#` relacionados com a parte de bases de dados.

- Todas as tabelas têm que começar pelo prefixo `tb`.
  Por exemplo, uma tabela na base de dados que represente os Clientes de uma aplicação deve ser nomeada como `tbClientes`.

- Todas as tabelas têm que conter os campos `ID`, `Sistema`, `Ativo`, `DataCriacao`, `UtilizadorCriacao`, `DataAlteracao`, `UtilizadorAlteracao` e `F3MMarcador`.
  É prática da empresa que na definição das tabelas através de `C#`, cada uma deva conter todos os campos mencionados em cima.

- Todas as chaves estrangeiras têm que começar por `ID`.
  Isto é, uma chave estrangeira (marcada com `[ForeignKey(...)]`), deve conter o prefixo `ID`. Por exemplo, uma chave como referência a uma tabela de Clientes poderá ser representada como `IDClientes`.

- O nome das propriedades deve ser o mais curto possível.
  Ou seja, o objetivo passará por definir um número máximo de carateres que permita verificar que o código siga esta prática estabelecida.

3.1.2 Regras na criação de funcionalidades

Princípios relativos à criação de funcionalidades no projeto de forma a estarem de acordo com a parte de dados do mesmo.

- Se o nome da tabela da base de dados for `tb<Entidade>`:
  - o nome para a classe modelo deve ser `<Entidade>`;
  - o nome para o repositório deve ser `Repositorio<Entidade>`;
  - o nome para o controlador deve ser `<Entidade>Controller`;

  Por exemplo, se o nome da tabela for `tbClientes`, então o nome para a classe modelo deve ser `Clientes`, para o repositório `RepositorioClientes` e para o controlador `ClientesController`. No nosso entendimento, os testes a desenvolver passarão por receber o caminho (*PATH*) para as pastas de Bases de Dados, Modelos, Repositórios e Controladores, e, a partir daí, verificar que o nome dos ficheiros utilizados em cada um destes segue o princípio estabelecido.

3.1.3 Regras do lado servidor

Estes princípios são tidos em conta na implementação do lado do servidor das aplicações desenvolvidas.

- Os *inputs* das funções devem começar sempre por `in`.
  No nosso entender, esta norma passa por colocar o prefixo `in` em todos os parâmetros das funções criadas, por exemplo: "`int soma (int inA, int inB) {...}`".

- Tipificar sempre as variáveis utilizadas:

  - Inteiros, utilizar prefixo `int` → ex^{o}: `public int intNumClientes;`
  - String, utilizar prefixo `str` → ex^{o}: `public string strNome;`
  - Boolean, utilizar prefixo `bln` → ex^{o}: `public bool blnDisponivel;`
  - Datas, utilizar prefixo `dt` → ex^{o}: `public DateTime dtFinal;`
  - Long, utilizar prefixo `lng` → ex^{o}: `public long lngNIF;`
  - List (of ..), utilizar prefixo `lst` → ex^{o}: `public List<int> lstNumeros;`
  - Dicionarios, utilizar prefixo `dic` → ex^{o}: `public Dictionary<int,Cliente> dicClientes;`
  - Array's, utilizar prefixo `arr` → ex^{o}: `public int[] arrInteiros;`

  A ideia passa por que toda a gente entenda o tipo de dados utilizados e para o que os mesmos servem.

- Documentar todos os métodos, definindo nós a seguinte estrutura para documentação:

```
/// <summary>
///     Descrição da funcionalidade do método.
/// </summary>
/// <param name="Param1">Descrição do parâmetro 1</param>
/// <param name="Param2">Descrição do parâmetro 2</param>
/// <returns>
///     Descrição do valor a "devolver".
/// </returns>
/// <example>
///     <code>
///         Exemplo de utilização do método.
///     </code>
/// </example>
```

3.2 Conhecimento e estudo de *refactoring* e *bad smells*

Através dos conhecimentos que já obtemos ao longo do nosso percurso e, tendo em conta a aprendizagem realizada por nós recentemente em *refactoring* de *software* e sobre *bad smells*, definimos as seguintes regras tendo em conta boas práticas que consideramos relevantes para o desenvolvimento de código melhor e mais percetível.

- Escrever apenas uma classe por cada ficheiro `C#`.
  De forma a contribuir para uma maior modularidade do projeto e melhor percepção do mesmo.

- Limite do número de métodos por classe.
  Pode ser interessante no contexto de uma equipa de desenvolvimento de *software* limitar o número de métodos que se podem escrever numa classe. Relacionado ainda com os *bad smells*, muitos métodos pode ainda indicar a possibilidade de a classe poder ser dividida em várias.

- Limite do número de linhas de um método.
  Quanto maior um método é, mais difícil é perceber o mesmo e mantê-lo. Para além disto, métodos longos oferecem o esconderijo perfeito para códigos duplicados indesejados.

- Limite do número de parâmetros de uma função.
  Com o objetivo de tornar o código mais legível e compacto, podendo ainda ajudar a revelar código duplicado que ainda não tinha sido identificado.

- Limite de comentários por método analisando a percentagem de comentários dentro deste.
  A presença de muitos comentário dentro de um método pode querer indicar que o mesmo não é tão fácil de perceber como seria de desejar. Desta forma, é de evitar que um método contenha demasiados comentários, podendo este ser dividido em vários métodos. "*The best comment is a good name for a method or class*".

4 Plano de desenvolvimento do Projeto

Com o objetivo de realizar um planeamento cuidado para o desenvolvimento do projeto em causa, recorremos ao *Microsoft Project* para distribuir as várias etapas do mesmo tendo em conta as fases do desenvolvimento e as datas a cumprir, tendo em conta a existência de uma entrega intermédia (13 de Abril de 2019) e de um entrega final (8 de Junho de 2019), ambas marcadas como *Milestones* no planeamento.

Importa fazer referência que no plano apresentado optamos por incluir a fase de `Estudo do domínio` relativa ao presente documento, a às fases de `Testes` e `Documentação` reservadas para a posterioridade da implementação dos *scripts*.

WBS	Nome da Tarefa	Duration	Start	Finish	Predecessors
1	**Estudo do domínio**	**13 days**	**Wed 27/03/19**	**Fri 12/04/19**	
1.1	Análise de static analysis	9 days	Wed 27/03/19	Mon 08/04/19	
1.2	Análise das boas práticas da F3M	2 days	Tue 09/04/19	Wed 10/04/19	2
1.3	Selecção de regras a implementar com base nas boas práticas da F3M	1 day	Thu 11/04/19	Thu 11/04/19	3
1.4	Selecção de regras a implementar com base no nosso conhecimento e estudo em refactoring e bad smells	1 day	Fri 12/04/19	Fri 12/04/19	4
2	**Design e especificação dos scripts de teste de normas de codificação**	**6 days**	**Mon 15/04/19**	**Mon 22/04/19**	**1**
2.1	Análise de requisitos	2 days	Mon 15/04/19	Tue 16/04/19	
2.2	Escolha das ferramentas a utilizar	2 days	Wed 17/04/19	Thu 18/04/19	7
2.3	Definição da arquitetura a utilizar	2 days	Fri 19/04/19	Mon 22/04/19	8
3	**Implementação dos scripts de análise de código C#**	**24 days**	**Tue 23/04/19**	**Fri 24/05/19**	**6**
3.1	**Regras com origem na F3M**	**18 days**	**Tue 23/04/19**	**Thu 16/05/19**	
3.1.1	Regras de criação de tabelas	6 days	Tue 23/04/19	Tue 30/04/19	
3.1.2	Regras de criação de funcionalidades	6 days	Wed 01/05/19	Wed 08/05/19	12
3.1.3	Regras do lado do servidor	6 days	Thu 09/05/19	Thu 16/05/19	13
3.2	**Regras propostas por nós**	**6 days**	**Fri 17/05/19**	**Fri 24/05/19**	**14**
3.2.1	Regras de "refactoring" e "bad smells"	6 days	Fri 17/05/19	Fri 24/05/19	14
4	**Testes**	**5 days**	**Mon 27/05/19**	**Fri 31/05/19**	**10**
4.1	Criação de casos de teste	4 days	Mon 27/05/19	Thu 30/05/19	
4.2	Execução dos testes	1 day	Fri 31/05/19	Fri 31/05/19	18
5	**Documentação**	**5 days**	**Mon 03/06/19**	**Fri 07/06/19**	**17**
5.1	Escrita do relatório	5 days	Mon 03/06/19	Fri 07/06/19	
6	**Milestones**	**40 days**	**Sat 13/04/19**	**Sat 08/06/19**	
6.1	Entrega intermédia	0 days	Sat 13/04/19	Sat 13/04/19	1
6.2	Entrega final	0 days	Sat 08/06/19	Sat 08/06/19	20

Figura 1: Plano de desenvolvimento do projeto definido.

Resultante deste planeamento surge o respetivo Diagrama de *Gantt*.

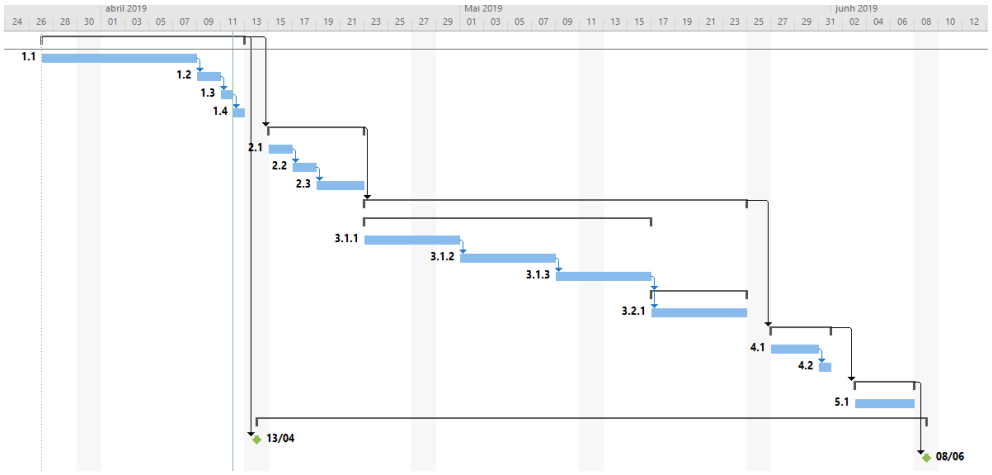

Figura 2: Diagrama de *Gantt* do projeto.

5 Conclusão

Com o estudo realizado a este altura do projeto, e tendo em conta a selecção de regras de codificação que consideramos relevantes, o trabalho futuro passa maioritariamente pela implementação dos *script's* que testem as mesmas em código desenvolvido (`C#`).

Como próximo passo, o mesmo será a análise de requisitos estabelecidos (normas de codificação) e perceber quais as melhores ferramentas e arquitetura a utilizar para conseguir o efeito desejado.

Depois da consequente implementação dos *scripts*, é ainda desejado realizar testes aos mesmos que verifiquem a sua qualidade, deixando ainda tempo para redigir/ultimar o relatório final do projeto.

6 Bibliografia

- "**What Is Static Code Analysis?**", acedido em 05/04/2019
  https://www.perforce.com/blog/qac/what-static-code-analysis
- "**Static Analysis vs Dynamic Analysis in Software Testing**", acedido em 05/04/2019
  https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/
- "**Recommended Tags for Documentation Comments (C# Programming Guide)**", acedido em 12/04/2019
  https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/recommended-tags-for-documentation-comments
- "**Code metrics values - Visual Studio**", acedido em 12/04/2019
  https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019
- "**Refactoring**", acedido em 12/04/2019
  https://sourcemaking.com/refactoring