3PUP - Controle de Qualidade

⚠ Esta página está em construção.

ⓘ **Nesta pagina**

Informações sobre os processos, técnicas, modelos, padrões, diagramas e outros artefatos para controle de qualidade dos entregáveis do projeto. Os itens mostrados aqui são uma compilação das diretrizes acordadas com o cliente e fazem parte do processo de garantia do projeto.

O conteúdo dessa página deve ser utlizado como parâmetro de aceite para a etapa de "Aprovar construção" e "Aprovar homologação" no fluxo de trabalho de cada *feature* do projeto.

1. Introdução

Este documento é o balizador do controle de qualidade de artefatos produzidos no 3PUP. Ele contém critérios de qualidade a serem seguidos para construção de interfaces de usuário, código-fonte e processo de testes do sistema a ser entregue.

2. Interface de Usuário

Esta seção cobre critérios de qualidade esperados para as interfaces de usuário do sistema. Como interface de usuário podem ser consideradas telas da aplicação, dispositivos de acionamento óptico, som, movimento ou qualquer outra interface de entrada ou saída de informação do sistema.

2.1.1. Telas do Sistema

O objetivo do controle de qualidade para as telas do sistema é manter uma aparência, comportamento e usabilidade consistentes ao longo de toda a aplicacao.

Os critérios para checagem de qualidade das telas do sistema são (1):

CHECKLIST 1: Checklist para verificação de telas de sistema

Número	Artefato	Elemento	Item	Teste a ser realizado	Status
1	Tela de sistema	Navegador (Browser)	Tipo e Versão	A tela, os seus controles e o comportamento de funcionamento está consistente em relação a(s) versão (ões) do(s) navegador(es) padrão(ões) especificado(s) pela arquitetura da aplicação ?	✓ ✗
2	Tela de sistema	Textos de Tela	Ortografia	Os textos estáticos e dinâmicos da tela, os valores de listas e de controles, os títulos de janela e de painéis, os textos de botões e as legendas estão ortograficamente corretos ?	✓ ✗
3	Tela de sistema	Textos de Mensagens	Ortografia	As mensagens de informação, as de auxílio, as de aviso, as de notificação, as de erro e as dicas de tela (*tooltips*) estão ortograficamente corretas ?	✓ ✗
4	Tela de sistema	Exibição de mensagens	Local e Momento	As mensagens de informação, as de auxílio, as de aviso, as de notificação, as de erro e as dicas de tela (*tooltips*) estão sendo exibidas nos locais corretos e nos momentos apropriados ?	✓ ✗
5	Tela de sistema	Teclas de Atalho	Funcionamento	Se aplicável, as teclas de atalho do sistema estão funcionando adequadamente, acionando os menus de tela, operações e ações esperadas ?	✓ ✗
6	Tela de sistema	Campos de Dados	Tabulação	A ordem de tabulação dos campos de formulário está condizente com a disposição do leioute dos campos da tela ?	✓ ✗
7	Tela de sistema	Elementos de Tela	Espaçamento	Os espaçamentos entre campos, bordas de campos, guias, painéis e entrelinhas está consistente ao longo de todos elementos da tela, e da tela em relação à outras telas do sistema ?	✓ ✗
8	Tela de sistema	Campos de Dados	Ordenamento	A disposição dos campos de formulário na tela estão de acordo com as premissas da especificação técnica ?	✓ ✗
9	Tela de sistema	Elementos de Tela	Leioute e Alinhamento	O alinhamento vertical e o alinhamento horizontal dos elementos em relação à tela, e dos elementos de tela entre si está condizente com o projeto gráfico ?	✓ ✗
10	Tela de sistema	Textos de Tela	Fontes e Formatação	Os estilos e os tamanhos de fonte, negritos, itálicos, sublinhados e outros efeitos de fonte estão aderentes ao projeto gráfico ?	✓ ✗

11	Tela de sistema	Elementos de Tela	Cores e Efeitos Visuais	As cores de primeiro e segundo plano de textos, as de caixas de mensagem, as de dicas de tela (*tooltips*), as de painéis, as de títulos e as de rótulos estão aderentes ao projeto gráfico ?	
12	Tela de sistema	Elementos de Tela	Imagens e Fundo de Tela	As imagens utilizadas nas telas, painéis e controles, as imagens de fundo, os degradês e as sombras estão aderentes ao projeto gráfico ?	
13	Tela de sistema	Ícones		Os ícones de telas, os de painéis, os de mensagem estão sendo adequadamente utilizados, incluindo tamanho, cores e local de apresentação ?	
14	Tela de sistema	Sons e Vídeo		Se aplicável, os efeitos sonoros e os vídeos associados à tela e os controles de tela estão corretos, em relação à tamanho de tela, volume sonoro e tempos ?	
15	Tela de sistema	Menus, Sub-menus e Itens de Menus		Os menus de ação, seus sub-menus e itens estão exibidos nos locais corretos, na ordem correta, com textos corretos e executam as funções corretas ?	
16	Tela de sistema	Navegação		Os links de navegação, os menus de navegação e o sequenciamento de telas do sistema estão condizentes com o projeto gráfico e o projeto técnico ?	
17	Tela de sistema	Botões e Links de Ação		Os botões de tela e os links de ação da tela estão nos locais corretos, executam as funções corretas ?	
18	Tela de sistema	Botão Voltar do Browser		A tela suporta sem erros o uso do botão "voltar" do navegador ?	
19	Tela de sistema	Controle de Dupla Submissão		Os botões de tela, teclas de atalho, menus, links de ação e outras operações que fazem submit de dados para o servidor fazem tratamento correto no caso de dupla-submissão do formulário ?	
20	Tela de sistema	Dependências entre Campos		As automações de dependências entre campos "preencher-e-mostrar", "preencher-e-habilitar", "limpar-e-ocultar", "limpar-e-desabilitar", "clicar-e-editar", "preencher-e-carregar" estão aderentes ao projeto técnico ?	
21	Tela de sistema	Eventos, Ajax e Comportamento		Os eventos de tela e de campos, como *OnClick(), OnChange(), OnFocus(), OnLostFocus()* e outros disparam os comportamentos esperados conforme o projeto técnico da tela ?	
22	Tela de sistema	Máscaras de Preenchimento		As máscaras para auxílio para o preenchimento de campos estão corretas para os tipos e formatos de dados, como campos numéricos, campos de texto, campos de data, campos de hora, campos de intervalo e campos de conteúdo monetário estão consistentes com o projeto técnico da tela ?	
23	Tela de sistema	Validação de Dados		As validações de dados, como campos obrigatórios, campos opcionais, campos dependentes, campos com domínio de valores estão sendo executadas no momento correto, sobre o campo correto e produzindo as mensagens de retorno corretas, conforme projeto técnico ?	
24	Tela de sistema	Internacionalização		Se aplicável, informações sensíveis à internacionalização (país, localidade, *timezone*, horário de verão, moeda) como textos, números datas e horários foram testados adequadamente usando configurações regionais do Sistema Operacional hospedeiro compatíveis com cada língua e localidade definida pelo projeto técnico ?	

3. Código-Fonte

Esta seção apresenta os critérios de qualidade para o código-fonte e regras de compilação (ex. geração de *warnings* em código-fonte) do sistema.

> **Dica**
>
> A utilização de todos os padrões abaixo, tal qual estão especificados, evita um grande número de *commits* desnecessários no projeto. Isso ocorre porque, simples ações na tela da IDE do usuário, como um "Control+Shift+F" (formatar código-fonte) ou um "Control+Shift+O" (organizar *imports*) impactam em alterações de no arquivo fonte, as quais derivam em *commits*, que seriam totalmente desnecessários se todos no projeto estiverem utilizando as regras aqui descritas.

3.1. Regras de Codificação

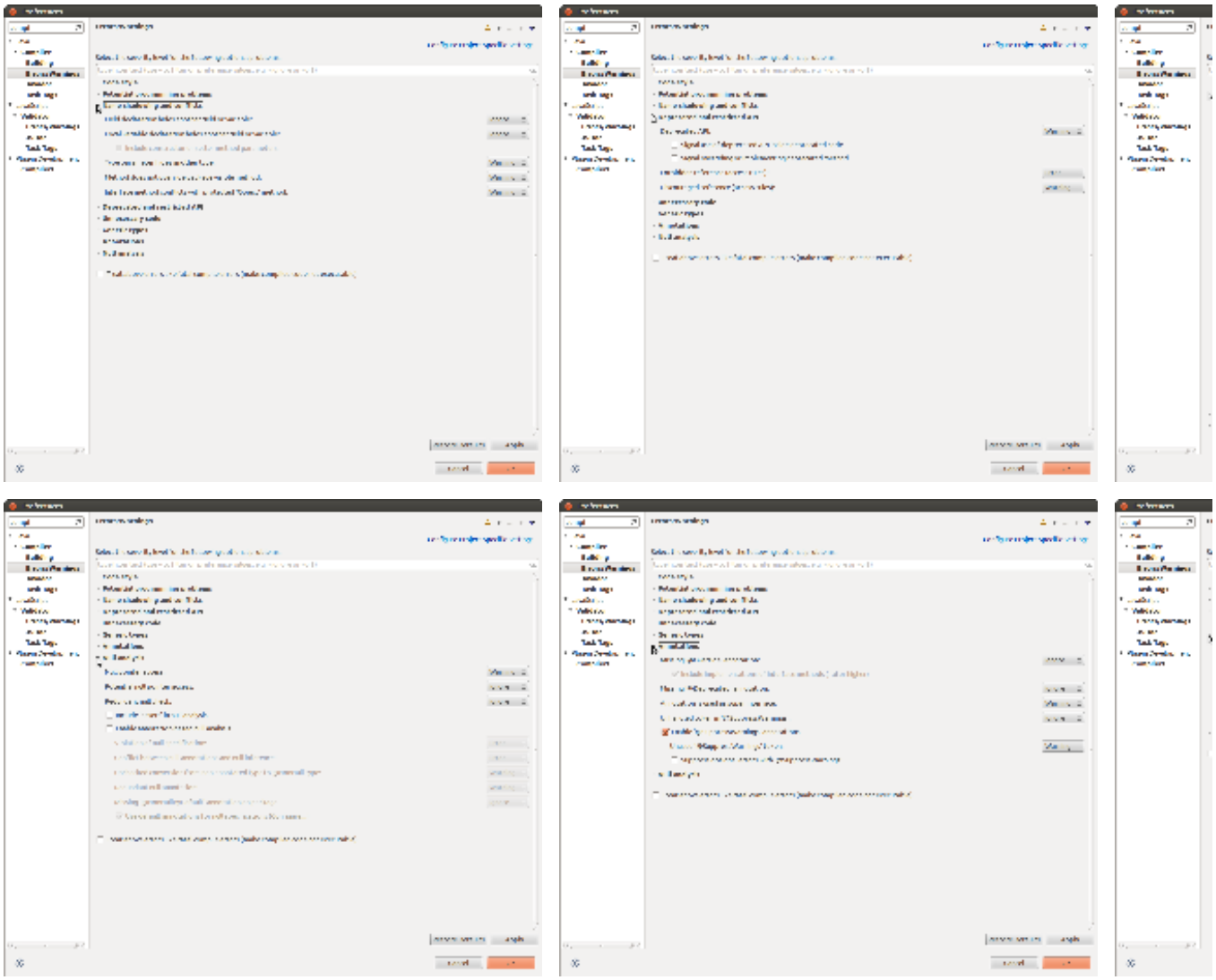
A *Integrated Development Environment* (IDE) para o ambiente é o EclipseIDE, na sua distribuição Juno. Opcionalmente, o JBoss Developer Studio e o SpringSource, ambas variantes do EclipseIDE são suportadas.

O objetivo do controle de qualidade para as regras de codificação no ambiente de desenvolvimento é fazer com que todos os envolvidos no desenvolvimento do sistema produzam código-fonte no mesmo formato e estrutura de escrita, de forma que exista uma isonomia do código-fonte ao longo de todo o processo de desenvolvimento e manutenção, a qual seja independente de preferências pessoais, vícios de programação, habilidades e experiência do desenvolvedor entre outros fatores comportamentais sucetíveis à gostos pessoais.

Independente da variante da IDE, as regras de configuração do ambiente de desenvolvimento devem seguir as premissas abaixo:

3.1.1. Compilação

As regras de configuração do compilador da IDE devem possuir as seguintes entradas:



Para ser consistente com o padrão de qualidade do projeto, as regras compilação da IDE do desenvolvedor devem seguir as configurações acima.

Para efeitos de teste de aceite, com o projeto aberto na IDE, as configurações em uso na IDE do desenvolvedor devem ser idênticas às acima.

3.1.2. Limpeza de Código-Fonte (Clean Up)

A limpeza de código-fonte diz respeito aos padrões para eliminação de conteúdo desnecessário ou incorreto dentro do código-fonte do sistema, tais como espaços em branco, falta de modificadores *final* em variáveis, ausência ou uso incorreto de de anotações @Overide, @SupressWarings entre outras, além de conversões desnecessárias, importações de dependências desnecessárias ou em nível de profundidade (*) desapropriados, linhas em branco, tabulações e identações incorretas, etc...

As regras de configuração para limpeza de código-fonte da IDE são assim definidas:

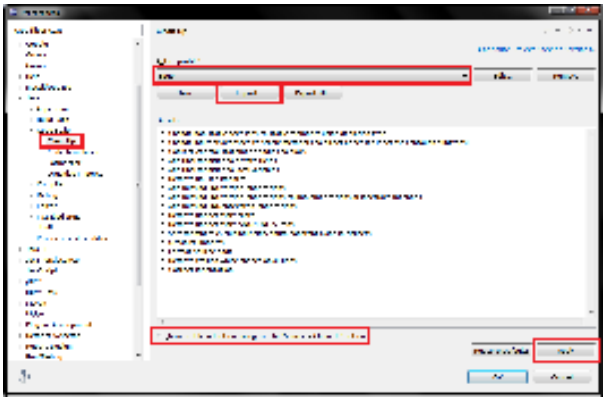
Data	Versão	Aprovador	Publicador	Local	Comentários
2014/Janeiro/17	1	(arquivo sob avaliação)	Yuri Morales Britto	Download	Arquivo oriundo do padrão 3PUP.

Para aplicar as regras de limpeza de código-fonte, deve ser realizada a importação do arquivo de configuração no ambiente de desenvolvimento*:*

- Vá até a opção "Windows > Preferences > Java > Code Style > Clean Up"
- Clicar em importar

- Verifique se o nome no campo "Active Profile" está como "3PUP" (Caso não esteja, o mesmo deve ser selecionado na combo de seleção do campo)
- Desmarcar a opção "Show profile selection dialog for the 'Source > Clean Up' action"
- Clicar em 'Apply'

Ao final da configuração, a tela deve estar igual a imagem abaixo:



~~Para ser consistente com o padrão de qualidade código-fonte do projeto, as regras limpeza de código-fonte da IDE do desenvolvedor devem seguir as configurações acima.~~

~~Para efeitos de teste de aceite, com o projeto aberto na IDE, as configurações em uso na IDE do desenvolvedor devem ser idênticas às acima.~~

Dica

O atalho de teclado "Alt+Shift+S > U" aplicam estas regras automaticamente.

Atenção

Ao executar um "Control+Shift+S > U" em um arquivo recém obtido do repositório versionado (*update*) e esta operação implicar alterações no conteúdo do arquivo, isso indica inevitavelmente que (i) ou a pessoa que realizou o *commit* anterior não estava usando as regras corretas de limpeza de código-fonte, ou (ii) o usuário atual não está as utilizando.

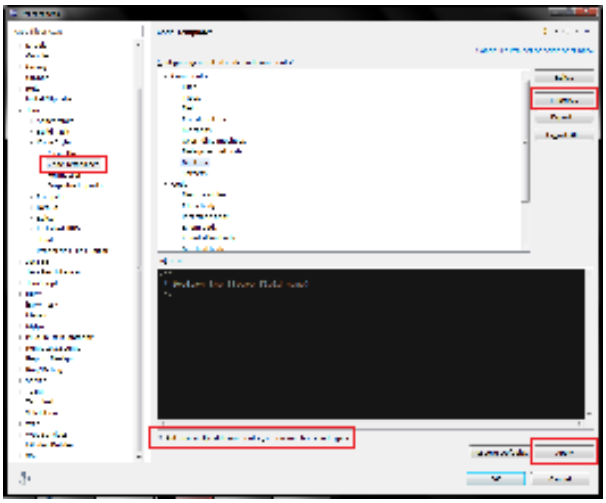
3.1.3. Modelos de Código-Fonte (Code Templates)

Modelos de código-fonte dizem respeito à padrões para estruturas como blocos de código-fonte, conteúdos padrões para métodos *get* e *set*, para construtores, para tratamento padrão de erros, entre outros. Também define padrões para escrita de comentários em código-fonte, como cabecalhos de arquivo, Javadoc para classes, variáveis, métodos, sobreescrita de métodos, delegação de chamadas, entre outros.

As regras de codificação para modelos de código-fonte, incluindo os comentários, são definidas no arquivo abaixo:

Data	Versão	Aprovador	Publicador	Arquivo	Comentários
2014/Janeiro/17	1	(arquivo sob avaliação)	Yuri Morales Britto	Download	Arquivo oriundo do padrão 3PUP.

Para aplicar as regras de modelos de código-fonte, deve ser realizada a importação do arquivo de configuração no ambiente de desenvolvimento, no menu *Window* > *Preferences* > *Java* > *Code Style* > *Code Templates*, opção *Import*. E, certifique-se que a opção *Automatically add comments for new methods and types* na tela relacionada, conforme abaixo:



Para ser consistente com o padrão de qualidade código-fonte do projeto, as regras modelos de código-fonte da IDE do desenvolvedor devem seguir as configurações acima.

Para efeitos de teste de aceite, com o projeto aberto na IDE, as configurações em uso na IDE do desenvolvedor, bem como as estruturas finais geradas a partir destes modelos (ex. o conteúdo de um metoto *getter*) devem continuar consistentes com este padrão.

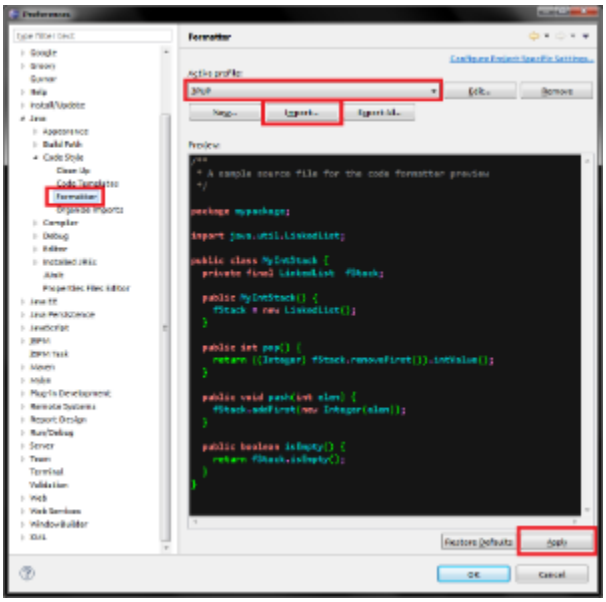
3.1.4. Formatação de Código-Fonte (Formatter)

As regras de formatação de código-fonte dizem respeito à padronização da estrutura de apresentação dos fontes do sistema. Enquanto as regras de limpeza de código discutidas anteriormente buscam ajustar o código-fonte em relação a um padrão, as regras de formatação definem este padrão. Regras como tamanho da linha de codigo-fonte, uso de espaços x tabulação, alinhamento de campos de classes, locais dos sinais de chaves de abertura de fechamento de blocos de código-fonte, uso de linhas em branco para espaçamento, identação, regras para declaração de comentários, entre várias outras.

As regras para formatação de código-fonte, incluindo os comentários, são definidas no arquivo abaixo:

Data	Versão	Aprovador	Publicador	Arquivo	Comentários
2013 /Maio/30	1	(aguardando cliente, sob responsabilidade de Leopoldo Barreiro)	Marcelo Mrack	Download	Arquivo oriundo e idêntico do padrão 3PUP utilizado pela fábrica de projetos da Redhat em Porto Alegre.

Para aplicar as regras de formatação de código-fonte, deve ser realizada a importação do arquivo de configuração no ambiente de desenvolvimento, no menu *Window* > *Preferences* > *Java* > *Code Style* > *Formatter*, opção *Import*, conforme abaixo:



Para ser consistente com o padrão de qualidade código-fonte do projeto, as regras formatação de código-fonte da IDE do desenvolvedor devem seguir as configurações acima.

Para efeitos de teste de aceite, com o projeto aberto na IDE, as configurações em uso na IDE do desenvolvedor, bem como as estruturas finais geradas a partir desse formato (ex. identação) devem continuar consistentes com este padrão.

> **Dica**
>
> O atalho de teclado "Control+Shift+F" aplicam estas regras automaticamente.

> **Atenção**
>
> Ao executar um "Control+Shift+F" em um arquivo recém obtido do repositório versionado (*update*) e esta operação implicar alterações no conteúdo do arquivo, isso indica inevitavelmente que (i) ou a pessoa que realizou o *commit* anterior não estava usando as regras corretas de formatação de código-fonte, ou (ii) o usuário atual não está asa utilizando.

3.1.5. Organização de Dependências (Organize Imports)

A importação de dependências de pacotes e classes em Java é recurso imprescindível para o desenvolvimento de aplicações. Entretanto, a utilização incorreta de importações deriva para problemas de dependências no sistema, tais como uso de arquivos JAR desnecessários, verbosidade excessiva em código-fonte e alta acoplabilidade (pacotes que dependem indevidamente de outros pacotes) no sistema.

Com o intuito de evitar ao máximo tais problemas no projeto, as regras para organização de dependências (*imports*) de código-fonte, são definidas no arquivo abaixo:

Data	Versão	Aprovador	Publicador	Arquivo	Comentários
2013 /Maio/30	1	(aguardando cliente, sob responsabilidade de Leopoldo Barreiro)	Marcelo Mrack	Download	Arquivo oriundo e idêntico do padrão 3PUP utilizado pela fábrica de projetos da Redhat em Porto Alegre.

Para ser consistente com o padrão de qualidade código-fonte do projeto, as regras organização de dependências da IDE do desenvolvedor devem seguir as configurações acima.

Para efeitos de teste de aceite, com o projeto aberto na IDE, as configurações em uso na IDE do desenvolvedor devem ser idênticas às acima.

> O atalho de teclado "Control+Shift+O" aplicam estas regras automaticamente.

> **Atenção**
>
> Ao executar um "Control+Shift+O" em um arquivo recém obtido do repositório versionado (*update*) e esta operação implicar alterações no conteúdo do arquivo, isso indica que (i) ou a pessoa que realizou o commit anterior não estava usando as regras corretas de organização de dependências corretas, ou (ii) o usuário atual não está as utilizando.

3.1.6. Análise de Código-Fonte

Além das regras de codificação utlizadas na IDE, duas ferramentas extras são utilizadas para o processo de montagem (*build*) da aplicação. São elas o PMD e o Checkstyle.

3.1.6.1. PMD

O PMD é um analisador de código-fonte. Ele localiza falhas de programação como variávies não utilizadas, blocos de tratamento de erros vazios, criação desnecessária de objetos e uma gama de outros elementos. Ele tem suporte para linguagem Java e Javascript e formatos XML e XSL. Adicionalmente, inclui suporte para deteção de "copy-and-paste" (CPD) de código-fonte, uma prática que pode ser nociva também. O CPD está disponível para Java, C, C++, C#, PHP, Ruby, Fortran, JavaScript. O PMD é altamente configurável e pode ser integrado de diversas forms nos processos de desenvolvimento.

As regras de configuração da ferramenta PMD definidas para este projeto estão no arquivo de configuração da ferramenta, que pode ser acessado abaixo:

Data	Versão	Aprovador	Publicador	Arquivo	Comentários
2013/05/30	1.0		Marcelo Mrack	Download	TODO Definir arquivo templayte

Estas regras precisam ser aplicadas nos três ramos integração contínua do projeto, *Desenvolvimento*, *Homologação* e *Produção*, via ferramenta de automação de *build*, e serem publicadas como artefato de construção.

Para ser considerado aprovado o código-fonte em relação ao PMD, nenhum erro deve ser reportado no relatório de análise.

3.1.6.2. Checkstyle

O Checkstyle é uma ferramenta de desenvolvimento que ajuda os programadores Java a produzirem código-fonte aderente a um determinado padrão de escrita. Regras como identação, tabulação, tamanho de linhas, sintaxe de variáveis e métodos e uma série de outros elementos podem ser definidas. Ele é altamente configurável e pode suportar diferentes padrões, como o Sun Code Conventions.

Importante

Excepcionalmente, no projeto SEPD a ferramenta Cheskstyle não é utilizada, conforme informação passada por Heliton Filho em 29-Maio-2013.

Uma padronização para uso desta ferramenta deve ser discutida em momento oportuno para novos projetos, sob responsabilidade de Leopoldo Barreiro esta evolução.

~~As regras de configuração da ferramenta Checkstyle definidas para este projeto estão no arquivo de configuração da ferramenta, que pode ser acessado abaixo:~~

Data	Versão	Aprovador	Publicador	Arquivo	Notas
~~2013/05/30~~	~~1.0~~	~~(aguardando cliente, sob responsabilidade de Leopoldo Barreiro)~~	~~Marcelo Mrack~~	~~Download~~	Aguardando padronização.

~~Estas regras precisam ser aplicadas nos três ramos integração contínua do projeto, *Desenvolvimento*, *Homologação* e *Produção*, via ferramenta de automação de *build*, e serem publicadas como artefato de construção.~~

~~Para ser considerado aprovado o código fonte em relação ao Chesktyle, nenhum erro deve ser reportado no relatório de análise.~~

4. Revisão de Código-Fonte

A revisão de código-fonte é parte da etapa de construção, ocorrendo imediateamnte após a programação e imediatamente antes do disparo dos testes.

O objetivo da revisão de código-fonte é fazer uma análise do que foi codificado para garantir que nada tenha passado despercebido antes de iniciar as baterias de teste, os quais seriam perdidos ou gerariam altos índices de falha caso tal processo não tenha sido realizado.

A revisão de código-fonte é uma atividade que deve ser desenvolvida obrigatoriamente por outra pessoa que não o codificador do programa. Pode ser outro desenvolvedor (conceito de programação em par) o mesmo o projetista ou arquiteto, dependendo da complexidade e outros fatores do artefato associado.

4.1. Lista de Verificação de Código-Fonte

As verficações que devem ser efetuadas no processo de revisão de código-fonte são assim sumarizadas:

CHECKLIST 2: Checklist para Teste de Revisão de Código-Fonte.

Número	Artefato	Elemento	Item	Teste	Meio	Status
1	Arquivo de Classe	Método público	Javadoc do método	O método possui Javadoc associado ?	Aferição visual ou Checkstyle	
2	Arquivo de Classe	Método público	Javadoc dos parâmetros	Se existentes, os parâmetros do método estão documentados no Javadoc ?	Aferição visual ou Checkstyle	
3	Arquivo de Classe	Método público	Javadoc do retorno	Se existente, o retorno do método está documentado no Javadoc ?	Aferição visual ou Checkstyle	
4	Arquivo de Classe	Método qualquer	Sintaxe do nome	O nome do método está em conformidade com o padrão 3PUP ?	Aferição visual	
5	Arquivo de Classe	Método qualquer	Sintaxe dos parâmetros	Os nomes dos parâmetros estão em conformidade com o padrão 3PUP ?	Aferição visual	
6	Arquivo de Classe	Método qualquer	Semântica do método	A semântica de execução do método está consistente com o seu nome e, se existente, com o Javadoc associado (ou seja ele faz o que diz fazer) ?	Aferição visual	
7	Arquivo de Classe	Método qualquer	Semântica do parâmetros	As funcionalidades dos parâmetros estão consistentes com o seu nome e, se existentel, o Javadoc associado (ou seja eles fazem o que dizem fazer) ?	Aferição visual	
8	Arquivo de Classe	Método qualquer	Semântica do retorno	Se aplivável, o método retorna as informações corretas, conforme o projeto técnico e, se existente, o Javadoc associado (ou seja, ele retorna o que diz retornar) ?	Aferição visual	
9	Arquivo de Classe	Método qualquer	Validação de parâmetros	Se existente, o método realiza ou delega o tratamento de validação e de erros em caso de parâmetros inconsistentes ou nulos ?	Aferição visual	
10	Arquivo de Classe	Método qualquer	Objetivida de	O método realiza apenas as operações estritamente necessárias para sua funcionalidade de negócio, conforme projeto técnico ?	Aferição visual	

11	Arquivo de Classe	Método qualquer	Desempenho	O método não possui nenhum *statement* que afete negativamente o desempenho do sistema ?	Aferição visual	✅❌
12	Arquivo de Classe	Método qualquer	Segurança	O método não possui nenhum *statement* que implique em falhas ou violação da segurança do sistema ?	Aferição visual	✅❌
13	Arquivo de Classe	Variável pública	Javadoc da variável	A variável possui Javadoc associado ?	Aferição visual ou Checkstyle	✅❌
14	Arquivo de Classe	Variável qualquer	Sintaxe do nome	O nome da variável está em conformidade com o padrão 3PUP ?	Aferição visual	✅❌
15	Arquivo de Classe	Variável qualquer	Semântica do nome	A semântica da variável está consistente com o seu nome e, se existente, com o Javadoc associado (ou seja ela faz o que diz fazer) ?	Aferição visual	✅❌
16	Arquivo de Classe	Classe pública	Javadoc da classe	A classe possui Javadoc associado ?	Aferição visual ou Checkstyle	✅❌
17	Arquivo de Classe	Classe qualquer	Sintaxe do nome	O nome da classe está em conformidade com o padrão 3PUP ?	Aferição visual	✅❌
18	Arquivo de Classe	Classe qualquer	Conteúdo da classe	Os métodos e/ou outras classes contidas na classe estão aderentes ao seu nome e, se existente, ao Javadoc associado (ou seja ela contém aquilo e somente aquilo que deveria conter) ?	Aferição visual	✅❌
19	Arquivo de Pacote	Pacote	Javadoc do pacote	O pacote possui Javadoc via arquivo *package-info.java* associado ?	Aferição visual ou Checkstyle	✅❌
20	Arquivo Pacote	Pacote	Sintaxe do nome	O nome do pacote está em conformidade com o padrão 3PUP (ex.: *x.p.t.o.umPacoteQualquerComNomeExtenso.utils.x.y.z*) ?	Aferição visual	✅❌
21	Pacote	Pacote	Conteúdo do pacote	As classes e enumerações contidas no pacote estão aderentes ao o seu nome e ao Javadoc associado (ou seja ele contém aquilo e somente aquilo que deveria conter) ?	Aferição visual	✅❌
22	Arquivo com pacotes	Módulo	Sintaxe do nome	O nome do módulo (arquivo JAR, WAR ou outro) está em conformidade com o padrão 3PUP ?	Aferição visual	✅❌
23	Arquivo com pacotes	Módulo	Conteúdo do módulo	O pacotes contidos no módulo estão aderentes ao seu nome e ao projeto técnico associado (ou seja ele contém aquilo e somente aquilo que deveria conter) ?	Aferição visual	✅❌
24	Arquivo com módulos	Sistema	Sintaxe do nome	O nome do sistema (arquivo JAR, WAR, EAR ou outro) está em conformidade com o padrão 3PUP (não considerando o sufixo de numeração) ?	Aferição visual	✅❌
25	Arquivo com módulos	Sistema	Conteúdo do sistema	Os módulos contidos no sistema estão aderentes ao seu nome e ao projeto técnico associado (ou seja, ele contém aquilo e somente aquilo que deveria conter) ?	Aferição visual	✅❌

4.2. Nomenclatura de Classes, Métodos e Variáveis

Quanto à nomenclatura de classes, métodos e variáveis, o padrão 3PUP utiliza o conceitos Domain Driven Design (DDD) para construção de artefatos. Ne ste sentido, a nomenclatura de classes, métodos, variáveis, pacotes e qualquer outro artefato de código-fonte segue o padrão dado pela sintaxe abaixo:

4.2.1. Nomes de Classes

Sintaxe:

```
Substantivo[Preposicao((Substantivo|SubstantivoConjuncaoSubstantivo)|(Adjetivo|AdjetivoConjuncaoAdjetivo)|
(SubstantivoAdjetivo|SubstantivoAdjetivoConjuncaoSubstantivoAdjetivo))]*
```

Exemplos:

- Cliente
- ClienteAtivo
- ClienteDoMarketing
- ClienteAtivoDoMarketing
- ClienteAtivoDoMarketingDeEmpresasDoRamoAutomobilisticoAtual
- ClienteAtivoDoMarketingEFinanceiroDeEmpresasDoRamoAutomobilisticoAtual

4.2.2. Nomes de Métodos

Sintaxe (apenas possui um verbo inicial frente à sintaxe de classes, o qual inicia-se em minúsculo):

```
verboSubstantivo[Preposicao((Substantivo|SubstantivoConjuncaoSubstantivo)|(Adjetivo|AdjetivoConjuncaoAdjetivo)|
(SubstantivoAdjetivo|SubstantivoAdjetivoConjuncaoSubstantivoAdjetivo))]*
```

Exemplos:

- buscar()
- buscarPedido()
- buscarPedidoDeUso()
- buscarPedidoDeUsoDosClientes()
- buscarPedidoDeUsoDosClientesAtivos()
- buscarPedidoDeUsoDosClientesNovos()
- buscarPedidoDeUsoDosClientesCorporativos()
- buscarPedidoDeUsoDosClientesCorportativosPorNome(String nome)

4.2.3. Nomes de Variáveis

Sintaxe (idêntica à sintaxe de classes, porém iniciando-se em minúsculo):

```
substantivo[Preposicao((Substantivo|SubstantivoConjuncaoSubstantivo)|(Adjetivo|AdjetivoConjuncaoAdjetivo)|
(SubstantivoAdjetivo|SubstantivoAdjetivoConjuncaoSubstantivoAdjetivo))]*
```

Exemplos:

- cliente
- clienteAtivo
- clienteDoMarketing
- clienteAtivoDoMarketing
- clienteAtivoDoMarketingDeEmpresasDoRamoAutomobilisticoAtual
- clienteAtivoDoMarketingEFinanceiroDeEmpresasDoRamoAutomobilisticoAtua

4.2.4. Operadores Relacionais de Quantificação

⚠️ Em construção.

Esta seção visa normatizar a nomenclatura de classes, variáveis e, principalmente, métodos quando operadores relacionais de quantificação, tais como:

1. buscarClientesCorporativosOndeNomeIniciaCom(parteDoNome);
2. buscarClientesCorporativosOndeNomeTerminaCom(parteDoNome);
3. buscarClientesCorporativosOndeNomeContem(parteDoNome);
4. buscarClientesCorporativosOndeIdadeMaiorQue(idadeMinima);
5. buscarClientesCorporativos(nomeIniciaCom);
6. buscarClientesCorporativosOndeNomeIniciaComOuIdadeMaiorQue(parteDoNome, idadeMinima);

Observe nestes exemplos (que são apenas idéias iniciais) os casos 1 e 5. Ambos possuem a mesma semântica, porém com construção - sintaxe - diferente. Cada um possui virtudes e fraquezas. Enquanto o caso primeiro favorece a clareza da API e a construção de testes testes unitários com melhor legibilidade, o quinto favorece a flexibilidade da refatoração interna sem dependências externas (ex.substituir o parâmetro "nomeIniciaCom" por "nomeTerminaCom" não implica em refatoração de código no resto do programa - embora obviamente a semântica seria afetada seriamente).

A tendência é que o padrão 3PUP siga o primeiro caso, porém isso exigirá ainda amplo estudo.

4.2.5. Outras Informações

- Este padrão favore a formação de nomes semelhantes à linguagem natural, facilitando a tradução de regras de negócio em "português estruturado" e por consequência, código-fonte um-para-um com tais regras.
- Nomes longos devem ser evitados sempre que possível, através da decomposição de estruturas de código o que, por consequência, irá promover a redução da complexidade do código-fonte.
- Para casos que os nomes não podem ser decompostos, é possível utilizar conjunções como "Ou" e "E". Observe o exemplo abaixo:

```
buscarPedidoDeUsoDosClientesCorporativosPorNomeEPorDataDeNascimento(String nome, Date dataDeNascimento)
buscarPedidoDeUsoDosClientesCoporativosPorNomeOuPorDataDeNascimento(String nome, Date dataDeNascimento)
```

    - Neste exemplo, o simples uso de polimorfismo não resolveria esta situação, ou seja um método `buscarPedidoDeUsoDosCliente(String nome, Date dataDeNascimento)` seria invariavelmente ambíguo!
- Observa-se que este padrão de nomes é consistente e permite construções extremamente complexas. Cita-se assim como exemplo o framework Grails, com atenção para a biblioteca Gorm de mapeamento objeto-relacional que utiliza este mesmo padrão para geração em tempo de execução de métodos de pesquisa na camada de persistencia (ex. *Book.findByTitleLikeOrReleaseDateLessThan(...)*).

- A outra vantagem inerente é a relação direta desses nomes com as *features* do sistema, que seguem o padrão AAROO ("Ator" realiza "Ação" para produzir um "Resultado" sobre um "Objeto" para atingir um "Objetivo" de negócio"), tal como exemplo "Gestor (é o ator) buscar (é a ação) pedidos de uso (é o resultado) dos clientes corporativos (é objeto, no caso uma lista) por nome para exportação em PDF (é o objetivo de negócio)", que poderia ser traduzido no método de fachada:

```
List<PedidoDeUso> pedidosDeUso = buscarPedidoDeUsoDosClientesCorporativosPorNome(nomeDoClienteCorporativo);
```

5. Testes

> ⚠ Esta seção não foi finalizada, dado o sistema não ter, segundo o Documento de Visão de Projeto, critérios para construção e execução de testes.
>
> As informações colocadas aqui são oriundas de outros projetos e estão a título de conhecimento apenas.

Testes para validar os critérios de aderência do sistema em relação aos seus requisitos são partes essenciais de um processo de controle de qualidade para desenvolvimento de sistemas.

Neste cenário, o Modelo V é uma referência de mercado, e é utilizado em conjuto com o Processo 3PUP da fábrica de software Redhat de Porto Alegre no seguinte formato:

Camada do Sistema	Teste	Produto testado	Demandante	Criador	Executor	Aprovador	Local	Etapa da *Feature*	Bloqueia
Requisitos do Sistema	Testes de Aceitação	Interfaces de Usuário	Usuário	Projetista	Usuário	Usuário	Ambiente de Homologação	95% - Em Homologação	Aprovar Homologação
Especificações Técnicas	Teste de Sistema (ou Teste Funcional)	Interfaces de Usuário	Projetista	Projetista	Projetista	Projetista	Ambiente de Homologação	95% - Em Homologação	Aprovar Homologação
Arquitetura, Módulos e Pacotes	Teste de Integração	Módulos e Pacotes	Projetista	Arquiteto	Projetista	Projetista	Ambiente de Desenvolvimento	65% - Em Construção	Aprovar Construção
Classes e Métodos	Testes Unitários	Classes e Métodos de classe	Projetista	Desenvolvedor	Outro desenvolvedor	Projetista	IDE	65% - Em Construção	Aprovar Construção

5.1. Testes de Unidade

Os testes de unidade, também conhecidos testes unitários representam o menor elemento testável do sistema. Eles objetivam identificar erros em relação à codificação de classes, métodos e outros recursos mínimos da aplicação.

Os testes de unidade e seus valores de aceitação (ex. esqueletos de classes, métodos e valores) são definidos pelo projetista do módulo do sistema e são codificados pelo desenvolvedor atribuído à feature. E, para execução do teste unitário, necessariamente outro desenvolvedor, que não o codificador da classe/método/recurso deve ser o responsável, a fim de evitar vícios de trabalho.

Os seguintes elementos devem ser verificados em um teste de unidade:

CHECKLIST 4: Checlist para aprovação da construção:

Número	Elemento	Item	Teste a ser realizado	Mecanismo de teste	Status
1	Método público	Teste unitário primário	O método possui um teste unitário associado e este executa com sucesso ?	Biblioteca JUnit	✅ ❌
	Método público	Testes unitários secundários	Se aplicável, o método possui testes unitário secundários associados e todos eles executam com suceso ?	Biblioteca JUnit	✅ ❌

5.2. Testes de Integração

A fazer.

5.3. Testes de Sistema (Funcionais)

A fazer.

5.4. Testes de Aceitação (realiado pelo Usuário)

A fazer.

6. Mais Informações

Nada consta.

7. Referências

Nada consta.

8. Anexos

Arquivo	Modificado
Arquivo PNG eclipse - preferences - java codestyle cleanup.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG eclipse - preferences - java codestyle codetemplates.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG eclipse - preferences - java codestyle formater profile.png	out 24, 2013 by Marcelo | Founder
Arquivo XML modelos - eclipse - preferences - java codestyle cleanup.xml	out 24, 2013 by Marcelo | Founder
Arquivo XML modelos - eclipse - preferences - java codestyle codetemplates.xml	out 24, 2013 by Marcelo | Founder
Arquivo XML modelos - eclipse - preferences - java codestyle formatter profile.xml	out 24, 2013 by Marcelo | Founder
Arquivo modelos - eclipse - preferences - java codestyle organize imports.importorder	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_21.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_26.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_29.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_31.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_33.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_36.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_38.png	out 24, 2013 by Marcelo | Founder
Arquivo PNG Screenshot from 2013-05-29 17_34_40.png	out 24, 2013 by Marcelo | Founder
Arquivo XML checkstyle.xml	jun 05, 2014 by Marcelo | Founder
Arquivo XML checkstyle-supressions.xml	jun 05, 2014 by Marcelo | Founder
Arquivo eclipse-preferences-4.2-original.epf	jun 05, 2014 by Marcelo | Founder
Arquivo XML 3layer - 3pup - modelos - eclipse - preferences - java codestyle cleanup.xml	jun 05, 2014 by Marcelo |

	Founder
Arquivo XML 3layer - 3pup - modelos - eclipse - preferences - java codestyle codetemplates.xml	jun 05, 2014 by Marcelo | Founder
Arquivo XML 3layer - 3pup - modelos - eclipse - preferences - java codestyle formatter profile.xml	jun 05, 2014 by Marcelo | Founder
Arquivo eclipse-preferences-4.2-black.epf	jun 05, 2014 by Marcelo | Founder

Baixar Todos