Qualiti Software Processes

Guia de Fatores de Qualidade de OO e *Java*

Versão 1.0

Este documento só pode ser utilizado para fins educacionais, no Centro de Informática da Universidade Federal de Pernambuco. Qualquer outro tipo de utilização deste documento tem que ser expressamente autorizada pela Qualiti Software Processes (www.qualiti.com.br).

© 2000 Qualiti Software Processes.

1 Introdução

Este documento apresenta diretrizes para programação orientada a objetos e *Java* com o intuito de promover a melhoria da qualidade do código escrito segundo os fatores de qualidade: modularidade, legibilidade, robustez e corretude. Assume-se que o código já tenha sido estruturado em camadas independentes conforme o Guia de Estruturação de Aplicações em Camadas Independentes.

2 Legibilidade

2.1 Segue-se o padrão de codificação Java.

2.2 Variáveis auxiliares são declaradas como variáveis locais dos métodos que as utilizam; não são declaradas como variáveis de instância da classe.

2.3 O método padrão "`equals`" de objetos complexos[1] usa o método "`public static boolean equals(Object a, Object b)`", da classe "`Funcoes`", que retorna "`(a == null) && (b == null)) || ((a != null) && (b != null) && (a.equals(b)))`".

2.4 Os métodos com tipo de retorno "`void`" não invocam o comando "`return`".

2.5 O comando condicional abreviado "`condição? comando : comando`" não é usado.

2.6 Constantes são declaradas para evitar o uso direto e repetido de valores específicos.

2.7 A cláusula de importação genérica, como "`import nome.do.pacote.*`", só é usada nos arquivos em que declara-se uma classe com o mesmo nome de uma classe definida em "`nome.do.pacote`".

2.8 O comando "`for`" só é utilizado quando sabe-se exatamente o número de iterações a serem realizadas.

2.9 Tipos enumerados, os quais contêm um número finito de elementos, são representados por interfaces que declaram apenas constantes. Veja o exemplo:

```
public interface DiasSemana {
    public static final int DOMINGO = 1;
    public static final int SEGUNDA = 2;
    public static final int TERCA = 3;
...
}
```

[1] Objetos que fazem referência a outros objetos

2.10 Desde que sejam equivalentes, utiliza-se um único comando "`try-catch-finally`" com várias cláusulas "`catch`" ao invés de vários comandos "`try-catch-finally`" cada um com apenas uma cláusula "`catch`".

3 Modularidade

3.1 Variáveis de instância são *private* ou *protected.*

3.1.1 Variáveis de instância são *private* e são acompanhadas por métodos *set,* com qualificador *protected,* caso elas precisem ser alteradas diretamente nas suas subclasses.

3.2 O construtor de uma classe "`X`" recebe como argumento os objetos que serão utilizados para inicializar os atributos de "`X`", ao invés de receber as informações necessárias para a criação desses objetos.

3.3 O código da GUI trabalha com os objetos da camada de negócio ("`Conta`", "`Cliente`", etc.) e não apenas com "`String`", "`int`", etc.

3.4 O código da GUI só verifica as regras de negócio relativas à validação de dados fornecidos pelos usuários; as demais regras ficam por conta da camada de negócio. (Por exemplo, a verificação de que o nome de um cliente não pode ser a *String* vazia é feita pela GUI)

3.4.1 O código da GUI não realiza novas operações, conceitualmente falando, sobre os objetos da camada de negócio. (Por exemplo, ao invés da GUI realizar um crédito em uma conta seguido de um débito em outra, a GUI chama diretamente o método transfere da camada de negócio)

3.5 Conceitos que podem ser diretamente representados por uma *String* ou um tipo primitivo ("`int`", "`double`", etc.) são representados por uma classe caso seja preciso realizar alguma operação sobre os elementos associados aos conceitos. Veja o exemplo:

```
public class AutorLivro {


    String nomeAutor;

    public String getUltimoNome() {
    ...
    }
    public String getNome() {
    ...
    }
}
```

3.6 As classes do sistema são distribuídas em pacotes.

3.6.1 Os pacotes agrupam classes conceitualmente relacionadas.

3.6.2 Os pacotes não têm dependências mútuas.

3.7 Classes são normalizadas no sentido de que elas não representam mais de um conceito.(Por exemplo, uma classe "`Cliente`" não tem variáveis de instância "`rua`" e "`cidade`", mas sim uma variável "`endereco`". Da mesma forma, uma classe `Recebimento` não tem variáveis de instância "`banco`" e "`agencia`", mas sim "`referenciaBancaria`")

3.8 O padrão de projeto *Item-Item description* é utilizado. Por exemplo, tem-se a classe "`Livro`", com código, localização na biblioteca, descrição, etc., e a classe "`DescricaoDeLivro`", com título, autor, ISBN, etc., a classe "`FitaDeVideo`" e a classe "`Filme`", a classe "`ItemDeVenda`" e a classe "`Produto`".

3.9 Métodos são declarados públicos (`public`) ou protegidos (`protected`) quando há justificativa.

3.10 Variáveis estáticas são usadas apenas em casos excepcionais, devendo haver uma boa justificativa para seu uso. (Por exemplo, o número da próxima conta a ser cadastrada é um atributo de "`CadastroContas`", e não uma variável estática de "`Conta`")

3.11 Métodos estáticos são usados apenas em casos excepcionais, devendo haver uma boa justificativa para seu uso. (Por exemplo, é bem justificado o uso de métodos estáticos para definir novas operações relacionadas a tipos primitivos, ou a tipos pré-definidos e finais, como "`String`".)

3.12 A camada de negócio não contém código relativo à GUI. (Por exemplo, o código da camada de negócio não contém chamadas ao método "`System.out.print`")

3.12.1 Os atributos de uma exceção, inclusive o atributo de "`Exception`", não são mensagens a serem impressas pela GUI, mas sim informações que possam ser utilizadas para gerar tais mensagens, independente da língua a ser utilizada por uma determinada configuração da aplicação.

3.12.2 A camada de negócio não verifica as regras de negócio relativas à validação de dados fornecidos pelos usuários, a menos que estas regras sejam essenciais para o correto funcionamento das operações de negócio. (Estas regras podem até ser implementadas através de métodos da camada de negócio, mas são verificadas pela GUI)

3.13 Métodos auxiliares, privados, são definidos para evitar duplicação de código nos métodos de uma classe.

3.14 As exceções de APIs específicas utilizadas pelas camadas de dados e comunicação não são propagadas para a camada de negócio; ao invés disso, estas exceções são trocadas por exceções genéricas.

4 Robustez

4.1 Situações de erro na execução de um método são indicadas por exceções.

4.1.1 Erros tratados de maneiras diferentes são indicados por exceções de classes diferentes. (Por exemplo, ao invés de usar "`IllegalArgumentException`", usa-se as suas subclasses "`EnderecoNuloException`" e "`ClienteNuloException`" caso os tipos de erros associados devam ser tratados de maneiras diferentes de outros erros causados por argumentos inválidos)

4.1.2 Exceções de classes diferentes, mas que devem ser tratadas do mesmo jeito em algumas situações, são agrupadas através da definição de uma superclasse comum, a ser utilizada na cláusula "`catch`" quando as exceções tiverem que ser tratadas do mesmo jeito.

4.2 Todas as exceções são tratadas, em lugares apropriados, dependendo de que providências queira-se tomar. (Por exemplo, exceções que correspondem a situações que devem ser reportadas aos usuários são tratadas na GUI)

4.2.1 Exceções não são tratadas por cláusulas do tipo "`catch (TipoException e) {}`" ou "`catch (TipoException e) {System.out.print("Erro Interno...");}`"

4.2.2 Uma exceção é tratada por uma cláusula do tipo "`catch (TipoException e) {throw new ErroInternoException(e);}`" caso tenha-se certeza de que a exceção não será levantada. ("`ErroInternoException`" é uma subclasse de "`RunTimeException`", de forma que não precisa ser declarada em cláusulas "`throws`")

4.3 Métodos e construtores que recebem referências para objetos como argumento geram indicações de erro caso alguma das referências seja "`null`", a menos que exista uma boa justificativa para aceitar uma referência "`null`".

5 Corretude

5.1 O contexto de uma chamada de método como "`o.m(argumentos)`" garante que a variável "`o`" não contém "`null`".

5.2 Cada classe tem um método "`public static void main(String[] argc)`"que testa os métodos mais críticos e não-triviais da classe.

5.2.1 Os métodos mais críticos e não-triviais são exaustivamente testados, explorando situações não equivalentes.

5.3 *Casts* são usados em contextos que garantem que eles serão bem sucedidos. (Por exemplo, após um teste com "`instanceof`".

5.4 Nenhum método possui uma cláusula de acesso mais abrangente do que a necessária para sua utilização.

5.5 Um método que não retorna "`void`" possui apenas um (**1**) comando "`return`" **.**

```
if (condicao) {
    a = x + VALOR_CONSTANTE;
    ..........
    return a;
}
return y - x; //ERRADO

TipoRetorno retorno;
if (condicao) {
    a = x + VALOR_CONSTANTE;
    ..........
    retorno = a;
}
else {
    retorno = y - x;
}
return retorno; //CORRETO
```

5.6 Objetos armazenados pelas coleções de negócio não são retornados por métodos da fachada, a menos que o tipo de retorno do método seja uma interface que contém apenas os métodos de leitura do objeto retornado, caso contrário, retorna-se *clones* ou versões protegidas com atualização proibida.

5.7 Redefinições de métodos **concretos** preservam o comportamento dos métodos originais; isto é, toda redefinição de método tem o mesmo efeito do método original sobre os atributos da superclasse, mas pode ter qualquer efeito sobre os atributos da subclasse.