



Professor: Gilmar Luiz de Borba
2018 – 2

una
Qualidade de Software



CONSIDERAÇÕES INICIAIS

CONTEÚDO PROGRAMÁTICO



1. Referências Principais (discussão)
2. Grandes Erros de Software
3. Um pouco de história
4. Visão Geral Sobre a Qualidade de Software (Conceitos)
5. Visão Geral Sobre Teste de Software (Conceitos)
6. Tipos de Testes de Software
7. Estágios de Testes de Software
8. Dimensões dos Testes de Software
9. Modelos de Testes de Software
10. Planejamento de Testes de Software (e Plano de Testes)

CONTEÚDO PROGRAMÁTICO



11. Tratamento de Erros
12. Refatoração
13. Testes de Unidade.
14. Código Limpo
15. Debugger
16. Junit – *Suite de Testes*
17. Mock

QUESTÕES



1 Conceitos: 1 – 9

2 Testes de Software, Métricas: 10 – 19

3 Modelos de Testes: 20 – 26

4 Planos de Testes: 27 - 33

5 Refatoração: 34 – 40

6 Métricas e Testes Unitários: 41 - 45

7 Código Limpo: 46 - 74

8 TDD e BDD: 75 - 86

PRÁTICAS



- 1 Refatoração
- 2 Testes Unitários
- 3 Suite de Testes
- 5 TDD
- 6 Métricas
- 7 Mock Objects
- 8 Código Limpo

SEMINÁRIO



Plenárias

Artigos sobre o tema Qualidade de Software

Processo

Máximo 30 minutos, mínimo 20 minutos.

Usar o próprio texto ou apresentação PowerPoint

Destacar, assunto, data do artigo e autores

Destacar pontos relevantes do artigo

Observar se o artigo traz uma pergunta no início e a responde no final

No caso de estudo de caso, relatar ...

REFERÊNCIAS PRINCIPAIS



Testes de Software – Leonardo Molinari

Teste de Software – Emerson Rios

Engenharia de Software – Sommerville

Introdução ao Teste de Software – Delamaro

Código Limpo – Robert Martin

Refactoring – Martin Fowler

CONSIDERAÇÕES ADICIONAIS

Distribuição de pontos

Atividades em Sala de Aula

Atividades no Laboratório

...



CONCEITOS

GRANDES ERROS DE SOFTWARE



CASO 1

CD Lançamento do game Rei Leão (Disney)
Natal de 1994 a Disney

CASO 4

Míssil Patriot – Sistema de Defesa dos EUA –
1991

CASO 2

Intel Pentium - "Bug" de Divisão de Ponto Flutuante –
1994

CASO 5

GALAXY NOTE 7

Segundo a própria Samsung, as falhas, que foram responsáveis por tirar o produto de circulação, estavam relacionadas à bateria. Segundo a fabricante sul-coreana, testes realizados revelaram alguns problemas.

CASO 3

NASA - Mars Polar Lander - 1999
Em 12/1999

CASO 6

CIA distribuição de gás aos soviéticos (1982)

CASO 7

Linhos da AT&T “morrem” (1990)

HISTÓRIA

- 
- 1** Pouco compromisso com a qualidade, iniciativas isoladas promoviam melhorias no produto final ... Testes: um mal necessário. (década de 1970)
 - 2** “Moda” era descobrir “BUGS” de software. (década de 1980.)
 - 3** No Brasil foi implantado o PBQP, Programa Brasileiro de Qualidade e Produtividade. (década de 1990.)
 - 4** Realizada pesquisa no Brasil com 282 empresas. (1993).
 - 5** Nova pesquisa de qualidade no setor de software brasileiro junto a 589 empresas de desenvolvimento de software (Ministério da Ciência e Tecnologia). (1997).
 - 6** Enfoque na qualidade de software, garantia durante todo o processo. (2000 =>)

c:\gilmar_borba\AQS\imagens\

CONCEITOS



Auditória

Investigação sistemática das atividades de um campo específico com o objetivo de averiguar se estão de acordo com os padrões estabelecidos. (*na*).

Qualidade

É o encontro com os requerimentos e especificações.
[...]

[...] produto ou serviço que faz o que o usuário precisa. ... “pronto para usar”. (MOLINARI, 2003, Pág:20)



Qualidade de Software

“A totalidade de características de um produto de software que lhe confere a capacidade de satisfazer necessidades explícitas e implícitas”.
(NBR 13596, 1996).

CONCEITOS



Garantia de Qualidade de Software SQA (Software Quality Assurance)

“[...] é a definição de processos e padrões que devem conduzir a produtos de alta qualidade e a introdução de processos de qualidade na fabricação. (SOMMERVILLE, 2011, pág. 455)

c:\gilmar_borba\AQS\imagens\

CONCEITOS



Controle de Qualidade de Software - CQA

O controle de qualidade é a aplicação desses processos de qualidade visando eliminar os produtos que não atingiram o nível de qualidade exigido.”

(SOMMERVILLE, 2011, pág. 455)

CONCEITOS



Garantia de Qualidade de Software

SQA (*Software Quality Assurance*)

Segundo Molinari (2007), um trabalho típico de SQA abrange seis dimensões:

- 1 Métodos e ferramentas de construção
- 2 Revisões formais
- 3 Estratégia de teste
- 4 Controle de documentação e sua história de mudanças
- 5 Procedimentos para garantir a adequação aos padrões de desenvolvimento
- 6 Mecanismos de medição

c:\gilmar_borba\AQS\imagens\

CONCEITOS

Como se compõe a SQA?



c:\prof_gilmar_borba\imagens\auditoria_testes\

CONCEITOS



SQA e CQA – Considerações adicionais

SQA (*Software Quality Assurance*):

Qualidade no nível *estratégico* durante o processo de desenvolvimento.

CQA (*Control Quality Assurance*):

Qualidade no nível *tático* e *operacional*.

CONCEITOS



CMM

(Capability Maturity Model) , é um popular conjunto de padrões.

CMM traz importantes conceitos para a realização de testes de software, uma empresa com um nível mais elevado de CMM facilita a realização dos testes e da qualidade do software.

Segundo o IEEE processo é uma sequência de passos realizados para atingir um objetivo.

CONCEITOS



CMM (Capability Maturity Model , é um popular conjunto de padrões)

- Surgiu nos EUA para avaliar a capacitação de fornecedores de software.
- É baseado no processo mas foca o produto (o produto final).
- Descreve o conjunto de resultados esperados.
- Também leva em consideração as pessoas e métodos, de forma integrada.

c:\gilmar_borba\AQS\imagens\

CONCEITOS



CMM

(Capability Maturity Model , é um popular conjunto de padrões)

- Induz a maturidade e consistência nos processos de software de uma empresa.
- Estabelece cinco níveis de maturidade para avaliar a capacidade de processo de software.
- A partir dos níveis de maturidade o processo permite integração com pessoas e tecnologias.

Nível 1 : inicial → Nível 2 : Gerenciado → Nível 3 : Definido

Nível 4 : Gerenciado Quantitativamente → Nível 5 : Otimizado

CONCEITOS



ISO 9000

ISO é a sigla de International Organization for Standardization (Organização Internacional para Padronização). Trata-se de uma entidade de padronização e normatização, e foi criada em Genebra, na Suíça, em 1947.

Tem como objetivo básico aprovar normas internacionais nos diversos campos do conhecimento humano, como normas técnicas, classificações de países, normas de procedimentos e processos, e etc. No Brasil, a ISO é representada pela ABNT (Associação Brasileira de Normas Técnicas).

CONCEITOS



ISO 9000

- É também (como CMM) um conjunto de padrões relacionados à qualidade.
- Tem como foco prioritário, o processo
- Consiste em considerar: fazer aquilo que diz que faz mas não significa dizer que aquilo é o mais correto.
- Seções da ISO 9000 que lidam com software: ISO 9000-1 e ISO 9000-3.

CONCEITOS

ISO 9000

- ISO 9000-1 se refere produtos em geral (projeto, desenvolvimento e instalação) ...*que inclui projetos de software também.*
- ISO 9000-3 é mais específica para software: desenvolver, fornecer, instalar e manter software.
- Estabelece pontos como: definir e planejar projetos de software; desenvolver e documentar planos de testes, documentar testes.

CONCEITOS



Falha ou BUG

“Um “bug” ou falha é um problema ou qualquer falha no software. Por exemplo um jogo que não funciona corretamente no computador, ou qualquer outra coisa que venha a ser castatrófica e que ninguém percebeu [...].” (MOLINARI, 2003, Pág.20).

Defeito

“É uma não conformidade em relação ao que o software se propõe a fazer, que diz que faz e não faz”. (MOLINARI, 2003, Pág.20).

Outros termos também são usados para se referenciar a falhas de
software ...

c:\gilmar_borba\AQS\imagens\

CONCEITOS



Erro 404 –
File Not Found

Os russos apenas usaram lápis no espaço!

CONCEITOS



Qualidade e Confiabilidade

Qualidade é o grau de excelência, se um produto possui alta qualidade, significa que vai ao encontro das necessidades do consumidor.

A confiabilidade é apenas um aspecto da Qualidade.



Verificação e Validação

Verificação é a confirmação que algo, o software vai ao encontro das especificações. [...] Validação é o processo de confirmação de que o software vai ao encontro dos requerimentos do usuário. (MOLINARI, 2003, Pág:96).

c:\gilmar_borba\AQS\imagens\

CONCEITOS



Segundo Molinari (2003), relacionado aos testes de software a *Precisão* se refere a acertar os mesmos valores independente da situação e a *Acurácia* é acertar os valores desejados.



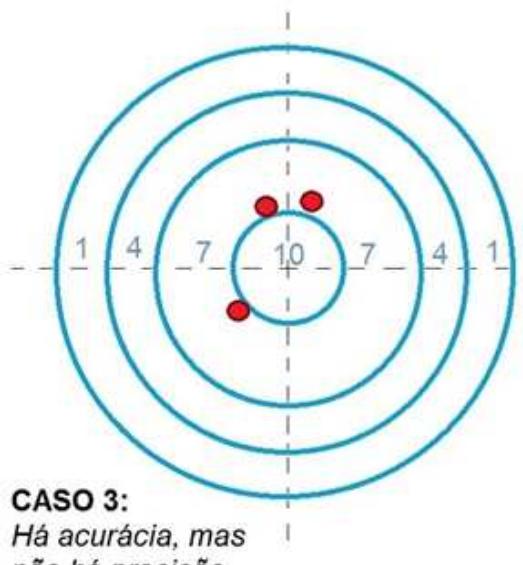
CASO 1:
Nao há nem precisão
nem acurácia



CASO 2:
Há precisão, mas
não há acurácia

CONCEITOS

Segundo Molinari (2003), relacionado aos testes de software a **Precisão** se refere a acertar os mesmos valores independente da situação e a **Acurácia** é acertar os valores desejados.



CONCEITOS

Quais são as situações que fazem um defeito ou bug acontecer?

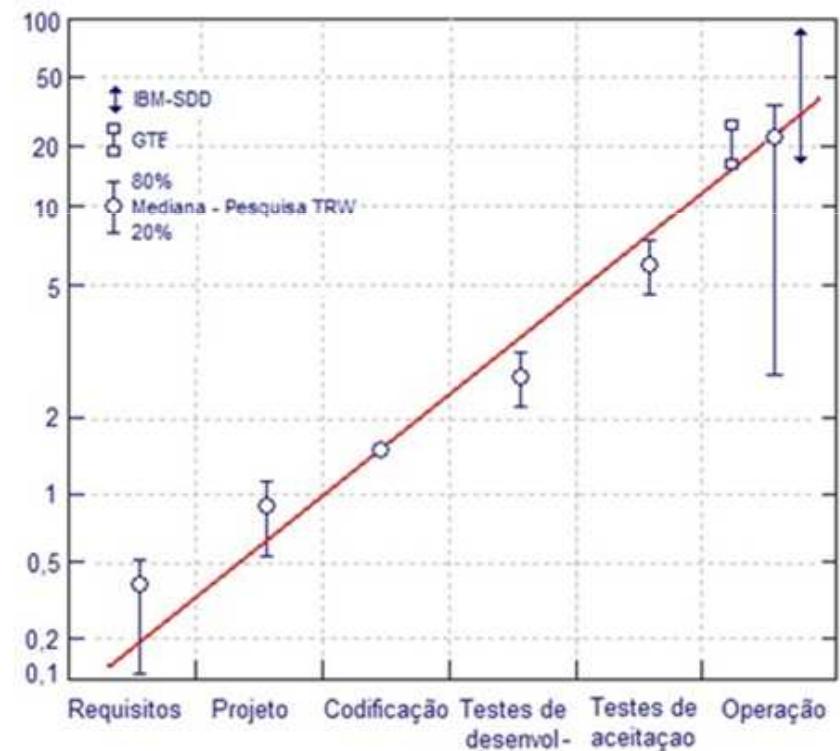
- Software não faz algo que a especificação diz que faz.
- Software faz algo que na especificação diz que não é para fazer.
- Software faz algo que a especificação
não menciona.
- Software é difícil de entender, difícil de usar, lento ou simplesmente
aos olhos do testador “o usuário não gostará”.

CONCEITOS

Regra 10 de Meyer



Regra 10 de Meyer e
os custos para
correção de erros de
Barry Boehm



Fonte: (Gane, 1983, página 7)

QUESTÕES



- (1)** Definir Qualidade de Software. Qual a relação entre a qualidade de software e a Engenharia de Requisitos de software?
- (2)** Explicar o termo Garantia de Qualidade de Software SQA (Software Quality Assurance). Diferenciar SQA e CQA.
- (3)** Como se compõe a SQA? Explicar cada um dos itens ou dimensões que compõem o SQA.
- (4)** Por que falar de CMM quando falamos de qualidade de software?

QUESTÕES



- (5)** Por que falar de ISO quando falamos de qualidade de software?
- (6)** Com relação ao processo e ao produto, diferenciar CMM e ISO.
- (7)** Diferenciar Defeito e Falha.
- (8)** Explicar a regra 10 de Meyers.
- (9)** Descreva alguns aspectos que proporcionam a um software ter qualidade.

c:\gilmar_borba\AQS\imagens\



TESTES DE SOFTWARE

TESTES DE SOFTWARE



O que é um teste de software?

1. Verificar se o software está fazendo o que deveria fazer, de acordo com os requisitos, e não o que não deveria fazer.
2. Processo de executar um programa ou sistema com a intenção de encontrar defeitos (testes negativos). (Meyer, 1979 apud Rios 2013).
3. Qualquer atividade que, a partir da avaliação de um atributo ou capacidade de um programa ou sistema seja possível determinar se ele alcança os resultados desejados. (Hetzl, 1988 apud Rios 2013).

TESTES DE SOFTWARE



Como cobrir todas as possibilidades em um teste?

Na prática, não se pode testar um programa por completo e garantir que ele ficará livre de *bugs*. É quase impossível testar todas as possibilidades de formas alternativas de entrada de dados , bem como testar as diversas possibilidades e condições criadas pela lógica do programador. (RIOS, 2013, pág. 10).

TESTES DE SOFTWARE



Como cobrir todas as possibilidades em um teste?

“[...] para testar a “calculadora do Windows” existem várias combinações. Para testar uma soma simples, podemos testar $1 + 1$ e não dar erro. Podemos testar a soma $1,99 + 3,3439$ e ocorrer um erro. O erro pode aparecer das mais variadas formas. Por isso que testar tudo se torna impossível.”

(MOLINARI, 2003, pág. 94)

TESTES DE SOFTWARE



Tipos de Testes de Software

Testes de Caixa Preta (Black Box)

Visam verificar a funcionalidade e a aderência aos requisitos, em uma ótica externa ou do usuário, sem se basear em qualquer conhecimento do código e da lógica interna do componente testado. (RIOS, 2013. pág.16)

Testes de Caixa Branca (White Box)

Visam avaliar a cláusula de código, a lógica interna do componente codificado, as configurações e outros elementos técnicos. (RIOS, 2013. pág.16)

Testes de Cinza (Grey Box) . . .



Testes Unitários

Estágio mais baixo da escala de testes aplicados nos menores componentes de código criados, visando garantir que estes atendam às especificações em termos de características e funcionalidade. Os testes unitários verificam o funcionamento de um pedaço do sistema ou do software, de forma isolada [...]. (RIOS, 2013. pág.16)

Testes de Integração

São executados em uma combinação de componentes para verificar se eles funcionam juntos de maneira correta, ou, seja, assegurar que as interfaces funcionem e que os dados estão sendo processados de forma correta, conforme as especificações. (RIOS, 2013. pág.16)

ESTÁGIOS DE TESTES DE SOFTWARE



Testes de Sistemas

São realizados pela equipe de testes, visando a execução do sistema como um todo, ou um subsistema (parte do sistema) dentro de um ambiente operacional controlado, para verificar a exatidão e a perfeição na execução de suas funções. Neste estágio de testes, a operação normal do sistema deve ser simulada sendo testadas todas as suas funções de forma mais próxima possível do que ocorrerá no ambiente de produção. (RIOS, 2013. pág.18)

Segundo Rios (2013) no contexto dos testes de Sistemas devem ser realizados os testes:

- de carga,
- de performance,
- de usabilidade,
- de compatibilidade,
- de segurança e recuperação.



Testes de Aceitação

São os testes finais de execução do sistema, realizados pelo usuários, visando verificar se a solução atende aos objetivos do negócio e os seus requisitos, no que diz respeito à funcionalidade e usabilidade, antes da utilização no ambiente de produção. (RIOS, 2013. pág.18)

OUTROS TIPOS DE TESTES DE SOFTWARE

(RIOS, 2013. PÁG.19-22)

1. Aceitação
2. Alfa
3. Back-to-back
4. Beta
5. Carga
6. Compatibilidade
7. Configuração
8. Desempenho (*performance*)
9. Embutidos
10. Estresse (*subtipo de teste de sistema*)
11. Funcionais

12. Instalação
13. Integração
14. Interoperabilidade
15. Qualidade de código
16. Recuperação
17. Recessão
18. Segurança
19. Sistema
20. Sobrevivência (Confiabilidade ou disponibilidade)
21. Unitário
22. Usabilidade

OUTROS TIPOS DE TESTES DE SOFTWARE

(RIOS, 2013. PÁG.19-22)



Agrupamento dos estágios do teste

Segundo Rios (2013) os estágios de testes podem ser agrupados em 3 partes:

Teste Geral	Teste Especializado	Teste de usuário
Unitário	Perfomance	Usabilidade
Integração	Estresse	Beta
Sistema	Segurança	Alfa
Régressão	Recuperação	Aceitação

c:\gilmar_borba\AQS\imagens\

Fonte: Rios (2013) - tabela 3.3 – página 39



O profissional de teste (*tester* ou testador) deve possuir entre outras, as seguintes habilidades:

- Ser um explorador.
- Ser um solucionador de problemas.
- Ser persistente.
- Ser criativo.
- Ser perfeccionista.

O PROFISSIONAL DA ÁREA DE TESTES DE SOFTWARE

A meta de um testador de software é achar “bugs” o mais cedo possível, e fazer com que sejam consertados com certeza. “Bug” esquecido ainda é “bug”. (MOLINARI, 2003, pág. 57)

c:\prof_gilmar_borba\imagens\auditoria_testes

Os testes habilitam as “-idades”.

(MARTIN, [et al], 2011, pág. 124)

c:\prof_gilmar_borba\imagens\auditoria_testes

Usabilidade Confiabilidade Portabilidade Interoperabilidade
Privacidade Facilidade de uso Velocidade Manutenibilidade

Dimensões do teste

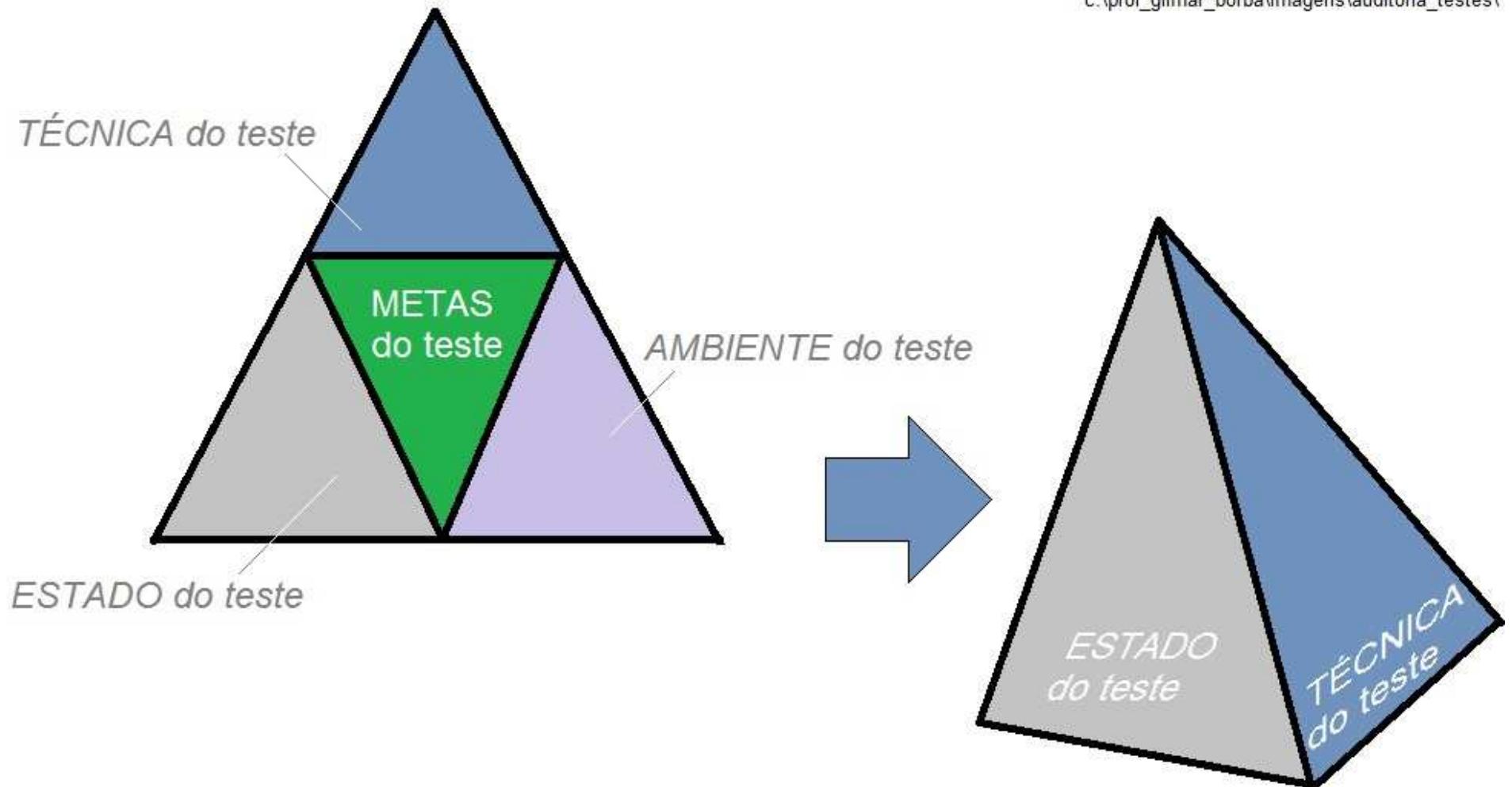
Para efetivamente identificar e eliminar os *bugs* devemos antes planejar os testes, planejamento é fundamental. Segundo vários autores, existindo variações entre o ponto de vista de cada um, existem várias dimensões no que se refere aos testes de software, são elas:

- 1** *O estado do teste, ou seja, o momento em que ele é aplicado.*
- 2** *A técnica usada para realizar o teste.*
- 3** *As metas do teste, o que deverá ser testado.*
- 4** *Onde será realizado o teste.*

DIMENSÕES DOS TESTES DE SOFTWARE

Dimensões do teste

c:\prof_gilmar_borba\imagens\auditoria_testes\





MÉTRICAS

Métricas

Métricas são usadas como indicadores de qualidade, ou seja, atestam se os objetivos definidos para um método, um módulo, ou para o projeto foram alcançados.

Por que medir?

1. Para melhorar a qualidade do produto.
2. Para possibilitar um melhor retorno de investimento.
3. Podem ser medidos: o processo, o teste em si, a automação do teste, etc.



Exemplos

1. % de defeitos encontrados.
2. % de defeitos resolvidos.
3. Métricas de código.
4. Métricas subjetivas.
5. Métrica de conferência e análise de teste.
1. Benchmark de performance. etc.

MÉTRICAS DE SOFTWARE



Atributos relacionados às métricas (MOLINARI, 2003, Pág.. 62)

1. Manutenibilidade.
2. Eficácia.
3. Confiabilidade.
4. Flexibilidade.
5. Usabilidade.
6. Robustez.
7. Portabilidade.



Objetivo das métricas

Auxiliar (dar suporte) a uma melhor gerência dos projetos de software.

Aumentar a qualidade dos artefatos de software (produto final).

Garantir que os usuários recebam produtos dentro das especificações e dos requerimentos. (lembra especificação é diferente de requerimento).

Quais são as informações apresentadas nas métricas?

Tamanho do software.

Complexidade.

Quantidade de defeitos do software.

Tipos de variáveis usadas.

Coesão dos métodos.

etc.

Obs: essas informações são normalmente indicativos da qualidade do software.

MÉTRICAS



Quais são as informações apresentadas nas métricas?

Tamanho do software.

Complexidade.

Quantidade de defeitos do
software.

Tipos de variáveis usadas.

Coesão dos métodos.

etc.

Obs: essas informações são normalmente indicativos da qualidade do software.



Complexidade ciclomática (complexidade condicional)

É uma métrica que indica a complexidade de um programa de computador.

Mede o número de caminhos independentes de um programa/código de software.

É baseada a partir de um grafo de fluxo de controle. Os nós correspondem a grupos indivisíveis de comandos.

É aplicada a módulos de um programa, métodos e classes.

Foi desenvolvida por Thomas J. McCabe em 1976.



Complexidade ciclomática (complexidade condicional)

“Sabemos que um dos grandes vilões na hora de mantermos um sistema legado são os trechos de código difíceis ou complicados de entender. Todo programador já se deparou com [...] códigos com péssimos nomes de variáveis ou muitas linhas de código em um único método, sem contar o excesso de responsabilidade. [...] um dos maiores responsáveis pela dificuldade em entender o que um código faz é o simples if (*na: case, switch, while ...*)”

Com essa simples instrução, o desenvolvedor faz com que o mesmo método possa responder de duas, três, quatro, N, ..., maneiras diferentes, de acordo com certas condições! E pior, o número de caminhos diferentes pode crescer muito rápido!..”



Complexidade ciclomática (complexidade condicional)

“Devemos, quase sempre, manter esse número de caminhos **o menor possível.**”

“Não há um número ideal para a complexidade de um método. Mas fato é que uma complexidade ciclomática de 20 ou 30 é demais e esse método deve ser reescrito. Mesmo quebrando-o em dois, é interessante observar a complexidade dos métodos de uma classe em conjunto.” Maurício AnicheMsc. Ciência da Computação – USP

Local: <http://blog.caelum.com.br/medindo-a-complexidade-do-seu-codigo/>
Acessado em: agosto/2016

Complexidade ciclomática ReservaPassagemArea (Sem refatorar)

Screenshot of the Eclipse IDE Metrics view for the file `TestePilha.java`. The table shows various code metrics with their total values, means, standard deviations, maximums, and the resource causing the maximum value. A large black arrow points to the McCabe Cyclomatic Complexity row.

Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum	Method
> Number of Parameters (avg/max per method)		1	0	1	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	main
> Number of Static Attributes (avg/max per type)	0	0	0	0	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> Specialization Index (avg/max per type)		0	0	0	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
Number of Classes	2					
> Number of Attributes (avg/max per type)	4	2	2	4	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> Number of Static Methods (avg/max per type)	1	0,5	0,5	1	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
Number of Interfaces	0					
Total Lines of Code	140					
> Weighted methods per Class (avg/max per typ	24	12	12	24	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> Number of Methods (avg/max per type)	0	0	0	0	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> Depth of Inheritance Tree (avg/max per type)		1	0	1	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> McCabe Cyclomatic Complexity (avg/max per	24	0	24	24	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	main
> Nested Block Depth (avg/max per method)	6	0	6	6	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	main
> Lack of Cohesion of Methods (avg/max per typ		0	0	0	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> Method Lines of Code (avg/max per method)	125	125	0	125	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	main
> Number of Overridden Methods (avg/max per	0	0	0	0	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	
> Number of Children (avg/max per type)	0	0	0	0	/PrjPilhaSemRefatorar/src/pkgPilha/TestePilha.java	

Complexidade ciclomática ReservaPassagemArea (refatorado)

Screenshot of the Eclipse IDE Metrics view showing cyclomatic complexity metrics for the 'ReservaPassagemArea' class (refactored). A black arrow points to the 'McCabe Cyclomatic Complexity' row.

Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum	Method
> Number of Parameters (avg/max per method)		0,889	0,314	1	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	main
> Number of Static Attributes (avg/max per type)	0	0	0	0	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Efferent Coupling (avg/max per packageFragment)		0	0	0	/PrjPilhaRefatorado/src/pkgPilha	
> Specialization Index (avg/max per type)		0	0	0	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Number of Classes (avg/max per packageFragment)	2	2	0	2	/PrjPilhaRefatorado/src/pkgPilha	
> Number of Attributes (avg/max per type)	4	2	2	4	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Abstractness (avg/max per packageFragment)		0	0	0	/PrjPilhaRefatorado/src/pkgPilha	
> Normalized Distance (avg/max per packageFragment)		0	0	0	/PrjPilhaRefatorado/src/pkgPilha	
> Number of Static Methods (avg/max per type)	9	4,5	4,5	9	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/PrjPilhaRefatorado/src/pkgPilha	
> Total Lines of Code	165					
> Weighted methods per Class (avg/max per type)	32	16	16	32	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Number of Methods (avg/max per type)	0	0	0	0	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Depth of Inheritance Tree (avg/max per type)		1	0	1	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Number of Packages	1					
> Instability (avg/max per packageFragment)		1	0	1	/PrjPilhaRefatorado/src/pkgPilha	
> McCabe Cyclomatic Complexity (avg/max per method)	3,556	3,201		12	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	main
> Nested Block Depth (avg/max per method)	2,333	0,943		4	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	filtrar
> Lack of Cohesion of Methods (avg/max per type)		0	0	0	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Method Lines of Code (avg/max per method)	137	15,222	10,581	36	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	main
> Number of Overridden Methods (avg/max per type)	0	0	0	0	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	
> Average Cyclomatic Complexity	2	2	2	2	/PrjPilhaRefatorado/src/pkgPilha/TestePilha.java	



Instalando Plugin do Metrics no Eclipse

No menu **HELP**

Escolher: Install New Software

Clicar sobre o botão **ADD** (na caixa de texto Working With)

Na caixa de diálogo: **Add Repository**

Informar *Metrics* (na opção name)

Informar: <http://metrics.sourceforge.net/update> (na opção location)

<http://metrics2.sourceforge.net/update> (na opção location)

Acionar **OK**



Instalando plugin do METRICS no Eclipse Galileo

Acionar Select All

Acionar NEXT (Ainda na caixa de diálogo Available Software)

(se a opção NEXT estiver desabilitada selecione e tire a seleção de Hide items that are already installed)

Acionar NEXT (Na caixa de detalhe Install Details)

Observe que Metrics plugin for Eclipse version 1.3.6 foi reconhecido

Aceitar os termos de uso da licença (Na caixas de diálogo Review Licenses)

Acionar FINISH

... O Plugin será instalado em alguns segundos ... talvez um minuto ...

Continuar (OK) a instalação caso ocorra a mensagem Warning!

Acione YES para reiniciar o Eclipse.

MÉTRICAS



Para Visualizar o Plugin o *METRICS* no Projeto

Acesse: Menu Window -> Show View -> Other

Expanda o ítem da árvore: Metrics

Selecione: Metrics View

Clique: **Ok**

Irá aparecer uma janela no rodapé da IDE

Para Ativar o Plugin do *METRICS* no Projeto

Clique com o botão direito do mouse sobre o nome do projeto no Explorer do Eclipse

Clique: Properties

Clique no Metrics na árvore de propriedades

Selecione o checkbox: **Enable Metrics**

QUESTÕES

- (10)** Estabeleça uma relação entre as disciplinas: Engenharia de Requisitos e Testes de Software.
- (11)** Qual(is) pode(m) ser a(s) consequência(s) de um site com retornos recorrentes para um link: "404 File or Directory no found ..."



QUESTÕES



- (12)** Descrever as principais diferenças entre os testes de Caixa Preta e Caixa Branca.
- (13)** Qual é a diferença entre os testes: ALFA e BETA?
- (14)** Qual é a diferença entre o teste de Carga e teste de Stress?

QUESTÕES



(15) Diferenciar Testes de Interoperabilidade e Testes de Integração.

(16) Com relação às habilidades do profissional da área de testes de software, associar as duas colunas:

- | | |
|---|---|
| (a) Ser um explorador.
(b) Ser um solucionador de problemas.
(c) Ser persistente.
(d) Ser criativo.
(e) Ser perfeccionista. | (1) deve adorar puzzles ...
(2) teste não tem meio termo, os detalhes são importantes.
(3) não ter medo de se aventurar em situações diversas.
(4) testar nem sempre é óbvio, às vezes não há um padrão.
(5) testar até estar completamente satisfeito. |
|---|---|

QUESTÕES



(17) Comente a frase de
Robert Martin (apud Fowler):
"Os testes habilitam as "-idades".

(18) O que são métricas? Por que medir o teste e a
automação do teste?

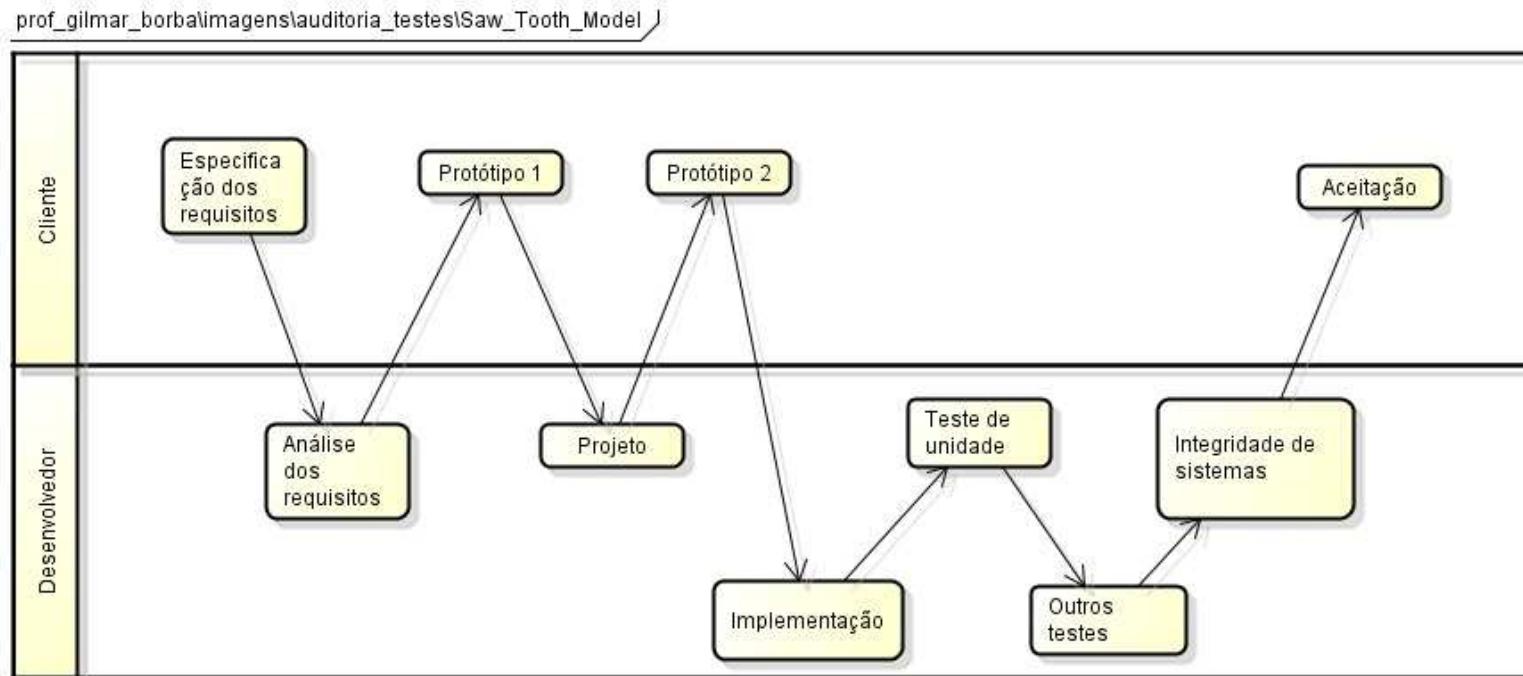
(19) Forneça 3 exemplos de métricas. Como construir uma
métrica



MODELOS DE TESTES

Saw Tooth

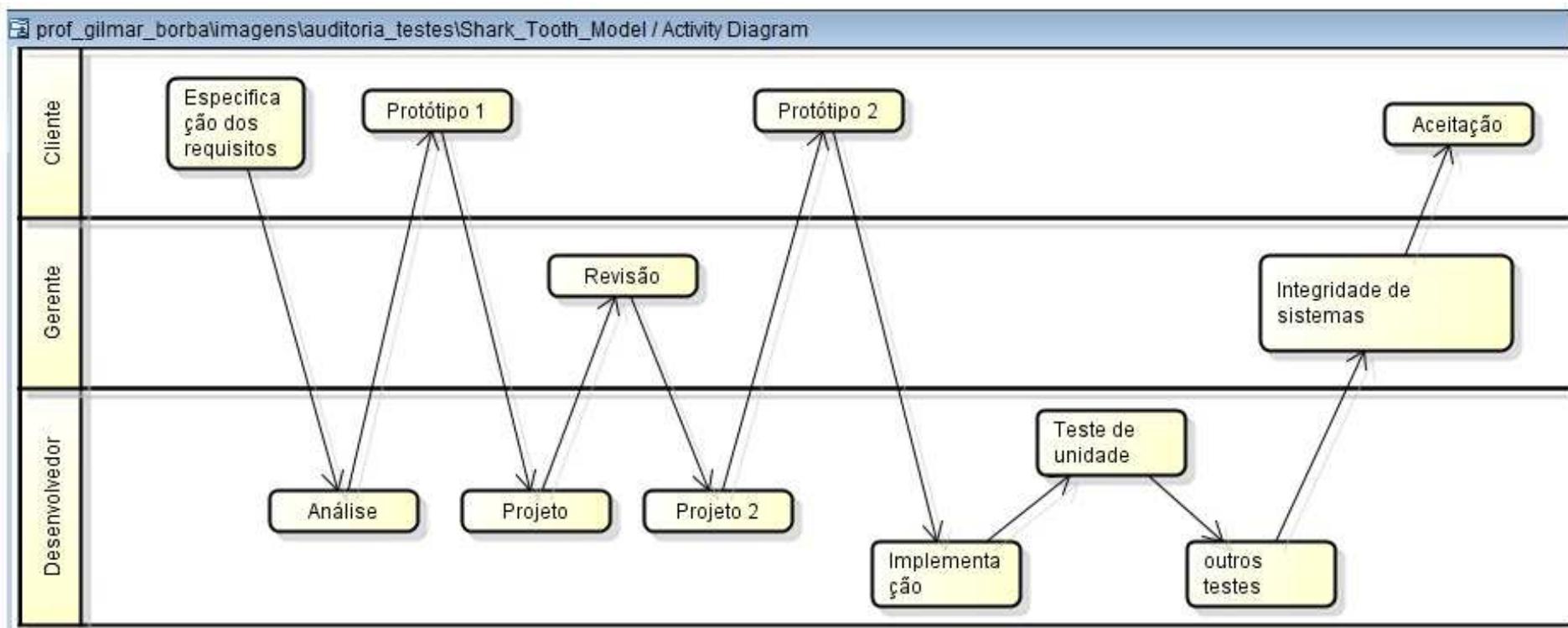
1. Mostra a interação entre usuário e desenvolvedor.
2. Faz o que foi pedido.
3. Somente há dois papéis: cliente e desenvolvedor.
4. O desenvolvedor faz a integração.



MODELOS DE TESTES DE SOFTWARE

Shark Tooth

- 1 . Mostra a interação entre usuário e desenvolvedor.
- 2 . Há também a participação de um novo ator: o gerente (revisor).
- 3 . O gerente faz a integração e a previsão.





Espiral

1. Cíclico, faz uso da prototipação.
2. Os testes estão explicitados (análise de risco, validação, requerimentos, desenvolvimento etc.)
3. O plano de teste pode ser feito após o projeto do sistema.
4. Segue a linha de código.
5. Não identifica atividades associadas com remoção de defeitos.

c:\gilmar_borba\AQS\imagens\

MODELOS DE TESTES DE SOFTWARE

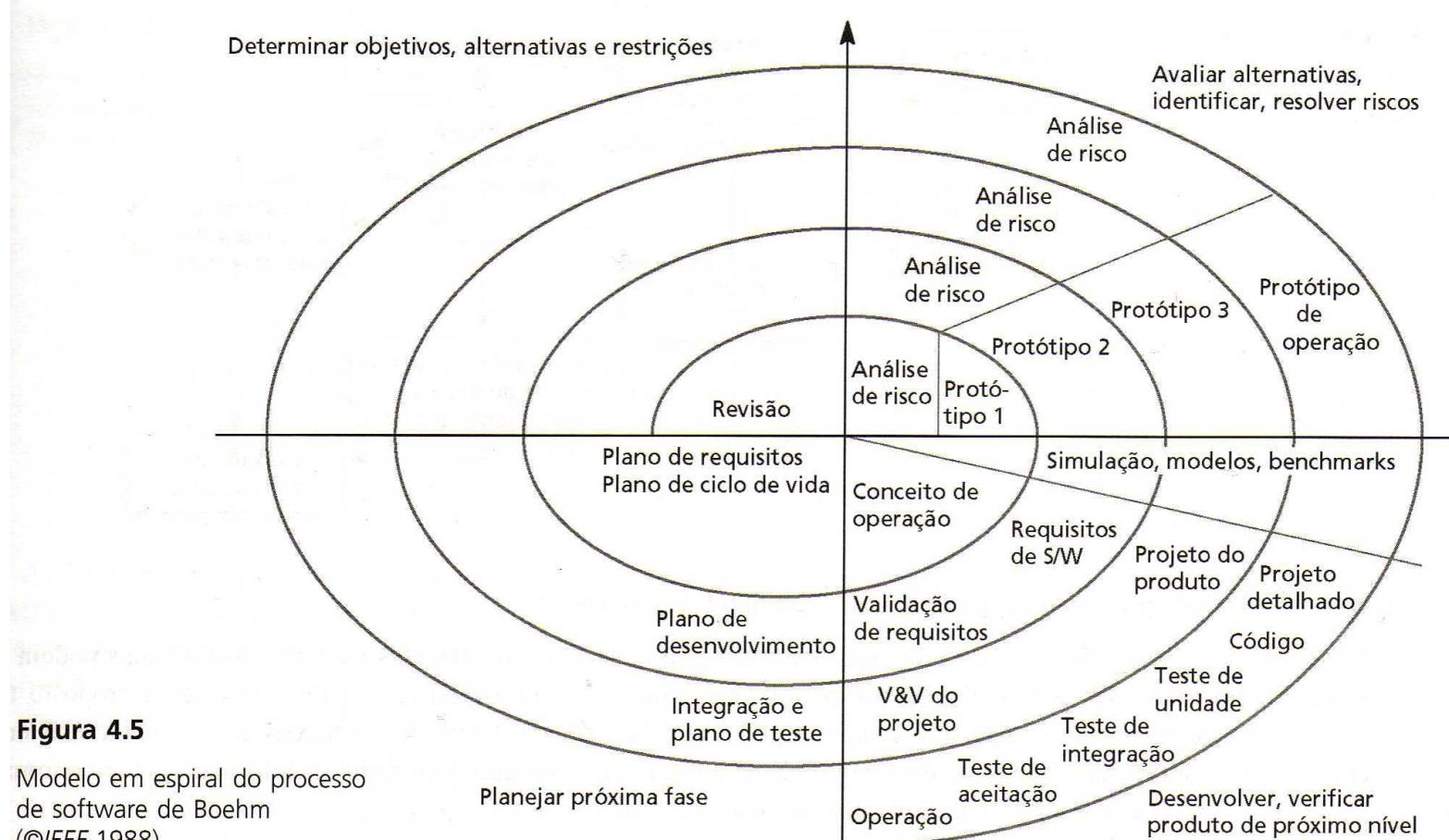


Figura 4.5

Modelo em espiral do processo de software de Boehm
(©IEEE, 1988).



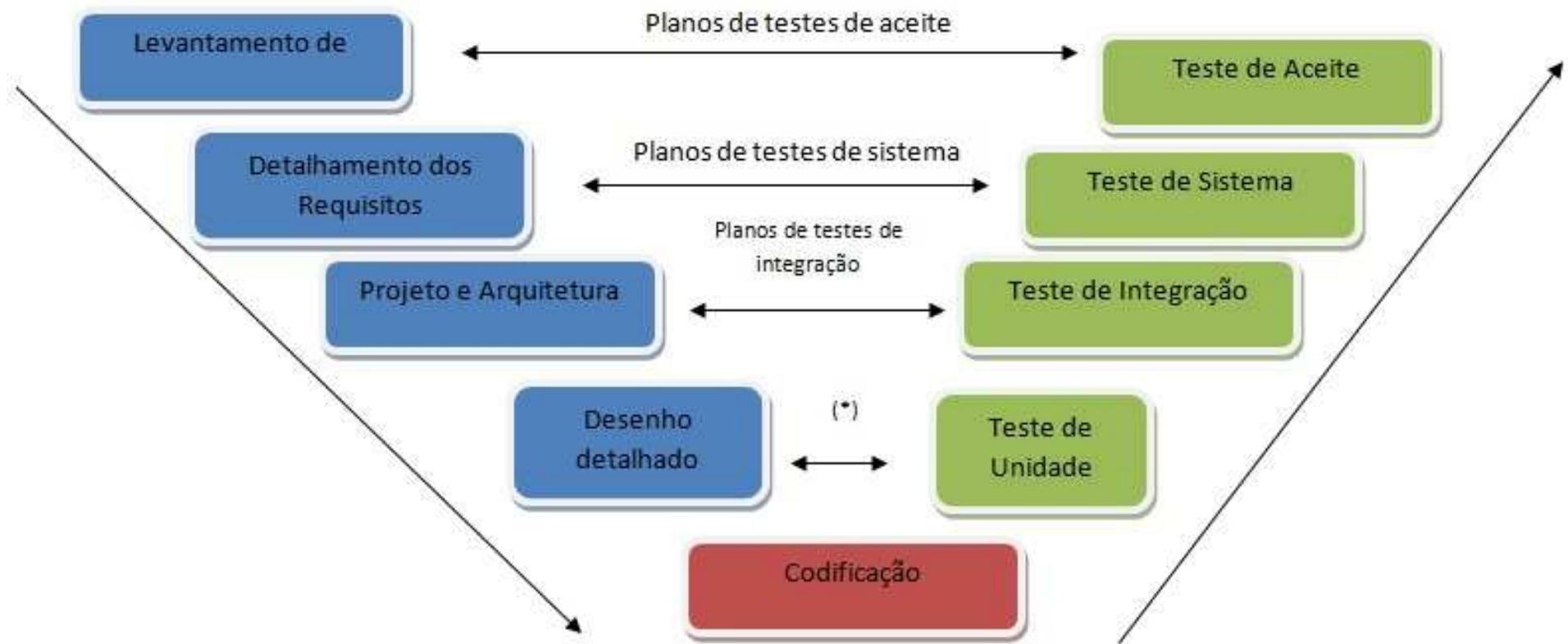
Modelos de Testes de Software - Modelo V

É um dos mais populares. V, significa verificação e validação.

Modelo V

- As fases de teste sempre estão associadas às fases de desenvolvimento.
- No lado esquerdo está a fase construtiva, do lado direito a fase destrutiva.
- Não identifica atividades associadas com remoção de defeitos.
- Testes de regressão não são aplicados em nenhum lugar deste modelo.

MODELOS DE TESTES DE SOFTWARE



c:\prof_gilmar_borba\imagens\auditoria_testes\



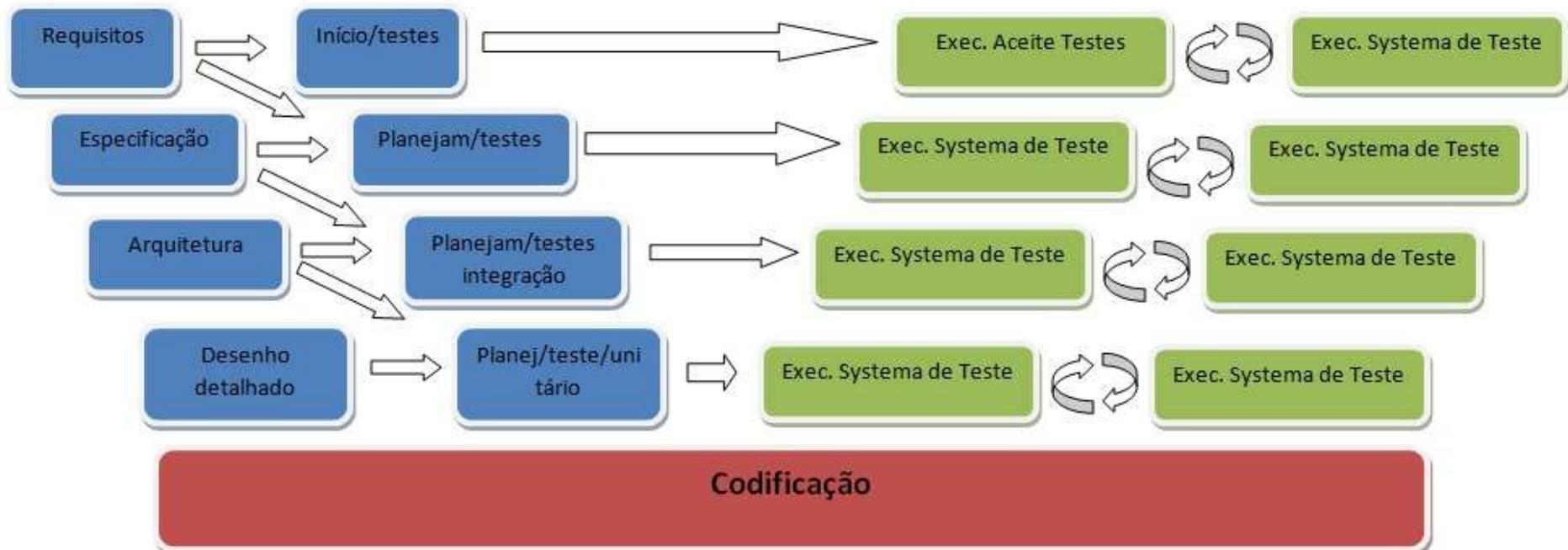
Modelo W

1. É baseado no modelo em V.
2. Para cada fase temos um teste correspondente de forma a eliminar os *bugs*.
3. Este modelo minimiza os custos pois há um número maior de testes.
4. A idéia é que para cada artefato gerado por uma atividade deve ter uma atividade de teste associada.
5. É usado com maior frequência em projetos baseados em risco para e-business relacionados a: usabilidade, desempenho, segurança, disponibilidade e funcionalidade.

MODELOS DE TESTES DE SOFTWARE



Modelo W



c:\prof_gilmar_borba\imagens\auditoria_testes\



Modelo “Butterfly”

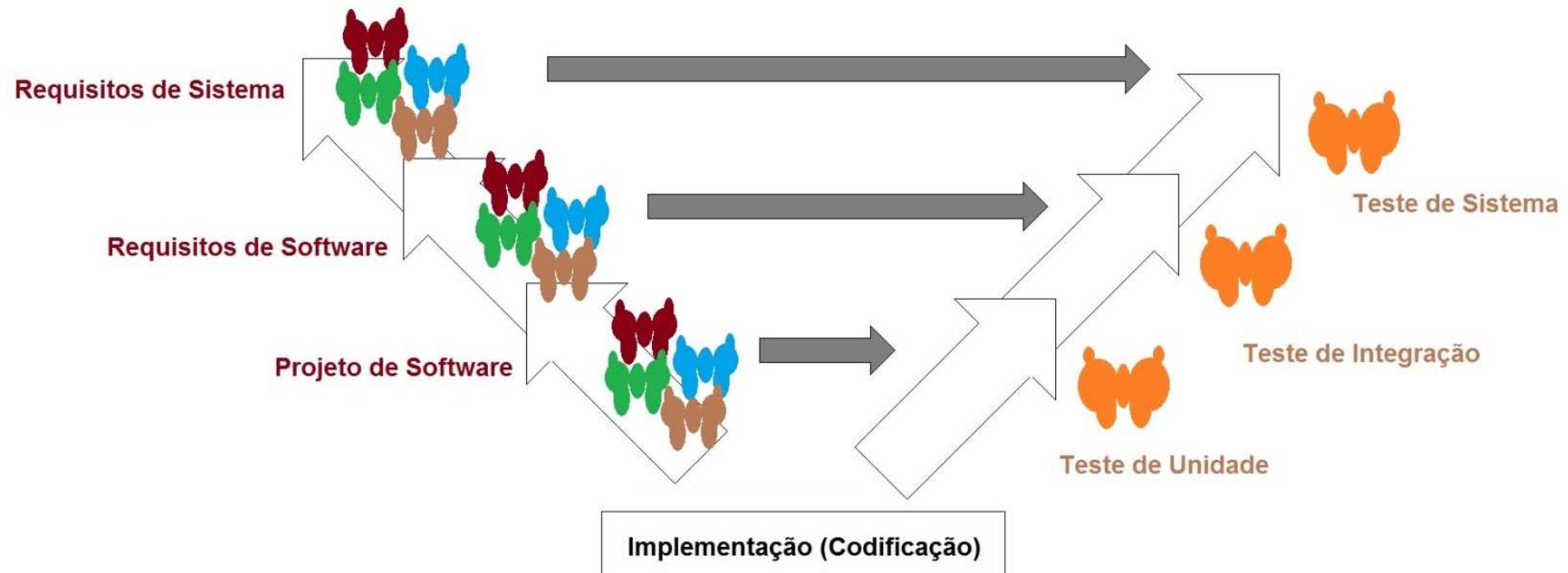
1. É também baseado no modelo em V.
2. Cada sub-atividade possui na prática pequenas fases de análise , projeto, produto e execução.
3. Cada subfase representa um componente que tem vida curta (*uma borboleta*).
4. Cada micro atividade possui, na prática pequenas fases de análise, projeto e execução (todas relativas a testes).
5. A asa esquerda da borboleta seria a etapa de análise (levantamentos iniciais), a parte direita seria a própria análise e a parte central seria a execução dos testes propriamente ditos.

c:\gilmar_borba\AQS\imagens\

MODELOS DE TESTES DE SOFTWARE



Modelo “Butterfly”

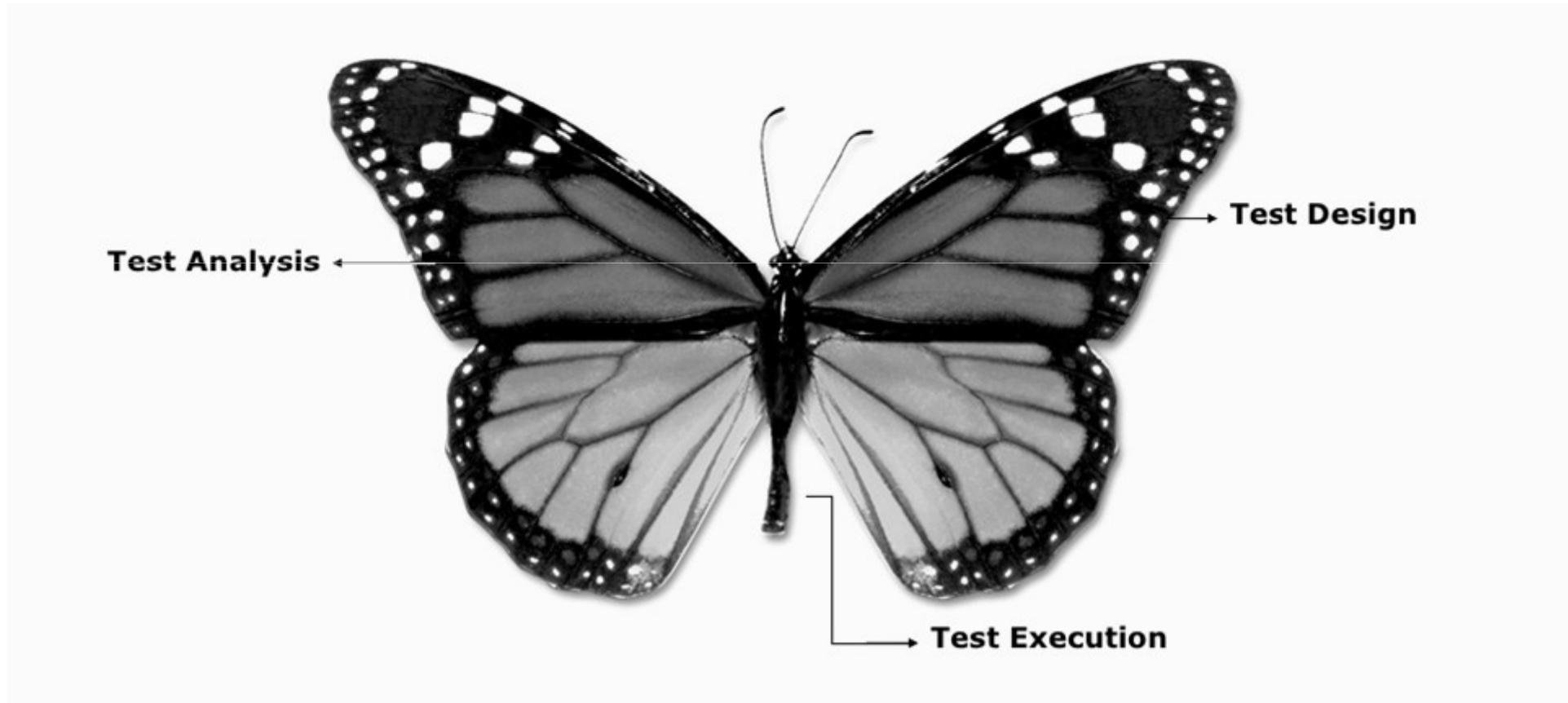


c:\prof_gilmar_borba\imagens\auditoria_testes\

MODELOS DE TESTES DE SOFTWARE



Modelo “Butterfly”

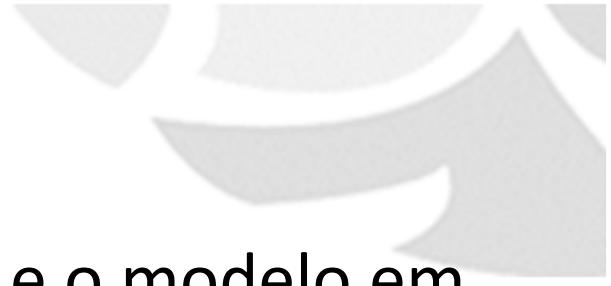


QUESTÕES



- (20)** Qual é a principal diferença entre os modelos de teste: Saw Tooth e Shark tooth.
- (21)** Há alguma desvantagem no modelo Shark tooth em relação ao modelo Saw Tooth?
- (22)** Qual é o modelo mais popular de testes de software?
- (23)** Descreva as características do modelo em "V".

QUESTÕES



- (24)** Comente sobre os testes de regressão e o modelo em "V".
- (25)** O fato de que no lado esquerdo está a fase construtiva e do lado direito a fase destrutiva, há alguma implicação nesse formato?
- (26)** Compare os modelos "W" e "Butterfly" de testes de software.

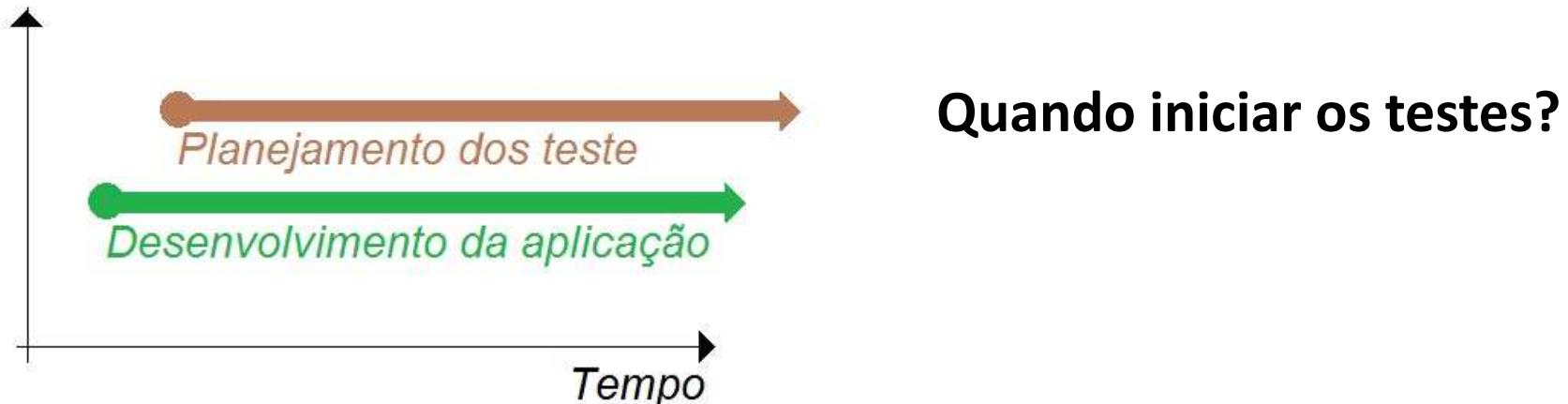
PLANEJAMENTO DE TESTES DE SOFTWARE

Planejamento de Testes

É o processo de preparação dos testes, observando quando iniciar, quando realizar e quando finalizar o teste.

Plano de Testes

É o artefato produzido no processo de planejamento de teste, é o resultado prático desse planejamento.



PLANEJAMENTO DE TESTES DE SOFTWARE

FACILITA	TEM COMO ENTRAVES
A organização das atividades de teste	O prazo (um limitador do escopo)
A objetividade dos testes	O perfil do elaborador do teste
A previsibilidade	O requisitos que podem ser frágeis
O acompanhamento dos testes	

c:\prof_gilmar_borba\imagens\auditoria_testes\

PLANEJAMENTO DE TESTES DE SOFTWARE

Segundo Molinari (2003) há alguns conceitos dentro do contexto do planejamento de testes:

Planejamento de Testes

Requerimentos de testes

Casos de Testes

Plano de Testes

PLANEJAMENTO DE TESTES DE SOFTWARE

Segundo Molinari (2003) há alguns conceitos dentro do contexto do planejamento de testes:

Procedimentos de Testes

Script de Teste

Ponto de verificação



Segundo Molinari (2003) os principais artefatos produzidos durante o processo de planejamento de testes são:

Test People

Test Manager, Test Designer, Test Implementor, Test support.

Test Material

Test Results

Test Plan Document

c:\gilmar_borba\AQS\imagens

QUESTÕES



- (27)** Diferenciar Planejamento de Testes e Plano de Testes.
- (28)** Quando deve ser iniciado o Planejamento de Testes?
- (29)** Descreva três facilitadores do Planejamento de Testes?
- (30)** Descreva três entraves do Planejamento de Testes?
- (31)** Diferencie Requerimento de Teste e Casos de Teste.

QUESTÕES



(32) Relacione as duas colunas

- | | |
|--|---|
| <p>(1) Test People
(2) Test Material
(3) Test Results
(4) Test Plan Document</p> | <p>(A) Todo o histórico e planejamento dos testes, incluindo o resultados.
(B) É o documento oficial que contém tudo: quem faz; o que faz, quais os requerimentos etc.
(C) É o que está usando. Aplicação a ser testada, os dados disponibilizados etc.
(D) Quem faz o teste, quem faz o que.</p> |
|--|---|

(33) Qual é o papel do Test Support?

O PLANO DE TESTES



A ideia básica trazida nas matrizes de teste é expor as entradas obrigatórias e opcionais, descobrir as variáveis que serão submetidas aos testes e traçar os cenários de testes com os correspondentes resultados esperados.

O PLANO DE TESTES



As vantagens deste método de expor os casos de teste são:

- a) É uma estratégia mais eficaz e precisa.
- b) Os testes redundantes tornam-se mais evidentes.
- c) Os testes redundantes podem ser eliminados antes de criar os scripts.
- d) A cobertura é menos propensa a ter problemas por defeitos e passar.
- e) Os resultados esperados são claramente definidos.

O PLANO DE TESTES

Fonte: Molinari (2003)

O PLANO DE TESTES



Para utilizar a matriz:

1. Identifique todas as entradas possíveis relacionados com o código que está sendo testado.
2. Identificar as possibilidades de cada entrada.
3. Assegurar que todos os insumos necessários são contabilizados (versus entradas opcionais)
4. Uma vez que as entradas foram definidas, coloque um X nas colunas numeradas para identificar o que será usado para que passe teste.
5. Em seguida, identificar o resultado esperado para essa combinação..

O PLANO DE TESTES

* Txn Type	RK	x	x	x	x	x	x	x			
* Delay (sec)	0	x									
	5		x						x		
	55			x							
	555				x						
	999					x					
	BLANK						x				
* INI setting of "Allowed="	VERDADEIRO	x	x	x	x	x	x				
	FALSO							x			
* inactivity timeout (milliseconds)	10000	x				x	x	x			
	15000		x								
	20000			x							
	60000				x						
EXPECTED RESULT		immed dial, 10 sec timeout	5 sec dial delay, 15 sec timeout	55 sec dial, 20 sec timeout	555 sec dial delay, 1 min timeout	999 sec dial, 10 sec timeout	immed dial, 10 sec timeout	No predial, 10 sec timeout after txn			

c:\prof_gilmar_borba\imagens\auditoria_testes\



REFATORAÇÃO

REFATORAÇÃO



Em geral, uma refatoração é tão simples que parece que não vai ajudar muito.

Mas quando se juntam 50 refatorações, bem escolhidas, em seqüência, o código melhora radicalmente. (Chaim apud Martin, 2007)

c:\gilmar_borba\AQS\imagens\

- É uma técnica de design muito usada.
- É Parte integrante do TDD (nos métodos ágeis)

(Para cada ciclo verifique se há refatorações a serem feitas).

REFATORAÇÃO

A refatoração não deve ser usada com o objetivo de implementar novas funcionalidades do código.

[...] o processo de mudar o design de seu código sem mudar o seu comportamento – o que ele faz permanece o mesmo mas como ele faz, muda.” (*SHORE, 2008, pág. 315*)

“Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.”

Martin Fowler.

REFATORAÇÃO



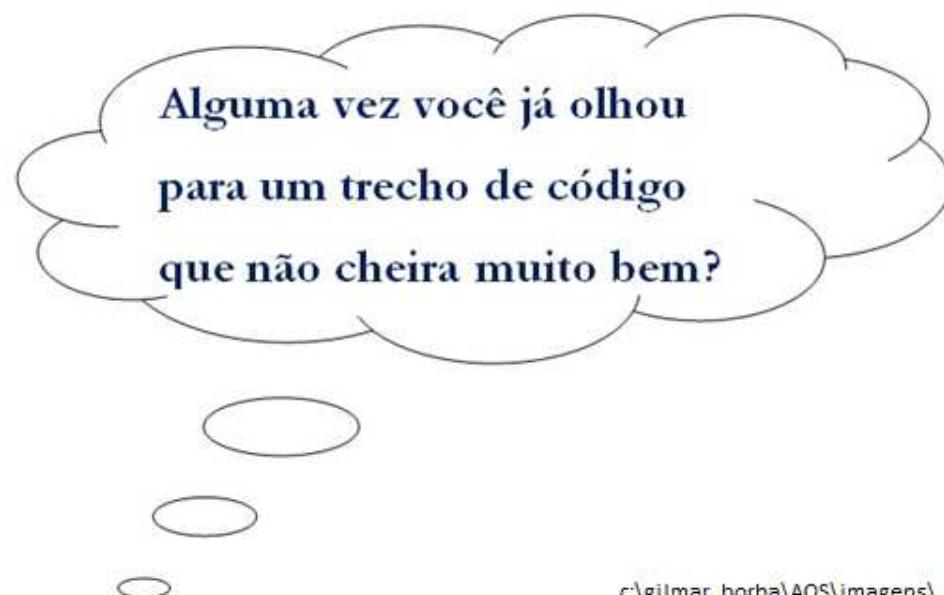
Limpar o código faz parte do processo de desenvolvimento, se a sujeira for deixada no código ficará mais difícil (\$) corrigi-lo depois.

TECHNICAL DEBT

REFATORAÇÃO

Limpar o código faz parte do processo de desenvolvimento, se a sujeira for deixada no código ficará mais difícil corrigi-lo depois.

BAD SMELLS



c:\gilmar_borba\AQS\imagens\

REFATORAÇÃO

Alguns exemplos de odores:

- Métodos longos
- Métodos que fazem muita coisa (*shotgun surgery*)
- Métodos super dimensionados
- Classes que dependem de detalhes de outras
(Inappropriate Intimacy)
- Classes invejosas (*feature envy*) uma classe que usa muito as funcionalidades de outra.

c:\gilmar_borba\AQS\imagens\

REFATORAÇÃO



"O desempenho pode piorar, otimizar não é o objetivo da refatoração". (Fowler, 2004)

Por que refatorar?

- Para melhorar o projeto do software.
- Para tornar o software mais fácil de entender.
- Para ajudar a encontrar falhas.
- Para auxiliar nas atividades de programação (rapidez).

"O desempenho pode piorar, otimizar não é o objetivo da refatoração". (Fowler, 2004)

REFATORAÇÃO



Como nos padrões de projeto, a refatoração é apresentada em um formato específico nos catálogos.

- 1** Nome.
- 2** Resumo (necessidade ...).
- 3** O que ela faz (objetivo).
- 4** Motivação (quando usá-la e quando não usá-la).
- 5** Mecânica (passo a passo).
- 6** Exemplo.

c:\gilmar_borba\AQS\imagens\



Refatoração De Mudança De Nomes *(Rename)*

(Métodos, Classes, Campos, Variáveis etc.)

Usado quando é necessário alterar um elemento do código de programação fazendo com que este tenha um significado melhor no contexto da aplicação.

Passos:

1. Alterar o nome do elemento
2. Alterar as referências
3. Verificar os possíveis conflitos



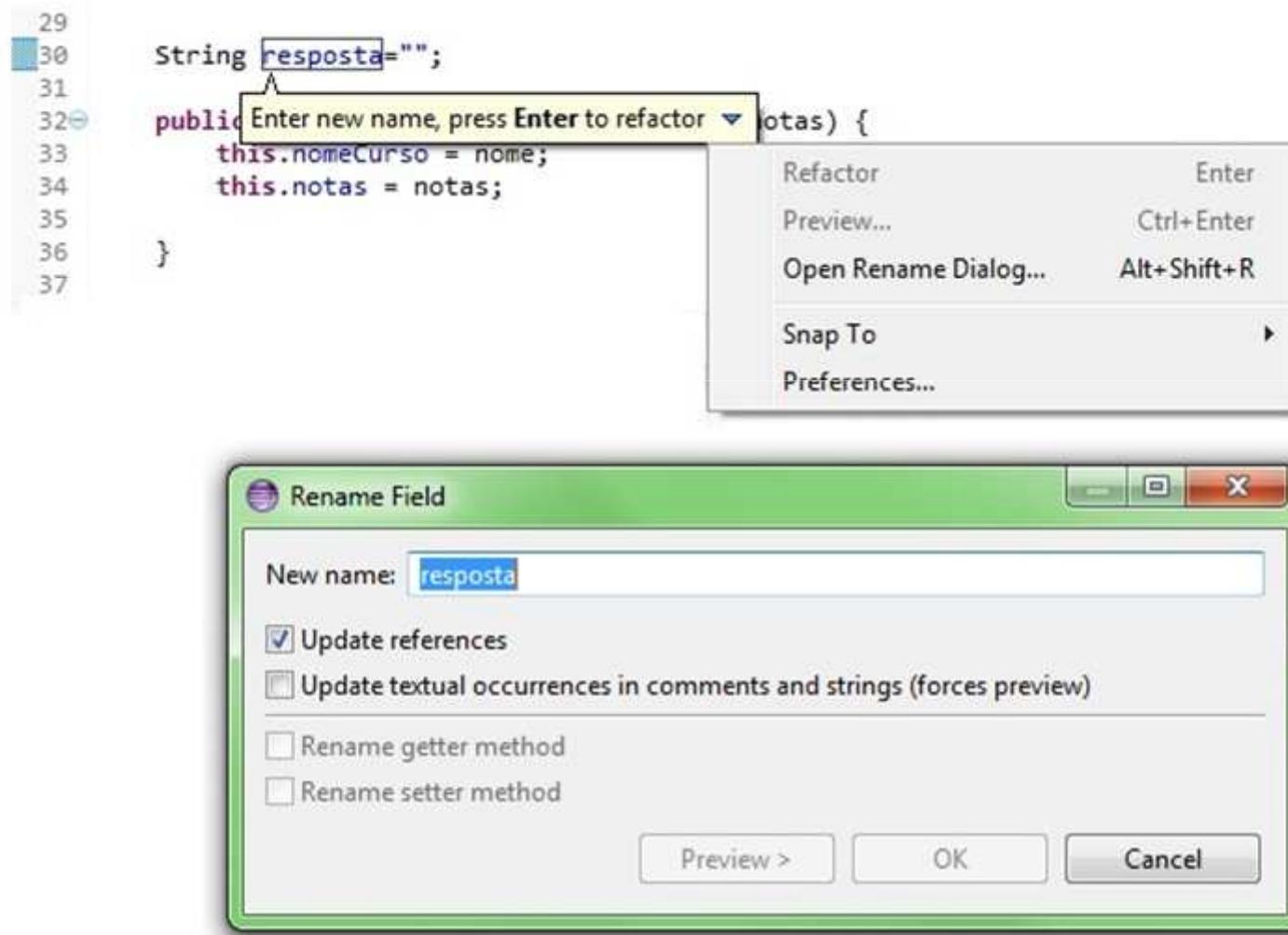
Refatoração De Mudança De Nomes *(Rename)*

No ambiente Eclipse

- É a refatoração mais usada no Eclipse.
- Aplicável a variáveis, classes, métodos, pacotes, pastas etc.
- Após renomear, todas as referências deverão ser renomeadas.
- No Eclipse, Alt+Shift+R: invoca a refatoração “Rename”.

REFATORAÇÃO

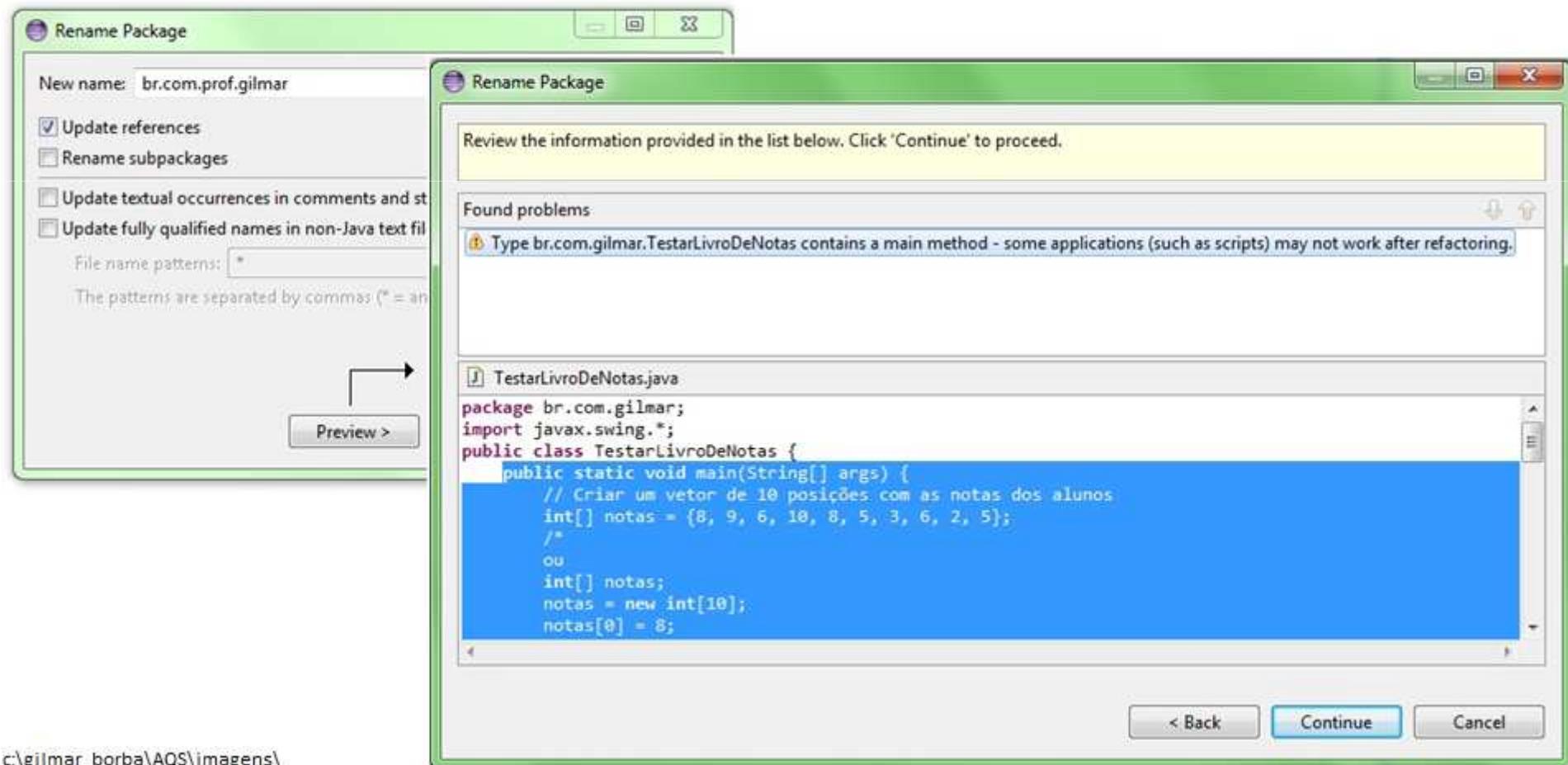
Refatoração De Mudança De Nomes (*mudar nome de variável*)



c:\gilmar_borba\AQS\imagens\

REFATORAÇÃO

Refatoração De Mudança De Nomes *(mudar nome de package)*





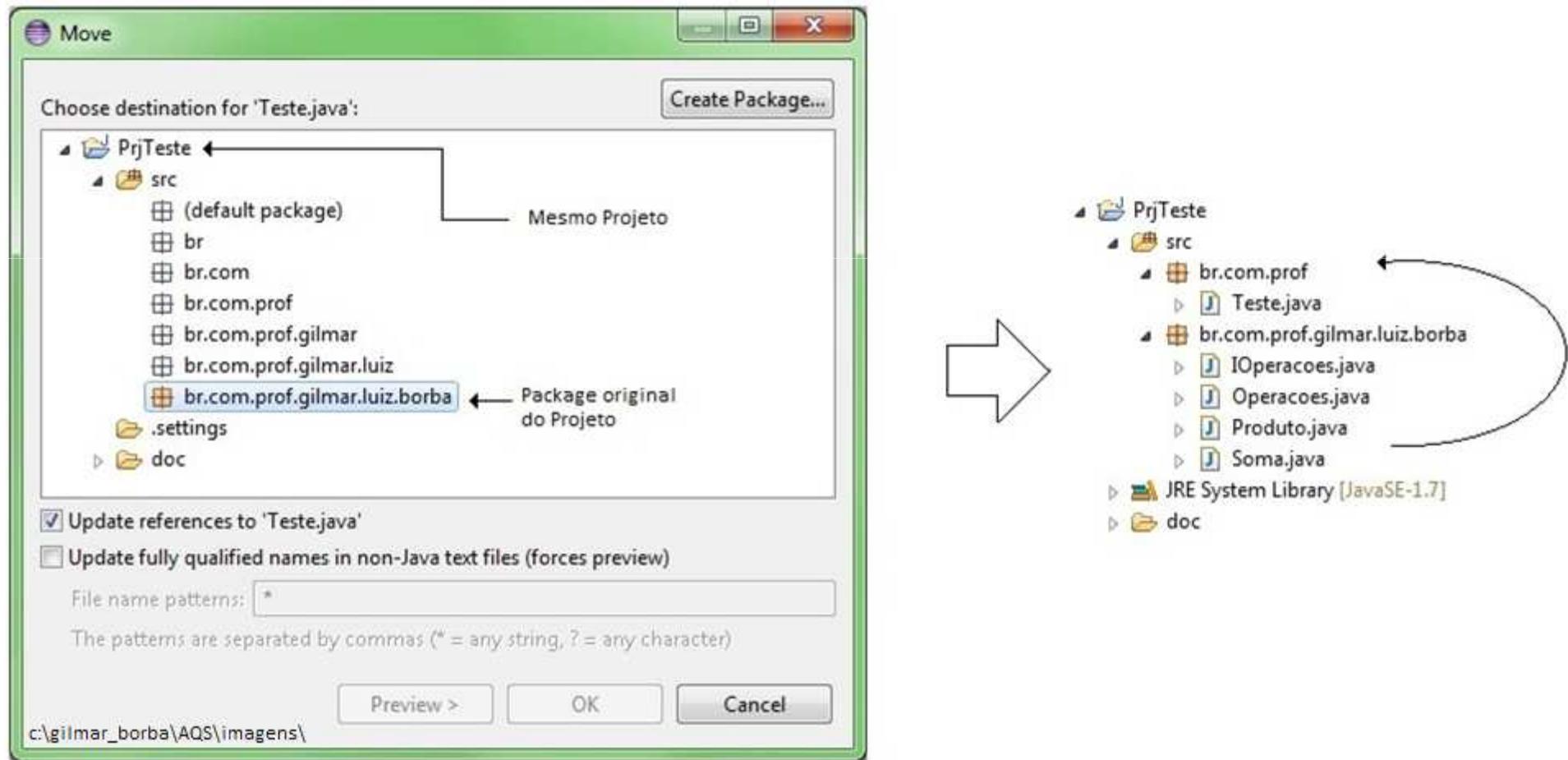
Refatoração De Mover Classe

(mover uma classe de um pacote para outro)

- Move uma classe de um pacote para outro.
- Faz fisicamente a remoção (para a pasta correspondente).
- Altera todas as referências à classe para se referir ao novo pacote.
- Também é possível fazer a remoção fisicamente, arrastando e soltando uma classe para um novo pacote (Package Explorer), neste caso a refatoração é feita automaticamente.

REFATORAÇÃO

Refatoração De Mover Classe *(mover uma classe de um pacote para outro)*



REFATORAÇÃO



Extração de variáveis locais *(Extract Local Variable)*

- Permite associar um resultado (de uma expressão) a uma nova variável local.
- É usada para simplificar o código.
- Permite modularizar o código, dividindo em várias linhas expressões complexas.

REFATORAÇÃO



Extração de variáveis locais *(Extract Local Variable)*

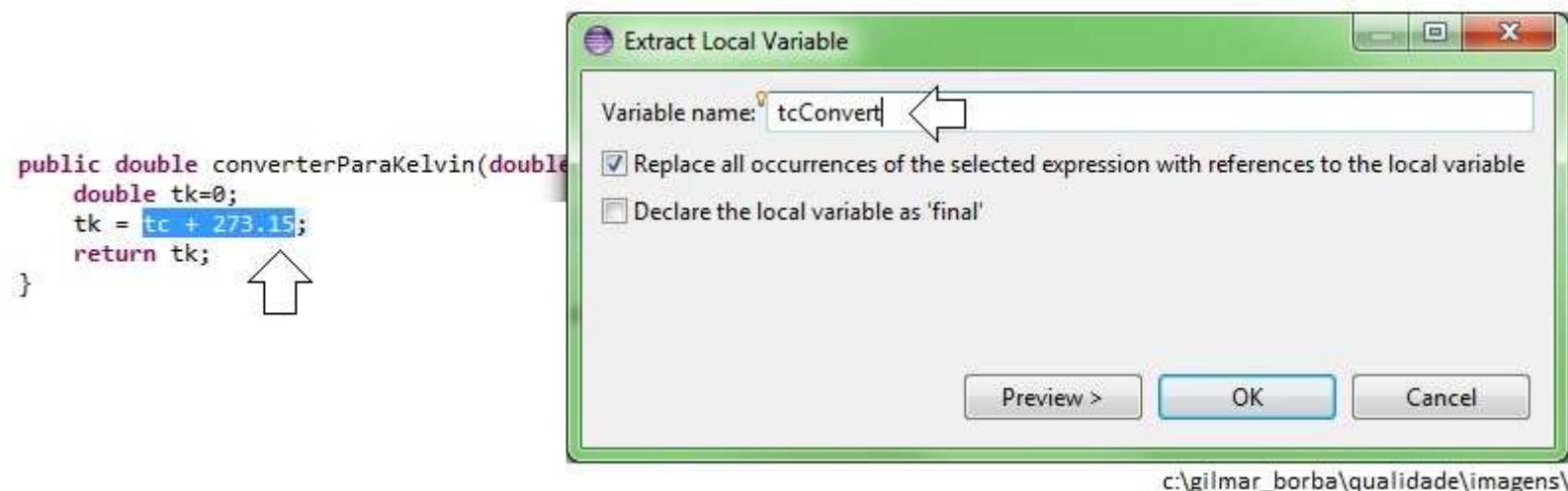
Usando o Eclipse

- A criação desta variável é feita automaticamente pelo IDE (usando recurso de REFACTORING)
- Selecionar a expressão matemática
- Informar um nome para variável

c:\gilmar_borba\AQS\imagens\

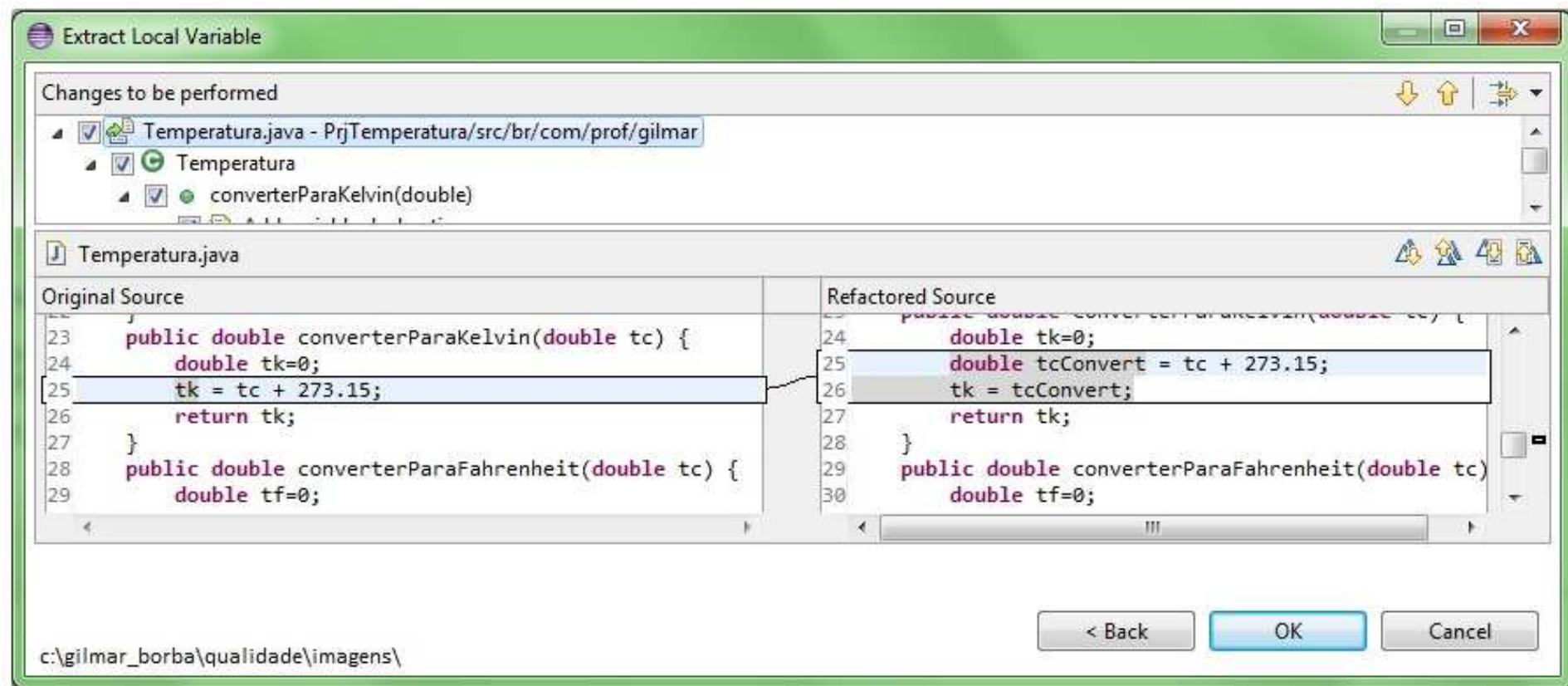
REFATORAÇÃO

Extração de variáveis locais (*Extract Local Variable*)



REFATORAÇÃO

Extração de variáveis locais (*Extract Local Variable*)





Extração de constantes (*Extract Constant*)

- Permite converter números ou cadeias de caracteres em constante (um campo final).
- Após a refatoração, todas as dependências a esses números ou literais apontarão para essa constante.

Passos:

1. Selecionar o valor ou expressão desejada
2. Informar o nome da constante (usar notação padrão)

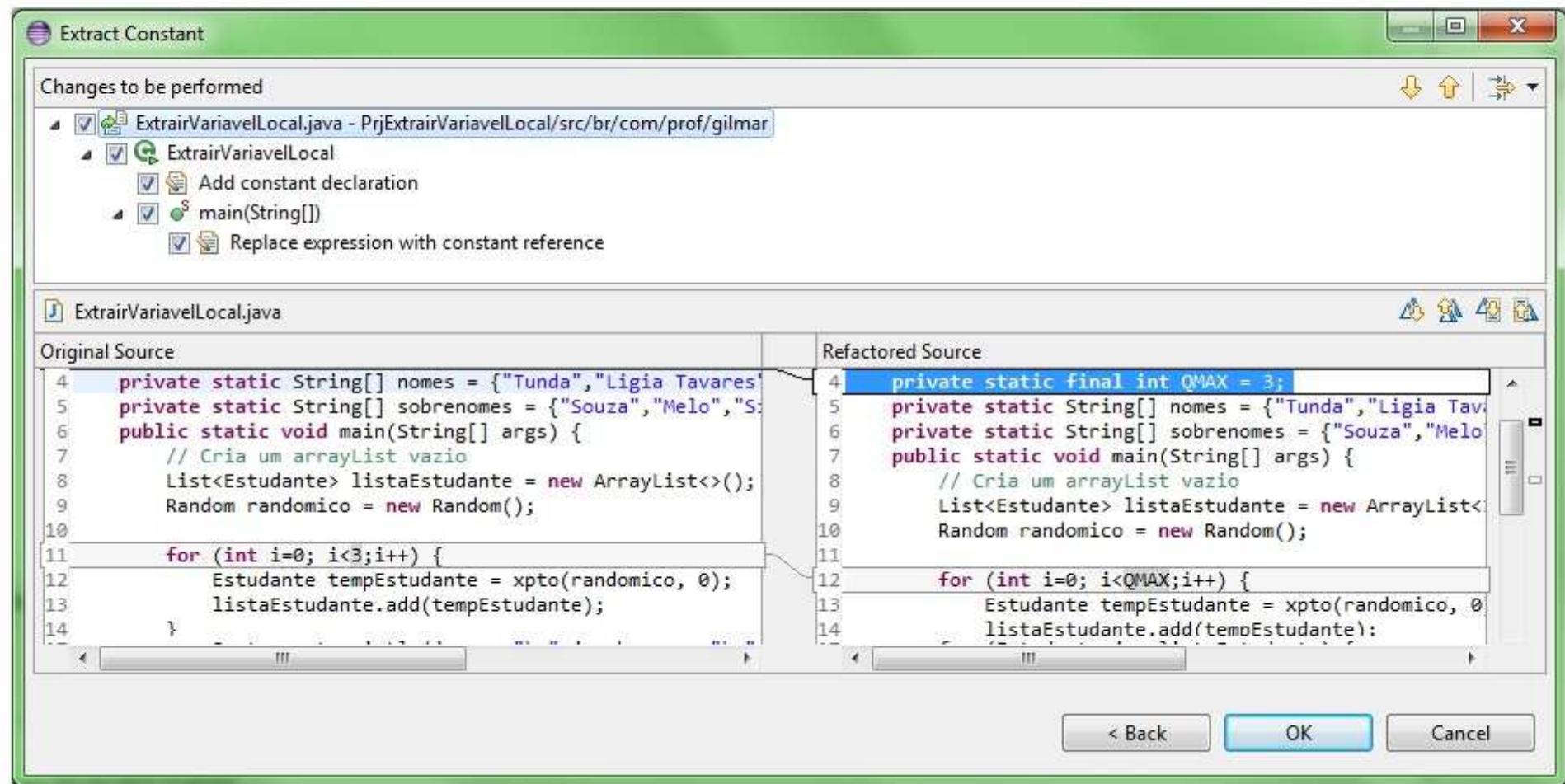
REFATORAÇÃO

Extração de constantes (*Extract Constant*)



REFATORAÇÃO

Extração de constantes (*Extract Constant*)



REFATORAÇÃO

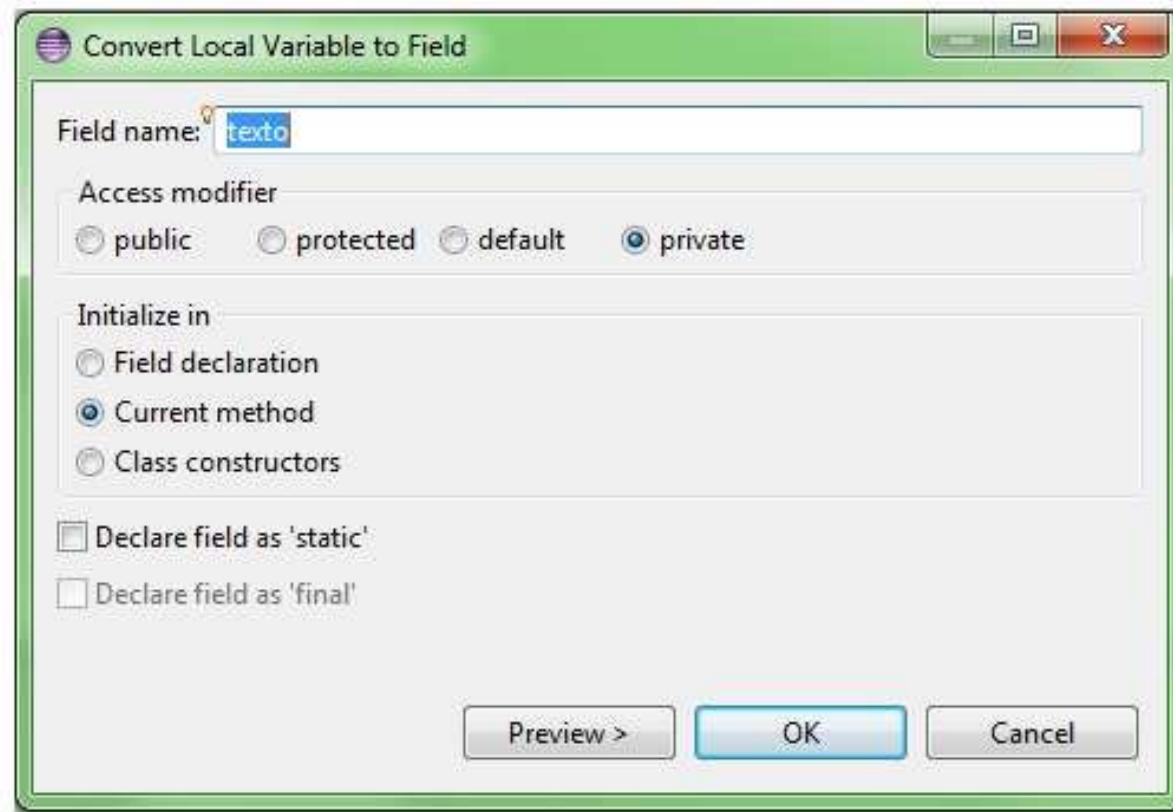
Converte variáveis locais para campo

- Converte uma variável local para um campo da classe.
- Todas as referências à variável local apontarão para o campo.

```
23     public void setSalario(double salario) {  
24         String texto="Salário deve ser maior que 0";  
25         if (salario >0)  
26             this.salario = salario;  
27         else  
28             throw new IllegalArgumentException(texto);  
29     }
```

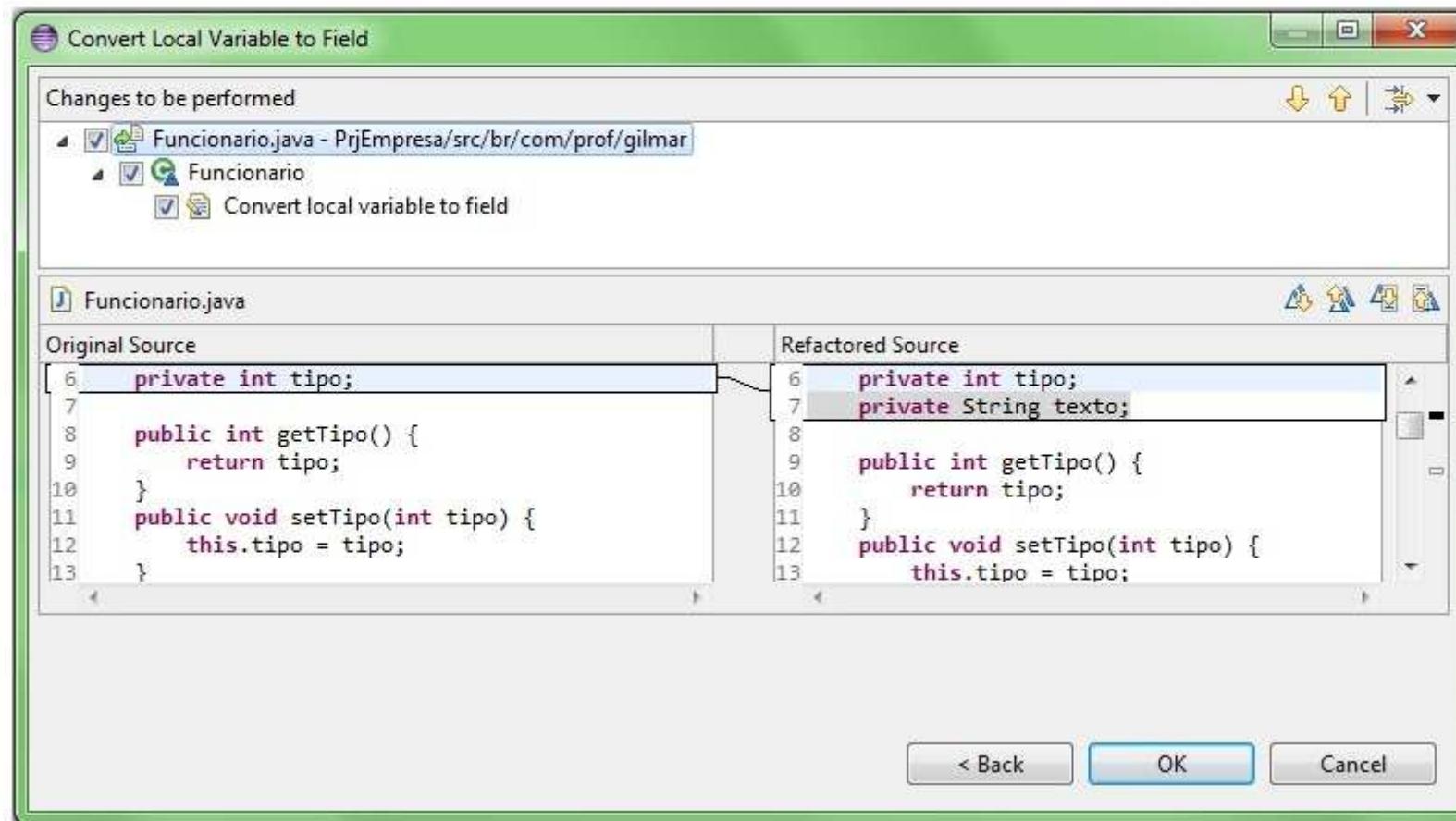
REFATORAÇÃO

Converte variáveis locais para campo



REFATORAÇÃO

Converte variáveis locais para campo





Extração de métodos

Extração de métodos é uma das refatoração muito comum. em métodos longos demais ou de difícil entendimento, extrair trechos de código e criar novos métodos para eles proporcionará uma melhora significativa para reutilização e manutenção futura desse código.

c:\gilmar_borba\AQS\imagens\

REFATORAÇÃO



Extração de métodos

Extração de métodos é uma refatoração muito comum. em métodos longos demais ou de difícil entendimento, extrair trechos de código e criar novos métodos para eles proporcionará uma melhora significativa para reutilização e manutenção futura desse código.

Passos:

1. Criar o novo método
2. Mover o trecho de código para esse novo método
3. Identificar os parâmetros
4. Identificar o retorno
5. Atualizar referências

Extract Method

Changes to be performed

- Extract Metodo.java - PrjExtrairMetodo/src/br/com/prof/gilmar
- Extract Metodo
- main(String[])
- Substitute statements with call to imprimirResultados
- Create new method 'imprimirResultados' from selected statements

ExtrairMetodo.java

Original Source	Refactored Source
<pre> 19 System.out.println("\n\nInforme o valor da mensalidade:"); 20 mensalidade = entrada.nextDouble(); 21 System.out.print("Informe o mês de referência: "); 22 mes = entrada.nextInt(); 23 Calendar calendario = Calendar.getInstance(); 24 mesAtual = calendario.get(Calendar.MONTH) + 1; 25 26 if (mes < mesAtual) 27 novaMensalidade = mensalidade + mensalidade * 10 / 100; 28 else if (mes == mesAtual) 29 novaMensalidade = mensalidade; 30 else { 31 mes = mesAtual; 32 novaMensalidade = mensalidade; 33 } 34 System.out.println("RESULTADO\n"); 35 System.out.println("Mês atual: " + mesAtual); 36 System.out.println("Mês de referência: " + mes); 37 System.out.println("Mensalidade: " + mensalidade); 38 System.out.println("Nova mensalidade: " + novaMensalidade); 39 contador++; 40 } // fim while 41 System.out.println("FIM NORMAL DA APLICAÇÃO"); 42 } // fim main 43 } // fim class 44 </pre>	<pre> 22 mes = entrada.nextInt(); 23 Calendar calendario = Calendar.getInstance(); 24 mesAtual = calendario.get(Calendar.MONTH) + 1; 25 26 if (mes < mesAtual) 27 novaMensalidade = mensalidade + mensalidade * 10 / 100; 28 else if (mes == mesAtual) 29 novaMensalidade = mensalidade; 30 else { 31 mes = mesAtual; 32 novaMensalidade = mensalidade; 33 } 34 imprimirResultados(); 35 contador++; 36 } // fim while 37 System.out.println("FIM NORMAL DA APLICAÇÃO"); 38 } // fim main 39 private static void imprimirResultados() { 40 System.out.println("RESULTADO\n"); 41 System.out.println("Mês atual: " + mesAtual); 42 System.out.println("Mês de referência: " + mes); 43 System.out.println("Mensalidade: " + mensalidade); 44 System.out.println("Nova mensalidade: " + novaMensalidade); 45 } 46 } // fim class 47 </pre>



Extração de Super Classe

Usada quando um conjunto de classes concretas possuem campos ou métodos comuns.

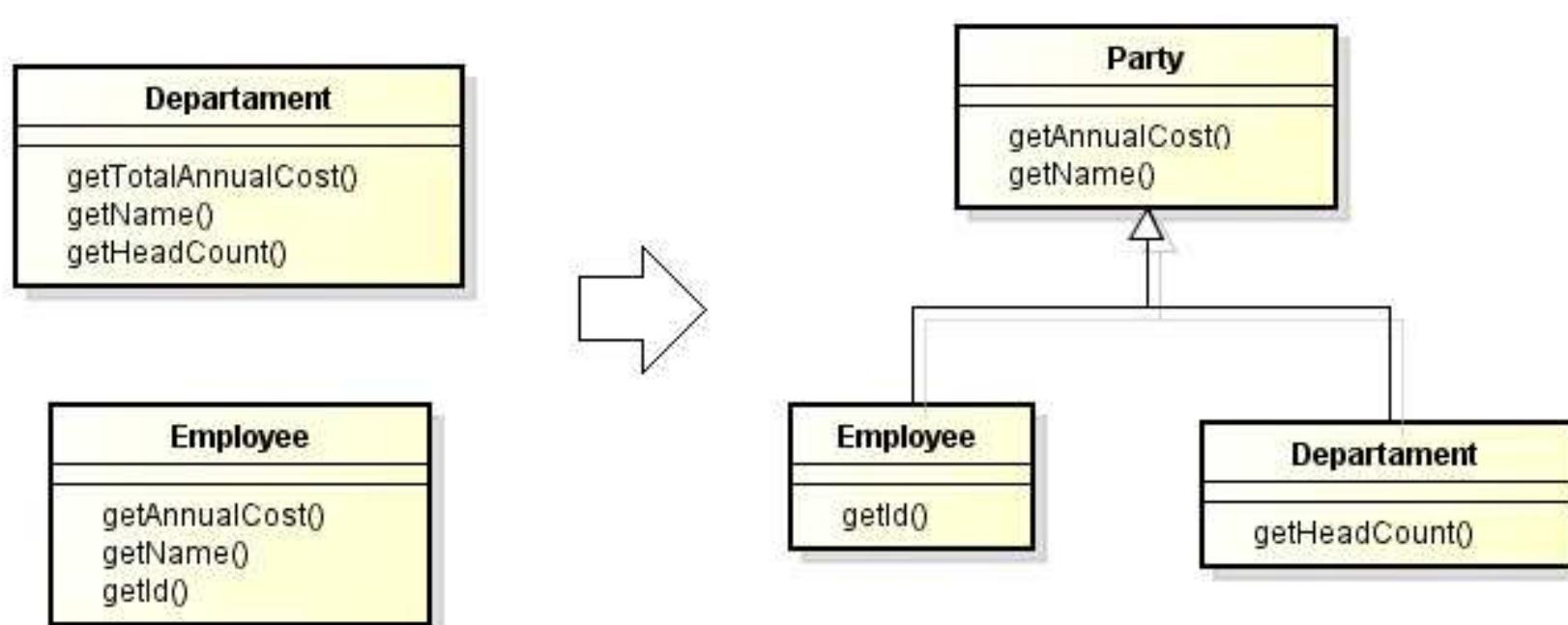
Passos:

1. Criar uma classe abstrata em branco
2. Fazer com que ela seja superclasse das demais classes
3. Subir (pull up) os métodos e campos um por um
4. Usar a abstração onde for mais adequado

REFATORAÇÃO

Extração de Super Classe

Usada quando um conjunto de classes concretas possuem campos ou métodos comuns.



c:\gilmar_borba\qualidade\imagens\



Substituir Condicional por Polimorfismo

Altera comandos condicionais substituindo-os com a invocação de métodos polimórficos.

Passos:

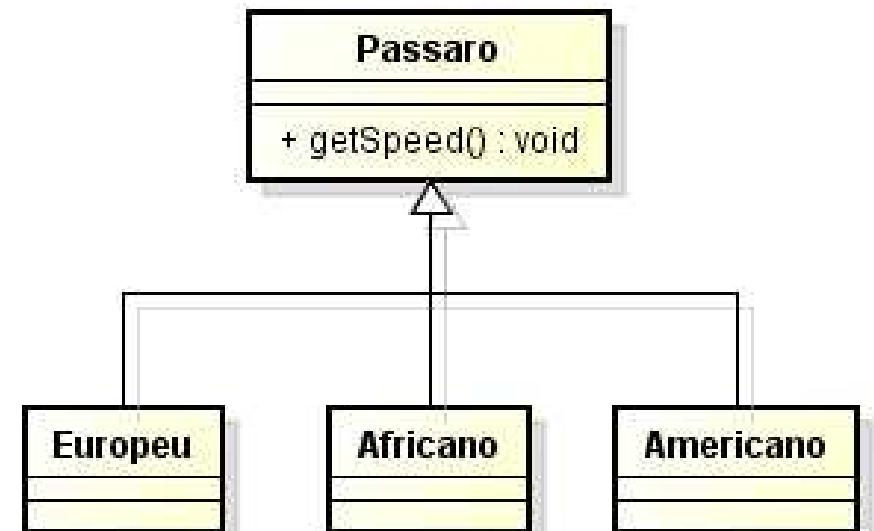
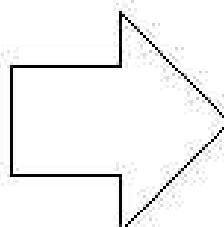
1. Criar uma subclasse para cada comportamento
2. Mover o comportamento para cada tipo da classe respectiva (pull up), para superclasse
3. Invocar o método respectivo fazendo chamadas polimórficas

REFATORAÇÃO

Substituir Condicional por Polimorfismo

Altera comandos condicionais substituindo-os com a invocação de métodos polimórficos.

```
static double getSpeed(int tipo) {  
    if (tipo == EUROPEU)  
        return 50.00;  
    if (tipo == AFRICANO)  
        return 120.00;  
    if (tipo == AMERICANO)  
        return 200.00;  
    throw new  
        RuntimeException("unreachable",  
    )
```



c:\gilmar_borba\qualidade\imagens\

REFATORAÇÃO



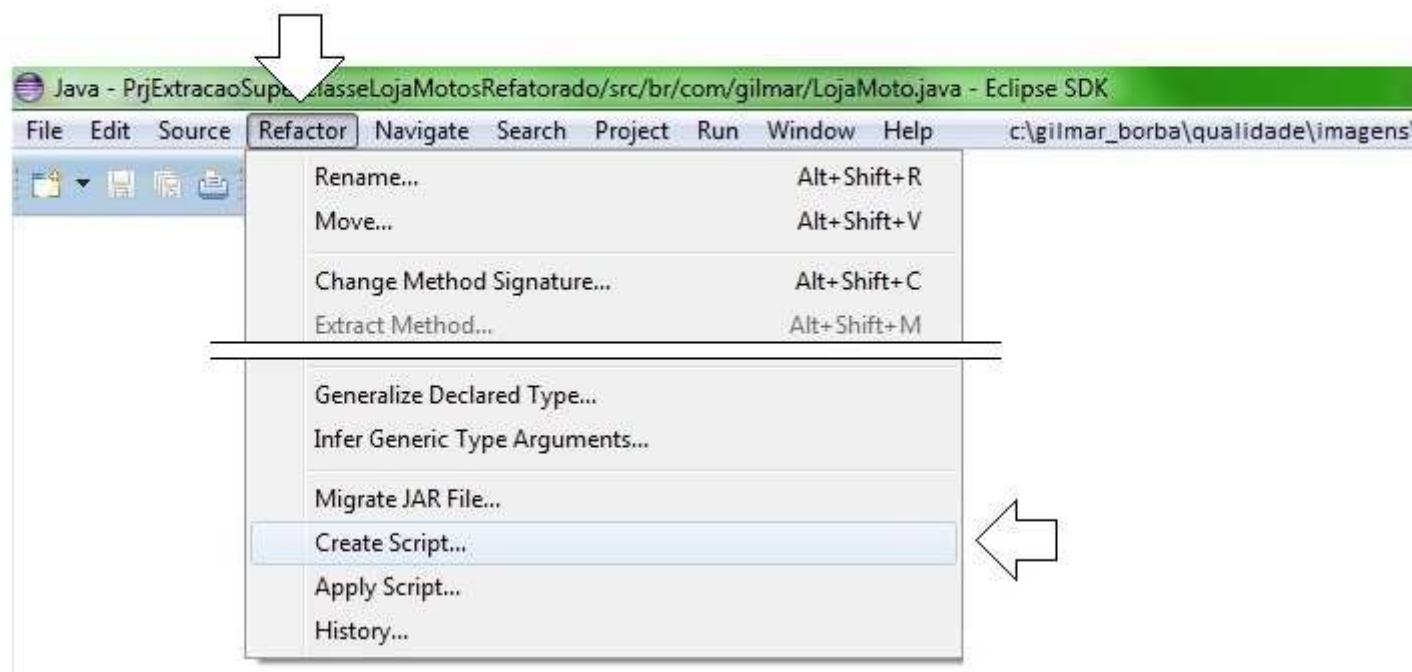
Scripts de refatoração

- Permitem mostrar as refatorações realizadas
- É importante distribuir o script junto com novas versões (código ou bibliotecas)
- Gera Script no formato .XML
- No Eclipse, acessar o menu Refatoração, em seguida Create Script.

c:\gilmar_borba\AQS\imagens\

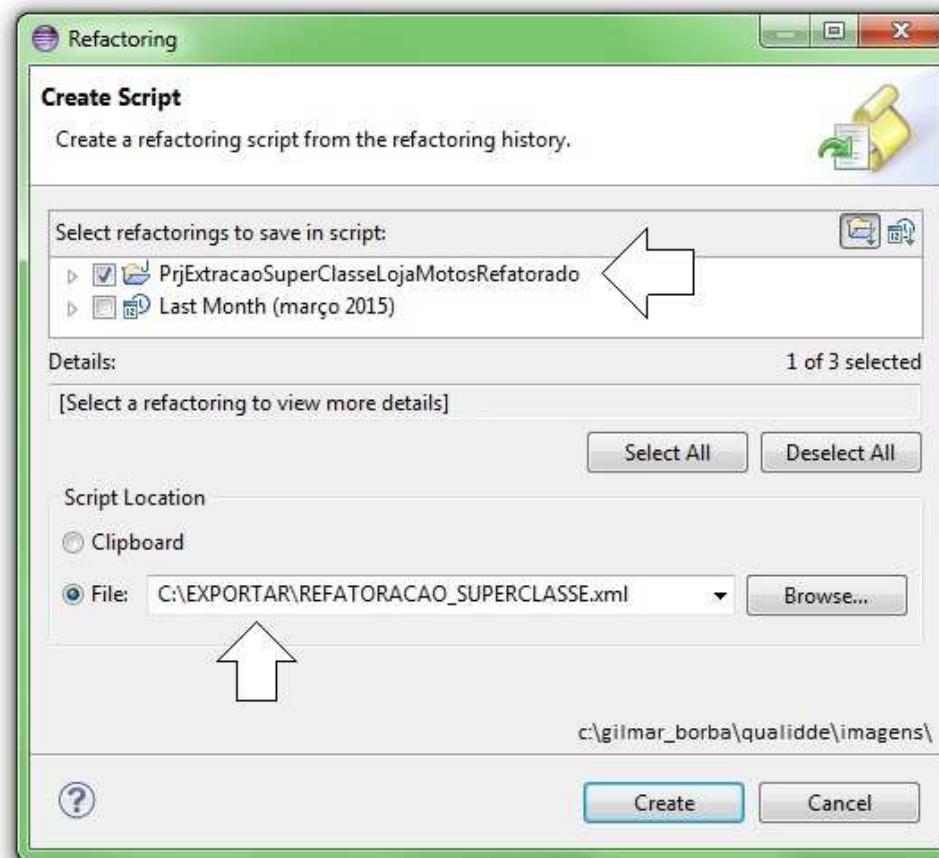
REFATORAÇÃO

Scripts de refatoração



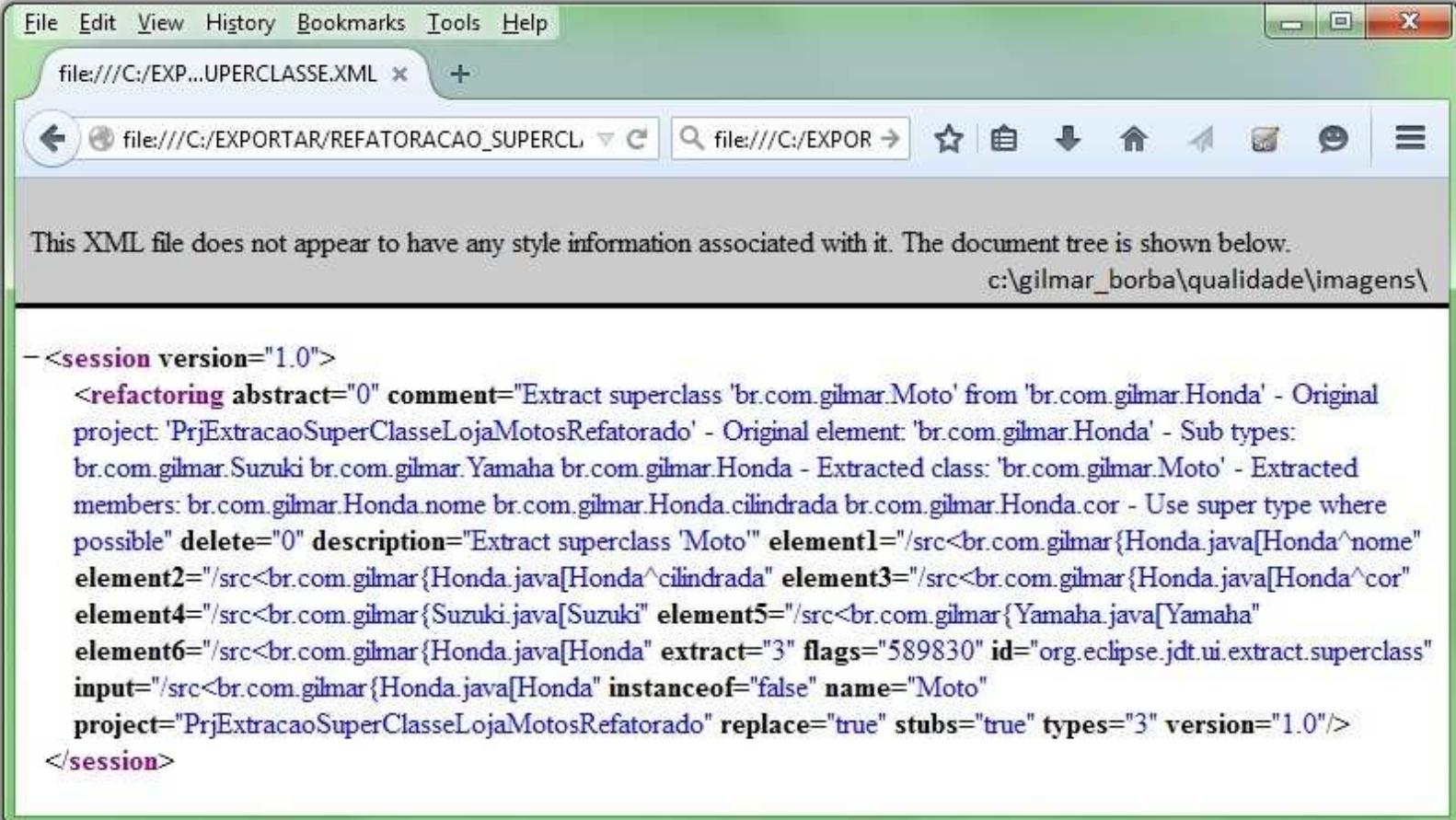
REFATORAÇÃO

Scripts de refatoração



REFATORAÇÃO

Scripts de refatoração



The screenshot shows a web browser window with a green header bar. The title bar says "file:///C:/EXP...UPERCLASSE.XML". The address bar shows "file:///C:/EXPORTAR/REFATORACAO_SUPERCL..." and has a dropdown arrow pointing down. Below the address bar are standard browser buttons: back, forward, search, and others. The main content area contains the following text:

This XML file does not appear to have any style information associated with it. The document tree is shown below.
c:\gilmar_borba\qualidade\imagens\

```
--<session version="1.0">
<refactoring abstract="0" comment="Extract superclass 'br.com.gilmar.Moto' from 'br.com.gilmar.Honda' - Original project: 'PrjExtracaoSuperClasseLojaMotosRefatorado' - Original element: 'br.com.gilmar.Honda' - Sub types: br.com.gilmar.Suzuki br.com.gilmar.Yamaha br.com.gilmar.Honda - Extracted class: 'br.com.gilmar.Moto' - Extracted members: br.com.gilmar.Honda.nome br.com.gilmar.Honda.cilindrada br.com.gilmar.Honda.cor - Use super type where possible" delete="0" description="Extract superclass 'Moto'" element1="/src<br.com.gilmar{Honda.java[Honda^nome"
element2="/src<br.com.gilmar{Honda.java[Honda^cilindrada" element3="/src<br.com.gilmar{Honda.java[Honda^cor"
element4="/src<br.com.gilmar{Suzuki.java[Suzuki" element5="/src<br.com.gilmar{Yamaha.java[Yamaha"
element6="/src<br.com.gilmar{Honda.java[Honda" extract="3" flags="589830" id="org.eclipse.jdt.ui.extract.superclass"
input="/src<br.com.gilmar{Honda.java[Honda" instanceof="false" name="Moto"
project="PrjExtracaoSuperClasseLojaMotosRefatorado" replace="true" stubs="true" types="3" version="1.0"/>
</session>
```

REFATORAÇÃO



Algumas conclusões sobre a Refatoração

- A refatoração torna o código mais simples e mais fácil de ser entendido.
- A refatoração consiste em fazer modificações simples, que não alteram a funcionalidade do código.
- A refatoração, quando feita de maneira adequada traz melhorias significativas no código.

REFATORAÇÃO



Algumas conclusões sobre a Refatoração

- A refatoração pode ser realizada de maneira manual ou apoiada em ferramentas.
- A idéia da refatoração é criar um design incremental e evolutivo.
- No TDD temos design o tempo todo, não há uma fase só de design, usando a refatoração o design vai emergindo aos poucos de acordo com a necessidade.

QUESTÕES



(34) Definir Refatoração.

(35) Estabeleça uma relação entre a refatoração e a dívida técnica.

(36) Um comentário inadequado dentro do código fonte pode ser um indicativo que necessitamos refatorar o código? Justifique.

(37) Forneça alguns exemplos de Bad Smells.

(38) Por que refatorar?

(39) Quais são os sintomas que mostram que seu código merece ser refatorado?

(40) Citar 5 tipos de refatoração de código.



JUNIT

JUNIT



É um framework open-source, para a criação de testes automatizados na linguagem de programação Java. Com o JUNIT verifica-se se cada método de uma classe funciona da forma esperada.

É uma forma (API) de fazer testes unitários em código Java. Então se você deseja verificar se todos os métodos de uma classe estão executando de forma esperada, usamos o JUNIT para verificar antes e fazer o deploy da aplicação. (MEDEIROS, 2012, 43).



O teste unitário testa o menor componente um sistema de software de maneira isolada.

Um teste unitário executa um método de maneira individual e compara os dados de entrada com uma saída conhecida.



Classe base (Que será testada)

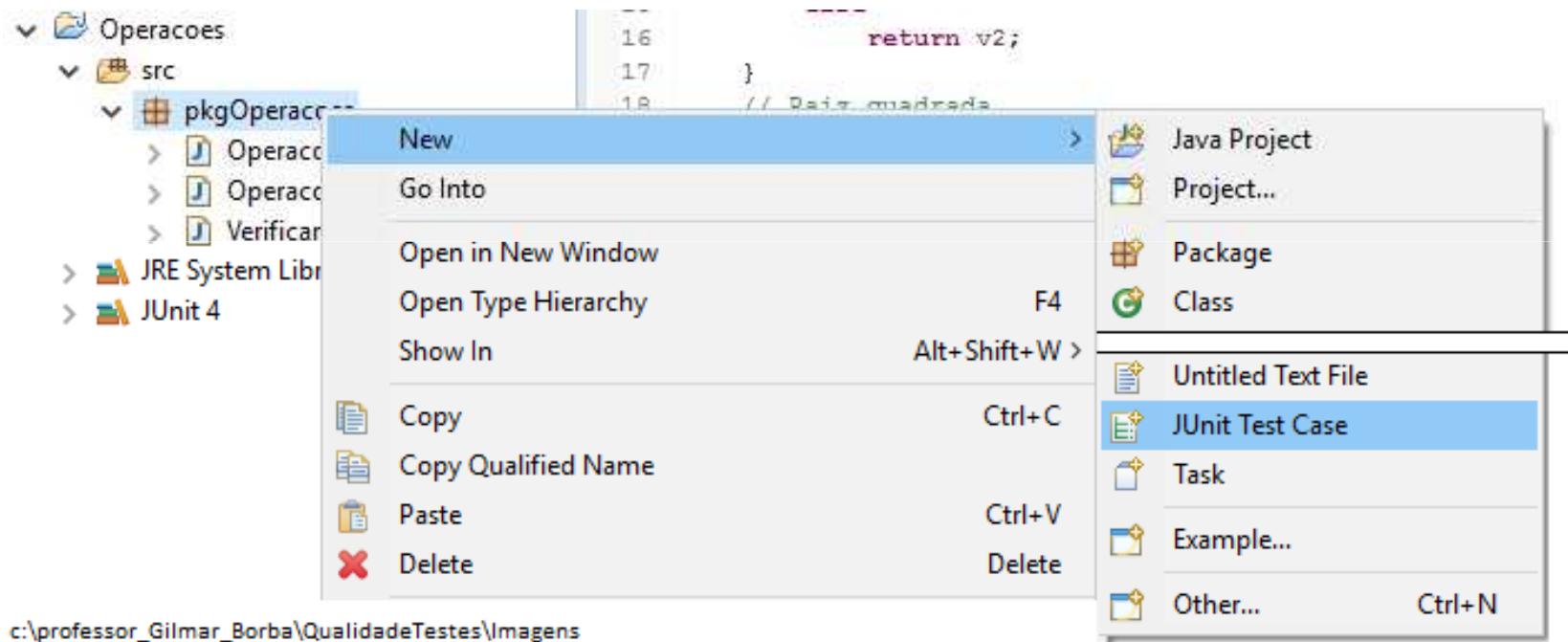
```
1 package pkgOperacoes;
2 public class Operacoes {
3     private double resultado = 0;
4     // Maior valor
5     public double maiorValor(double v1, double v2) {
6         if (v1 > v2)
7             return v1;
8         else
9             return v2;
10    }
11    // Menor valor
12    public double menorValor(double v1, double v2) {
13        if (v1 < v2)
14            return v1;
15        else
16            return v2;
17    }
18    // Raiz quadrada
19    public double raizQuadrada(double v1) {
20        if (v1 >=0)
21            resultado = Math.sqrt(v1);
22        else
23            throw new IllegalArgumentException("O Valor não pode ser negativo");
24        return resultado;
25    }
c:\professor_Gilmar_Borba\QualidadeTestes\Imagens
```



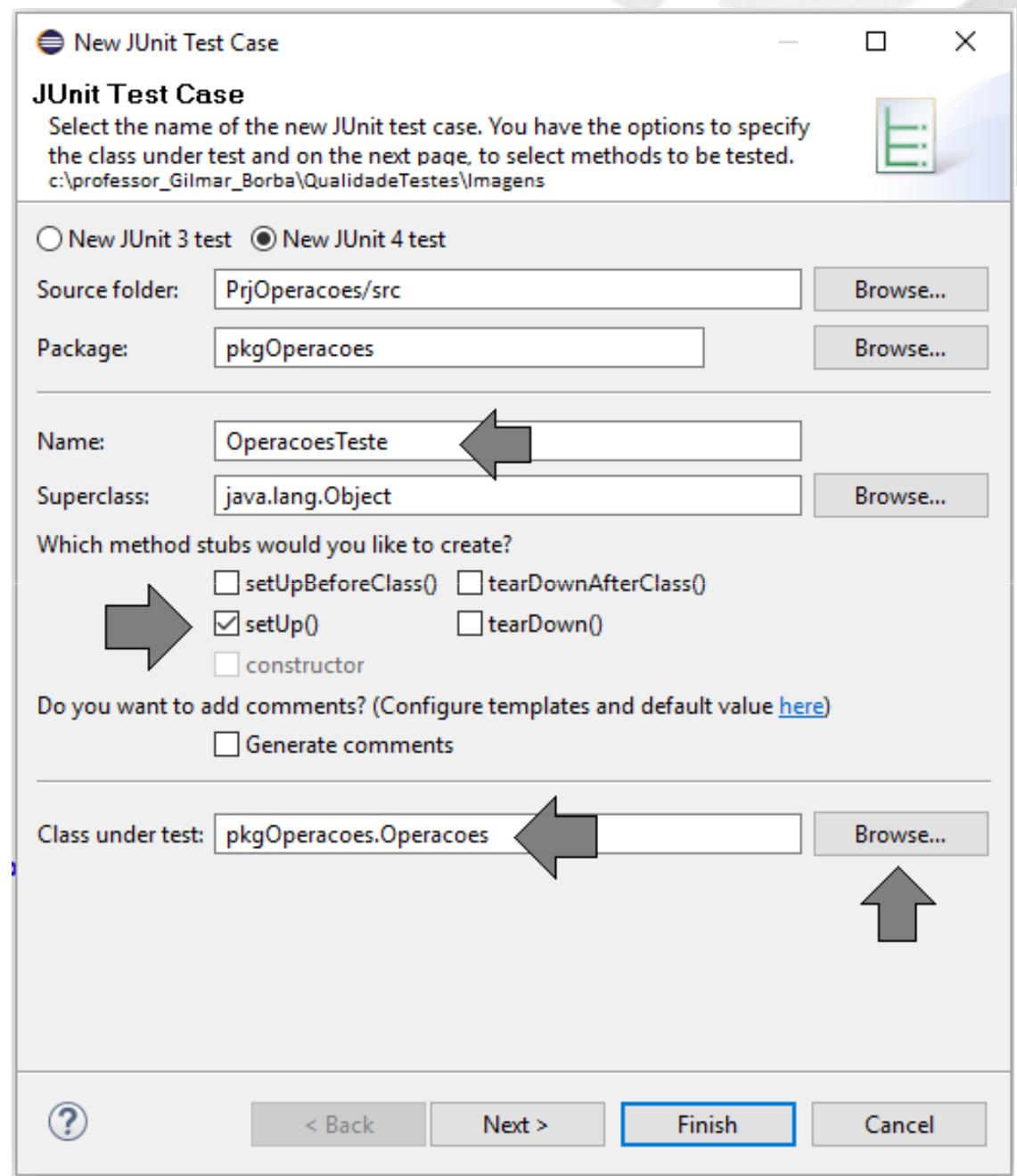
Classe base (*Que será testada*)

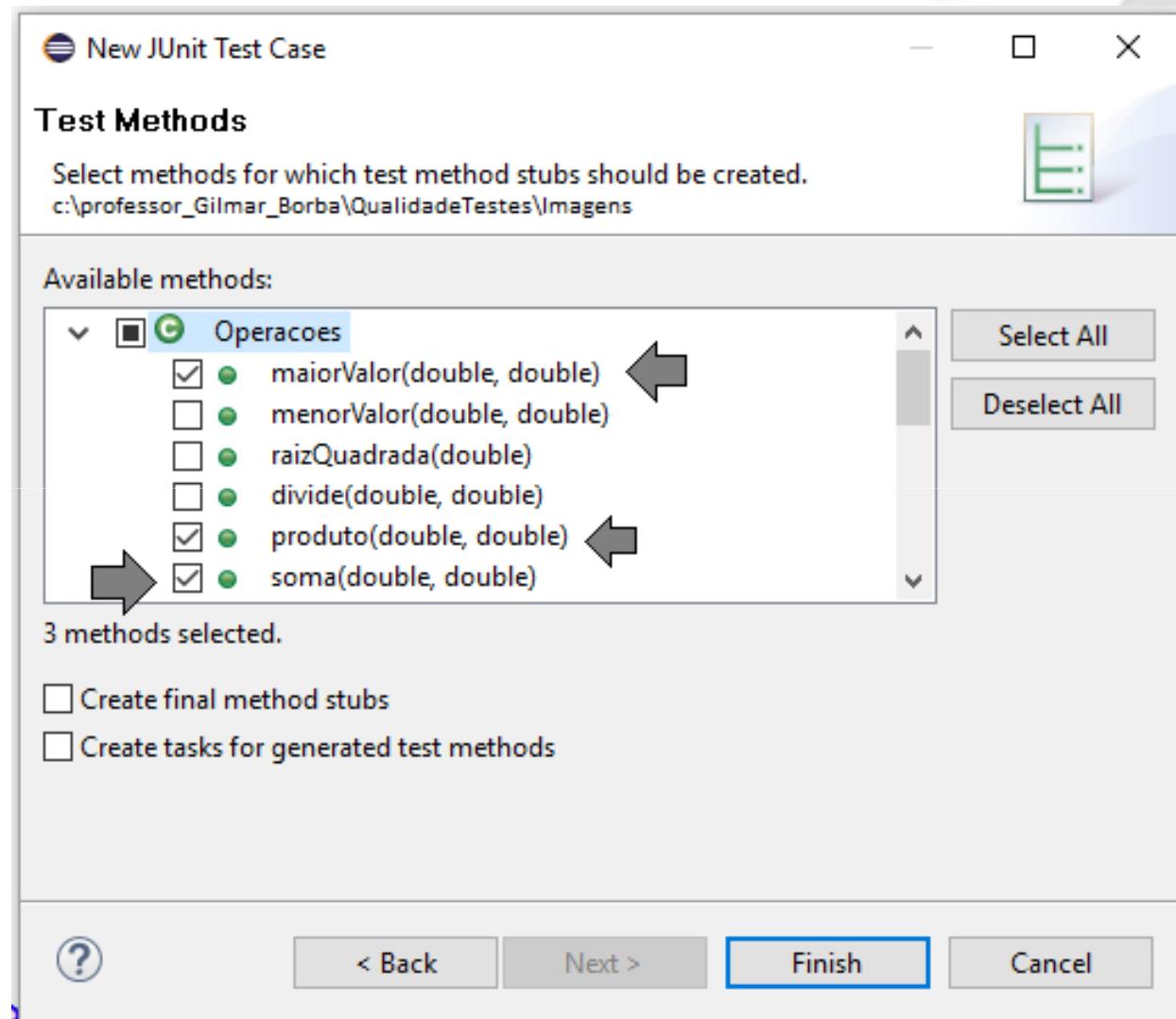
```
25
26 // Divisão
27 ⊕ public double divide(double v1, double v2) {
28     try {
29         resultado = v1 / v2;
30         return resultado;
31     }catch (Exception e) {
32         System.out.println("Um erro ocorreu"+e.getMessage());
33     }
34     System.out.println("O(s) valor(es) não pode(m) se nulos (Zeros!)");
35     throw new IllegalArgumentException("O(s) Valor(es) não pode(m) "+
36     "ser nulo(s)");
37 }
38 // Produto - multiplicacao
39 ⊕ public double produto(double v1, double v2) {
40     resultado = v1 * v2;
41     return resultado;
42 }
43
44 // Soma
45 ⊕ public double soma(double v1, double v2) {
46     resultado = v1 + v2;
47     return resultado;
48 }
49 }
50
c:\professor_Gilmar_Borba\QualidadeTestes\Imagens
```

Criar a classe de teste



JUNIT





JUNIT

```
1 package pkgOperacoes;
2
3 import static org.junit.Assert.*;
4
5 public class OperacoesTeste {
6
7     @Before
8     public void setUp() throws Exception {
9         }
10
11     @Test
12     public void testMaiorValor() {
13         fail("Not yet implemented");
14     }
15
16     @Test
17     public void testProduto() {
18         fail("Not yet implemented");
19     }
20
21     @Test
22     public void testSoma() {
23         fail("Not yet implemented");
24     }
25
26     }
27
28
29 }
30 c:\professor_Gilmar_Borba\QualidadeTestes\Imagens
```

JUNIT

```
1 package pkgOperacoes;
2 import static org.junit.Assert.*;
3 public class OperacoesTeste {
4     Operacoes op;
5     @Before
6     public void setUp() throws Exception {
7         op = new Operacoes();
8     }
9     @Test
10    public void testMaiorValor() {
11        assertEquals("CASO 1: ", 16, op.maiorValor(16, 9), 0);
12    }
13    @Test
14    public void testProduto() {
15        assertEquals("CASO 2: ", 144, op.produto(16, 9), 0);
16    }
17    @Test
18    public void testSoma() {
19        assertEquals("CASO 2: ", 25, op.soma(16, 9), 0);
20    }
21    @Test
22    public void testVerdadeiro() {
23        assertTrue(op.maiorValor(16, 9) < 25);
24    }
25    @Test
26    public void testFalso() {
27        assertFalse(op.maiorValor(16, 9) > 25);
28    }
29}
30}
31}
```

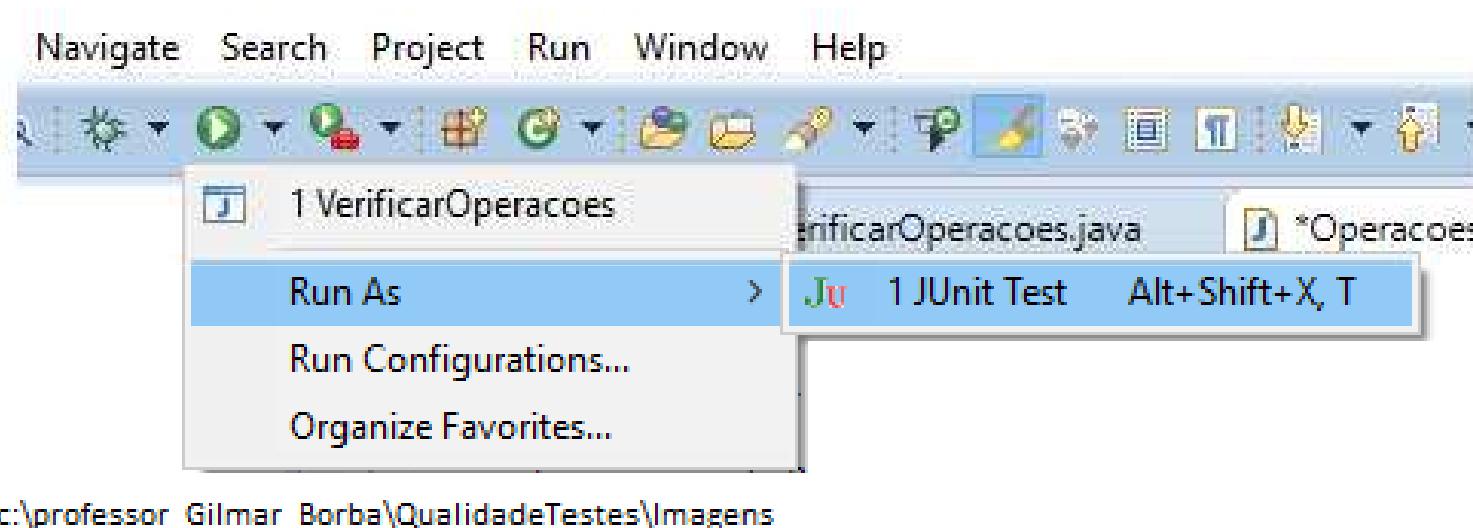
c:\professor_Gilmar_Borba\QualidadeTestes\Imagens

JUNIT

```
1 package pkgOperacoes;
2 import static org.junit.Assert.*;
3 public class OperacoesTeste {
4     Operacoes op; 1
5     @Before
6     public void setUp() throws Exception {
7         op = new Operacoes(); 2
8     }
9     @Test
10    public void testMaiorValor() {
11        assertEquals("CASO 1: ", 16, op.maiorValor(16, 9), 0); 3
12    }
13    @Test
14    public void testProduto() {
15        assertEquals("CASO 2: ", 144, op.produto(16, 9), 0);
16    }
17    @Test
18    public void testSoma() {
19        assertEquals("CASO 2: ", 25, op.soma(16, 9), 0);
20    }
21    @Test
22    public void testVerdadeiro() {
23        assertTrue(op.maiorValor(16, 9) < 25);
24    }
25    @Test
26    public void testFalso() {
27        assertFalse(op.maiorValor(16, 9) > 25);
28    }
29}
30}
31}
```

c:\professor_Gilmar_Borba\QualidadeTestes\Imagens

JUNIT



JUNIT

The screenshot shows the Eclipse IDE interface with the JUnit perspective active. The top bar includes tabs for 'Package Explorer', 'JUnit' (which is selected), and other views like 'VerificarOperacoes.java' and 'OperacoesTeste.java'. The 'JUnit' tab displays the test results: 'Runs: 5/5', 'Errors: 0', and 'Failures: 1'. Below this, the 'Failure Trace' section shows a single assertion error: 'java.lang.AssertionError: CASO 1: expected:<9.0> but was:<16.0>' at line 30 of 'OperacoesTeste.java'. The main workspace contains two Java files: 'Operacoes.java' and 'OperacoesTeste.java'. 'Operacoes.java' defines a class with methods for addition, multiplication, and comparison. 'OperacoesTeste.java' contains JUnit tests for these methods, including a failing test for the 'maiorValor' method.

```
1 package pkgOperacoes;
2 import static org.junit.Assert.*;
3 public class OperacoesTeste {
4     Operacoes op;
5     @Before
6     public void setUp() throws Exception {
7         op = new Operacoes();
8     }
9     @Test
10    public void testMaiorValor() {
11        assertEquals("CASO 1: ", 9, op.maiorValor(16, 9), 0);
12    }
13    @Test
14    public void testProduto() {
15        assertEquals("CASO 2: ", 144, op.produto(16, 9), 0);
16    }
17    @Test
18    public void testSoma() {
19        assertEquals("CASO 3: ", 25, op.soma(16, 9), 0);
20    }
21    @Test
22    public void testVerdadeiro() {
23        assertTrue(op.maiorValor(16, 9) < 25);
24    }
25    @Test
26    public void testFalso() {
27        assertFalse(op.maiorValor(16, 9) > 25);
28    }
29}
30}
31}
32}
```

JUNIT

The screenshot shows the Eclipse IDE interface with the JUnit perspective selected. The top bar includes tabs for 'Package Explorer', 'JUnit' (which is active), and 'VerificarOperacoes.java'. Below the tabs, a message states 'Finished after 0,024 seconds'. A summary bar shows 'Runs: 5/5', 'Errors: 0', and 'Failures: 0' with a green progress bar. The left side features a tree view under 'pkgOperacoes.OperacoesTeste [Runner: JUnit 4] (0,000 s)' containing five test methods: 'testVerdadeiro', 'testFalso', 'testSoma', 'testProduto', and 'testMaiorValor', all marked with a green checkmark. The bottom left shows a 'Failure Trace' section with a small icon. The right side displays the source code for 'Operacoes.java' and 'OperacoesTeste.java'. The code for 'Operacoes.java' defines a class 'Operacoes' with methods like 'maiorValor', 'produto', and 'soma'. The code for 'OperacoesTeste.java' contains JUnit test cases for these methods, using assertions like 'assertEquals' and 'assertTrue'.

```
1 package pkgOperacoes;
2 import static org.junit.Assert.*;
3 public class Operacoes {
4     Operacoes op;
5     @Before
6     public void setUp() throws Exception {
7         op = new Operacoes();
8     }
9     @Test
10    public void testMaiorValor() {
11        assertEquals("CASO 1: ", 16, op.maiorValor(16, 9), 0);
12    }
13    @Test
14    public void testProduto() {
15        assertEquals("CASO 2: ", 144, op.produto(16, 9), 0);
16    }
17    @Test
18    public void testSoma() {
19        assertEquals("CASO 2: ", 25, op.soma(16, 9), 0);
20    }
21    @Test
22    public void testVerdadeiro() {
23        assertTrue(op.maiorValor(16, 9) < 25);
24    }
25    @Test
26    public void testFalso() {
27        assertFalse(op.maiorValor(16, 9) > 25);
28    }
29 }
30 }
```

QUESTÕES



- (41) Citar três objetivos das métricas.
- (42) Quais são as informações apresentadas nas métricas?
- (43) Definir Complexidade ciclomática.
- (44) O que é o Teste Unitário?
- (45) O que é o JUNIT?



CÓDIGO LIMPO



Qualidade de Código e o Código limpo

Conceitos iniciais

Falar sobre código é falar do passado? Esse assunto está ultrapassado?

Bobagens! Nunca nos livraremos do código, pois eles representam os detalhes dos requisitos. Em certo nível, não há como ignorar ou abstrair esses detalhes; eles precisam ser especificados. E especificar requisitos detalhadamente de modo que uma máquina possa executá-los é programar – e tal especificação é o código. (MARTIN (1), 2011:2).

Software em funcionamento mais que documentação abrangente

[...] Os dois melhores documentos para transferência de informações para novos membros são o código e a equipe. O código não mente sobre o que faz. Pode ser difícil extrair o fundamento lógico e o objetivo do código, mas ele é a única fonte de informação clara. A equipe tem na cabeça de seus membros o mapa sempre mutante do sistema. O modo mais rápido e eficiente de registrar esse mapa no papel e transferi-lo para outros é por meio da interação humana. (MARTIN (2), 2011:33).

CÓDIGO LIMPO

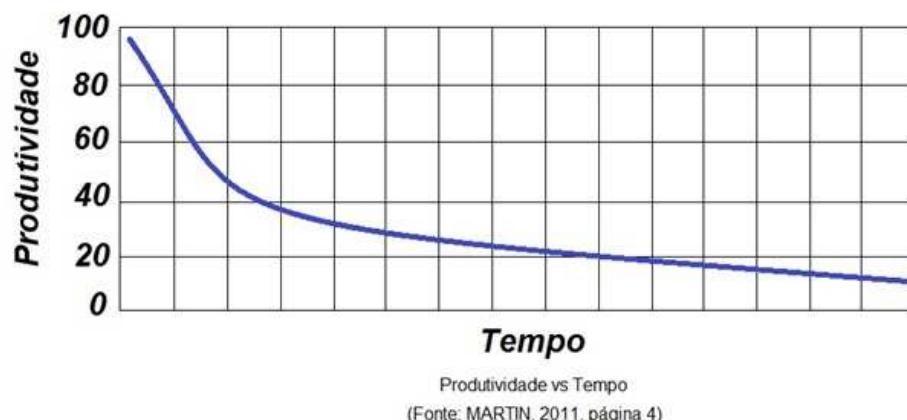
Qualidade de Código e o Código limpo

Conceitos iniciais

O custo de ter um código fonte confuso

[...] Conforme a confusão aumenta a produtividade da equipe diminui, assintoticamente aproximando-se de zero. Com a redução da produtividade, a gerência faz a única coisa que ela pode; adiciona mais membros ao projeto na esperança de aumentar a produtividade. Mas esses novos membros não conhecem o projeto do sistema, não sabem a diferença entre uma mudança que altera o propósito do projeto e aquela que o atrapalha. (MARTIN (1), 2011:4).

c:\prof_gilmar_borba\imagens\auditoria_testes\



CÓDIGO LIMPO



Qualidade de Código e o Código limpo

Conceitos iniciais

O que é um código limpo?

Gosto de meu código elegante e eficiente. A lógica deve ser direta para dificultar o encobrimento de *bugs*, as dependências mínimas para facilitar a manutenção, o tratamento de erro completo de acordo com uma estratégia clara e o desempenho próximo do mais eficiente de modo a não incitar as pessoas a tornarem o código confuso com otimizações sorrateiras. (MARTIN (1) apud STROUSTRUP, 2011:7).

Bjarne Stroustrup, criador do C++ e autor do livro A linguagem de programação C++.



O que é um código limpo?

Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas.

(MARTIN (1) apud BOOCH, 2011:8).

Grady Booch, autor do livro Object Oriented Analysis and Design Applications.

Eu poderia listar todas as qualidades que vejo em um código limpo, mas há uma predominante que leva a todas as outras. Um código limpo sempre parece que foi escrito por alguém que se importava. [...]

(MARTIN (1) apud FEATHERS, 2011:10).

CÓDIGO LIMPO

Nomes Significativos



O que é um código limpo?

Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem testes unitários e de aceitação, nomes significativos; ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa; possui poucas dependências, as quais são explicitamente declaradas e oferecem um API mínimo e claro.

Você sabe quando está criando um código limpo quando cada rotina que você lê se mostra como o que você esperava. Você pode chamar de código belo quando ele também faz parecer que a linguagem foi feita para o problema.

(MARTIN (1) apud CUNNINGHAM, 2011:9).

Ward Cunningham é o criador do conceito de WIKI, criador do Fit, cocriador da Programação Extrema (exTreme Programming). Incentivador dos padrões de projeto. Líder do Smalltalk e da OO.



Nomeando componentes de um código

```
8 static public void main(String args[]) {  
9     // a, b1, b2, h correspondem a: área, bases e altura do trapézio  
10    // r é a variável de repetição, q é a quantidade de cálculos  
11    // realizados  
12    double a, b1, b2, h;  
13    int r, q;  
14    // variáveis para cálculos finais:  
15    double acumuladorFinalS, acumuladorFinalM;  
16    a = 0;  
17    q = 0;  
18    acumuladorFinalS = 0;  
19    acumuladorFinalM = 0;  
20    for (r=1;r<=5;r++) {
```

c:\prof_gilmar_borba\imagens\auditoria_testes\

CÓDIGO LIMPO

Nomes Significativos

Nomeando componentes de um código

(1)Ter cuidados com nomes que não revelam o propósito.

(2)Ter cuidado ao usar nomes muito parecidos.

(3)Usar nomes pronunciáveis.

(4)Evitar o mapeamento mental.

... mais?

```
8 static public void main(String args[]) {  
9     // a, b1, b2, h correspondem a: área, bases e altura do trapézio  
10    // r é a variável de repetição, q é a quantidade de cálculos  
11    // realizados  
12    double a, b1, b2, h;  
13    int r, q;  
14    // variáveis para cálculos finais:  
15    double acumuladorFinalS, acumuladorFinalM;  
16    a = 0;  
17    q = 0;  
18    acumuladorFinalS = 0;  
19    acumuladorFinalM = 0;  
20    for (r=1;r<=5;r++) {
```

CÓDIGO LIMPO

Nomes Significativos



Nomeando componentes de um código

- 1 - Use nomes que revelem o propósito.

`int et; //tempo decorrido em segundos (início e final da iteração).`

Esse nome não revela nada, talvez o autor do código pensou em “*elapsed time*”.

```
int tempoDecorridoSegundos;  
int tempolnicioFim;
```

- 2 - Evite informações erradas. (MARTIN (1), 2011:19)

`String sisco, hp, aix; //tempo decorrido em segundos (início e final da iteração).`

[...] seriam nomes ruins de variáveis, pois são nomes de plataformas Unix ou variantes. Mesmo se estiver programando uma hipotenusa e hp parecer uma boa abreviação, o nome pode ser mal interpretado. (MARTIN (1), 2011:19).



Nomeando componentes de um código

3 - Cuidado ao usar nomes muito parecidos.

```
double valorTotalDasRsDoOrcamentoAtual; // receitas  
double valorTotalDasDsDoOrcamentoAtual; // despesas  
double valorTotalDosCsDoOrcamentoAtual; // custos
```

- As variáveis são muito semelhantes.
 - Pode levar a uma má interpretação.
- Possibilidade de erro ao escolher a variável errada no recurso auto-completar.

4 - Usar com cuidado as letras “L”, “I” , “O” e “o”

```
int l, O;  
l = 100;  
O = 50  
if (l ==O)  
...  
...
```

c:\prof_gilmar_borba\imagens\auditoria_testes

CÓDIGO LIMPO

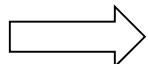
Nomes Significativos



Nomeando componentes de um código

5 - Faça distinções significativas.

```
public static void copiaCaracteres(char c1[], char c2[])
{
    for (int i=0; i<c1.length; i++) {
        c2[i] = c1[i];
    }
}
```



```
public static void copiaCaracteres(char origem[], char destino[])
{
    for (int i=0; i<origem.length; i++) {
        destino[i] = origem[i];
    }
}
```

c:\prof_gilmar_borba\imagens\auditoria_testes



Nomeando componentes de um código

6 - Use nomes pronunciáveis

Nomeamos nossas variáveis, classes, objetos etc. ao longo de todo o código, fazemos isso constantemente ao programar e já que devemos fazer isso é bom fazer **bem feito**. Uma variável de difícil pronúncia, trará problemas nas discussões e decisões sobre o projeto.

7 - Evite codificações

Na atividade de programação lidamos normalmente com codificações; mas códigos, só serviria para aumentar ainda mais possíveis problemas. É bom lembrar que nomes codificáveis são também de difícil pronúncia e escrita. Exemplo: end001_Orc_CadCli, cep002_Rec_CadFor.



Nomeando componentes de um código

8 - Prefixos de variáveis membros

```
private String m_descricaoProcedimento; // descrição do procedimento médico
```

Não é necessário; hoje as “IDEs” já destacam esse tipo de variável. Em um sistema de maior porte a tendência é que, aos poucos, os programadores ignoram esses prefixos, focando apenas na parte representativa do nome. Martin (2011) afirma que:

[...] As pessoas aprendem rapidamente a ignorar o prefixo (ou sufixo) para visualizar a parte significativa do nome. Quanto mais lemos o código, menos prefixos enxergamos. No final, estes se tornam partes invisíveis e um indicativo de código velho. (MARTIN (1), 2011:24).



Nomeando componentes de um código

9 - Evite mapeamento mental

Há nomes, no universo da escrita de códigos, que já estão chancelados pela comunidade de desenvolvedores, por exemplo: as variáveis “i” e “j” usadas nas iterações. Nestes casos, não há motivo para usar “r” ou “x”.

10 - Nomes de Classes e Métodos

Observar que nomes de classes devem representar substantivos, tais como Cliente, Fornecedor, Conta, Receita, Despesa etc. Os métodos devem ser verbos, como por exemplo: salvarDadosVendas, realizarPagamento etc. Sobre nomes de métodos, veja mais informações em: <http://wiki.netbeans.org/NetBeansJavaBeansTutorial>

11 - Selecione uma palavra por conceito

Nomes de métodos e classes devem ser consistentes e ter representatividade por si só. A idéia é, quando for necessário, buscar o nome dessa classe ou método sem a necessidade de uma busca alternativa. Use um contexto significativo.

CÓDIGO LIMPO

Nomes Significativos

Nomeando componentes de um código

Mais ...

(MARTIN, 2011:25-30)

c:\prof_gilmar_borba\imagens\auditoria_testes\



QUESTÕES



(46) O que é um código limpo?

(47) O que você entende por coesão, no contexto da programação orientada a objetos?

(48) O que você entende por acoplamento, no contexto da programação orientada a objetos?

(49) No que se refere à nomenclatura, descreva se os exemplos ao lado foram usados corretamente ou não. Justifique.

- (1) int t; // tempo decorrido em segundos
- (2) double valorTotalDasRsDoOrcamentoAtual; // receitas
double valorTotalDasDsDoOrcamentoAtual; // despesas
- (3) int O1;
float I2;
O1 = 0;
I2 = 1;
- (4) String slang_relatedFinalUser
float bcr33cttdr33;
bcr33cttdr33 = 0.0;
end001_Orc_CadCli
- (5) for (int r=1; r<=100000;r++) {
// faça ...
}
- (6) Public Class RealizarCompraAPrazo {
}
- (7) String EngenheiroSoftware

QUESTÕES



- (50)** Segundo Martin (2011) para nomear corretamente um elemento do código é bom selecionar palavra por conceito ou dentro de um contexto, explique essa diretiva.
- (51)** Segundo Martin (2011) para nomear corretamente um elemento do código é bom evitar trocadilhos, explique essa diretiva.
- (52)** Segundo Martin (2011), é um bom procedimento adicionar um contexto significativo, no que se refere a nomenclatura de variáveis, métodos etc. Justifique.



1 - As funções devem ser pequenas

[...] por cerca de quatro décadas tenho criado funções de tamanhos variados. Já escrevi diversos monstros de 3000 linhas; bastante funções de 100 a 300 linhas; e funções que tinham apenas de 20 a 30 linhas. Essa experiência me ensinou que , ao longo de muitas tentativas e erros, as funções devem ser pequenas. (MARTIN (1), 2011:34).

Cada função deve contar uma história.



COESÃO

É a medida da intensidade da associação funcional dos elementos em um módulo. (PAGE-JONES,1988:128).

[...] É uma medida do quanto fortemente relacionadas e focalizadas são as responsabilidades de uma classe. É extremamente importante assegurar que as responsabilidades atribuídas a cada classe sejam altamente relacionadas. Em outras palavras, o projetista deve construir classes de tal forma que cada uma delas tenha alta coesão. (BEZERRA, 2002:207).

CÓDIGO LIMPO

Funções



2 - As funções devem ter blocos e endentação

As funções não devem ser grandes e ter estruturas aninhadas, é recomendado um nível de endentação de um ou dois (avanços).

3 - Evitar seções dentro das funções.

Quando uma função apresenta várias seções, isso é um indício que aquela função faz mais de uma coisa.

4 - Nomes descritivos.

Lembre do princípio de Ward.

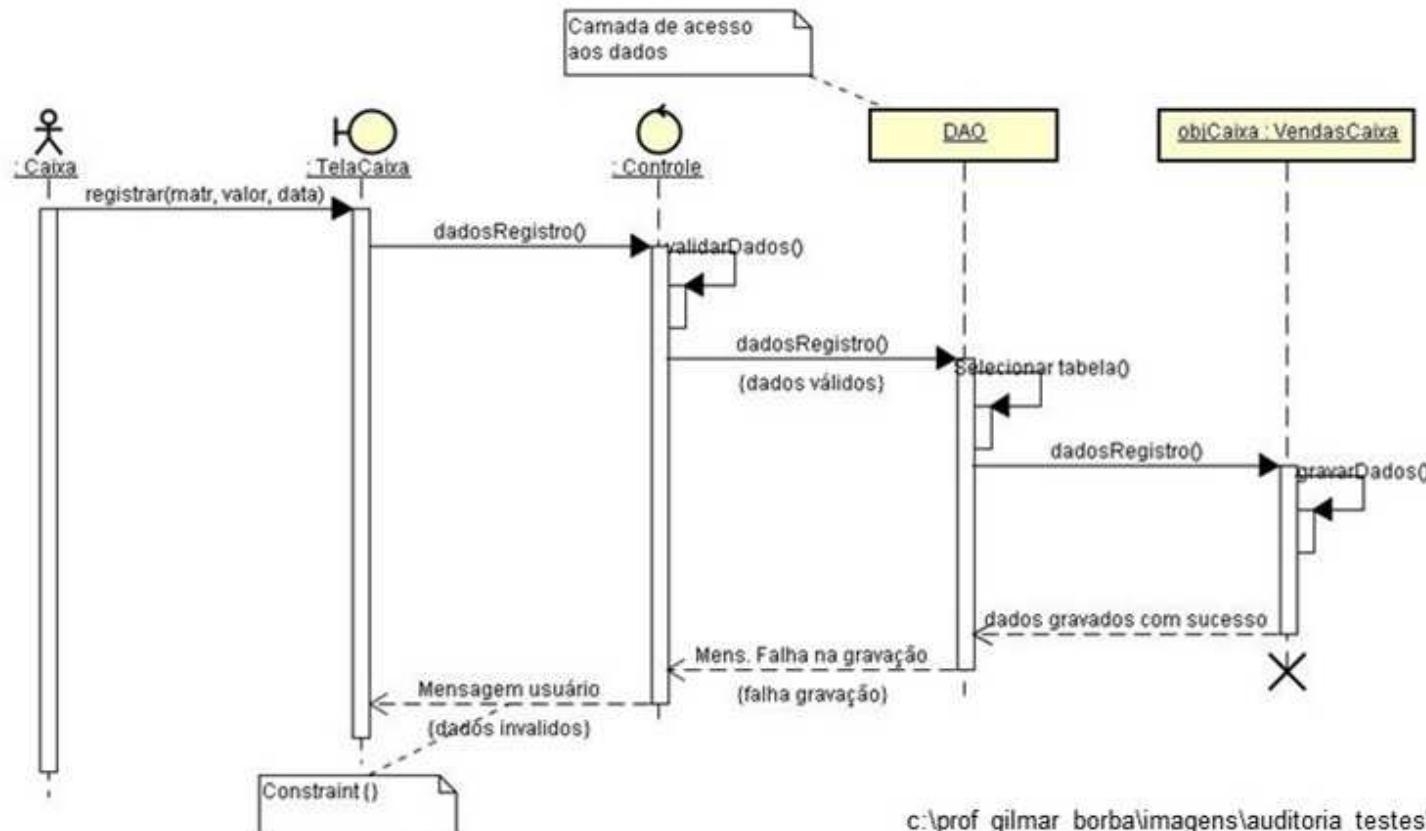
“Você sabe que está criando um código limpo quando cada rotina que você lê é como você esperava”.

CÓDIGO LIMPO

Funções

5 - Verificar o nível de abstração de cada função.

Todas as instruções dentro de uma função deve ter o mesmo nível de abstração.



c:\prof_gilmar_borba\imagens\auditoria_testes\



6 - Parâmetros de funções

Cuidados:

O parâmetro não está no mesmo nível de abstração da função.

Os parâmetros dificultam testes.

Os parâmetros de saída são mais difíceis de entender do que os parâmetros de entrada.

(MARTIN, 2011:40)

Segundo Martin (2011) a quantidade ideal de parâmetros para uma função é Zero!

Mônades: um parâmetro

Díades: dois parâmetros.

Tríades: Três parâmetros. ... Devem ser evitadas.

Mais de três parâmetros não devem ser usadas, a não ser que seja por um motivo muito especial.

CÓDIGO LIMPO

Funções

6 - Evite efeitos colaterais.

Ocorre quando a função promete fazer uma coisa, mas ela também faz outras coisas escondidas.

```
public static void verificarUsuarioSenha(String nome, String senha) {      c:\prof_gilmar_borba\imagens\auditoria_testes\  
    // ...  
    if (usuario.encontrarUsuarioSenha(nome, senha))  
    {  
        System.out.println("Usuario encontrado ...");  
        // Fecha conexão com um banco específico  
        try {  
            con.close();  
        } catch (SQLException e) {  
            System.out.println(e.getMessage());  
        }  
        // ... Inicializa uma nova seção  
        Session.initialize();  
        // ...  
        try {  
            InitialContext ic = new InitialContext();  
            DataSource ds = (DataSource)ic.lookup(dbName);  
            con = ds.getConnection();  
            // ... Verifica novas alterações cadastrais do usuário  
        } catch (Exception e) {  
            throw new Exception("Não foi possível abrir conexão com o banco de dados: " +e.getMessage());  
        }  
        // ...  
    }  
    else  
        System.out.println("> Usuário Não encontrado!!!!");
```



O princípio da Responsabilidade Única

SRP (Single Responsibility Principle)

De acordo com Page-Jones (1988), a coesão é a medida da força de uma associação funcional de atividades de processamento (normalmente dentro de um único módulo). (PAGE-JONES, 1988:381). A coesão de um módulo ou classe pode ser verificada de vários pontos de vista, como por exemplo: funcional, lógico, temporal etc.

Martin (2) (2011), define o Princípio da Responsabilidade Única:

uma classe deve ter apenas um motivo para mudar.

A idéia é: uma função ou classe, uma única responsabilidade!



SRP (Single Responsibility Principle)

Ao implementar uma classe, é importante que esta tenha uma "mentalidade única", ou seja, uma única responsabilidade. Quando uma classe possui muitos objetivos (ou responsabilidades) é importante decompor esses objetivos de forma a ter classes preferencialmente atômicas, Martin (2) (2011) esclarece essa questão:

[...] cada responsabilidade é um eixo de mudança. Quando os requisitos mudarem, a alteração se manifestará por meio de uma mudança na responsabilidade entre as classes. Se uma classe assumir mais de uma responsabilidade, ela terá mais de um motivo para mudar. (MARTIN (2), 2011: 135).

[...] Se uma classe tem mais de uma responsabilidade, as responsabilidades se tornariam acopladas. Mudanças em uma responsabilidade podem prejudicar ou inibir a capacidade da classe de cumprir as outras. Esse tipo de acoplamento leva a projetos frágeis que estragam de maneiras inesperadas quando alterados. (MARTIN (2), 2011: 135).

CÓDIGO LIMPO

Funções



7 - Prefira exceções a retornos de códigos de erros.

A validações com funções que retornam código de erro geram estruturas aninhadas excessivas, violação do princípio da responsabilidade única e um código confuso.

```
package Qualidade;
import javax.swing.*;
/**
 *
 * @author gilmar
 */
public class DivideSemTratamento {

    final static String DIVI_ZERO = "<< Erro de divisão por zero. >>";

    static public int divide(int numerador, int denominador) {
        return (numerador/denominador);
    }

    public static void main(String[] args) {
        int num = Integer.parseInt(JOptionPane.showInputDialog("Numerador: "));
        int den = Integer.parseInt(JOptionPane.showInputDialog("Denominador: "));
        if (den==0) {
            JOptionPane.showMessageDialog(null, "Erro. Mensagem original: "+DIVI_ZERO);
        }
        else {
            float resultado = divide(num, den);
            JOptionPane.showMessageDialog(null, "Resultado: "+resultado);
        }
    }
}
```

CÓDIGO LIMPO

Funções

Prefira exceções a retornos de códigos de erros ... Continuação.

Esses blocos são feios por si só. Eles confundem a estrutura do código e misturam o tratamento de erro com o processamento normal do código. Portanto, é melhor colocar as estruturas *try* e *catch* em suas próprias funções. (MARTIN (1), 2011:46).

```
package Qualidade;
import javax.swing.*;
/**
 *
 * @author gilmar
 */
public class DivideComTratamento {

    static public void divide(int numerador, int denominador) throws ArithmeticException {
        try {
            float resultado = (numerador/denominador);
            JOptionPane.showMessageDialog(null, "Resultado: "+resultado);
        }
        catch (ArithmeticException e) {
            JOptionPane.showMessageDialog(null, "Erro. Possivel divisão por Zero.\n"+
                "Mensagem original: "+e.getMessage());
        }
    }

    public static void main(String[] args) {
        try {
            int num = Integer.parseInt(JOptionPane.showInputDialog("Numerador: "));
            int den = Integer.parseInt(JOptionPane.showInputDialog("Denominador: "));
            divide(num, den);
        }
        catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Erro. Dados não permitidos.\n"+
                "Mensagem original: "+e.getMessage());
        }
    }
}
```

c:\prof_gilmar_borba\imagens\auditoria_testes\

CÓDIGO LIMPO

Funções



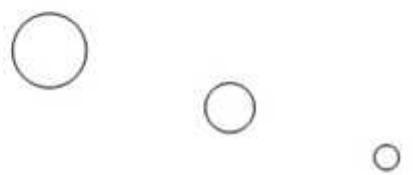
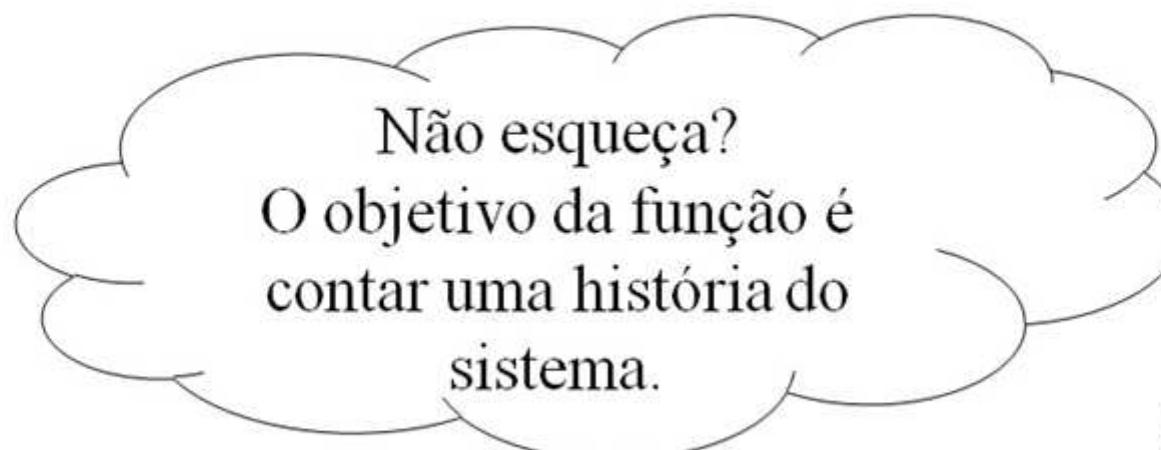
8 - Objetos como parâmetros.

No caso de funções que necessitem de muitos parâmetros, é possível passar um objeto como parâmetro, esse objeto poderia hospedar as propriedades (atributos) necessários.

```
Circulo desenharCirculo(float c1, float c2, float raio);  
Circulo desenharCirculo(Ponto objCentro, float raio);
```

9 - Escolha bons nomes para as funções.

Funções devem ser nomeadas como verbos, de modo a indicar precisamente seu(s) objetivo(s).



QUESTÕES



(53) O que é uma função?

(54) O que é um procedimento?

(55) Baseado no conceito da Coesão (questão 47) porque as funções devem ser pequenas?

(56) Porque as funções devem ter nomes descritivos?

(57) O que é uma função com efeito colateral?

(58) O que você entende por "Princípio da Responsabilidade Única"?

COMENTÁRIOS

“Nada pode ser tão útil quanto um comentário bem colocado” [...]. (MARTIN (1), 2011:53).

1 – Não use comentário para explicar um código ruim.

Não use comentários demais, normalmente quem faz isso, faz para tentar explicar a bagunça feita no próprio código.

2 – Procure se explicar no próprio código.

```
// Verifica descontos para o funcionário baseado nas contribuições de Assistência médica e bla  
bla bla  
if ( funcionario.descontos == DPS && funcionario.descontos == DTE && funcionario.salario >= 10000)  
    if (funcionario.idade <=35 ...
```

Uma solução:

```
if ( funcionario.contribuinteParaDescontos())
```

COMENTÁRIOS

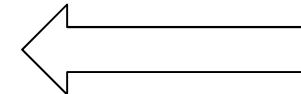


3 – Comentários legais e comentários informativos.

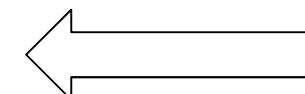
Direitos autorais, Data de implementação/alteração podem ser inseridos no cabeçalho de um código.

```
/*
Programa.....: SMPJ3050
Descricao ..: Emissão de carteira
Sistema.....: SAIJ - Sistema de Apoio a Informação-Jurídica
Arq./dados...: FI000000/FI100000/FI200000'
Desenvolvimento...: Gilmar Luiz
Data.....: Maio de 1993 - Plano de Saúde

Alterado por ..:
Data ..:
*/
```



```
// Retorna a instância da resposta tratada
Protected abstract RespostaMensagem instanciaMensagem();
```



Nesse caso, estamos fornecendo informações básicas sobre um determinado trecho do código.

COMENTÁRIOS



4 – Alerta sobre consequências.

Este é um tipo válido de comentário. Alertar outros programadores sobre um processo que poderá causar eventuais danos ou como no exemplo, uma demora excessiva no processamento. (MARTIN (1), 2011:58).

COMENTÁRIOS

4 – Alerta sobre consequências.

Este é um tipo válido de comentário. Alertar outros programadores sobre um processo que poderá causar eventuais danos ou como no exemplo, uma demora excessiva no processamento.
(MARTIN (1), 2011:58).

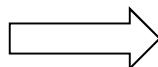
```
// ATENÇÃO:  
// Não execute essa função a menos que  
// você tenha muito tempo disponível.  
// O arquivo FATOSVENDAS possui alguns  
// Teras de dados.  
public class testarArquivoArmazemDados {  
    try  
    {  
        con.Open();  
        SqlCommand cmd = new SqlCommand(comando, con);  
        //...  
        while( ... )  
        {  
            .... comandos  
        }  
    }  
    finally  
    {  
        con.Close();  
    }  
}
```

COMENTÁRIOS

5 – JavaDocs.

- São bons para auxiliar a documentação de APIs públicas. Documentação bem apresentada e automatizada.

... porém



- Podem ser enganadores.

- Não há uma regra para o seu uso.

- Usar com moderação.

```
public class Estatistica {  
    String verbo;  
    String pluralSingular;  
    /**  
     * <b>Esse método tem como objetivo processar, classificar e imprimir os  
     * dados estatísticos de acordo com cada condição</b>  
     * @param Produto  
     * @param Quantidade  
     * @return Uma string  
     * @see A função processaImprimeEstatistica retorna um string com  
     * @see Quantidade, produto, verbo.  
     */  
    public String processaImprimeEstatistica(String produto, int quantidade) {  
        processaCasosParaImpressao(quantidade);  
        // Imprime resultado  
        return String.format(" => %s %s %s",produto,verbo, quantidade,pluralSingular);  
    }  
}
```

c:\prof_gilmar_borba\imagens\auditoria_testes\

COMENTÁRIOS



COMENTÁRIOS

6 – Evitar ...

Murmúrios: comentários desnecessários, feitos a partir do sentimento imediato do programador e que ficará, posteriormente, como um enigma dentro do código.

```
// Gilmar: não esquecer, verificar se a tabela já está povoada ou vazia ...
// A tabela é a seguinte (ReltableTemp) é usada para este relatório e para o
// relatório de Conferência do Orçamento
// Esta tabela será usada somente uma vez ...
```

Comentários redundantes: veja o exemplo:

```
// Essa função processa, classifica e imprime mensagem padronizada.
public String processarClassificarImprimirMensagemPadronizada(String descricao,
    double valor) {
    ...
}
```

Comentários longos: evitar de explicar muito, a partir de um comentário, toda vez que o módulo for alterado.

COMENTÁRIOS

6 – Evitar ...

Murmúrios: comentários desnecessários, feitos a partir do sentimento imediato do programador e que ficará, posteriormente, como um enigma dentro do código.

```
// Gilmar: não esquecer, verificar se a tabela já está povoada ou vazia ...
// A tabela é a seguinte (RelatableTemp) é usada para este relatório e para o
// relatório de Conferência do Orçamento
```

Comentários redundantes: veja o exemplo:

```
// Essa função processa, classifica e imprime mensagem padronizada.
```

```
public String processarClassificarImprimir MensagemPadronizada(String descricao, double valor) {
    ...
}
```

Comentários longos: evitar de explicar muito, a partir de um comentário, toda vez que o módulo for alterado.

COMENTÁRIOS

// Alterações no programa de atualização bancária:

//

// 14/02/2006: Implementado "stub" do módulo após reunião com Vitor e Glória (financeiro). foram implementados também o chamador das principais funções. Implementação feita por Naziazeno.

//

// 27/02/2006: Término da codificação dos módulos pendentes. foi decidido após reunião que todos os lançamentos das contas devem ter origem na finalização (efetiva) das vendas. Verificar "flag" de finalização. Alteração feita por Gilmar Luiz.

//

// 12/05/2006: Alteração feita para limpar os campos após realização da pesquisa por numero da conta, data do lançamento e periodo. Também foi implementada a exclusão do lançamento corrente após rotina de atualização. Alteração feita por Gilmar Luiz.

//

// 03/03/2007: Finalizado solicitação número 047/07: O módulo agora verifica se o lançamento possui contrapartida. Anteriormente o registro ficava marcado como contrapartida e após novo lançamento esse histórico era descartado. Solicão: Herbert, a pedido da Bernadete do Contas a Pagar. Alteração feita por Zemprônia.

//

// 07/07/2007: Alteração da última implementação feito pela Zemprônia. Se o lançamento tiver contrapartida o histórico deverá ser excluído automaticamente. Não tem sentido mais esse histórico, se continuar, teremos informações redundantes no banco. eu!

//

// 12/05/2008: Codificação de alteração: se o lançamento bancário é do tipo "C" (cheque) não poderá ocorrer a exclusão deste lançamento. Deverá ser usada a operaçao de exclusão de lançamento de cheque pelo módulo de Exclusão de cheque. Esse módulo está sendo desenvolvido, ver solicitação 078/08. Qualquer dúvida, falar com o Naziazeno. eu!

//

// 15/06/2009: Implementado função para lançar contas no centro de custo. Gilmar Luiz.

//

// 25/08/2009: Desfeito saldo de lançamento, implementado anteriormente. Todos os lançamentos que foram gerados após emissão do cheque não devem ser mais excluidos. Solicitação Urgento do Ticio (nossa melhor cliente). eu!

//

// ...

c:\prof_gilmar_borba\imagens\auditoria_testes\

COMENTÁRIOS

Evitar ...

Comentários ao lado de chaves de fechamento: Esse tipo de comentário faz sentido em funções longas com uma quantidade razoável loops aninhados, mesmo assim, deve ser usado com cautela para evitar o amontoado de comentários desnecessários.

```
/**  
 * @see TestarEstatistica  
 * @author Gilmar.  
 * @param quantidade  
 */  
public void processaCasosParaImpressao(int quantidade) {  
    if (quantidade==0) {  
        naoHaProduto();  
    } // fim se quantidade == 0 ←  
    else if (quantidade==1) {  
        haUmProduto();  
    } // fim se quantidade == 1 ←  
    else {  
        haMuitosProdutos();  
    } // fim se quantidade != 0 e != 1 ←  
}
```

c:\prof_gilmar_borba\imagens\auditoria_testes\

COMENTÁRIOS



Evitar ...

Códigos comentados: **condenável!!!**

```
// inputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

“Outros que vissem esse código não teriam coragem de excluir os comentários.” (MARTIN (1), 2011:68).

Comentário em HTML ...

COMENTÁRIOS

Evitar ...

Comentário em HTML: Uma aberração! Comentários em HTML dificultam a leitura dos comentários onde deveria ser fácil a sua leitura e entendimento.

```
/**  
 * <font face = arial color=green size=8>Comentário</font>  
 * <br><b>Esse método tem como objetivo processar, classificar</b>  
 * <br><b> e imprimir os dados estatísticos de acordo com cada</b><br>  
 * <b>condição</b><br>  
 * @author gilmar  
 * @param Produto Quantidade  
 */
```



Conclusões

“Comentários não são como a Lista de Schindler. Não são o “bom puro” de fato, eles são, no máximo, um mal necessário.”

(MARTIN (1), 2011:53)

QUESTÕES



(59) Informe duas situações que exemplificam a segunda parte da citação abaixo:

“Nada pode ser tão útil quanto um comentário bem colocado. Nada consegue amontoar um módulo mais do que comentários dogmáticos e supérfluos. Nada pode ser tão prejudicial quanto um velho comentário mal feito que dissemina mentiras e informações incorretas.”

(MARTIN (1), 2011:53).

(60) Forneça um exemplo de comentário legal

(61) Com relação ao uso de comentários, analise e comente os dois códigos a seguir:
CÓDIGO1

```
// Verifica descontos para o funcionário baseado nas contribuições de Assistência  
// médica baseado no salário bruto e renda anual.
```

```
if (funcionario.descontos == DPS && funcionario.descontos == DTE && funcionario.salario >= 10000)  
    if (funcionario.idade <=35 ...
```

CÓDIGO2:

```
if (funcionario.eContribuinteParaDescontos())
```

...

QUESTÕES



(62) Discorra sobre o uso da linguagem HTML para comentar trechos de código em outra linguagem como Java ou C#.

(63) Comente sobre as vantagens e desvantagens do JavaDocs.

(64) O comentário inserido está adequado? justifique.

```
// Essa função processa, classifica e imprime mensagem padronizada.  
public String processarClassificarImprimir MensagemPadronizada(String descricao, double valor) { ... }
```

(65) O comentário inserido está adequado? justifique.

```
// retorna o dia do mês ....  
public String retornaDia(Date data);
```

FORMATAÇÃO



“Primeiro de tudo, sejamos claros. A formatação do código é importante. Importante demais para se ignorar e importante demais para ser tratada religiosamente. Ela serve como uma comunicação, e essa é a primeira regra nos negócios de um desenvolvedor profissional. (MARTIN (1), 2011:76).

Formatação vertical

- Depende do tamanho das classes.
- Arquivos pequenos costumam ser mais fáceis de se entender.



A metáfora do Jornal

(MARTIN (1), 2011:77)

O código fonte deve ser como um artigo de jornal, o nome deve ser simples mas descritivo. Deve ser suficiente para dizer se estamos no módulo certo ou não.

Espaçamento vertical entre conceitos

A maioria dos códigos é lida de cima para baixo, da direita para esquerda.

FORMATAÇÃO

```
/**  
 * Teste Observer  
 * @author gilmar  
 */  
import java.util.Observable;  
public class DadosDoTempo extends Observable {  
    // (1) Cria uma variável de instância "temperatura"  
    private double temperatura;  
    public DadosDoTempo() {  
        temperatura = 36;  
    }  
    ←  
    // (2) Faz uma chamada a setChange para indicar que o  
    //      estado mudou, antes de chamar NotifyObservers()  
    public void medidasAlteradas() {  
        // (3) Indica simplesmente que algo mudou  
        setChanged();  
        System.out.println("Medida alterada...");  
        // (4) Notifica os observadores quando alguma mudança  
        //      ocorreu  
        notifyObservers();  
        System.out.println("Observador notificado ...");  
    }  
    ←  
    public void setTemperatura(double temperatura) {  
        this.temperatura = temperatura;  
        System.out.println("TEMPERATURA NO SETTEMPERATURA: "+  
                           temperatura);  
        medidasAlteradas();  
    }  
    ←  
    public double getTemperatura() {  
        System.out.println("TEMPERATURA NO GETTEMPERATURA: "+  
                           temperatura);  
        return temperatura;  
    }  
}
```



Declaração de variáveis

As variáveis devem ser declaradas o mais próximo possível de onde serão usadas. No caso de funções pequenas, as variáveis locais devem ficar no topo de cada função.

Declaração de variáveis de instância

Devem ser declaradas no início da classe.



Funções dependentes

“Se uma função chama a outra, elas devem ficar verticalmente próximas e a que chamar deve ficar acima da que for chamada, se possível. Isso dá um fluxo natural ao programa. Se essa convenção for seguida a fim de legibilidade, os leitores poderão confiar que as declarações daquelas funções virão logo em seguida após o seu uso. (MARTIN (1), 2011:82).

FORMATAÇÃO

Funções dependentes

c:\prof_gilmar_borba\imagens\auditoria_testes\

```
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    // ...
    public Response makeResponse(FitNesseContext context, Request request) throws Exception {
        String pageName = getPageNameOrDefault(request, "FrontPage");
        loadPage(pageName, context);
        if (page==null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }

    private String getPageNameOrDefault(Request request, String defaultPageName) {
        // ...
        return pageName;
    }

    protected void loadPage(String resource, FitNesseContext context) throws Exception {
        // ...
    }

    private Response notFoundResponse(FitNesseContext context, Request request) throws Exception {
        // ...
    }
    // ...
}
```

Fonte: (MARTIN, 2011:82-83)

FORMATAÇÃO



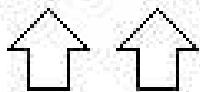
Formatação horizontal

Qual deve ser o tamanho de uma linha de código? “Programadores claramente preferem linhas curtas.” (MARTIN (1), 2011:85). Espaçamento contínuo horizontal

Verificar espaçamento entre atribuições, operadores de atribuição, destacando o lado esquerdo e direito da escrita.

Formatação horizontal

```
private void MedidaLinha(String linha) {  
    contadorLinha++;  
    int tamanhoLinha = linha.length();  
    totalCaracteres += tamanhoLinha;  
    //...  
}
```



FORMATAÇÃO



Espaçamento contínuo horizontal (continuação)

Usar espaçoamento para destacar certas estruturas, muitas vezes prejudica a interpretação e afasta os olhos do propósito real.

```
public class Funcionario {  
    private String nome;  
    private String endereco;  
    private String cep;  
    // ...  
}
```

Endentação

Deve ser feita de acordo com a hierarquia, por exemplo: classes individuais dentro de arquivos, métodos dentro das classes, blocos dentro dos métodos e assim sucessivamente. A ideia é tornar visível essa hierarquia.

FORMATAÇÃO



Endentação

Deve ser feita de acordo com a hierarquia, por exemplo: classes individuais dentro de arquivos, métodos dentro das classes, blocos dentro dos métodos e assim sucessivamente.

A idéia é tornar visível essa hierarquia. Verifique os dois exemplos de código:

```
while (!key(KEY_ESC)) {  
    rest(10);  
    if(sair==true){  
        return 0;}  
    if(morreu == true){  
        stop_sample (som_w);  
        SAMPLE *som_f = load_wav("final.wav"); // Faz o upload da música do jogo.  
        play_sample (som_f, 255, 128, 1000, 0); //Carrega a musica do jogo.  
        _sleep( 1000 * 10 );  
        break;} else {  
        atualizar();  
        renderizar();}}
```

gilmar

FORMATAÇÃO



Endentação (continuação)

```
while (!key(KEY_ESC)) {
    rest(10);
    if(sair==true){
        | return 0;
    }
    if(morreu == true){
        | stop_sample (som_w);
        | SAMPLE *som_f = load_wav("final.wav"); // Faz o upload da música do jogo
        | play_sample (som_f, 255, 128, 1000, 0); //Carrega a musica do jogo.
        | _sleep( 1000 * 10 );
        | break;
    } else {
        | atualizar();
        | renderizar();
    }
}
```

gilmar

Fonte: Projeto Jogo Thriller da Gripe. Desenvolvido pela equipe de alunos de ADS: Marina, Luan, Samuel, Gustavo, Leonardo, Rômulo e Davison, do Centro Universitário UNA, primeiro semestre de 2009, usando C++ e biblioteca Allegro.h.



Conclusões

- As regras de formatação devem ser da equipe.
- A equipe deve escolher um estilo de formatação e todos devem usá-lo.

As decisões envolvem itens como (entre outros): como será o estilo de indentação, como serão nomeadas as variáveis, classes, funções, em quais situações serão usados os comentários etc.

- Códigos fontes com estilos diferentes só dificultarão o seu entendimento.
- Um código de qualidade possui uma série de documentos de fácil leitura.

QUESTÕES



(66) Descreva como devem ser declaradas as variáveis no contexto da formatação.

(67) Descreva como devem ser implementadas as funções dependentes no contexto da formatação.

(68) O que você entende por formatação horizontal? Qual deve ser o tamanho ideal desse tipo de formatação em um código?

(69) O que você entende por formatação vertical?

(70) Como devem ser feitas as regras de formatação de código por uma equipe? Faça uma breve descrição.

Tratamento de Erros



“[...] O tratamento de erro domina completamente muito código fonte. Quando digo que “domina”, não quero dizer que eles só fazem tratamento de erro, mas que é quase impossível ver o que o código faz devido a tantos tratamentos de erros espalhados. Esse recurso é importante, mas se obscurecer a lógica, está errado. (MARTIN (1), 2011:103).

“Há muitos meios populares de lidar com erros. Mais comumente, o código de tratamento de erros é intercalado ao longo de todo o código de um sistema. Os erros são tratados nos lugares dentro do código em que os erros podem ocorrer. A vantagem dessa abordagem é que um programador que lê código pode ver o processamento de erros na vizinhança imediata do código [...]” (DEITEL, 2002:740).

Tratamento de Erros

“[...] O problema com esse esquema é que o código pode se tornar “poluído” com o processamento de erros. Fica mais difícil para um programador preocupado com o aplicativo em si ler o código e determinar se ele está funcionando corretamente.” (DEITEL, 2002:740).

Considerações Iniciais

- Incorporar uma estratégia de tratamento de exceções no início do projeto.
- Exceções são usadas para lidar com situações excepcionais, que o código não consegue controlar.
- Exceções são usadas para lidar com componentes de código (por exemplo, de terceiros) que não foram projetados para tratar exceções.
- O tratamento de exceções é relativamente lento em comparação com tratamento de erros locais.

Blocos try / catch / finally (na linguagem JAVA)

Tratamento de Erros



Blocos try / catch / finally (na linguagem JAVA)

```
try {  
    instruções possíveis de disparar uma condição de exceção  
}  
catch(Tipo referência) {  
    instruções para processar a exceção  
}  
finally {  
    liberação de recursos  
}
```

Use exceções ao invés de retornar códigos

- Exceções são recursos não encontrados em linguagens antigas.
- Verificações de erros eram feitas ao longo do código, imediatamente após a chamada do código.

Tratamento de Erros

```
package Qualidade;
import javax.swing.*;
/**
 *
 * @author gilmar
 */
public class DivideSemTratamento {

    final static String DIVI_ZERO = "<< Erro de divisão por zero. >>";

    static public int divide(int numerador, int denominador) {
        return (numerador/denominador);
    }

    public static void main(String[] args) {
        int num = Integer.parseInt(JOptionPane.showInputDialog("Numerador: "));
        int den = Integer.parseInt(JOptionPane.showInputDialog("Denominador: "));
        if (den==0) {
            JOptionPane.showMessageDialog(null, "Erro. Mensagem original: "+DIVI_ZERO);
        }
        else {
            float resultado = divide(num, den);
            JOptionPane.showMessageDialog(null, "Resultado: "+resultado);
        }
    }
}
```

c:\prof_gilmar_borba\imagens\auditoria_testes\

Tratamento de Erros



Use exceções ao invés de retornar códigos

```
package Qualidade;
import javax.swing.*;
/**
 *
 * @author gilmar
 */
public class DivideComTratamento {

    static public void divide(int numerador, int denominador) throws ArithmeticException {
        try {
            float resultado = (numerador/denominador);
            JOptionPane.showMessageDialog(null, "Resultado: "+resultado);
        }
        catch (ArithmeticException e) {
            JOptionPane.showMessageDialog(null, "Erro. Possivel divisão por Zero.\n"+
                "Mensagem original: "+e.getMessage());
        }
    }

    public static void main(String[] args) {
        try {
            int num = Integer.parseInt(JOptionPane.showInputDialog("Numerador: "));
            int den = Integer.parseInt(JOptionPane.showInputDialog("Denominador: "));
            divide(num, den);
        }
        catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Erro. Dados não permitidos.\n"+
                "Mensagem original: "+e.getMessage());
        }
    }
}
```

Tratamento de Erros



Criar um “template” try-catch-finally

É importante criar um escopo try-catch-finally quando há a suspeita que o código pode lançar alguma exceção; o código dentro da parte `try` indica o que será realizado pelo programa, a parte `catch` assegura que o programa ficará em um estado consistente caso algo de errado ocorra; o código posicionado na parte `finally` será obrigatoriamente executado, normalmente ele é usado para liberação de recursos.

Exceções verificadas

“Se você lançar uma exceção a ser verificada a partir de um método em seu código e o método catch estiver três níveis acima, será preciso declará-la na assinatura de cada método entre você e o catch. Isso significa que uma modificação em um nível mais baixo do software, pode forçar a alteração de assinaturas em muitos níveis mais altos. (MARTIN (1), 2011:107).

Isso é uma violação do princípio Aberto Fechado.

Tratamento de Erros



O Princípio Aberto-Fechado (Open-Closed Principle)

As classes devem estar abertas para a extensão, mas fechadas para modificação.

Nosso objetivo é permitir que as classes sejam facilmente estendidas para incorporar um novo comportamento sem modificar o código existente. O que conseguimos se fizermos isso? Projetos que são resistentes a mudanças e suficientemente flexíveis para assumir novos recursos para atender aos requisitos que mudam. (FREEMAN, 2007:87).

Tratamento de Erros

Exceções devem ser apresentadas dentro do contexto

Sempre que possível, criar mensagens de erro informativas e passá-las juntamente com a exceção. É de bom procedimento mencionar a operação que falhou e o tipo de falha ocorrida.

Não retorne *null* e não passe *null* como parâmetros

Esse cuidado poderia ficar entre os primeiros da lista, quando falamos de exceções e código limpo.

“retornar *null* dos métodos é ruim, mas passar *null* é pior ainda”. (MARTIN (1), 2011:111).

Tratamento de Erros



Não retorne *null* e não passe *null* como parâmetros (continuação)

```
public void registerItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.getItemRegistry();  
        if (registry != null) {  
            Item existing = registry.getItem(item.getID());  
            if (existing.getBillingPeriod().hasRetailOwner()) {  
                existing.register(item);  
            }  
        }  
    }  
}
```

(MARTIN, 2011:110)

O que aconteceria, em tempo de execução, se *persistentStore* fosse *null*?

Tratamento de Erros

Não retorne *null* e não passe *null* como parâmetros (continuação)

```
public class MetricsCalculator {  
    public double xProjection(Point p1, Point p2) {  
        return (p2.x-p1.x)* 1.5;  
    }  
}  
...  
calculator.xProjection(null, new Point(12,13)):
```



(MARTIN, 2011:111)

O que aconteceria, em tempo de execução, com a chamada do método *xProjection* acima?

Tratamento de Erros



Conclusões

- Um código limpo deve ser legível, não deve confundir o desenvolvedor, deve ter nomes significativos etc. mas também deve ser robusto.
- Ser legível e elegante não deve conflitar com sua robustez.
- Tratamentos de erros devem ser enxergados independente da lógica principal do programa.

QUESTÕES



- (71)** Qual é a desvantagem em ter o código de tratamento de erros juntamente com a própria lógica do negócio?
- (72)** Quando devem ser usadas as exceções?
- (73)** Explique os blocos try/catch/finally.
- (74)** Explique o princípio OCP (Open Closed Principle), ou Princípio Aberto Fechado. Porque ele é importante para um código com qualidade?



TDD

Test Driven Development



Introdução

"I like the notion of working the program, like an artist works a lump of clay.."
Ward Cunningham

TDD



Introdução

- Em meados da década de 1990, não se ouvia falar em TDD (Test Driven Development).
- Os testes eram feitos manualmente e em seguida o código era descartado.
- Atualmente há várias ferramentas que automatizam os testes unitários.

As três leis do TDD

Primeira lei:

. Não se deve escrever qualquer código de produção antes de escrever um teste de unidade de falhas.

Segunda lei:

. Não se deve escrever mais que um teste unitário do que o necessário para falhar, não compilar é falhar.

Terceira lei:

. Não deve escrever mais códigos de produção do que o necessário para aplicar o teste de falha atual.

TDD

SIMPLICIDADE

A quantidade de código de teste é tão significativa quanto o código de produção. Por esse motivo, devemos manter os testes limpos.

“[...] Os códigos de testes são tão importantes quanto o código de produção. Ele não é um componente secundário. Ele requer raciocínio, planejamento e cuidado. É preciso mantê-lo tão limpo quanto o código de produção. (MARTIN (1), 2011:124).

Quanto maior for a cobertura dos testes, melhor será → Simplicidade, portabilidade, legibilidade, manutenibilidade etc. do produto final.

Os testes devem ser simples e objetivos

“[...] vão direto ao ponto e usam apenas os tipos de dados e funções que realmente precisam”. (MARTIN (1), 2011:127).

TDD

FEEDBACK

Testes devem ser fácil e rápido de entender. Deve ser usado, quando possível, uma afirmação por teste.

“[...] Há uma escola de pensamento que diz que cada função de teste em um teste do Junit deve ter um e apenas uma instrução de afirmação (assert). (MARTIN (1), 2011:130).

Os códigos de testes e códigos de produção.

Os testes são tão importantes no projeto do que o próprio código de produção. Por esse motivo é necessário manter os testes limpos e objetivos.

... Os testes e o código de produção são escritos juntos, os testes apenas alguns segundos mais adiantados. (MARTIN, Robert, 2011, 23)

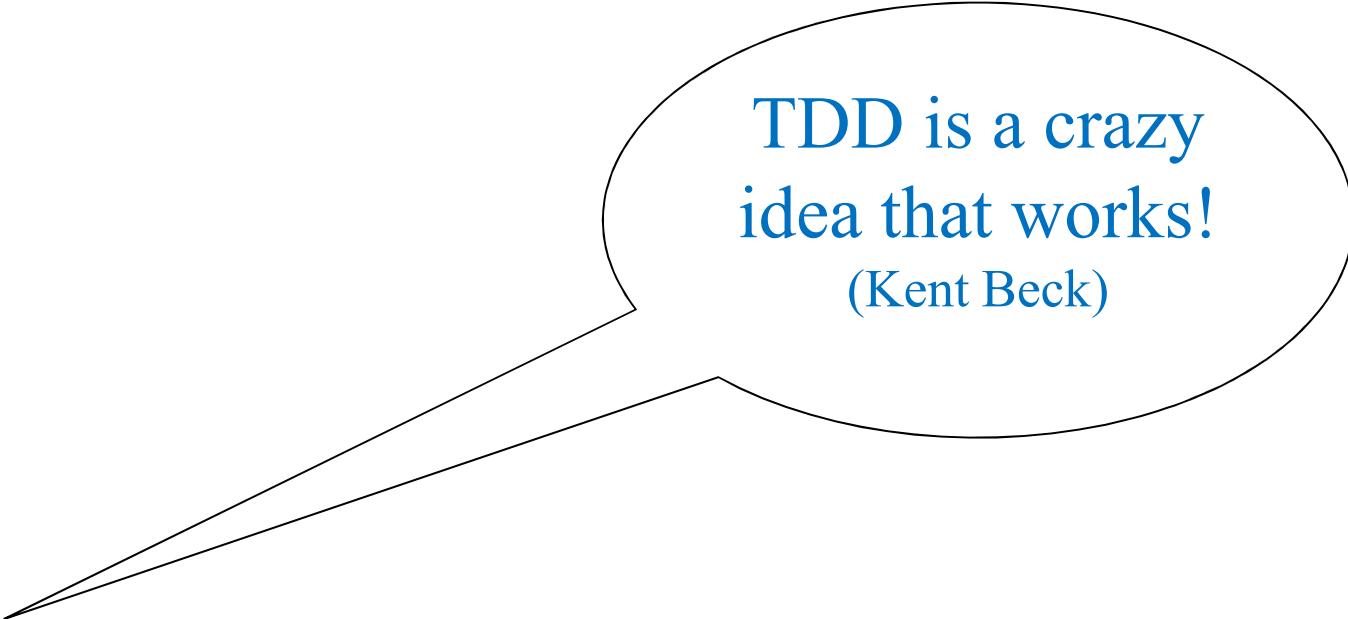
Com a prática, terminará mais de 20 ciclos em uma hora. Porém, não se concentre demais em sua velocidade. Isso poderá tentá-lo a pular a refatoração [...]. (SHORE, James; WARDEN, Shane, 2008, 300)



Outras definições

TDD

É uma técnica de desenvolvimento e projeto na qual os testes são criados antes do código de produção.



**TDD is a crazy
idea that works!**
(Kent Beck)

O desenvolvimento orientado a testes, ou TDD (em inglês), é um ciclo rápido de testes, codificação e refatoração. Ao adicionar um recurso, uma dupla pode fazer dezenas destes ciclos, implementando e refinando o software em pequenos passos, até que não haja nada a ser adicionado e nada mais a ser removido.

(Shore, Warden, 2008 apud Jansen, Saiedian, 2005)

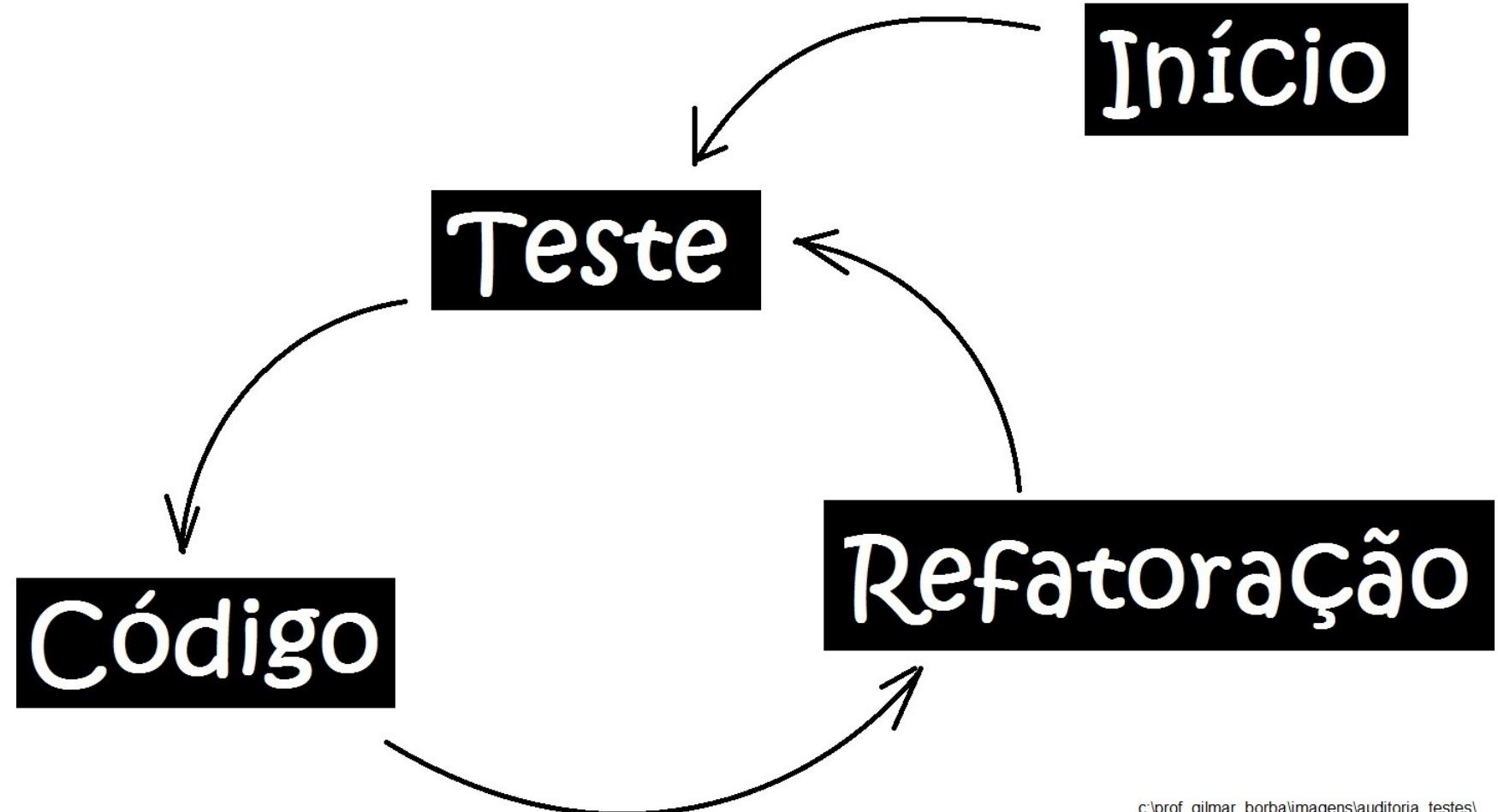
Ciclo



"So today, let's write a program simply. But let's also realize that tomorrow, we're going to make it more complex, because tomorrow it's going to do more".

Ward Cunningham

O ciclo do TDD



c:\prof_gilmar_borba\imagens\auditoria_testes\

O ciclo do TDD



c:\prof_gilmar_borba\imagens\auditoria_testes\

TDD – Código de Teste

“Após o término do TDD, os testes permanecem. Eles são verificados com o restante do código e agem como uma documentação viva do código. O mais importante é que os programadores façam todos os testes com (quase) todos os aplicativos, garantindo que o código continue funcionando como se intencionava originalmente. Se alguém accidentalmente mudar o comportamento do código – por exemplo com uma refatoração mal produzida – os testes falham, sinalizando o erro.
(Shore, Warden, 2008).



Código Limpo
Habilidades práticas do Agile Software.
Robert C. Martin - 2011



O ciclo do TDD

Red – Green – Refactor – Red – Green –
Refactor – Red - Green – Refactor – Red – Green
– Refactor – Red - Green – Refactor – Red –
Green – Refactor – Red - Green – Refactor – Red
– Green – Refactor – Red –

TDD – Como funciona

PASSO 1 – Adicionar o teste

- Código existe
- Código não existe
- API
- Pair Programming

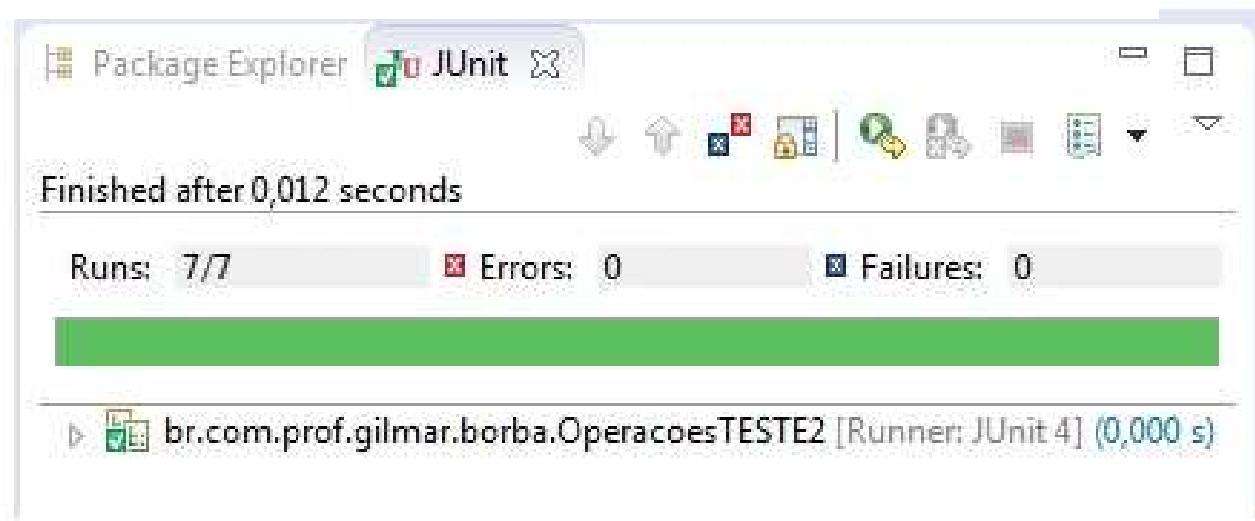
TDD – Como funciona

PASSO 2 – Fazer o teste passar

- Codificar em termos de comportamento
- Fazer o teste ficar verde
- Simplicidade, pequenos incrementos
- Você ainda não sabe os nomes dos métodos, classes, pensar na API



TDD – Como funciona



PASSO 3 – Refatorar

- Limpar o código
- Verificar as possibilidades de refatoração
- variável, constante, métodos, super classe etc.
- Questionamentos

TDD – Como funciona



PASSO 4 – Baby Steps

- O segredo é permanecer no controle
- Pequenos incrementos
- Alternância entre código de teste e código de produção
- Tem a ver com ritmo

TDD – Como funciona



PASSO 5 – Rodar os testes com frequência (automatizados)

- Não se concentrar na velocidade
- Cuidado para não pular a refatoração
- Criar “suites” para automatizar os testes e aplicá-los

sempre que necessário

TDD – Como funciona

Cria a classe
que faz a
conexão com o
banco ...

Cria a classe que
implementa aquela
regra de negócio ...

Vamos
testar?

Refatora!

Aplica o
padrão X
ai ...

Desacopla a
classe para
fazer o teste!

TDD x BDD



BDD um acrônimo para *Behavior Driven Development*, que significa desenvolvimento orientando a comportamentos.

Enquanto o TDD permite a escrita de testes e os valida de forma que eles funcionem, o **BDD**, permite escrever o comportamento, ou o problema (oriundo dos requisitos). O **BDD** é mais amigável para o usuário final.

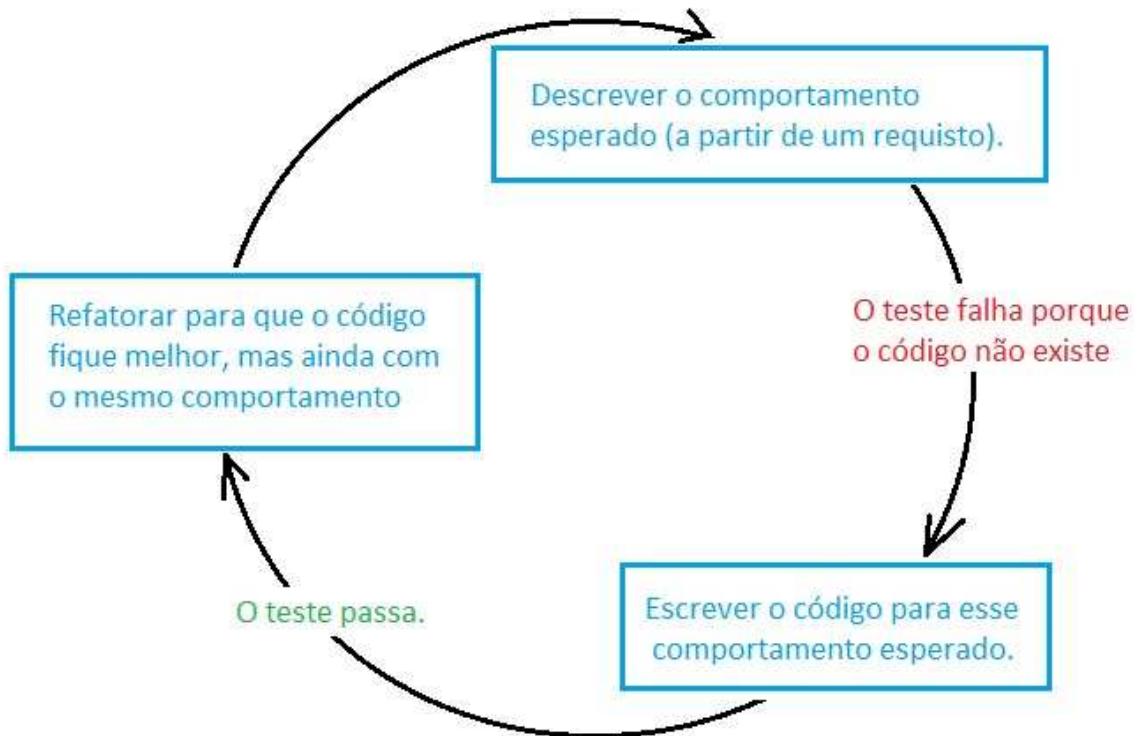
TDD x BDD



Existem frameworks específicas (por exemplo *RSpec*, *Cucumber* e o *PHP Spec*) que permitem que sejam descritos os requisitos (usando uma pseudo linguagem) informando como as coisas devem se comportar.

As ferramentas usadas no mercado permitem:

- 1)** escrever uma especificação em uma “pseudo linguagem”;
- 2)** transformar a especificação em uma função de teste. Isso porque esses frameworks implementam o TDD para guiar a realização do BDD.



TDD x BDD

O **BDD** é, de certa forma, uma evolução natural do TDD, as definições inseridas no BDD acabam sendo testadas efetivamente usando o TDD.

No **BDD** o desenvolvimento é guiado pelos comportamentos que o sistema apresenta. Assim, esse comportamento é priorizado em relação ao teste unitário. O **BDD** não exclui a execução do TDD.



Questões para discussão

TDD



(75) Test Driven Development é uma metodologia específica de teste de software? Justifique.

(76) Como o TDD funciona? Justifique a partir das 3 principais etapas desta técnica de design.

(77) Explique o termo Baby Steps.

(78) Por que aplicar os teste de unidade ao invés de um teste geral/único? Justifique.

(79) O desenvolvedor nunca deve testar seu próprio código? Justifique.

(80) É mais interessante usar o TDD para classes existentes ou para classes que ainda não foram criadas? Justifique.

TDD



- (81) Por que aplicar os teste de unidade ao invés de um teste geral/único? Justifique.
- (82) O desenvolvedor nunca deve testar seu próprio código? Justifique.
- (83) É mais interessante usar o TDD para classes existentes ou para classes que ainda não foram criadas? Justifique.
- (84) Uma questão para reflexão: Todo código com problema (erro ou falha) tem um culpado (cliente ou desenvolvedor), de quem é a culpa?
- (85) Se o design fica por conta dos programadores como que você controla a qualidade do código?
- (86) Qual é a diferença entre BDD e TDD

REFERÊNCIAS PRINCIPAIS



Referências Principais

DELAMARO , Eduardo, MALDONADO José Carlos, JINO Mário (Organizadores). Introdução ao teste de software. – Rio de Janeiro : Elsevier, 2007.

FOWLER, Martin. Refatoração: Aperfeiçoamento e Projeto. Editora Bookman, 2004. 365 páginas

MOLINARI, Leonardo. Teste de Software – Produzindo Sistemas Melhores e Mais Confiáveis. – São Paulo : Editora Érica, 2003.

RIOS, Emerson; Moreira, Trayahú Filho. Teste de Software. - Rio de Janeiro, RJ : Alta Books, 2013.

SOMMERVILLE, Ian. Engenharia de Software. – 9. ed. – São Paulo : Pearson Prentice Hall, 2011.

Referências Relevantes

CÔRTES, Mário Lúcio. Modelos de Qualidade de Software. Editora da Unicamp, 2001.

LARMAN, Craig. Utilizando UML e padrões. Uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. Editora Bookman, 2007.

MARTIN, Robert C. [et al]. Código Limpo - Habilidades práticas do Agile Software. – rio de Janeiro, RJ. Alta Books, 2011.

WEBER, Kival Chaves; ROCHA, Ana Regina Cavalcanti da. Qualidade E Produtividade Em Software. 3. ed. São Paulo: Makron Books, 1999

ROCHA, Ana Regina Cavalcanti. Qualidade de Software. Editora: Makron Books, 2001.

REFERÊNCIAS PRINCIPAIS



Refatoração de Software. DCC/IME-USP. Prof. Dr. Marcos L. Chain - EACH - USP

Disponível em: <http://www.ic.unicamp.br>

Acessado em: janeiro/2015

Refatoração: Melhorando a Qualidade de código Pré-Existente. Cursos de Verão 2007 - IME/USP. Alexandre Freire e&Paulo Cheque.

Disponível em: <http://ccsl.ime.usp.br>

Acessado em: Dezembro/2014

MEDEIROS NETO, Camilo Lopes de. TDD na prática. Rio de Janeiro. Editora Ciência Morderna Ltda. 2012.

SHORE, James; WARDEN, Shane. **A arte do desenvolvimento Ágil**. Editora Alta Books Ltda, 2008.

MARTIN, Robert C. **Princípios, padrões e práticas ágeis em C#**. Porto Alegre:Bookman, 2011. (2)

Guerra, Eduardo. TDD - **Uma ideia maluca que funciona!**

Disponível em: <https://www.youtube.com/watch?v=4nodh48zbLM>

Acessado em: abril/2015