Universidade Federal de Goiás
Instituto de Informática
Curso: Engenharia de Software

Disciplina: Técnicas Avançadas em Construção de Software

2º semestre/2016

Políticas de Desenvolvimento Projeto ANTENADO

Professor: Marcelo Ricardo Quinta

Discentes:

Alexandre Lara
Bruno Andrade
Dyego de Oliveira
Pablo Almeida
Weiner Silva

Goiânia, 09 Dezembro de 2016

Sumário

1. Introdução
2. Nomenclatura
3. Comentários
4. Classes métodos e variáveis
5. Commits
6. Testes
7. Integração Contínua
8. Métricas
9. Glossário

1. Introdução

Neste documento são apresentadas as políticas inerentes ao projeto ***ANTENADO***. Estas políticas devem ser seguidas obrigatoriamente, a fim de se manter a qualidade do código do projeto. Também são descritas neste documento as métricas a serem utilizadas para verificar o cumprimento destas políticas.

2. Nomenclatura

1. **Classes:** Os nomes de classes devem ser substantivos com a primeira letra de cada palavra em maiúsculo. Os nomes das classes devem ser simples e descritivos de acordo com as tabelas 11-1 e 11-2 do livro *[1].*

   Ex1 pode: public class FULLMASTERS(){..}

   Ex2 não pode: public class fullmasters){..}

   Ex3 não pode: public class full_masters(){..}

2. **Métodos:** Os nomes dos métodos devem ser verbos e descritivos.

   Ex1 pode: public int rotacaograus(){..}

   Ex2 não pode: public int rotacao_graus(){..}

   Ex3 não pode: public class int RotacaoGraus(){..}

3. **Constantes:** Devem ter todas as letras maiúsculas e em caso de nome composto, devem ser separados por um underline “_”.

   Ex1 pode: INTEGRAR_EQUATION;

   Ex2 não pode: INTEGRAR-EQUATION;

   Ex3 não pode: integrar_equation;

4. **Variáveis:** As variáveis devem ser escritas com a primeira em minúsculo em caso de nome composto a primeira letra da segunda palavra deve ser em maiúsculo (camelCase).

   Ex1 pode: alfaDeclaration;
   Ex2 não pode: AlfaDeclaration;

   Ex3 não pode: alfa_declaration;

3. Comentários

1. Sempre comente o essencial, explique para que serve:
    a. Cada método;
    b. As principais variáveis;
    c. Faça uma breve descrição no início de cada classe e interface.

2. A estrutura básica de um comentário de documentação tem como característica principal, o uso de uma barra e dois asteriscos**(/**)** no início e no seu final, possui um asterisco e uma barra **(*/).**
    a. Para comentários de uma linha utilize a seguinte formatação:

```
/** Exemplo básico de um comentário com apenas uma linha */
```

    b. Para comentário de mais de uma linha utilize a seguinte formatação:

```
/** Exemplo básico de um comentário
*  com mais de uma linha.
*/
```

4. Classes métodos e variáveis

1. O número de linhas de uma classe deve ser inferior ou igual a 500.
2. Classes devem ser bem definidas, visando a uma alta coesão, ou seja, cada classe deve dar suporte a um único e bem definido papel ou responsabilidade.
3. Em todo código fonte deve se ter no máximo 5% de blocos duplicados.

Ex:

```
public void methodTest(int a, int
  b){ int x = a + b;
  int y = a*b;
  return x - y;
 }
public void methoTest2(int b, int
  a){ int x = a + b;
  int y = a*b;
```

```
        return x - y;
    }
```

4. Métodos devem ser destinados a apenas uma função e devem ter no máximo 30 linhas;
5. A **complexidade ciclomática**[1] dos métodos não deve ultrapassar o número 10;
6. Ao utilizar a expressão Try Catch deve se obrigatoriamente efetuar o Log da exceção usando a classe Log do android (android.util.Log).

Ex1 pode: try { … }

```
catch(Exception e) {
        Log.d("", "", e);
};
```

Ex2 não pode: try { … }

```
catch(Exception e) {
        e.printStackTrace()
};
```

7. Utilize a palavra final para dados constantes e referências de objetos constantes. A palavra-chave final impede a mutação não desejada dos dados de instância em classes.

Ex:
```
private static final String URL_CONNECTION = "https://...";
```

8. **Indentação:** Deve-se utilizada a indentação padrão do android studio. Atalho para indentação: CTRL + ALT + L

9. **Declarações de variáveis e constantes:**
   a. Deve ser realizada apenas uma declaração por linha;

Ex1 pode:
```
private String nome;
private int id; private
int idade;
```

Ex2 não pode:
```
private String nome;
    private int id, idade;
```

   b. Declarações devem ser realizadas somente no início de cada bloco;

c. Sempre que for possível, a inicialização de variáveis devem ser realizadas no ato da declaração.

Ex1 pode:
```
private String acesHigh = "Iron Maiden";
```

Ex2 não pode:
```
private String acesHigh;
…
acesHigh = "Iron Maiden";

if(acesHigh == "Iron Maiden" ){};
```

5. Commits

5.1. Somente podem ser efetuados commits sem erros fatais (código compilando).

5.2. Os commits devem seguir o seguinte padrão:

Commit: *<nome da alteração realizada>*

Descrição: *<descrição da alteração realizada >*

5.3. Para tratamento de issues detectados no Sonar, os commits devem seguir o padrão abaixo:

Commit: *Correcting Sonar issue <id do issue no sonar>*;

Descrição do commit: *On: <link do issue no sonar>*.

ou para issues similares (issues com mesma descrição e dentro de um mesmo arquivo):

Commit: *Correcting similar Sonar issues*;

Descrição do commit: *On:*

<link do 1º issue no sonar>

<link do 2º issue no sonar>

<link do 3º issue no sonar>

[...].

6. Testes

6.1. O código deve ter cobertura de testes superior ou igual a 60%.

6.2. Os testes unitários devem ter 100% de sucesso;

6.3. Classes de testes devem ser construídas somente no pacote destinado a esse fim**(Pacote 'androidTests");**

7. Integração Contínua

O processo de integração contínua deve seguir os seguintes passos:

1. Commits devem ser realizados utilizando a ferramenta github. Endereço do projeto no git: https://github.com/BrunoVieiraAndrade/RiskLocations/
2. Pelo menos uma vez ao dia o build do projeto deve ser realizado na ferramenta Jenkins[2]. Endereço Jenkins do projeto: http://162.243.102.98/.
3. Após a realização do build as estatísticas do projeto devem ser verificadas na ferramenta SonarQube[3]. Endereço SonarQube[3]: http://45.55.51.163/.
4. Caso sejam identificados issues no SonarQube[3], estes devem ser corrigidos imediatamente e um novo commit deverá ser realizado.

8. Métricas

8.1. **Linhas de código:** Esta métrica será utilizada para representar a quantidade de linhas de código de uma classe. Como apresentado acima, a quantidade de linhas de uma classe não poderá ser superior a 500. Esta métrica será verificada automaticamente após a execução de cada build através da ferramenta SonarQube[3].

8.2. **Complexidade ciclomática[1]** *[7]*: Esta métrica será utilizada para representar a complexidade de um método, e não deve ser superior a 10. Esta métrica será verificada automaticamente após a execução de cada build através da ferramenta SonarQube[3].

8.3. **Blocos duplicados:** Esta métrica será utilizada para representar a existência de blocos de códigos duplicados no projeto. O projeto não deve possuir nenhum bloco duplicado. Esta métrica será verificada automaticamente após

a execução de cada build através da ferramenta SonarQube[3].

8.4. **Código lixo**: Esta métrica será utilizada para representar variáveis, imports e expressões que não estão sendo utilizados no projeto. Esta métrica será verificada automaticamente após a execução de cada build através da ferramenta SonarQube[3].

8.5. **Cobertura dos testes:** Esta métrica será utilizada para representar a porcentagem de código coberto por testes. Esta métrica será verificada automaticamente após a execução de cada build através da ferramenta SonarQube[3].

8.6. **Testes com sucesso:** Esta métrica será utilizada para representar a porcentagem de testes que obtiveram sucesso. Todos os testes unitários devem ter 100% sucesso. Esta métrica será verificada automaticamente após a execução de cada build através da ferramenta SonarQube[3].

8.7. **NOA (Number of Attributes):** Número de Atributos calcula o número de atributos de uma classe. Obtemos zero quando a classe analisada não possui atributos e estamos atribuindo o máximo 10.

8.8. **MaxNesting (Maximum Nesting Level):** *[4]* Nível Máximo de Estruturas Encadeadas calcula o número de caminhos linearmente independentes no método analisado, conhecido como complexidade ciclomática. Seu valor mínimo é 1 e não existe um limite máximo para o seu resultado, mas estamos usando no máximo 10.

8.9. ***SLOC:*** **(Soma do Número de linhas A métrica):** SLOC calcula a soma do número de linhas efetivas de todos os métodos de uma classe. Ou seja, seu resultado é a soma dos valores do LOC de cada método da classe no máximo 10.

8.10. **NCC (Number of Classes Called):** *[4]* Número de Classes Chamadas calcula o número de classes das quais métodos são chamados ou atributos são acessados por um método. Nesse calculo não contamos classes que tenham relação hierárquica com a classe do método em análise. Essa métrica é uma adaptação da *FDP (Foreign Data Providers),* ***que*** calcula o número de classes das quais atributos são acessados. Seu valor varia entre zero e o número total de classes do sistema. Obtemos zero quando o método não utiliza elementos de outras classes.

8.11. **NRP:** Número de Repasses de Parâmetros calcula o número de parâmetros recebidos pelo método em análise e repassados como argumento em chamadas a outras operações pertencentes a sua classe. Calculamos a média de NRP quando precisamos avaliar a média do número de repasses de parâmetros por método de uma classe.

8.12. **NEC (Number of External Calls):** *[6]* Número de Chamadas Externas calcula o número de métodos e atributos externos acessados por um método. NEC é uma adaptação da métrica *ATFD (Access To Foreing Data),* que calcula o número de atributos de classes não relacionadas que são acessados diretamente ou através de chamadas a métodos de acesso.

8.13. **LCOM4 (Lack of Cohesion of Methods):** *[3]* Calcula a falta de coesão dos métodos analisada e da classe seu máximo e 2.

9. Glossário

[9.1] Complexidade ciclomática: A complexidade ciclomática de uma seção do código fonte é a quantidade de caminhos independentes pelo código. Por exemplo, se o código fonte não contém estruturas de controle senão sequenciais a complexidade é 1, já que há somente um caminho válido através do código. Se o código possui somente uma estrutura de seleção contendo somente uma condição, então há dois caminhos possíveis, aquele quando a condição é avaliada em verdadeiro, e aquele quando a condição é avaliada em falso.

[9.2]Jenkins: Jenkins é um sistema de Integração Contínua (CI), projetado para fazer builds automáticos de um projeto a partir de gatilhos pré-definidos (periódico, a cada push no repositório, ao acessar uma URL, etc).A ideia do Jenkins é garantir que apenas builds de sucesso possam ser publicados em produção.

[9.3] SonarQube: SonarQube é um software open-source que se propõe a ser a central de qualidade do seu código-fonte, lhe possibilitando o controle sobre um grande número de métricas de software, e ainda apontando uma série de possíveis bugs.

[9.4] Integração Contínua: “Integração Contínua é uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva

software coeso mais rapidamente.” Martin Fowler.

10. Referências Bibliográficas

[1] MCCONNELL, Steve **Code complete**. Pearson Education, 2004.

[2] MARTIN, Robert C. et al. Código Limpo: Habilidades Práticas do Agile Software.**Rio de Janeiro-RJ: Alta Books**, 2009

[3] HITZ, Markus. Chidamber and Kemerer's metrics suite: a measurement theory perspective. **Software Engineering, IEEE Transactions on**, v. 22, n. 4*, p. 267-271*, 1996. *Citado na pág. 37*

[4] LANZA, Michele; MARINESCU, Radu. **Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. Springer Science & Business Media, 2007. *Citado na pág. 4, 35, 36.*

[5] LORENZ, Mark; KIDD, Jeff. **Object-oriented software metrics: a practical guide**. Prentice-Hall, Inc., 1994. *Citado na pág. 37.*

[6] MARINESCU, Radu. Measurement and quality in object-oriented design. In:**Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on**. IEEE, 2005. p. 701-704. *Citado na pág. 35.*

[7] MCCABE, Thomas J. A complexity measure. **Software Engineering, IEEE Transactions on**, n. 4, p. 308-320, 1976. *Citado na pág. 36.*