Documento de Qualidade do Código

Técnicas Avançadas em Construção de Software

Marcelo Ricardo Quinta

Bruno Martins
Douglas Soares Pereira
Lucas Fernandes
Lucas Ferreira
Murilo Sotelo

Versão 1.0
01/03/2016

Proposta

Este documento contem os padrões definidos a serem utilizados durante o desenvolvimento de software visando maior qualidade, manutenibilidade e legibilidade.

Cada um é responsável por manter e melhorar o código e também este documento caso exista alguma críticas e/ou sugestão.

Os padrões e as regras serão validados através de Integração Contínua utilizando o Jenkins e o SonarQube para análise de código.

Será aceitável Duplicidade de Código de 1%, Complexidade Ciclomática de até 250, Complexidade de Função de até 3, Complexidade de Classe de até 10 e apenas violações do tipo Minor[1]. Mas sempre com o objetivo de zerar todos os problemas encontrados.

[1] Existem algumas exceções em que será aceitável violações diferentes de Minor quando uma implementação padrão (por exemplo, a implementação padrão de Navigation Drawer, da Google) fugir dos padrões definidos e não for possível alterar.

Bibliografia recomendada:

- Beautiful Code - Leading Programmers Explain How They Think, Andy Oram e Greg Wilson, O'REILLY, 2007;
- Clean Code - A Handbook of Agile Software Craftsmanship, Robert C. Martin, Prentice Hall, 2009;
- The Clean Coder - A Code of Conduct for Professional Programmers, Robert C. Martin, Prentice Hall, 2011;

Nomes

Os nomes utilizados durante a codificação devem ser claros e diretos visando um entendimento mais fácil e rápido resultando em economia de tempo.

Eles também devem ser pronunciáveis para, novamente, auxiliar no entendimento e na comunicação entre a equipe durante uma discussão.

Foi escolhido que, por padrão e para universalização, o código deverá ser escrito em inglês com raras exceções que devem ser discutidas entre a equipe sobre casos de palavras regionais e/ou em casos isolados.

Seguindo isso se tornará mais fácil a utilização da busca sempre que necessária.

Também deve ser evitado abreviações e ambiguidades.

O objetivo do nome é você ler e entender o que ele é/faz sem perder tempo e precisar dar muitas voltas no código para compreender, sempre respeitando três questões:

- Porque existe?
- O que faz?
- Como é usado?

Exemplos de nomes serão aprofundados nas explicações seguintes.

Errado	**Correto**
int nf;	int notaFiscal;
float hpt;	Float hypotenuse;
public boolean TemIdade();	public boolean canDrive();
public class infopessoa;	public class Person;

Funções

Nomes de funções devem iniciar com letra minúscula e não devem utilizar em hipótese alguma caracteres especiais. Em caso de nomes compostos, utiliza-se minúsculo para a primeira letra da primeira palavra enquanto outras palavras do nome composto devem iniciar com letra maiúscula. O nome da função deve fazer referência à classe e sempre ser definido em forma de verbo, pois assim o mesmo passa a ideia de ação.

Errado	Correto
public boolean TemIdade(int idade) { return idade >= 18; }	public boolean canDrive(Integer age) { return age>= 18; }
public void velocidade(Car carro) { carro.speed++; }	public void increaseSpeed(Car car) { car.speed++; }

Funções devem ser pequenas e limitadas em até 40 linhas. Funções grandes normalmente são complexas, com baixa qualidade no código e executam múltiplas ações.

Cada Função deve fazer apenas uma única ação. Caso uma função execute mais de uma ação ela está perdendo o seu sentido e pode causar confusão, sendo assim, é melhor criar uma função para cada ação e continuar mantendo a coesão e a organização.

Errado	Correto
public void velocidade(Boolean aumenta, Car carro) { if (aumenta) carro.velocidade++; else carro.velocidade--; }	public void increaseSpeed(Car car) { car.speed++; } public void decreaseSpeed(Car car) { car.speed--; }

Trate o retorno da função e não retorne null pois poderá causar mais problemas no código ou adicionar “ifs” desnecessários e, se necessário, retorne o conjunto vazio.

Funções devem receber no máximo 4 (quatro) parâmetros. A existência de muitos parâmetros pode criar confusão e transparece desordem, falta de coesão, possibilidade de múltiplas ações e complexidade.

Errado	Correto
public void donoDoCarro(String nome, int idade, boolean sexo, int CPF, long datadenascimento, Car carro) { carro.dono.nome = nome; carro.dono.idade = idade; carro.dono.sexo = sexo; carro.dono.cpf = CPF; carro.dono.datadenascimento= datadenascimento; }	public void updateCarOwner(Car car, Person person) { car.owner = person; }

Parâmetros

Nomes de Parâmetros devem iniciar com letra minúscula e não utilizar caractere especial. Em caso de nomes compostos a primeira letra da primeira palavra deve ser minúscula e a primeira letra das demais palavras devem iniciar maiúsculas.

Como já foi dito em Funções, a quantidade de parâmetros está limitado em até 4 (três). Em métodos construtores, a quantidade de parâmetros está limitado em até 10 (dez).

Errado	Correto
public boolean canDrive(int Age) { return age>= 18; }	public boolean canDrive(int age) { return age>= 18; }

Comentários

Os comentários devem ter alguns focos para a melhor compreensão do código e utilização do mesmo. Comentários poderão descrever de forma sucinta um bloco de código mais complexo, fornecer informações sobre algum valor retornado de um método abstrato, esclarecer a utilização de algum parâmetro e alertar possíveis consequências da utilização de algum bloco de código.

Errado	Correto
//Método que aumenta a velocidade public void increaseSpeed(Car car) { car.speed++; } public void decreaseSpeed(Car car) { car.speed--; }	**// A velocidade possui um valor padrão!** **//Utilize este método somente quando esse** **//valor realmente precisar ser aumentado.** public void increaseSpeed(Car car) { car.speed++; } public void decreaseSpeed(Car car) { car.speed--; }

Formatação

Deve-se ter cuidado em relação ao números de linhas de código, caso o código-fonte possua muitas linhas há grande risco de que o mesmo esteja confuso, duplicação de código, exercendo funções alem das quais deveria, entre outros.Para evitar que esses erros ocorram por excesso de código bem como um arquivo muito grande, sera imposto um limite de linhas por arquivo. Um arquivo pode ter no máximo 300 (trezentas) linhas.

A formatação vertical sera usada para separar conceitos, dar sentido e fluidez ao código. Declare varias sempre próximas de onde as mesmas serão usadas, para manter a continuidade deixa as variáveis juntas como no exemplo a seguir

Errado	Correto
Integer idade; String nome String endereço String telefone public boolean canDrive(int idade) { return age>= 18; }	Integer idade; String nome; String endereço; String telefone; public boolean canDrive(int idade) { return age>= 18; }

Como demonstrado no exemplo espaçamento entre variáveis gera uma quebra de raciocínio dificultando a leitura do código, porem olhando no exemplo correto ao olhar para a classe já podemos perceber o nome da classe quais variáveis a mesma possui bem como as funções.

Seguindo a mesma linha de raciocínio espaçamento entre estruturas de repetição bem como decisão devem ser separadas por um espaço simples

A ideia em geral da formatação vertical e que conceitos íntimos devem ficar juntos verticalmente, bem como funções dependente. Para finalizar, por convecção do java as instancias de variáveis devem ser declaradas no inicio da classe

Para finalizar com a formatação vertical temos a endentação, ela estrutura o código fonte em hierarquias, por exemplo em um arquivo existe a classe, os métodos dentro da classe, os blocos de código dentro dos métodos, e ate mesmo blocos de código dentro dos blocos. Para torna a visualização dessa hierarquia intuitiva edenta-se as linhas de código de acordo com sua hierarquia, métodos dentro de classes são endentados uma nível a direita, implementações do método um nível a direita da declaração do método, assim por diante

Errado	Correto
public class Pessoa { Integer idade; String nome; public Pessoa(int idade, String nome) { this.idade = idade; this.nome = nome; } public boolean canDrive(int idade) { return age>= 18; } }	public class Pessoa { Integer idade; String nome; public Pessoa(int idade, String nome) { this.idade = idade; this.nome = nome; } public boolean canDrive(int idade) { return age>= 18; } }

Na formatação horizontal devemos ter cuidado com o numero de caracteres em uma linha, poucos caracteres podem deixar de revelar o proposito como um todo, porem muitos caracteres podem tornar aquela linha extensa e difícil de entender, para evitar os problemas citados anteriormente cada linha poderá ter no máximo 180 (cento e oitenta) caracteres.

O uso do espaço em branco, horizontalmente, serve para associar coisas que estão intimamente relacionadas e para desassociar as que estão francamente relacionadas. Exemplificando o conceito citado acima, em uma sentença os operadores de atribuição devem ficar entre espaços em brancos mostrando assim o lado esquerdo e direito da sentença, os quais tem sua importância por isso indentifica-los de maneira fácil e intuitiva é essencial,

Errado	Correto
public Pessoa(int idade, String nome) { this.idade=idade; this.nome=nome; }	public Pessoa(int idade, String nome) { this.idade = idade; this.nome = nome; }

Objetos e Estruturas de Dados

Os objetos devem possuir regras de acesso aos atributos, ou seja, acesso através de métodos de forma que a implementação seja ocultada. Chamadas de métodos subsequentes não devem ser utilizadas, devendo instanciar objetos da classe desejada e chamar o método presente nessa classe e assim sucessivamente.

Errado	Correto
String nome = context.person.name;	Context context = this.getContext(); Person person = context.getPerson(); String name = person.getName();

Devem também ser usados objetos de transferência de dados, evitando assim criar funções de comunicação com o banco de dados que recebem muitos parâmetros, prejudicando o entendimento do código.

Errado	Correto
public salvarContato(String email, String nomeContato, String telefone) { //salva o contato }	public salvarContato(Contato contato) { //salvo o contato } public class Contato { private String nomeContato; private String email; private String telefone; ... }

Classes e Interfaces

As classes e interfaces deverão seguir o padrão descrito na sessão “Nomes” desse documento. Devemos ainda adicionar algumas regras que devem ser seguidas, como por exemplo que o nome da classe deve sempre referenciar o seu conteúdo, assim como obedecer o padrão CamelCase além de ser um verbo.

Clases e Interfaces devem possuir no máximo 300 (trezentas) linhas com exceção das Classes Anônimas que devem possuir no máximo 50 (cinquenta) linhas.

Errado	Correto
public class EnviadoraDeEMAIL { ... }	public class EmailService { ... }

Tentei sempre criar apenas uma classe por arquivo. Criar duas classes dentro de um mesmo arquivo pode dificultar o entendimento e possivelmente causar problemas na execução do código. Só crie uma Classe Anônima quando realmente necessário e seja utilizado apenas dentro da Classe “mãe”, use o bom senso.

Erros e Limites

O tratamento de erros é um recurso importante, mas deve ser utilizado com cautela, pois não queremos que obscurecer a lógica do nosso código devido a uma grande quantidade de trechos destinados apenas a tratamento de erros. Para que possamos criar um código limpo e robusto sem deixarmos de tratar os erros, é necessário que sigamos uma série de boas práticas, tais como:

- Utilizar exceções em vez de retornar códigos;
- Utilizar exceções não verificadas;
- Fornecer exceções com contexto;
- Não retonar null;
- Não passar null;
- Criar estruturas try-catch-finally para poder cancelar a execução a qualquer momento;

O controle de limites do nosso software é uma prática bastante complexa e relativa pois, dentro de um sistema, podemos utilizar pacotes de outros fabricantes, podemos utilizar código livre ou até desenvolver nossos componentes dependendo da necessidade. Para que possamos manter os limites do nosso software, é importante que essas decisões sobre o uso de código de terceiros não comprometam a legibilidade do código.

Uma técnica muito recomendável para que os limites de software fiquem em níveis aceitáveis é o Teste de Aprendizagem que, nestes testes, chamamos a API do código externo como faríamos ao usá-lo em nosso aplicativo. Basicamente estaríamos controlando os experimentos que verificam nosso conhecimento daquela API. Estes testes se focalizam no que desejamos saber sobre a API. Resumindo, os testes de aprendizagem são experimentos preciso que ajudam a aumentar nosso entendimento.

É importante lembrar que modificações/alterações de software ocorrem o tempo todo, ainda mais em seus limites, por isso o código nos limites precisa de uma divisão clara e testes para definir o que devemos esperar. É preferível que evitemos que grande parte do nosso código enxergue particularidade dos de terceiros, pois é muito melhor depender de algo que temos controle em vez de utilizarmos algo que pode acabar nos controlando.

Testes

Código de teste deve ser tratado da mesma maneira que o código de produção, logo, ao escrever o teste unitário, deve-se seguir todas as diretrizes citadas acima. Ao se fazer um teste o mesmo deve testar somente uma função, cada teste, uma função.

Errado	Correto
public void testarAceleracao(){ Car car = new Car(); car.accelerate(); car.turnLeft(); car.stop() Assert.assertEquals(0, car.getSpeed()); Assert.assertEquals(0, car.getFuelCapacity()); Assert.assertEquals(0, car.getPassengerCapacity()); }	public void testarAceleracao(){ Car car = new Car(); car.accelerate(); Assert.assertNotEquals(0, car.getSpeed()); }

É de grande importância que o código teste somente uma funcionalidade, como já foi citado, caso o teste fique grande ou teste mais de uma função o mesmo tem altas chances de ser confuso e mal escrito. Os teste não devem possuir dependência entre si, pois possíveis mudanças no código de produção bem como no teste poder influenciar os resultados dos testes, causando duvida em relação ao o que gera a falha.

Em resumo os teste devem seguir 5 (cinco) regras, do acrônimo FIRTS(Fast, Independent, Repeatable, Self-validating, Timely).

- Rápidos
  - Os teste devem ser executados com rapidez, caso o teste seja demorado poderá existir relutância em executa-lo, podendo assim não encontrar falhas cedo quando é fácil concerta-las;

- Independentes
  - Como dito acima os teste devem ser independentes, caso um teste influencia no resultado de outros teste, a falha de um teste causara a falha de todos os outros dificultando uma analise acerca das falhas;

- Repetitivos

    - Os teste devem ser executados em qualquer ambiente, seja no de produção ou em um computador de uso pessoal;

- Autovalidaveis
    - Os teste devem possuir uma saída booleana, logo ou o teste tem sucesso ou ele falha, caso os teste não possuem essa saída os resultados se tornam subjetivos exigindo uma validação formal;

- Pontuais
    - Os teste de unidade devem ser criados antes do código de produção, isso evita um código de produção complexo demais para ser testado ou ate mesmo um que não possa ser testado;