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

8 de Junho 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 utilização 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}: `int intNumClientes;`
  - String, utilizar prefixo `str` → ex^{o}: `string strNome;`
  - Boolean, utilizar prefixo `bln` → ex^{o}: `bool blnDisponivel;`
  - Datas, utilizar prefixo `dt` → ex^{o}: `DateTime dtFinal;`
  - Long, utilizar prefixo `lng` → ex^{o}: `long lngNIF;`
  - List (of ..), utilizar prefixo `lst` → ex^{o}: `List<int> lstNumeros;`
  - Dicionários, utilizar prefixo `dic` → ex^{o}: `Dictionary<int,Cliente> dicClientes;`
  - Array's, utilizar prefixo `arr` → ex^{o}: `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 obtivemos 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 a uma fase inicial do projeto, a às fases de `Testes` e `Documentação` reservadas para a posterioridade da implementação dos *scripts* e da *GUI* (*Graphical User Interface*).

WBS	Nome da Tarefa	Duração	Início	Conclusão
1	**Estudo do domínio**	**13 dias**	**Qui 21/03/19**	**Seg 08/04/19**
1.1	Análise de static analysis	9 dias	Qui 21/03/19	Ter 02/04/19
1.2	Análise das boas práticas da F3M	2 dias	Qua 03/04/19	Qui 04/04/19
1.3	Selecção de regras a implementar com base nas boas práticas da F3M	1 dia	Sex 05/04/19	Sex 05/04/19
1.4	Selecção de regras a implementar com base no nosso conhecimento e estudo em refactoring e bad smells	1 dia	Seg 08/04/19	Seg 08/04/19
2	**Design e especificação dos scripts de teste de normas de codificação**	**6 dias**	**Ter 09/04/19**	**Ter 16/04/19**
2.1	Análise de requisitos	2 dias	Ter 09/04/19	Qua 10/04/19
2.2	Escolha das ferramentas a utilizar	2 dias	Qui 11/04/19	Sex 12/04/19
2.3	Definição da arquitetura a utilizar	2 dias	Seg 15/04/19	Ter 16/04/19
3	**Implementação dos scripts de análise de código C#**	**24 dias**	**Qua 17/04/19**	**Seg 20/05/19**
3.1	**Regras com origem na F3M**	**18 dias**	**Qua 17/04/19**	**Sex 10/05/19**
3.1.1	Regras de criação de tabelas	6 dias	Qua 17/04/19	Qua 24/04/19
3.1.2	Regras de criação de funcionalidades	6 dias	Qui 25/04/19	Qui 02/05/19
3.1.3	Regras do lado do servidor	6 dias	Sex 03/05/19	Sex 10/05/19
3.2	**Regras propostas por nós**	**6 dias**	**Seg 13/05/19**	**Seg 20/05/19**
3.2.1	Regras de "refactoring" e "bad smells"	6 dias	Seg 13/05/19	Seg 20/05/19
4	**Interface Gráfica**	**4 dias**	**Ter 21/05/19**	**Sex 24/05/19**
4.1	Implementação de interface gráfica	4 dias	Ter 21/05/19	Sex 24/05/19
5	**Testes**	**5 dias**	**Seg 27/05/19**	**Sex 31/05/19**
5.1	Criação de casos de teste	4 dias	Seg 27/05/19	Qui 30/05/19
5.2	Execução dos testes	1 dia	Sex 31/05/19	Sex 31/05/19
6	**Documentação**	**5 dias**	**Seg 03/06/19**	**Sex 07/06/19**
6.1	Escrita do relatório	5 dias	Seg 03/06/19	Sex 07/06/19
7	**Milestones**	**40 dias**	**Sáb 13/04/19**	**Sáb 08/06/19**
7.1	Entrega intermédia	0 dias	Sáb 13/04/19	Sáb 13/04/19
7.2	Entrega final	0 dias	Sáb 08/06/19	Sáb 08/06/19

Figura 1: Plano de desenvolvimento do projeto definido.

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

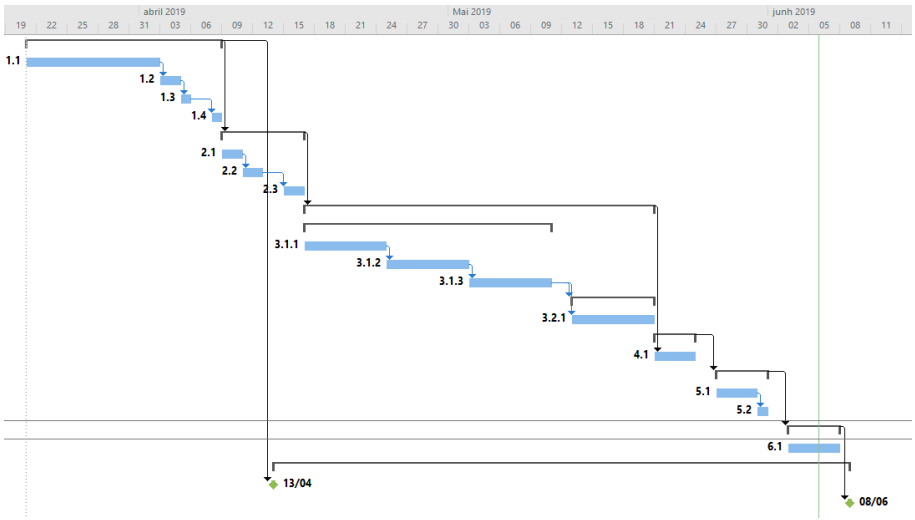


Figura 2: Diagrama de *Gantt* do projeto.

5 Ferramentas utilizadas na solução

Para a implementação dos *scripts* foi escolhida a utilização da linguagem de programação *Python* por ser uma linguagem de *scripting*, que permite um uso facilitado de expressões regulares, algo que consideramos fundamental para este trabalho, assim como a juntar ao facto do grupo de trabalho já ter experiência de utilização da mesma, sendo que esta contém um vasto suporte *online*.

Para a leitura dos ficheiros `C#` foi utilizada a biblioteca `fileinput`, sendo utilizada a biblioteca `re` para o uso de *regex's*. Quanto à interface para o utilizador implementada após os *scripts*, foi utilizada a biblioteca `Tkinter`.

6 Implementação e Testes

Nesta seccção pretendemos mostrar algumas decisões tomadas de forma a conseguir testar as regras por nós apresentadas anteriormente para cada um dos quatro *scripts*.

6.1 *Script* 1 - Regras de criação de tabelas

6.1.1 Implementação das regras

- Todas as tabelas têm que começar pelo prefixo `tb`

  Espera-se que as tabelas sejam definidas por uma anotação do tipo `[Table("tbArtigos")]`, sendo neste caso o nome da tabela "`tbArtigos`". Sendo assim, de forma a testar esta regra é preciso encontrar o nome da tabela, sendo utilizado o seguinte *regex*:

  ```
  table_name = re.match(r'\s*\[Table\(\"(\w*)\"\)\]\s*', line)
  ```

- Todas as tabelas têm que conter os campos `ID`, `Sistema`, `Ativo`, `DataCriacao`, `UtilizadorCriacao`, `DataAlteracao`, `UtilizadorAlteracao` e `F3MMarcador`

  Espera-se que as campos sejam definidos por uma anotação do tipo `[Column("Sistema")]`, sendo neste caso o nome do campo "`Sistema`". Sendo assim, de forma a testar esta regra é preciso encontrar o nome dos campos, sendo utilizado o seguinte *regex*:

  ```
  field_name = re.match(r'\s*\[Column\(\"(\w*)\"\)\]\s*', line)
  ```

  Para além disto, é utilizado um dicionário com todos os campos necessários, desta forma caso um campo seja encontrado o seu valor é marcado a 1, e caso não seja encontrado o seu valor mantêm-se em 0. Para além disto, é uma forma simples de introduzir modificações no futuro sobre os campos requiridos.

  ```
  required_fields = {
      'ID': 0,
      'Sistema': 0,
      'Ativo': 0,
      'DataCriacao': 0,
  ```

```
    'UtilizadorCriacao': 0,
    'DataAlteracao': 0,
    'UtilizadorAlteracao': 0,
    'F3MMarcador': 0
}
```

- Todas as chaves estrangeiras têm que começar por `ID`

  Espera-se que as chaves estrangeiras sejam definidas por uma anotação do tipo `[ForeignKey("Idtaxa")]`, sendo neste caso o nome da chave estrangeira "`Idtaxa`". Sendo assim, de forma a testar esta regra, é preciso encontrar o nome das chaves estrangeira, sendo utilizando o seguinte *regex*:

  ```
  foreign_key = re.match(r'\s*\[ForeignKey\(\"(\w*)\"\)\]\s*', line)
  ```

- O nome das propriedades deve ser o mais curto possível

  Foi definido por nós a título de exemplo que o tamanho máximo para as propriedades deveria ser de 20 carateres.

  Tendo em conta os exemplos de código `C#` recebido, uma possível declaração de um propriedade pode ser:

  ```
  public virtual ICollection<TbProcessamentoUtentes> TbProcessamentoUtentes { get; set; }
  ```

  Sendo assim, para conseguir obter o nome de todas as propriedades e tentar cobrir o máximo de possibilidades de declarações (como os níveis de acessibilidade ou o uso opcional de `virtual` ou `override`) é utilizado o seguinte *regex*:

  ```
  property = re.match(r'\s*(private|public|protected|internal|protected internal|
             private protected)(\s+(virtual|override))?\s+[A-Za-z\<\>?\[\]]+\s+
             (\w+)\s*\{\s*get\s*;\s*set\s*;\s*\}\s*', line)
  ```

6.1.2 Testes e Resultados obtidos

Através da execução do *script* referindo qual a pasta que contem os ficheiros que se pretendem analisar obtemos um relatório com informação sobre cada uma das regras testadas, sendo o processo o mesmo para todos os restantes *scripts*. De referir ainda a utilização de cores no relatório devolvido, de forma a ser mais amigável para o utilizador e mais fácil de distinguir a informação relevante.

No exemplo de execução apresentado em baixo que testa a criação de tabelas com o ficheiro `TbArtigos.cs`, podemos através do *output* recebido perceber que o mesmo respeita a primeira regra (nome da tabela), tem em falta o campo `ID` para cumprir com a segunda rega, cumpre a terceira regra relativa às *foreign keys* declaradas e, quanto à ultima regra, duas propriedades não cumprem com o tamanho máximo estabelecido.

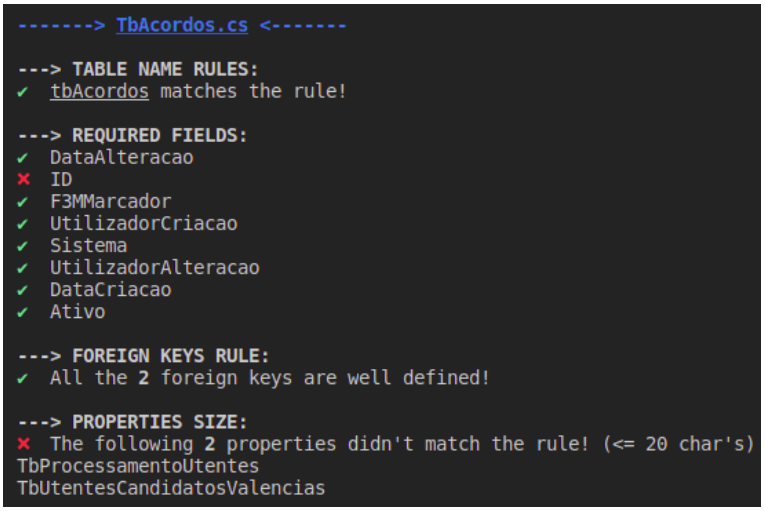


Figura 3: Resultados do *script* 1 relativo ao ficheiro `TbAcordos.cs`.

6.2 *Script* 2 - Regras na criação de funcionalidades

6.2.1 Implementação das regras

- Se o nome da tabela da base de dados for `tb<Entidade>`, o nome dos modelo deve ser `<Entidade>`, o nome do repositório `Repositorio<Entidade>` e o nome do controlador `<Entidade>Controller`

  Para executar este *script*, o mesmo necessita de receber como parâmetros:

  - Ficheiro `C#` com a definição do contexto das tabelas
  - *path* para pasta que contem os `Modelos`
  - *path* para a pasta que contem os `Repositórios`
  - *path* para a pasta que contem os `Controladores`

  Um primeiro passo para a realização deste *script* passa pela obtenção das entidades definidas no ficheiro de contexto das tabelas. Nesse mesmo ficheiro, um exemplo de uma declaração (neste caso da entidade `Acordos`) é:

```
public virtual DbSet<TbAcordos> TbAcordos { get; set; }
```

  Sendo assim, de forma a encontrar estas entidades é utilizada a seguinte expressão regular:

```
table = re.match(r'\s*public\s+virtual\s+DbSet\<[A-Za-z0-9\_]+\>\s+
        ([A-Za-z0-9\_]+)\s*\{\s*get\s*\;\s*set\s*;\s*\}', line)
```

  Depois disto, e através da leitura do nome dos ficheiros em cada um dos *path's* recebidos, são feitos os testes necessários por forma a perceber quais os que já se encontram declarados e os que se encontram em falta.

6.2.2 Testes e Resultados obtidos

É importante de referir que este *script* tem duas formas de execução: o modo normal onde os resultados apresentam apenas o número de documentos em falta para cada tipo pretendido, e um segundo modo mais completo que indica quais os que estão em falta.

Na execução mostrada de seguida de acordo com o modo mais completo, percebemos que são apresentadas as tabelas definidas, e quais os modelos, repositórios e controladores já definidos (comum aos dois modos), sendo nos resultados apresentados não só o número de ficheiros em falta como o nome dos mesmos.

```
---> TABLES:
TbAcordos
TbAreas
TbArtigos
TbUtentes
TbUtentesValencias
TbValencias
TbArtigosAnexos

---> MODELS:
Artigos.cs
Areas.cs
TbUtentes.cs
Valencias.cs
UtentesValencias.cs
Acordos.cs

---> REPOSITORIES:
RepositorioValencias.cs
RepositorioUtentes.cs
UtentesValenciasRepository.cs

---> CONTROLLERS:
ValenciasController.cs
UtentesValenciasController.cs
UtentesController.cs

--> RESULTS:
×  Missing 2 from 7 models according to tables!
Utentes.cs
ArtigosAnexos.cs
×  Missing 5 from 7 repositorys according to tables!
RepositorioAcordos.cs
RepositorioAreas.cs
RepositorioArtigos.cs
RepositorioUtentesValencias.cs
RepositorioArtigosAnexos.cs
×  Missing 4 from 7 controllers according to tables!
AcordosController.cs
AreasController.cs
ArtigosController.cs
ArtigosAnexosController.cs
```

Figura 4: Resultados do *script* 2 em modo completo.

6.3 *Script* 3 - Regras do lado servidor

6.3.1 Implementação das regras

- Os *inputs* das funções devem começar sempre por `in`

  Para testar esta regra é preciso encontrar os métodos das classes recebidas, sendo que um exemplo de um método pode ser:

  ```
  public async Task<List<object>> Execute(long id, long text)
  ```

  Sendo assim, é utilizado o seguinte *regex* para capturar os mesmos (não sendo necessário em nossa opinião capturar o método construtor da classe):

  ```
  function_in_line = re.match(r'\s*(public|private)\s+(async\s+)?(void|[A-Za-z]
                     [A-Za-z0-9\<\>\_?\[\]]*)\s+[A-Z][A-Za-z0-9\_]*\s*\(.*\)', line)
  ```

A partir deste ponto, para cada função é feito o *parsing* dos seus parâmetros (caso os mesmos existam) e verificar se estes cumprem ou não a regra definida.

- Tipificar sempre as variáveis utilizadas

  De acordo com as tipificações que as declarações de variáveis devem seguir já apresentadas anteriormente, é primeiro que tudo conseguir "capturar"todas as declarações de variáveis, sendo que exemplos de declarações podem ser:

```
bool blnDisponivel = false;
DateTime dtFinal;
```

  Para tal é utilizado o seguinte *regex*:

```
variable_test = re.findall(r'([A-Za-z0-9\[\]\_\<\>,?]+)\s+([A-Za-z0-9\_.]+)\s*
                (?:=[^;]+\s*)?;', line)
```

  A partir do momento em que se consegue obter o tipo (por exemplo `DateTime`) e o nome da variável (por exemplo `dtFinal`) é testado se a declaração cumpre com a regra definida. Para tal, é definido no *script* um dicionário com as correspondências de `tipo` e `prefixo` segundo expressões regulares de forma a fazer os testes pretendidos.

```
variables = {
    r'^int$': r'^int',
    r'^string$': r'^str',
    r'^bool$': r'^bln',
    r'^DateTime$': r'^dt',
    r'^long$': r'^lng',
    r'^List<[^\>]+>$': r'^lst',
    r'^Dictionary<[^\,]+,[^\>]+>$': r'^dic',
    r'^[A-Za-z0-9\_]+\[\]$': r'^arr'
}
```

  Tal como referido num exemplo anterior, este dicionário permite modificações futuras quanto às regras que as variáveis declaradas devem cumprir.

- Documentar todos os métodos

  De forma a testar a documentação proposta já apresentada anteriormente, é verificado se a função contem uma declaração de documentação anterior à sua declaração. Se a mesma tiver, é verificado se a estrutura da mesma está de acordo com a definida por nós. Para tal é aplicado o seguinte *regex* à documentação recolhida (tendo em conta que a parte referente aos parâmetros tanto pode não aparecer no caso da função não ter parâmetros como aparecer mais do que uma vez):

```
correct_documentation = re.match(r'<summary>[^<]+</summary>
                        ((?:<param name="[^\"]+">[^<]+</param>)*)
                        <returns>[^<]+</returns><example>
                        <code>[^<]+</code></example>', documentation)
```

Se a estrutura estiver correta, é obtido o nome dos parâmetros declarados na mesma de forma a verificar se o nome destes está de acordo com os definidos na declaração da função:

```
d_params = re.findall(r'<param name="([^\"]+)">[^<]+</param>', params)
```

6.3.2 Testes e Resultados obtidos

No exemplo de execução apresentado de seguida, conseguimos perceber quais os métodos que não cumprem quer a primeira regra relativa ao nome do parâmetro declarados, assim como a informação relativa à documentação (segunda regra). Para além disto são ainda apresentadas as variáveis recolhidas e se as mesmas cumprem ou não a tipificação definida.

```
-------> UtentesController.cs <-------

---> FUNCTIONS INPUT'S:

-> Function public async Task<List<object>> Get(long id, Task<List<object>> task):
✘  Missing 'in' in the input parameter id
✘  Missing 'in' in the input parameter task
-> Function public void Put(int id, [FromBody] string value):
✘  Missing 'in' in the input parameter id
✘  Missing 'in' in the input parameter value

---> FUNCTIONS DOCUMENTATION:

✘  public async Task<List<object>> GetByQuarto(int inid): no documentation
✘  public async Task<List<object>> Get(): Parameters of function are different from the documentation
        Function params = []
        Documentation params = ['Param1']
✘  public void Post([FromBody] string invalue): no documentation
✘  public void Delete(int inid): no documentation
✘  public async Task<List<object>> Get(long id, Task<List<object>> task): Parameters of function are different from the documentation
        Function params = ['id', 'task']
        Documentation params = ['Param1', 'Param2']
✘  public void Put(int id, [FromBody] string value): no documentation

---> VARIABLES NAME RULE:

-> Function public void Delete(int inid):
✔  Dictionary<int,Cliente> dicclientes
✔  int intname
✔  int[] arrnames
✘  int nameNoint
✘  bool verdadeiro
-> Function public void Put(int id, [FromBody] string value):
✔  string strNome
✔  string str2
✔  bool blnDisponivel
✔  DateTime dtFinal
✔  long lngNIF
✔  List<int> lstNumeros
✔  int intclientes
✘  string ya
✘  bool true
✔  Dictionary<int,Cliente> dict
✔  int[] array
```

Figura 5: Resultados do *script* 3 relativo ao ficheiro `UtentesController.cs`.

6.4 *Script* 4 - Regras relativas a *refactoring* e *bad smells*

6.4.1 Implementação das regras

- Escrever apenas uma classe por cada ficheiro `C#`

  Para conseguir testar esta regra, é necessário encontrar as classes do ficheiro a ser analisado e, sabendo que classes em `C#` podem ser definidas da seguinte forma:

  ```
  public class UtentesController : ControllerBase { ... }
  public class TestController { ... }
  ```

  É então utilizada a seguinte expressão regular:

```
class_in_line = re.match(r'\s*(?:public|private)\s+class+\s+([^:{\n]+)', line)
```

- Limite do número de métodos por classe

  Para concretizar esta regra, torna-se necessário encontrar os métodos do ficheiro a ser analisado e atribuir o mesmo à respetiva classe a que está a ser feito o *parsing*. Para tal, são utilizados *regex's* como os apresentados anteriormente para a detecção de métodos.

- Limite do número de linhas de um método

  Foi definido por nós a título de exemplo que o número máximo de linhas para um método deveria ser de 10 linhas.

  Sendo assim, a partir do momento em que um novo método é reconhecido, foi preciso ter especial atenção para perceber quando o método chega ao seu fim. Para tal, é contado o número de carateres do tipo { e } de forma a que quando o número de ocorrências de um e de outro é o mesmo podemos concluir que o método chegou ao fim, tendo ainda em conta que estes carateres dentro de *strings* não devem ser tidos em conta.

- Limite do número de parâmetros de uma função

  Foi definido por nós a título de exemplo um limite de 4 parâmetros por método. Os parâmetros são assim obtidos e testados quando se deteta uma função com os *regex's* anteriormente referidos.

- Limite de comentários por método analisando a percentagem de comentários dentro deste

  Foi definido por nós a título de exemplo um limite de 20% de comentários por método. Para testar esta regra foram contabilizados o número de caracteres no interior de uma função, assim como os carateres respetivos a comentários, de forma a poder efetuar uma divisão "*carateres_de_comentarios / carateres_totais*" e obter a percentagem pretendida.

  Entre as várias expressões regulares utilizadas, algumas relativas à detecção de comentários são de seguida apresentadas.

```
regexA =  re.search('\/\*.*\*\/',line)     # /* ... */
regexB = re.search('\/\/.*', line)         # // ...
regexC = re.search('\/\*.*',line)          # /* ...
regexEndComment = re.search('.*\*\/',line) # ... */
```

6.4.2 Testes e Resultados obtidos

No exemplo de execução apresentado de seguida, conseguimos perceber os vários testes efetuados ao ficheiro em causa, sendo que o mesmo só tem uma classe definida no mesmo (primeira regra), cumpre quer com o número de métodos por classe (segunda regra) como com o número de linhas e parâmetros por método (terceira e quarta regra), tendo apenas um método que não cumpre com a percentagem de comentários desejada (quinta regra).

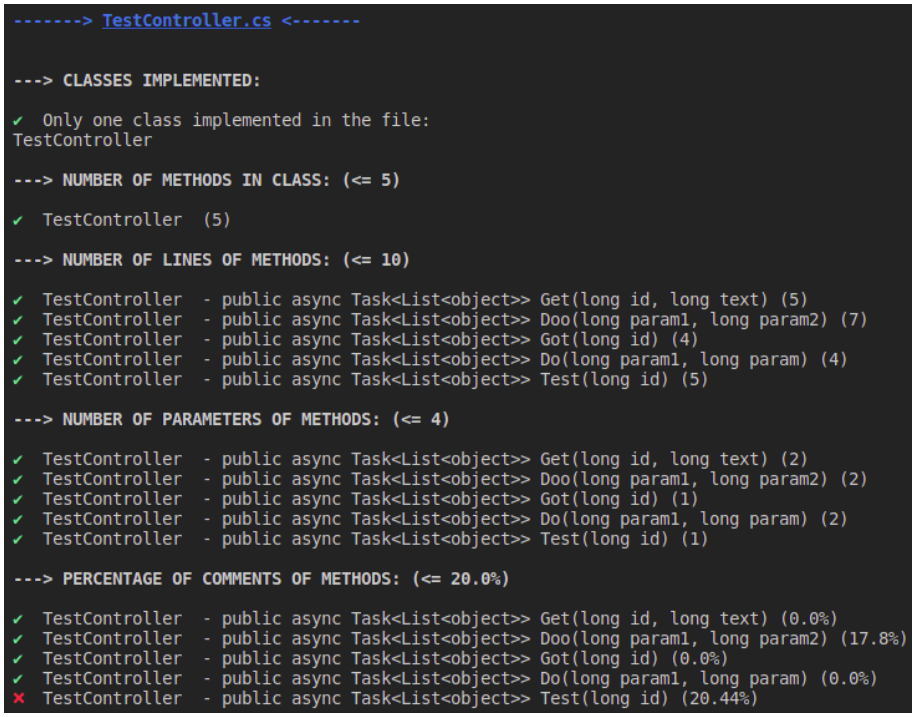


Figura 6: Resultados do *script* 4 relativo ao ficheiro `TestController.cs`.

7 Interface gráfica

Por forma a facilitar o uso dos *scripts* pelo utilizador final dos mesmos, foi decidido desenvolver uma interface que permita a execução dos mesmos e o visionamento do relatório de cada execução na própria interface.

Para tal, e como referido anteriormente no relatório, foi utilizada a biblioteca `Tkinter` em `Python`. O resultado final obtido é apresentado de seguida.

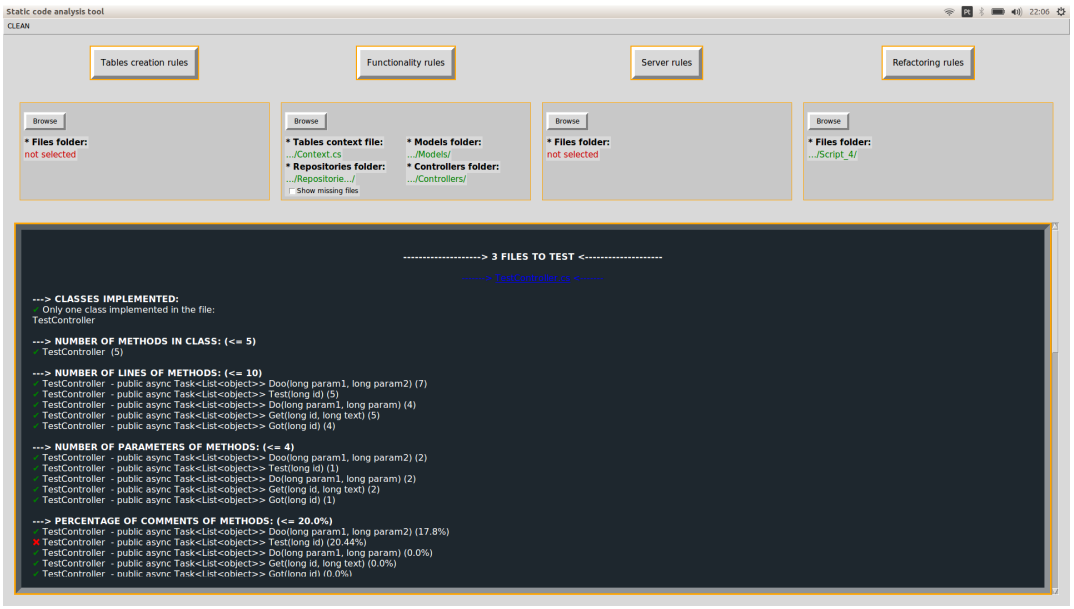
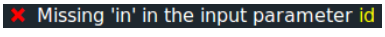


Figura 7: Interface gráfica desenvolvida.

Como se pode ver pela imagem, a ideia passa por existir um botão para cada um dos quatro *scripts* desenvolvidos, sendo que as pastas e ficheiros necessárias à execução são apresentadas em baixo de cada um e podem ser selecionados através de cada um dos botões "`Browse`". Ao executar os *scripts*, os resultados anteriormente apresentados na linha de comandos passam agora a ser também apresentados pela via da interface se a mesma for utilizada. De referir ainda a *checkbox* existente no segundo *script* referente aos dois modos de realização do mesmo, assim como o botão "`CLEAN`"que permite limpar todos os campos e a consola.

Para além da construção da própria interface, foi ainda necessário alterar os *scripts* para passarem a ter duas formas de imprimir os resultados: uma para a linha de comandos e outra para a interface. Por exemplo, para imprimir o resultado:



Ao imprimir para o terminal, o código utilizado é o seguinte:

```
print('\033[91m\033[1m' + u'\u274C' + '\033[0m  ' + "Missing 'in' in the input
      parameter \033[93m" + inp + "\033[0m")
```

Ao passo que para imprimir na interface gráfica o código utilizado é:

```
bad =  u'\u274C'
text.insert(INSERT, bad, ["red","bold"])
text.insert(INSERT, "  Missing 'in' in the input parameter ")
text.insert(INSERT, inp, "yellow")
```

8 Conclusão

Depois de feito um estudo à área de *static analysis* e ao problema em mãos, foram escolhidas algumas regras para teste, não só regras utilizadas pela *F3M*, mas também regras propostas por nós com base no nosso conhecimento de *refactoring* e *bad smells*.

Após este estudo e de efetuada a escolha das ferramentas ideais para desenvolver os *scripts* de teste, foram implementados e testados *scripts* com ficheiros de teste `C#` idealizados por nós e com base nos ficheiros recebidos pela *F3M* de forma a validar o desempenho e resultados dos *scripts*.

De forma a conseguir um uso mais agradável destes *scripts* foi ainda desenvolvida uma interface gráfica para utilização dos mesmos, em forma de um executável.

Com a realização deste projeto, podemos concluir que *Static analysis* tem um papel fundamental na prevenção precária de erros e problemas no código assim como verificar o mesmo conforme as regras estabelecidas (quer seja por uma empresa ou por um grupo de trabalho), visto haver a possibilidade de serem testadas muitas caraterísticas de forma sistemática sempre que há alterações no código.

9 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
- "**Regular expression operations**", acedido em 07/06/2019
  https://docs.python.org/2/library/re.html
- "**The Tkinter Grid Geometry Manager**", acedido em 07/06/2019
  https://effbot.org/tkinterbook/grid.htm
- "**The Tkinter Pack Geometry Manager**", acedido em 07/06/2019
  https://effbot.org/tkinterbook/pack.htm
- "**Accessibility Levels (C# Reference)**", acedido em 07/06/2019
  https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels