

Cleuton Sampaio

# Guia de Campo do Bom Programador

Como desenvolver software  
Java EE com qualidade

Utilizando métodos, técnicas e  
ferramentas modernas, entre  
elas: Maven, Continuum,  
Sonar, jMock etc.

Prefácio  
Aquino Botelho  
Professor e  
Coordenador  
Instituto Infnet



Cleuton Sampaio

# **Guia de Campo do Bom Programador**

Como desenvolver software  
Java EE com qualidade



*Gostaria de dedicar este livro a:*

*Meus filhos: Rafael, Tiago, Lucas e Cecília.  
As verdadeiras razões disto tudo;*

*Minha esposa: Fátima, que sempre me apoiou;*

*Meu pai: Cleuton, que já está lançando seu próximo livro.*

*Contrariando meu costume, gostaria de colocar mais de uma citação importante, cujo significado guia a elaboração deste livro.*

*The psychological profiling (of a programmer) is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.*

(“O perfil psicológico (de um programador) é principalmente a capacidade de mudar os níveis de abstração, de nível baixo a alto nível. Para ver algo em pequenos detalhes e ver algo no conjunto geral.” – Traduzido pelo autor).

Professor Donald Knuth

*If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.*

(“Se você quiser programadores mais eficazes, descobrirá que eles não deveriam desperdiçar seu tempo depurando o código-fonte, eles não deveriam introduzir os bugs, para começar.” – Traduzido pelo autor).

Professor Edsger W. Dijkstra

*...if you're afraid to change something it is clearly poorly designed.*

(“...se você tem medo de mudar alguma coisa, ela é claramente mal projetada.” – Traduzido pelo autor).

Martin Fowler

*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.*

(“Há duas maneiras de construir um projeto de software: Uma maneira é fazê-lo tão simples que obviamente não há deficiências e a outra maneira é fazê-lo tão complicado que não existem deficiências óbvias.” – Traduzido pelo autor).

Sir Charles Antony Richard Hoare

## Prefácio

Escrever o prefácio para uma obra de Cleuton Sampaio é um privilégio. Profissional com grande experiência em arquitetura e no desenvolvimento de software, tem a valiosa capacidade de transmitir os seus conhecimentos em diversos livros e nas salas de aula. Seus alunos e ex-alunos, muitos dos quais conheço, são unânimes em reconhecer a excelência de suas apresentações.

Seus conhecimentos de Gerência de Projetos – Cleuton é PMP –, aliados aos de engenheiro e arquiteto de software, mostram um perfil, cada vez menos raro, de profissional que alia essas áreas do conhecimento humano para conceber sistemas. Isso permite a ele enxergar com clareza os projetos dos sistemas de software, mantendo-se centrado na realidade.

Desenvolver software é também uma arte e, como tal, demanda criatividade. Software é conhecimento e conhecimento é cumulativo, carregando complexidade e evoluindo com rapidez. Em adição, os desenvolvedores são otimistas. Sempre achamos que podemos fazer qualquer sistema com qualidade e dentro dos prazos e dos custos. A realidade tem demonstrado ser diferente.

Escrever código computacional nos prazos e custos previstos, adequados aos padrões de mercado, com boa manutenibilidade e flexibilidade, em harmonia com o ambiente de TI da empresa, capaz de executar com bom desempenho e otimização dos recursos telecomputacionais, é trabalho para profissionais do padrão de Cleuton Sampaio.

Paradoxalmente, quanto mais frameworks, processos, métodos e técnicas de desenvolvimento aparecem, em muitos casos, pior fica o resultado de um projeto de software. A literatura sobre estes assuntos é farta e dispersa. Este livro consegue a rara façanha de compilar os principais assuntos que impactam um projeto de software, oferecendo ao leitor um roteiro de alto nível para consecução desse tipo de projeto.

Várias técnicas são abordadas no livro, como a estrutura do SWEBOK, SCM

(Software Configuration Management), TDD (Test Driven Development) e CI (Continuous Integration), além de diversas ferramentas, mostradas sempre de forma simples e prática.

Ao oferecer um projeto integrado ao longo de seus capítulos, permite, ao final da leitura, uma visão prática e integrada dos assuntos abordados. Ele ensina como desenvolver e testar um software de forma profissional, gerando um produto com qualidade.

Mais importante, ainda, é a divulgação da vasta experiência de Cleuton Sampaio como desenvolvedor e arquiteto de software. E como todos sabemos: experiência não se compra, se adquire com o tempo e muito esforço.

As empresas contratam desenvolvedores que escrevem um código-fonte inteligível e manutenível, conhecem padrões de projeto e são capazes de escrever algoritmos de alto desempenho para a situação que se apresenta.

Este livro, “Guia de Campo do Bom Programador”, coloca um holofote sobre as questões destacadas. Sem dúvida, mais uma obra para a estante dos bons desenvolvedores, arquitetos e engenheiros de software.

### ***Aquino Botelho***

*Engenheiro de Computação e Consultor de TI, sócio fundador da All Files Consultoria e Informática e colaborador da Performance Sistemas e Métodos, é coordenador dos cursos de pós-graduação em Engenharia de Software da Faculdade de Tecnologia Infnet.*

# Sumário

## **Introdução**

Sobre o tema

Por que Java?

E por que não Python ou Ruby?

Quem é o público-alvo deste livro?

Por que “programador”? Não seria melhor “engenheiro de software”?

Complementos do livro

## **1. Problemas com o desenvolvimento de software**

Organização é a chave do sucesso

Mas o que é sucesso?

A influência das técnicas e ferramentas

## **2. Qualidade de software**

Qualidade e requisitos

Qualidade e capacidades (RNF)

Trade offs

Qualidade e aderência a padrões

Mas o que significa usar padrões?

Padrões de codificação

Padrões de interoperabilidade

Funcionalidade

Conformidade

- Testes
- Adequação
- Acurácia
- Segurança
- Interoperabilidade
- Confiabilidade
  - Maturidade
  - Tolerância a falhas
  - Recuperabilidade
  - Conformidade
- Usabilidade
  - Inteligibilidade
  - Apreensibilidade
  - Operacionalidade
  - Atratividade
  - Conformidade
- Eficiência
  - Comportamento em relação ao tempo
  - Utilização de recursos
- Manutenibilidade
- Portabilidade

### **3. Os suspeitos de sempre**

#### **1. Software deve ser fácil de entender e manter**

- A única constante é a mudança
- Código autodocumentado
- Simplificar a implementação
- Utilizar padrões de projeto
- Empregar mecanismos e interfaces padrões de mercado
- Evitar dependências desnecessárias



Procurar alcançar baixo acoplamento e alta coesão interna  
Fazer a coisa certa

## **5. Soluções**

As três regras de ouro

Programar o menos possível

Chamar os “universitários”

Trabalhar com transparência

Os monstros sagrados

Edsger Dijkstra

Donald Knuth

A base teórica

Meilir Page-Jones: Projeto Estruturado de Sistemas

GoF: Design Patterns – Elements of Reusable Object-Oriented Software

Robert C. Martin: Principles Of Object Oriented Design

Alur, D.; Crupi, J.; Malks, D.: Core J2EE Patterns

Joshua Bloch: Effective Java (Java Efetivo)

Martin Fowler: Refatoração – Aperfeiçoando o projeto de código existente

Maurice Naftalin, Philip Wadler: Java Generics and Collections

Derek C. Ashmore: The J2EE Architect’s Handbook

Roger S. Pressman: Engenharia de software

## **6. Práticas e técnicas**

Momentos

Antes

Durante

Depois

Ferramentas

Sobre o projeto exemplo

## **7. Antes da construção do código-fonte**

O que “arquitetura” tem a ver com “ambiente de desenvolvimento”?

O moderno engenheiro de software tem que gerenciar

A intenção desta parte do livro

Visão geral da arquitetura

Mecanismos de interface entre sistemas

Interface com o usuário

Interface com o sistema de pedidos

Interface com a loja virtual

Interface com a transportadora

Mitigando riscos técnicos

O ambiente de desenvolvimento

O sistema operacional

O kit de desenvolvimento Java

O ambiente de desenvolvimento

Crie um ambiente virtual

Gestão de configuração de software

Gestão e planejamento

Identificação da configuração

Controle da configuração

Liberação e entrega

Contabilização e estado

Auditoria

Como implementar o SCM

Escolha de um sistema de controle de versão

Instalação do Subversion no Linux e no Windows

Configuração do Eclipse para usar o SVN

Gestão de montagem (build) e de componentes

Apache Maven

Apache Archiva

Relacionamento do Maven com o Archiva

Configuração do Maven e do Archiva

- Criação de um arquétipo Maven
- Como utilizar o Maven dentro do Eclipse
- Colocando o projeto exemplo dentro do ambiente
- Conclusão

### **3. Durante a construção do projeto de software**

- Antipatterns: o que deve ser evitado
  - Não basta ser simples
- O princípio da solução
  - Soluções de problemas de software
  - Conclusão
- Padrões e princípios de projeto
  - Sugestões adicionais de normas de codificação
  - Evite o uso de literais numéricos ou textuais
  - Escreva código autodocumentado
  - Comentários
  - Princípios de projeto orientado a objetos
- Organização do código-fonte
  - “Layers” (camadas)
  - “Layers” e “tiers” (camadas e nós)
  - Organização do código-fonte em camadas
  - Dependência entre camadas e pacotes
  - Coesão e acoplamento
  - Complexidade
  - Controle automático de métricas
- Testes unitários
  - Evite introduzir bugs, para começar
  - Testes unitários devem ser limitados
  - Cobertura dos testes
  - Teste de componentes altamente acoplados
- Organize sua rotina de trabalho

- Divida seu tempo
- Divida o trabalho de acordo com o tempo
- Não deixe tarefas incompletas
- Conclusão

## **9. Depois do código pronto**

- Revisões
  - Promover a gestão do conhecimento
  - Otimização do processo
- Integração
- Configurações de ambiente
- Testes de integração
  - Anatomia de um teste de integração
  - Separação dos testes de unidade
- Processo de liberação
- Implementando a integração contínua
  - Instalação e configuração do Continuum
  - Conclusão
- Analisando a qualidade do código-fonte
  - Geração de informações do projeto
  - Adicionando relatórios de qualidade
  - Utilizando o SONAR
  - Como melhorar a qualidade

## **10. FCQDC**

### **Referências bibliográficas**

# Introdução

Este livro é fruto de uma conversa que tive com alguns alunos, em uma noite chuvosa de 2010. Estávamos para começar as aulas e, devido à chuva, muitos alunos faltaram. Como estava com a matéria adiantada, e, para não prejudicá-los, resolvi deixar o tema em aberto. Eu e os poucos “gatos pingados” começamos a conversar descontraidamente sobre desenvolvimento de software. Quando notei, estava dando uma aula sobre tudo o que eu achava importante para a qualidade do software.

Foram mais de duas horas falando sobre: SCM, processo de desenvolvimento, métricas de qualidade, princípios de projeto orientado a objeto, boas práticas, ferramentas, testes etc. Então, quando já estava apagando a luz para ir embora, um dos alunos perguntou: “Por que você não escreve um livro sobre isso tudo que nos falou?” E “voilà”: o livro está aqui!

Diferentemente dos meus outros livros, este não é uma referência para um software ou framework específico, mas exatamente o que o nome diz: um guia de campo, ou seja, algo que você leva embaixo do braço para suas aventuras pelo mundo da criação de software.

Ele é escrito em linguagem totalmente descontraída, como uma conversa, pois procurei escrever da mesma forma que falei aos meus alunos, na referida “aula”. Estou colocando todo o peso dos meus trinta (e muitos) anos de experiência e conhecimento acumulados, misturando com pitadas de problemas reais, que acompanhei nessa longa jornada. Embora não desejasse criar uma obra acadêmica, tive que acrescentar o “estado da arte” da construção de software, com literatura, conselhos e técnicas dos melhores mestres do assunto.

Após terminar o livro, entreguei a alguns amigos para revisão técnica e um deles me perguntou: “Qual a parte que você considera mais importante?” E eu respondi sem hesitar: o capítulo “o princípio da solução”, que mostra como as soluções começam e onde estão as causas de tantos problemas com software. O resto são dicas, métodos, técnicas e organização. Vou reproduzir algumas das

perguntas deles para poder dar uma ideia do trabalho.

## **Sobre o tema**

Neste livro, focamos muito em alguns aspectos da qualidade de software, para os quais as soluções estão ao alcance dos desenvolvedores (os bons programadores). Em especial, alguns aspectos como: adaptabilidade, flexibilidade e testabilidade podem ser imensamente melhorados através de atitudes proativas, tomadas em determinados momentos do trabalho de desenvolvimento.

Depois de lecionar por tantos anos, e ver como os projetos de software ainda falham, resolvi que deveria fazer alguma coisa. O pedido dos meus alunos foi um indicador de que eu teria algo mais a contribuir, além de escrever sobre Java... Busquei inspiração em algumas fontes, como a famosa palestra do professor Dijkstra “The Humble Programmer” (<http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html>). Então, resolvi qual deveria ser a linha a seguir: mostrar os problemas e as soluções para aumentar a qualidade dos projetos de software, em uma abordagem baseada nos três momentos de um projeto: antes, durante e depois da construção do código-fonte.

## **Por que Java?**

Bem, para começar, é a linguagem de programação mais popular, de acordo com o índice TIOBE (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>). Em segundo lugar, é a plataforma com a qual trabalho há muitos anos e, em terceiro lugar, a maioria das ferramentas para ela é gratuita, o que barateia muito os custos para um autor nacional.

Bem que eu gostaria de fazer uma versão .NET também, mas dependo da boa vontade dos fornecedores de software para me cederem as ferramentas necessárias. Vamos ver...

## **E por que não Python ou Ruby?**

São duas linguagens muito utilizadas, sem dúvida. Mas Python ocupa a oitava posição na lista TIOBE, e Ruby, a décima segunda. Se eu tivesse que abordar outra linguagem, poderia voltar meus esforços para Objective-C, que, além de estar na quinta posição, vem crescendo muito devido ao seu uso no iOS.

## **Quem é o público-alvo deste livro?**

Bem, eu diria que arquitetos, engenheiros de software e desenvolvedores em geral e, em especial, os recém-formados ou que estejam desatualizados. De acordo com os meus artigos no blog ([“www.obomprogramador.com”](http://www.obomprogramador.com)), o interesse deverá ser muito grande, pois mesmo os que já trabalham há algum tempo com Java e Java EE ficam desatualizados rapidamente.

Talvez não seja apropriado para quem não é profissional de software, pois exige conhecimento e experiência prévia em programação. Para mim, profissional de software é aquele que conhece a tecnologia com a qual esta trabalhando.

Na minha opinião (e nos meus mais de trinta anos construindo software), software deve ser criado por profissionais de software. Isto também se aplica a todos os profissionais envolvidos no “caminho crítico” de um projeto de software, como:

- Analistas de requisitos;
- Gerentes de projeto;
- Gerentes de configuração.

Todos que estão envolvidos diretamente no projeto de um software devem saber construí-lo, conhecendo as tecnologias, atividades e riscos envolvidos.

Se isto não acontece em uma equipe, o projeto está destinado ao fracasso. Para mim, é simples e cartesiano.

## **Por que “programador”? Não seria melhor “engenheiro de software”?**

Programador é aquele que cria programas de computador, o que nós chamamos de “software”. É um termo mais simples, claro e direto, que engloba: “analistas”, “engenheiros de software”, “engenheiros de computação”, “desenvolvedores”, “arquitetos de software” e diversos outros termos.

Além de ser um termo mais forte, houve muitos “luminares” da ciência da computação que se denominavam “programadores”, entre eles o professor Edsger Dijkstra ([http://pt.wikipedia.org/wiki/Edsger\\_Dijkstra](http://pt.wikipedia.org/wiki/Edsger_Dijkstra)). Logo, por que não calçamos as “sandálias da humildade” e aceitamos logo o que somos?

## Complementos do livro

Neste livro, usamos um projeto Java EE Web (Microblog) como exercício para o Maven, Archiva, Continuum e SONAR. Você pode baixá-lo, assim como todos os outros complementos, acessando o blog “[www.obomprogramador.com](http://www.obomprogramador.com)”.

Porém, deixe-me falar um pouco sobre o sistema exemplo, o Microblog. Eu precisava ter um exemplo pronto de sistema para demonstrar o uso das principais ferramentas e técnicas do livro. Pensei em desenvolver um sistema específico para isto, porém, fugiria do foco. Então, peguei um software que eu havia criado há tempos e usei como exemplo para tudo. Você não precisa dele para poder entender o livro, mas, certamente, vai se beneficiar se tiver um projeto pronto.

O Microblog é um projeto do tipo “Dynamic Website”, feito em Java EE, com camada de apresentação em JSF/Facelets e persistência com Hibernate, uma combinação muito comum em sistemas corporativos. Esta é a razão de eu ter criado um Arquétipo Maven para este tipo de projeto. Porém, o Microblog está longe de ser um sistema perfeito, pois foi criado apenas para exemplificar o uso das tecnologias Java EE, e não para ser um exemplo. Mesmo assim, funciona razoavelmente bem. Se você quiser usá-lo, assim como os outros complementos do livro, fique à vontade. Estou liberando todos os complementos sob licença Apache, versão 2 (<http://www.apache.org/licenses/LICENSE-2.0.html>).



# 1.

## Problemas com o desenvolvimento de software

*The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.*

Edsger W. Dijkstra, “The Humble Programmer” (Communications of the ACM, 1972).

(“O programador competente está plenamente consciente do tamanho estritamente limitado de seu próprio crânio, por isso ele trabalha na tarefa de programação em profunda humildade, e entre outras coisas, ele evita truques “espertos” como se fossem a Peste.” – Tradução e interpretação do autor).

Antes de prosseguir neste livro, temos que estabelecer claramente nossa meta. O que queremos alcançar, enquanto programadores? E por que queremos fazê-lo? Então, gostaria de propor uma questão a você, leitor ou leitora:

- *Você acha que existe algum problema geral com a construção de software?*

Muitas pessoas responderiam que sim, porém, um número considerável de pessoas poderia responder “não”. Eu fiz esta pergunta a algumas pessoas e, em um universo de cem desenvolvedores de software, tive as respostas:

- 62 pessoas responderam que existe um problema geral com a construção de software;
- 30 pessoas responderam que não existe problema;
- 8 pessoas responderam que não sabem;

Minha opinião pessoal é que existe um problema geral (que afeta diversos tipos de empresas, projetos e profissionais) com o desenvolvimento de software, mas

eu gostaria de saber a opinião dos outros colegas. Fiquei bastante surpreso com o número de desenvolvedores que considera não haver problema algum com a construção de software.

Então, para esclarecer os motivos, restringi os dois grupos de maior impacto (“sim” e “não”), fazendo uma segunda pergunta: “por quê?” Procurei agrupar as respostas mais repetitivas e obtive os seguintes motivos:

**Existe problema porque:**

- a. O software não está sendo entregue com qualidade, gerando insatisfação;
- b. O custo de desenvolvimento aumentou, mas a qualidade não seguiu a mesma proporção;
- c. A multiplicidade de plataformas e técnicas dificulta a formação de programadores eficazes.

**Não existe problema porque:**

- a. Hoje em dia, os desenvolvedores saem muito mais preparados da universidade, graças à evolução das técnicas de Engenharia de Software;
- b. Os problemas são oriundos da má gestão de projetos e não da construção do software;
- c. São decorrências naturais do aumento de complexidade do software;
- d. Existem técnicas e ferramentas modernas para evitá-los.

Claramente, fiquei mais confuso do que estava antes. Então resolvi procurar dois amigos meus: um Gerente em uma grande empresa e um Empresário, dono de uma pequena consultoria de TI. O primeiro é basicamente usuário de TI e contrata serviços em consultorias e fábricas de software. Já o segundo produz o software como meio de vida. Fiz a ambos as mesmas perguntas e as respostas foram:

**(Gestor de TI)** Sim, há um problema geral com a construção de software, e os motivos são:

- a. Entregas fora do prazo;
- b. Estimativas irreais;

- c. Grande quantidade de problemas com o software entregue.

**(Consultor de TI)** Sim, há um problema geral com a construção de software, e os motivos são:

- a. Diminuição da tolerância dos clientes;
- b. Profusão de técnicas e ferramentas, o que encarece os custos;
- c. Dificuldade em estimar e gerenciar cronogramas;
- d. Aumento da exigência dos clientes;
- e. Aumento da concorrência.

Mesmo com visão diferenciada, ambos demonstraram que existe um problema generalizado com a construção de software, cujo efeito principal é o aumento do número de problemas no software entregue, além da consequente insatisfação dos clientes.

Eu posso constatar que os problemas existem. No momento em que escrevo este livro, ainda trabalho em uma grande empresa de software e TI, estando alocado a um setor que presta apoio ao desenvolvimento. Minha responsabilidade é fornecer soluções, avaliar código-fonte e resolver problemas que estejam dificultando o desenvolvimento. Nesta posição, frequentemente me deparo com os problemas gerais aos quais estou me referindo, e com sua principal consequência: má qualidade do código-fonte produzido.

O Professor Dijkstra (11.5.1930 – 6.8.2002), cuja brilhante citação abre este capítulo, foi um grande visionário, além de ser um dos primeiros a apontar os problemas do desenvolvimento de software. Uma das causas apontadas por ele é o excesso de autoconfiança (“truques espertos”) dos desenvolvedores, que preferem se garantir em uma profusão de jargões e técnicas, ao invés de procurar atender ao cliente da melhor maneira possível.

Antes de finalizar este “preâmbulo”, vou propor uma questão: imagine que você tenha comprado um carro zero quilômetro, cheirando a novo. Você toleraria um barulho persistente do lado esquerdo? Você aceitaria a explicação da concessionária, de que este barulho é uma característica e que não atrapalha a funcionalidade? Você aceitaria esperar por um prazo indeterminado até receber

seu carro? Você aceitaria pagar um preço variável por ele?

Bem, estas são as consequências da falta de qualidade dos produtos de software:

- Não atendem a todas as especificações do cliente;
- São entregues fora do prazo combinado;
- Custam mais do que foi orçado;
- Não funcionam de maneira satisfatória.

Alguns podem alegar que prazo, custo e requisitos são questões de gestão de projetos, portanto, não são de responsabilidade dos desenvolvedores. Eu repudio esta alegação, pois, como desenvolvedor e gerente de projetos (certificado PMP), eu sei que grande parte da responsabilidade recai mesmo sobre os ombros dos engenheiros de software.

## Organização é a chave do sucesso

Em todos os projetos bem-sucedidos que eu já vi, havia uma característica comum: organização. É claro que não basta organização para que um projeto seja bem-sucedido, mas ela pode ajudar a encontrar os problemas mais cedo, evitando altos custos.

Equipe capacitada e coesa, uso de processos e técnicas consagrados, cuidado com os riscos, escolha adequada de ferramentas e organização geral do projeto, normalmente, são ingredientes para o sucesso.

### **Mas o que é sucesso?**

De acordo com o PMI, um projeto bem-sucedido é aquele que todo o escopo solicitado foi entregue no prazo e custo combinados. Simples e cartesiano. Atender ao escopo significa que os requisitos funcionais e não funcionais negociados, ou seja, constantes do planejamento, foram implementados e aprovados pelo Cliente.

O prazo e custo são, geralmente, critérios difíceis de aceitar. Vamos a um

exemplo: você contrata um pedreiro para reformar sua cozinha. Se ele terminar depois do prazo, ou gastar mais do que o orçamento inicial, você vai ficar com raiva, certo? Mas e se ele terminar antes do prazo, ou gastar menos do que o valor orçado? Isso vai te prejudicar?

A resposta é sim. Por quê? Porque você comprometeu parte do seu tempo e do seu orçamento com a obra e, por causa disto, deixou de fazer várias coisas. O valor das outras coisas é chamado de “custo de oportunidade”.

Por exemplo, ele disse que iria demorar dois meses: março e abril. Você combinou com sua cara-metade (esposa ou marido) e um marcou as férias para março e o outro para abril, de modo a pegar as crianças e levar a um restaurante. Assim, vocês ficarão sem poder viajar em julho, que é o mês de férias escolares, mas, pelo menos, sempre haverá um de vocês por perto, para providenciar material, levar as crianças para comer fora etc. Só que o pedreiro termina em vinte dias! Acabou com as férias da família inteira! Se vocês soubessem que ele demoraria só isso, poderiam pensar em outra solução, como cada um pedir para adiantar dez dias de férias aos respectivos patrões. Em outras palavras, vocês deixarão de viajar em julho, aguentando a reclamação das crianças.

E o custo? Bem, é legal custar menos que o previsto, mas, dependendo da diferença, pode significar prejuízo. Vamos supor que você não tinha R\$ 10.000,00 para reformar a cozinha, então, ao invés de “raspar o porquinho”, você preferiu pegar um empréstimo no banco. No final, ele gasta apenas R\$ 6.000,00. Se você tivesse pegado as suas economias, poderia pegar um empréstimo bem menor para pagar a obra. Mas agora é tarde! Você tem que pagar os juros da dívida de R\$ 10.000,00.

Quando se trata de empresas, esses problemas são ainda mais graves, pois o custo de oportunidade pode gerar até mesmo demissões. Vamos imaginar um sistema cujo prazo inicial foi de dez meses. Muitos projetos e investimentos serão adiados, na espera de que o novo sistema demore este tempo. Se ele ficar pronto em quatro meses, a empresa terá perdido inúmeras oportunidades. E, quanto ao custo, a mesma coisa acontece, pois a empresa compromete parte de seu orçamento e deixa de fazer investimentos e aplicações financeiras, porque vai necessitar do dinheiro.

Não estou dizendo que você deva fazer “contas de chegar” e ficar enrolando (e

gastando) só para cumprir o planejamento. Isto será descoberto pelo Cliente. O que quero dizer é que você tem que estimar corretamente as atividades do projeto e elaborar um cronograma de alocação de recursos que lhe permita executar o projeto de acordo com o que foi combinado.

## **A influência das técnicas e ferramentas**

Hoje em dia, é quase impossível construir software sem o uso de técnicas e ferramentas especiais. Foi-se o tempo em que programar era apenas escrever código-fonte! Agora, temos que gerenciar a configuração de software, a montagem e o uso de componentes, os testes, a integração dos componentes do projeto, analisar estaticamente o código-fonte, enfim, diversas atividades que visam garantir a qualidade do projeto como um todo.

É o que vamos ver neste livro, pois esta é a essência de um “Guia de Campo”!

## 2.

# Qualidade de software

*Homens comuns sabem complicar coisas simples. Gênios sabem simplificar coisas complicadas.*

(Anônimo)

De acordo com Pressman (PRESSMAN):

*Em sentido mais geral, qualidade de software é a satisfação de requisitos funcionais e de desempenho explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas que são esperadas em todo o software desenvolvido profissionalmente.*

Lembra daquele velho ditado: “Quanto mais eu rezo, mais assombração me aparece!”? Pois é, quanto mais frameworks, processos, métodos e técnicas de qualidade aparecem, pior fica o resultado de um projeto de software.

Existe muita literatura sobre qualidade de software (veremos mais adiante), mas o que é qualidade de software? Segundo a Wikipédia ([http://pt.wikipedia.org/wiki/Qualidade\\_de\\_software](http://pt.wikipedia.org/wiki/Qualidade_de_software)):

*A qualidade de software é uma área de conhecimento da engenharia de software que objetiva garantir a qualidade do software através da definição e normatização de processos de desenvolvimento. Apesar dos modelos aplicados na garantia da qualidade de software atuarem principalmente no processo, o principal objetivo é garantir um produto final que satisfaça às expectativas do cliente, dentro daquilo que foi acordado inicialmente.*

Resumidamente, é isso mesmo! Porém, a norma ISO 9001 – 2000, referenciada por: IEEE, 2004, diz que:

*Is the degree to which a set of inherent characteristics fulfills requirements.*

Traduzindo: é o grau em que um conjunto de características atendem aos requisitos. Logo, qualidade é uma medida contínua (em contraponto à discreta), ou seja, existe uma escala livre na qual as características de um produto podem

variar. Vamos supor o seguinte gráfico:

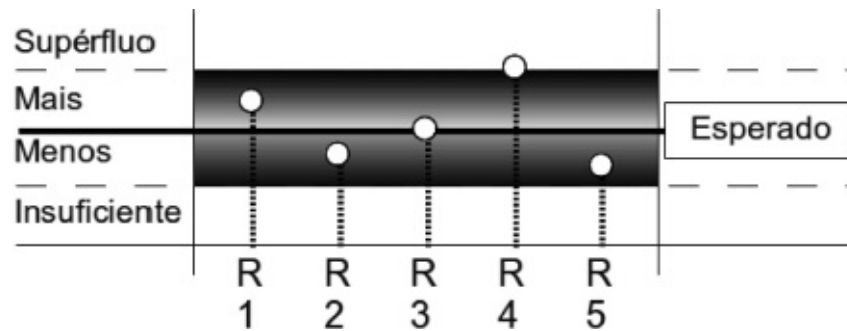


Ilustração 1: Satisfação dos clientes

Vamos relaxar na crítica e tentar entender como isto funciona. Dado um conjunto de requisitos (R1, R2 etc), é dito que sua implementação pode variar e que desta variação advém a satisfação do cliente. Há uma linha imaginária que separa a satisfação da insatisfação (Mais que o esperado / menos que o esperado). Se a implementação dos requisitos por um software ultrapassa esta linha, significa que atende mais do que o esperado. Porém, se ultrapassa muito, entra na área do supérfluo, gerando insatisfação porque o cliente está pagando por algo que não foi solicitado.

Se a implementação dos requisitos por um software está abaixo desta linha, entra na área da insatisfação e, novamente, se está abaixo de um mínimo, gera a rejeição. As áreas escuras dos gradientes representam os níveis onde a insatisfação começa. Há uma pequena faixa de tolerância que, normalmente, é difícil de determinar.

Vamos exemplificar melhor...

Você é o dono de uma empresa comercial e solicitou a um consultor que criasse um sistema para administrar suas vendas e seu estoque. Você deu algumas diretrizes a ele e passou os requisitos funcionais. Agora, vamos imaginar alguns cenários:

### **1 – O consultor entrega um sistema feito em “xBase”:**

O sistema é feito em um ambiente compatível com linguagem xBase (“Clipper”, por exemplo), que roda apenas em modo “Prompt de Comando” e exige versão



específica de sistema operacional desktop (Microsoft Windows XP, por exemplo). Ele utiliza armazenamento local de dados, feito em um compartilhamento de rede.

O sistema executa todas as funções solicitadas e funciona em ambiente multiusuário. Então, você ficaria satisfeito ou não?

## **2 – O consultor entrega um sistema feito em plataforma “web”:**

O sistema é feito em plataforma “web”, funcionando na intranet da empresa. É compatível com qualquer navegador web e não faz exigência de sistema operacional desktop. É simples, rápido e fácil de instalar e manter.

O sistema executa as funções solicitadas, embora não atenda exatamente a todos os requisitos da forma como foram especificados.

Qual dos dois cenários vai lhe satisfazer mais? No primeiro, você recebeu um código feito em uma linguagem antiga, que exige plataforma específica e que pode criar um “gargalo” para sua evolução; porém, atende a todos os requisitos. No segundo, você recebeu um código atualizado, com boa manutenibilidade e flexibilidade, que possui baixo impacto em seu ambiente de TI, porém não implementa todos os requisitos da maneira que você pediu.

Não é uma avaliação fácil! Muitos clientes leigos considerariam o primeiro cenário aceitável, embora reclamassem um pouco da interface. O que estão deixando de enxergar é a evolução! Existem duas capacidades (ou requisitos não funcionais) muito importantes, que estão sendo negligenciadas aqui: manutenibilidade e flexibilidade. Você, como cliente, pode acabar ficando “preso” a um único fornecedor por muito tempo.

Outros diriam que o segundo cenário é preferível, afinal, sendo um sistema moderno e com boa manutenibilidade, seria possível alterá-lo para corrigir os problemas ou implementar as partes que faltam. Porém, devemos lembrar que estaríamos aceitando algo que não está 100% adequado aos requisitos, logo, assumimos riscos imprevistos. Por exemplo: e se as alterações necessárias forem estruturais? E se o sistema possuir baixa manutenibilidade e flexibilidade?

O que quero dizer com esta simulação é que a qualidade de um software não é

medida apenas de forma superficial (atender a tudo / atender parcialmente) e certamente não é binária (atende / não atende). Existem limites superiores e inferiores para cada característica implementada, e é dever do desenvolvedor saber quais são eles.

## Qualidade e requisitos

Em 1996, o Dr. Yoji Akao desenvolveu um método chamado “QFD – Quality Function Deployment” (Implantação da Função de Qualidade), visando garantir a transformação dos desejos do cliente em características de qualidade de um produto. Sempre citada em literatura e concursos públicos, a IFQ (como é conhecida aqui) classifica os requisitos implementados em um produto como:

- **Normais:** são declarados explicitamente pelos clientes (ou usuários). Sua ausência pode gerar um grau variado de insatisfação;
- **Esperados:** nem sempre são declarados explicitamente, porém são esperados de um produto. Sua ausência gera um alto grau de insatisfação;
- **Excitantes:** elevam o grau de satisfação quando presentes, mas não geram insatisfação quando ausentes.

É de extrema importância saber distinguir quais requisitos se encaixam em qual classificação. Por exemplo, a ausência de um requisito esperado (que pode sequer ser citado) pode ter um impacto maior do que a de um requisito normal. Requisitos excitantes podem compensar o resto até certo ponto, a partir do qual passam a ser considerados supérfluos.

O SWEBOK (IEEE, 2004) prega que devemos gerenciar a qualidade do projeto (Software Quality Management) através de um processo definido, com etapas que se iniciam junto com o próprio projeto em si, incluindo o planejamento e a verificação dos itens de qualidade definidos.

O objetivo é aumentar o grau de aderência aos requisitos, sejam funcionais ou não funcionais, esperados, normais ou excitantes.

Portanto, concluímos que, em grande parte, a qualidade de um software está refletida no grau de atendimento aos seus requisitos.

# Qualidade e capacidades (RNF)

Todo sistema possui capacidades (ou Requisitos Não Funcionais), que são suas características de operação e uso. A Wikipédia tem uma boa definição:

*In systems engineering and requirements engineering, a non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. This should be contrasted with functional requirements that define specific behavior or functions. The plan for implementing functional requirements is detailed in the system design. The plan for implementing non-functional requirements is detailed in the system architecture.*

([http://en.wikipedia.org/wiki/Non-functional\\_requirement](http://en.wikipedia.org/wiki/Non-functional_requirement))

(“Em engenharia de sistemas e engenharia de requisitos, um requisito não funcional é um requisito que especifica os critérios que podem ser usados para julgar a operação de um sistema, em vez de comportamentos específicos. Isto deve ser contrastado com os requisitos funcionais que definem o comportamento ou as funções específicas. O plano de implementação dos requisitos funcionais é detalhado no projeto do sistema. O plano para implementação de requisitos não funcionais é detalhado na arquitetura do sistema.” – Tradução via Google Translator).

Um dos itens de qualidade que logo surge é a usabilidade. Um software ou um website é considerado bom quando é fácil de usar, ou seja, as informações estão bem localizadas, é fácil acessar o que se precisa e seu uso é intuitivo. Porém, existem diversos outros itens para avaliação de qualidade em um software e nem sempre conseguimos atender a todos eles.

## Trade offs

Não é fácil nem prático atender a todos os requisitos de qualidade possíveis, pois isto exigiria um alto investimento e, como todo projeto possui restrições orçamentárias, às vezes é necessário fazer escolhas (“trade-offs”) entre itens de qualidade, privilegiando alguns em detrimento de outros. Por exemplo, um software com altíssima usabilidade pode ser altamente calcado em componentes visuais e Ajax, sacrificando um pouco o desempenho em prol de uma interface mais rica. Isto não é problema, desde que a escolha esteja amparada, documentada e reconhecida pelo Cliente.

Existem itens de qualidade menos perceptíveis à primeira vista, porém extremamente importantes, como a “testabilidade”, por exemplo. Se não podemos testar adequadamente um sistema, podemos esperar erros e reclamações.

Pressman (PRESSMAN) afirma que existem dois grupos de fatores que afetam a qualidade de um software:

- Fatores diretos, que podem ser medidos através de métricas simples, como erros por ponto de função;
- Fatores indiretos, mais subjetivos ou que exigem maior observação, como: usabilidade ou manutenibilidade.

A norma ISO/IEC 9126 (ISO 9126) define um modelo para determinar características de qualidade de um sistema. Ela estabelece um conjunto de seis fatores principais de avaliação de qualidade de software:

- **Funcionalidade:** se o software atende às necessidades do usuário;
- **Confiabilidade:** se o software consegue desempenhar suas funções dentro das condições estabelecidas;
- **Usabilidade:** a facilidade de uso do sistema;
- **Eficiência:** se os recursos consumidos estão compatíveis com o valor adicionado pelo uso do sistema;
- **Manutenibilidade:** se o sistema é fácil de manter (alterar) ou mesmo de acrescentar funcionalidade (estender);
- **Portabilidade:** capacidade (ou facilidade) de ser transferido (ou utilizado) de um ambiente para outro.

Cada fator é dividido em subfatores, que especificam melhor os requisitos de qualidade, conforme a ilustração 2.

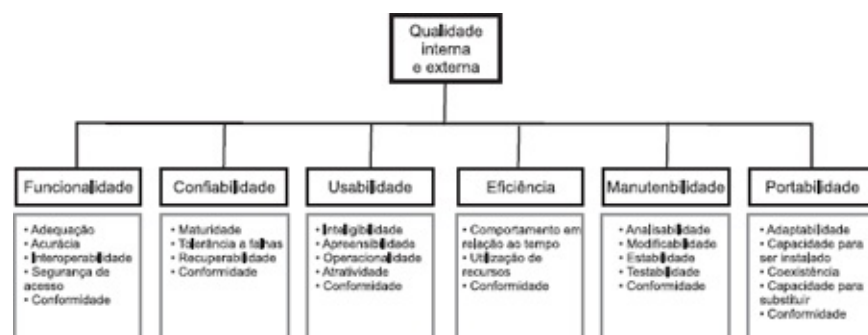


Ilustração 2: Decomposição dos fatores de qualidade – Imagem da Wikimedia commons

Na verdade, a tradição literária e tecnológica modifica um pouco estas qualidades. Por exemplo, “confiabilidade” é entendida como:

- **“dependability”**: reflete a confiança que o usuário tem no sistema, ou seja, o quanto ele espera que o sistema funcione e dê os resultados esperados, ou então apresente mensagens de erro consistentes;
- **“reliability”**: a habilidade de um sistema ou componente de executar suas funções requeridas, dentro das condições preestabelecidas, dentro do período de tempo esperado.

Na norma ISO/IEC 9126, o fator “Confiabilidade” engloba tolerância a falhas e recuperabilidade, que é a característica de realizar transações efetivas ou então desfazê-las, não deixando os dados em um estado desconhecido.

Igualmente, a norma 9126 não trata explicitamente de um fator muito discutido na literatura e nos meios profissionais, que é o desempenho (performance). Na norma 9126, a “performance” (ou desempenho) está incluída nos subfatores de “Eficiência”. Existem várias definições para o termo, desde a engenharia até a computação. No contexto que estamos discutindo, a performance está relacionada com o tempo de resposta percebido pelo usuário de um sistema. Então posso conceituá-la como:

*A quantidade de trabalho útil realizado, comparado com os recursos e o tempo gastos.*

Pode ser medida também através do tempo de resposta de transações. Veja, eu incluí a comparação com os recursos também, então não basta sabermos apenas o tempo, mas também a quantidade de recursos gastos (alocados).

A literatura é vaga quando se trata de mensurar performance. Alguns autores pregam que pode ser medida apenas com base no tempo de resposta, outros avaliam como a quantidade de ciclos de CPU consumidos (recursos), mas, quando avaliamos a qualidade de um software, devemos pensar em ambos (tempo e recursos consumidos).

Vamos ver um exemplo. Suponha que um determinado aplicativo precisa executar uma transação, a qual vamos denominar “XPTO”. Temos duas

implementações:

- “A” executa “XPTO” em cinco segundos (em média), porém eleva o consumo de CPU em X% e aloca Y kbytes de memória;
- “B” executa “XPTO” em doze segundos (em média), porém eleva o consumo de CPU em Z% e aloca T kbytes de memória.

Se  $X < Z$  e  $Y < T$ , então não resta dúvidas: a implementação “A” é mais eficiente que a implementação “B”. Porém, se “B” consome menos CPU e/ou menos memória, então devemos avaliar com cuidado a performance de ambas. Talvez seja melhor negociar os sete segundos restantes em prol de melhor consumo de recursos.

Porém, chamo sua atenção para as observações “em média”! Devemos avaliar o grau de confiança nesta “média”, antes de tomarmos uma decisão.

Outro fator a ser considerado quando se trata de performance é a Manutenibilidade! Talvez a solução “B” seja mais fácil de se compreender e alterar do que a solução “A”.

E, finalmente, “Funcionalidade” está relacionada com os requisitos funcionais. Um sistema, para ser aceito, deve atender 100% aos requisitos funcionais, ou então será rejeitado pelo Cliente. Porém, a norma inclui outros subfatores no conceito de funcionalidade.

Da mesma forma, a norma ISO 9126 deixa de falar sobre uma qualidade que o mercado considera muito importante: o grau de aderência a padrões!

## **Qualidade e aderência a padrões**

Vamos falar novamente sobre padrões. Você verá isto muitas vezes neste livro, pois é muito importante que um desenvolvedor conheça, divulgue e reforce a necessidade de adequação a padrões de mercado.

### **Mas o que significa usar padrões?**

Existem diversos padrões dentro das disciplinas de TI e Engenharia de Software,

veja alguns exemplos:

- Protocolos padrões: SOAP, HTTP;
- Padrões para interoperabilidade de web services: WS-I (OASIS);
- Padrões de projeto: Composite entity (GAMMA et al), Session Facade (ALUR et al);
- Padrões de segurança: OWASP (OWASP), JAAS (Oracle JAAS);
- Padrões de codificação e de linguagem: (BLOCH);
- Componentes padronizados: SAX parser, DOM parser, Apache HTTP Client etc;
- Frameworks padrões: Java EE, Spring, Hibernate etc.

Alguns padrões são regulamentados por entidades, como os padrões Java JSR (JCP) ou os padrões para web services do grupo OASIS (OASIS). Outros são padrões “de fato”, ou seja, são amplamente utilizados pelo mercado e há pouca probabilidade de errar ao escolhê-los – por exemplo, os componentes fornecidos pelo grupo Apache (<http://www.apache.org>).

Se um sistema utiliza ao máximo soluções padrões, ele será de fácil manutenção, fácil integração e com maior vida útil. Ao contrário, se um sistema utiliza soluções proprietárias ou caseiras, há sempre a tendência a ficar desatualizado ou sem suporte.

## **Padrões de codificação**

Por exemplo, é fundamental adotar um padrão de codificação a ser utilizado pela equipe de desenvolvimento. A importância de padrões de codificação é explicitada pela famosa citação de Martin Fowler:

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

(“Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que os humanos possam entender.” – Traduzido pelo Autor).

(Martin Fowler)

Padronizar a codificação não é “frescura”, pois facilita o entendimento e a divulgação do código-fonte.

## **Padrões de interoperabilidade**

É de fundamental importância que as interfaces de um sistema sigam padrões de mercado. Por exemplo, o HTTP é um protocolo padrão, logo, por que reinventar a roda? Alguns sistemas com os quais me deparei abusam da criatividade, implementando protocolos proprietários e utilizando portas não convencionais. Por quê? Apenas para criar problemas com a área de segurança de rede?

E os padrões de transporte de dados? XML e esquemas são uma forma reconhecida mundialmente, mas será que basta usar XML? Existem várias organizações que se preocuparam em criar esquemas XML para uma variedade de áreas de conhecimento, entre elas:

- Esquemas para relatórios financeiros: XBRL (eXtensible Business Reporting Language), vários esquemas para troca de informações financeiras;
- Passagens e viagens: OTA (OpenTravel Alliance), uma biblioteca de esquemas XML para a indústria de viagens;
- e-Commerce e Supply-Chain: GS1 (<http://www.gs1us.org/standards/ecommerce/xml>).

Com raríssimas exceções, sempre existirá um esquema XML para o que você quiser trafegar, logo, em vez de usar a “criatividade destrutiva”, procure utilizar padrões de mercado.

## **Funcionalidade**

Segundo a norma ISO/IEC 9126, a funcionalidade determina se o sistema atende às necessidades do usuário, dentro do contexto em que foi elaborado, de maneira satisfatória. Isto pode ser comprovado através de testes funcionais (e não funcionais também), supervisionados pelo Arquiteto de software.

A funcionalidade pode ser dividida nos seguintes subfatores:



- **Adequação:** indica o grau de adequação das funções implementadas às necessidades do usuário;
- **Acurácia** (ou precisão): indica a precisão dos resultados fornecidos;
- **Interoperabilidade:** como o sistema interage com outros sistemas externos;
- **Segurança:** proteção do sistema aos princípios básicos da segurança da informação;
- **Conformidade:** indica o grau de atendimento das funções implementadas às necessidades do usuário.

À primeira vista, adequação e conformidade parecem sinônimos. Porém, conformidade é se as funções estão seguindo as necessidades explícitas dos usuários. Uma não conformidade é um erro na interpretação dos requisitos. Adequação é como as funções estão atendendo às necessidades do usuário. Podem estar mais adequadas, aumentando a satisfação, ou menos adequadas. Um exemplo seria:

*O sistema deverá registrar todas as transações recebidas dos representantes para posterior processamento.*

Diante desta necessidade, vamos imaginar duas implementações:

1. **Upload de arquivos texto:** se baseia em arquivos enviados por HTTP via upload no web site do sistema. Estes arquivos são gravados no servidor e um “script” se encarrega de ler e enviar para a rotina que os atualiza no banco de dados.

Esta alternativa funciona e atende às necessidades do usuário, logo, ela apresenta conformidade. Porém, trata-se de uma solução frágil, pois “upload” de arquivos via HTTP não é um dos meios mais seguros, juntamente com o processamento via “CRON”. Isto pode gerar processamento duplicado, perda de informação etc. Logo, não é uma solução muito adequada.

2. **Web Service/Mensagem transacional:** através de um Web Service usando WS-I, o representante é autenticado via certificado digital e envia um documento de atualização assinado digitalmente, recebendo um “timestamp” também assinado. O Web Service então envia uma mensagem

*assíncrona transacional para ser processada posteriormente. O Message Driven Bean recebe a mensagem, verifica se já foi atualizada e a insere no banco de dados. Caso haja algum problema, uma exceção é gerada e comunicada, não havendo perda de informação.*

A segunda implementação, apesar de mais complexa (e talvez mais cara), atende à necessidade e de maneira mais adequada, evitando problemas decorrentes de implementação.

## Conformidade

Para alcançar a **Conformidade**, é necessário rastrear e validar a implementação de todos os requisitos que compõem o incremento a ser construído (assumindo-se que a equipe utiliza um processo iterativo e incremental). O rastreamento normalmente é feito pelo Analista de requisitos, que pode utilizar um software como o Requisite Pro, da Rational. Porém, cabe ao Arquiteto de software zelar para que as inspeções de código-fonte sejam realizadas e que o critério de conformidade tenha sido incluído.

Porém, é necessário zelar para que os testes funcionais sejam eficazes, apresentando ampla cobertura; logo, precisamos conversar sobre testes.

## Testes

De acordo com Pressman (PRESSMAN), existem vários tipos de teste, a saber:

- **Teste de unidade:** objetiva descobrir não conformidades em uma unidade do projeto, uma classe ou um componente;
- **Teste de integração:** permite avaliar se todos os componentes (ou unidades) estão funcionando em conjunto, ou se algum problema foi introduzido por um novo componente;
- **Teste de validação:** se concentra na conformidade com os requisitos. Todo o sistema é exercitado através de roteiros de testes, cujo objetivo é descobrir desvios dos requisitos originais;
- **Teste de sistema:** objetiva avaliar as características sistêmicas integradas (software + hardware) do projeto. São avaliadas questões como:

recuperação, segurança, desempenho (carga) e estresse (situações anormais).

Os vários testes possuem objetivos diferentes. Normalmente, os testes de unidade e de validação podem ser utilizados para avaliar a conformidade de um sistema com os requisitos.

Os testes podem ser avaliados de outra forma:

- Testes **caixa-preta**: realizados conhecendo-se apenas a interface do software;
- Testes **caixa-branca**: realizados com conhecimento da implementação do software.

O objetivo do teste caixa-preta é avaliar a conformidade de um módulo, sem entrar em detalhes quanto à sua implementação. Embora sejam mais simples de realizar, requerem uma massa de dados muito bem estruturada. Normalmente, os roteiros de testes são preparados antes da construção do sistema. Por exemplo, um teste de validação pode ser caixa-preta.

Já o teste caixa-branca tem por objetivo exercitar o máximo possível de cada componente do sistema, procurando passar por todos os caminhos lógicos possíveis. Isto requer o conhecimento da estrutura interna de cada módulo a ser testado, de modo a criar uma massa de testes bem abrangente. Testes caixa-branca são difíceis de desenvolver e nem sempre possíveis. O ideal é testar os componentes mais críticos desta forma e avaliar, através de ferramentas automatizadas, o nível de cobertura das massas de teste.

Utilizando o cálculo de “complexidade ciclomática” e o método de caminho básico, citados por Pressman (PRESSMAN), é possível criar uma massa de testes que percorra todos os caminhos, ou pelo menos os mais importantes, de um componente. Porém, ferramentas automatizadas, como o “sonar” ([www.sonarsource.org](http://www.sonarsource.org)) podem nos indicar a quantidade de código que foi executada com base em nossa massa de teste. Este fator é conhecido como “cobertura de testes” e pode ser utilizado para avaliar a eficácia de nossos testes de forma automática.

O Sonar também avalia a complexidade ciclomática (o número de caminhos independentes em uma estrutura de código, ver PRESSMAN (páginas 319 a 321), indicando se temos módulos mais ou menos complexos.

Não existe muito consenso a respeito do índice de cobertura de testes, porém, é geralmente aceito que acima de 80% de cobertura em um sistema é um bom indicador da qualidade dos testes.

Porém, caso o sistema apresente módulos com alta complexidade ciclomática, o índice acima de 80% pode não ser suficiente. Tom McCabe (McCabe, 1976 apud PRESSMAN) estabeleceu que módulos com complexidade ciclomática acima de dez deveriam ser refatorados, pois apresentariam problemas de testabilidade.

Logo, quando se fala em conformidade, o papel do arquiteto de software é garantir que os testes estejam cobrindo boa parte do código-fonte, zelando para que sejam realizados testes de regressão a cada mudança nos requisitos.

## Adequação

Conforme já mencionamos, a adequação aos requisitos é um conceito mais subjetivo, pois o software pode estar em conformidade sem estar adequado.

Pressman (PRESSMAN) descreve a técnica de Implantação da Função de Qualidade (IFQ), que classifica os requisitos em:

- **Requisitos normais:** explicitados e exigidos pelo Cliente;
- **Requisitos esperados:** implícitos e não declarados, porém esperados pelo Cliente. Sua ausência vai causar grande insatisfação;
- **Requisitos excitantes:** se implementados, vão aumentar muito a satisfação do Cliente.

O Arquiteto de software pode auxiliar o Analista de requisitos a tentar detalhar e descobrir o máximo possível de requisitos esperados, além de tentar selecionar alguns requisitos excitantes, de modo a aumentar a satisfação do usuário e a sensação de adequação do sistema.

Certa vez, quando eu desenvolvi um sistema de Open Market, passei alguns dias observando o trabalho dos usuários, tanto na operação da mesa de open como na retaguarda. E isto foi muito útil para saber o que mais os incomodava e o que mais os satisfaria. Esta é uma boa atitude a ser tomada, de modo a aumentar a adequação do sistema.

## Acurácia

A acurácia ou precisão está relacionada intimamente à qualidade dos testes executados, porém, existem medidas proativas que podem auxiliar, como: testar valores ou situações limítrofes.

Certa vez, eu peguei um grande problema apenas informando “-1” em um parâmetro. Ninguém jamais considerou a possibilidade do usuário digitar este valor, logo, não se preocuparam em validar se era menor que zero.

Cálculos com datas, números reais e percentuais são muito sujeitos às idiossincrasias da linguagem de programação, logo, se o sistema trabalha com uma precisão numérica muito sensível, seria bom criar provas de conceito e recomendações para realização de cálculos, repassando aos programadores e testando cuidadosamente seus produtos de trabalho.

## Segurança

Segurança tem dois aspectos:

- **Segurança da informação:** medidas para garantir os princípios básicos de confidencialidade, autenticidade e integridade;
- **Segurança de uso:** medidas para impedir que pequenos erros tenham grandes consequências.

Segurança da informação visa explicitamente garantir:

- **Autenticidade:** quem gerou a mensagem é realmente quem diz ser. Quem se logou no sistema é realmente quem diz ser;
- **Confidencialidade:** somente o destinatário correto leu a mensagem. Quem

acessou a função no sistema realmente tem a autorização para fazer isto;

- **Integridade:** a mensagem não foi adulterada ao longo do caminho. O Banco de Dados não foi alterado fora do sistema e os dados estão íntegros.

Como vamos falar explicitamente sobre a disciplina de segurança, não vamos entrar em detalhes neste momento. Porém, o desenvolvedor precisa pensar em segurança, mais especificamente em como adotar medidas proativas para sua implementação. A adoção de padrões de segurança (mais uma vez: padrões) como o OWASP (OWASP) podem auxiliar, além de inspeção de código através de ferramentas automatizadas (como o Sonar [[www.sonarsource.org](http://www.sonarsource.org)]).

Já a segurança de uso pode ser detectada através de testes de validação. E pode ser controlada através da criação de um protótipo de interface de usuário bem elaborado, no qual se impeça a execução de operações críticas sem autorização e sem aviso.

Um bom exemplo de falha de segurança de uso foi um sistema que eu tive a oportunidade de ver funcionando. Tratava-se de um sistema de mensagens entre um avião e o controlador de despacho. Caso o piloto notasse algum problema, registrava isto no sistema de “MEL – Minimum Equipment List” (lista de equipamentos mínimos). São problemas que não impedem o voo, porém requerem atenção e, caso sejam em determinada quantidade, a aeronave deve ir para a manutenção. No referido sistema, havia uma caixa (“checkbox”) cujo rótulo era “CA” e, caso fosse selecionada, desmarcava todos os problemas encontrados no avião, sem avisar ao usuário.

Por incrível que pareça, existem vários sistemas em produção com problemas deste tipo, ou seja, permitem uma transação crítica sem muito aviso ou confirmação.

## Interoperabilidade

É a capacidade do sistema interagir com sistemas externos, presentes ou futuros. Um sistema que possui boa interoperabilidade expõe ou consome dados através de interfaces padronizadas, com grande aceitação no mercado.

O desenvolvedor de software pode reforçar a interoperabilidade através do uso criterioso de interfaces padrões de mercado. É muito importante prestar atenção aos requisitos, desenhando um modelo arquitetural que contemple todas as interfaces necessárias, escolhendo mecanismos e protocolos padrões de mercado para elas.

## Confiabilidade

Segundo a norma ISO/IEC 9126, a confiabilidade indica se o produto mantém seu nível esperado de desempenho, dentro das condições preestabelecidas. Este fator pode ser dividido em subfatores:

- **Maturidade:** significa a capacidade do sistema em evitar falhas devido a problemas no software\*;
- **Tolerância a Falhas:** representa a capacidade do software em manter o funcionamento adequado mesmo quando ocorrem defeitos nele ou nas suas interfaces externas;
- **Recuperabilidade:** foca na capacidade de um software se recuperar após uma falha, restabelecendo seus níveis de desempenho e recuperando os seus dados;
- **Conformidade:** se o sistema segue os requisitos de confiabilidade estabelecidos no plano de projeto (requisitos não funcionais).

\*Há uma certa discórdia na literatura técnica sobre o conceito de “maturidade”. Alguns autores concordam que também pode ser a frequência com que o sistema apresenta falhas.

## Maturidade

Maturidade, conforme já explicamos, está relacionada à capacidade do sistema em evitar falhas e também à frequência com que elas ocorrem. Um sistema maduro apresenta poucos problemas e, quando o faz, consegue contornar a situação ou avisar aos usuários com clareza, facilitando a sua resolução.

É claro que maturidade somente é alcançada com o tempo. Conforme o uso, o sistema vai sendo depurado no dia a dia, até que alcança uma situação na qual

não ocorrem mais problemas frequentes, e, caso ocorram, o próprio sistema consegue evitar falhas ou avisar aos usuários. Este processo demora algum tempo e, na minha opinião, é muito difícil que um sistema já saia da “fábrica” maduro.

Outro dia, eu estava vendo um documentário sobre a construção do carro de passeio mais veloz do mundo, o **Bugatti Veyron**. Antes de ser entregue, o Veyron passa por diversos testes, tanto de velocidade como de resistência, rodando pelo menos 500 km antes que seja entregue ao seu proprietário. Isto é para aumentar a confiabilidade do produto, forçando os problemas mais comuns a aparecerem ainda na fábrica.

Uma bateria de testes de carga e “stress” bem elaborada, além de uma massa de teste com alto índice de cobertura, pode ajudar a “amadurecer” um pouco mais o sistema, antes que seja posto em produção. Antigamente, havia o que se chama de “paralelo”, no qual o sistema novo era processado juntamente com o sistema antigo, de modo a evitar probleminhas não detectados nos testes. Hoje em dia, com os métodos “ágeis”, a pressa de entregar alguma coisa é tão grande que não se faz mais processamento paralelo.

Outra medida proativa é a confecção de código-fonte bem fatorado, que aumenta a manutenibilidade e a testabilidade do sistema. Veremos isto mais adiante.

## Tolerância a falhas

Vamos ver mais algumas definições (LAPRIE):

- **Falha:** um estado não especificado ou não testado de software ou hardware;
- **Erro:** uma alteração inesperada no estado lógico do sistema, derivada de uma ou mais falhas;
- **Defeito:** a percepção de um ou mais erros no sistema, por parte dos usuários.

Falhas acontecem. E podem ser causadas por hardware ou software. Um problema em um componente do sistema pode causar uma ou mais falhas, que podem resultar em erros e defeitos.



Diz-se que um software é tolerante a falhas quando mantém as suas especificações, mesmo quando a plataforma computacional falha. As falhas podem ter como causas:

- **Problemas na especificação:** entendimento incorreto, especificação incorreta de algoritmos, arquitetura equivocada, especificação de plataforma de software e/ou hardware inadequada etc;
- **Problemas na implementação:** falta de conformidade, codificação de baixa qualidade, componentes inadequados e falta de testes efetivos;
- **Problemas externos:** fatores socioeconômicos (greves etc), desvios das especificações (plataforma insuficiente), interferência eletromagnética, problemas naturais (enchentes, inundações etc).

Construir um sistema tolerante a falhas é, antes de mais nada, criar uma arquitetura eficaz, especificar adequadamente os requisitos não funcionais e zelar pela qualidade do código e dos componentes utilizados, ou seja, tudo o que o arquiteto de software precisa fazer.

Além disto, a Tolerância a Falhas pode ser reforçada através de mecanismos de detecção de erros, como classes validadoras, por exemplo, e uso de transações eficazes, ou seja, ou a transação toda é executada sem problemas, ou seus efeitos são suspensos e o sistema retorna ao estado anterior.

Proteção contra falhas de hardware pode ser obtida com recursos externos, como redundância (“clusters”), por exemplo. É necessário que o arquiteto de software analise os requisitos não funcionais do Cliente (ou ajude a extraí-los, caso o analista de requisitos não o faça), de modo a projetar uma arquitetura que inclua mecanismos de tolerância a falhas, tanto de software como de hardware.

## **Recuperabilidade**

Se um sistema é tolerante a falhas, deve possuir também boa recuperabilidade. Isto significa que ele pode retornar ao seu estado original de funcionamento após a ocorrência de uma ou mais falhas. O uso de mecanismos de redundância, como “clusters” de servidores e sistemas RAID (Redundant Array of Inexpensive Disks), pode ajudar a melhorar a recuperabilidade de um sistema, mas precisam ser especificados pelo desenvolvedor.

No código-fonte, é necessário que esteja previsto o uso de transações, sejam locais ou distribuídas, de modo a possibilitar o retorno dos dados ao estado original, caso haja alguma falha. Além disto, mecanismos de relatório de exceções são imprescindíveis para o rastreamento de problemas.

## **Conformidade**

Conforme já falamos anteriormente, conformidade significa se a implementação está seguindo a especificação. Se algum requisito deixou de ser implementado, o sistema apresenta uma inconformidade (ou não conformidade) com a especificação.

Quando falamos em requisitos não funcionais, sempre notamos diversos problemas. É comum vermos documentos de arquitetura e modelos de requisitos muito bonitos, porém sem fazer referência ou sem esclarecer o que o usuário espera do comportamento do sistema.

Os maiores problemas relativos à confiabilidade de um sistema estão relacionados com falhas na especificação de requisitos não funcionais. O motivo é a falta de experiência dos analistas de requisitos e gerentes de projeto, no que diz respeito ao desenvolvimento de sistemas corporativos. Muitas vezes, considera-se que a equipe pode resolver “pepinos” durante a implantação do sistema. Eu já ouvi absurdos do tipo: “é só botar mais servidores que resolve o problema”.

O desenvolvedor deve se assegurar de que os requisitos não funcionais estejam claramente especificados, por exemplo:

- Qual é tempo de resposta (médio, máximo e mínimo) esperado?
- Qual é o número estimado de acessos simultâneos, inclusive em horários de “pico”?
- Qual é o tempo médio de recuperação de falhas esperado?

Estas três perguntas ajudam a definir a arquitetura do sistema, justificando a especificação de mecanismos de redundância e de distribuição de carga.

# Usabilidade

Hoje em dia, com “tablets”, “smartphones” e “internet TVs”, os sistemas estão cada vez mais presentes na nossa vida. Logo, boa usabilidade é uma qualidade muito importante em um sistema de informação. Eu sou da época em que os usuários recebiam pilhas de formulários contínuos e viviam pedindo “pelo amor de Deus” para incluirmos mais um campo. Hoje, eles recebem a informação em seus “smartphones” em qualquer parte do globo.

Usabilidade pode ser dividida em várias características:

- **Inteligibilidade:** facilidade em entender o que o sistema faz;
- **Apreensibilidade:** facilidade de aprender a usar o sistema, uso intuitivo, por exemplo;
- **Operacionalidade:** se o sistema é fácil de operar e se avisa sobre erros de digitação;
- **Atratividade:** o que o sistema oferece para atrair usuários ou consumidores, seja uma interface gráfica atraente, ou recursos especiais;
- **Conformidade:** se a usabilidade do sistema está de acordo com o que foi especificado.

É muito importante prestar atenção à usabilidade logo no início do projeto, evitando retrabalhos no momento da apresentação (testes de validação) ao Cliente.

## Inteligibilidade

Hoje em dia possuímos diversos profissionais especializados em interface com usuário. Desde arquitetura de informação até projeto gráfico, tudo é voltado para aumentar a usabilidade do sistema, em especial a sua inteligibilidade. Uma boa medida proativa é reforçar a criação de protótipos de interface, mostrando-os aos clientes e potenciais usuários, de modo a avaliar se conseguem utilizar o sistema.

O desenvolvedor pode ajudar reforçando a necessidade do protótipo de interface, também procurando fatorar as interfaces muito complexas.

## **Apreensibilidade**

No início da minha carreira, sempre incluíamos uma atividade de “treinamento do usuário”, de modo a capacitar os futuros usuários do sistema. Hoje em dia, o sistema tem que ser intuitivo, ou seja, possibilitar ao usuário seu aprendizado através do próprio uso. Até mesmo porque não seria prático treinar cem mil usuários da Internet no uso do seu website.

Da mesma forma, um protótipo de interface pode ajudar a criar uma interface fácil e intuitiva. Outro aspecto importante é envolver algum “arquiteto de informação”, que é um profissional especializado em criar estruturas de informação para websites, de modo a facilitar a localização das informações mais importantes, além da personalização (mostrar as informações de acordo com o perfil do usuário que está acessando o site).

## **Operacionalidade**

Da mesma forma que as duas características anteriores, a operacionalidade envolve uma interface simples, fácil e intuitiva, que também ofereça proteção contra erros de operação ou digitação.

O envolvimento de um arquiteto de informação, de um projetista de interface e a elaboração de um protótipo podem auxiliar a incluir características que melhorem a operacionalidade antes da construção do sistema em si.

## **Atratividade**

Existem sistemas que realmente jamais deveriam ter sido criados. Alguns são tão ruins que apresentam atratividade negativa para os usuários. É claro que nem todos os sistemas precisam ser muito atrativos, porém, um certo grau de atratividade auxilia em sua divulgação e uso.

Eu já vi vários sistemas “caírem no esquecimento” por falta de atratividade. Os usuários simplesmente foram deixando de usar o sistema, até que seja desativado por falta de uso.

Vou dar um exemplo de atratividade negativa. Em uma empresa, resolveram criar um sistema de bater o ponto via intranet. Colocaram alguns computadores

na entrada do prédio e logo formavam-se enormes filas para bater o ponto de entrada. O tal sistema impedia que o usuário batesse ponto caso sua senha expirasse e, além disto, exigia a troca de senha no mesmo momento, com algumas regras bem rígidas para sua formação. Resultado: todo mês havia brigas e desentendimentos na fila do ponto. Isto é um exemplo de atratividade negativa.

## Conformidade

Seria atribuição do gerente de projeto alocar um especialista em interface gráfica, ou mesmo um arquiteto de informação. São profissionais caros e, normalmente, utilizados na forma de “pool” ou contratados externamente nas empresas. É necessário que seja feito um esboço de interface (conhecido como “wireframe”) um projeto gráfico, e que sejam aprovados pelo Cliente antes de mais nada. Assim, os requisitos de usabilidade podem ser capturados de maneira eficaz, evitando retrabalho futuro.

O desenvolvedor pode sugerir a criação de protótipos de interface que podem ajudar a capturar os requisitos de usabilidade do sistema.

## Eficiência

A eficiência de um sistema é a principal medida de seu desempenho, seja em tempo de resposta ou em consumo de recursos. Um sistema eficiente apresenta tempos de resposta compatíveis com o especificado (e os desvios dentro do esperado), além de consumir recursos de forma proporcional ao serviço que estão prestando.

As duas características da eficiência de um sistema são:

- **Comportamento em Relação ao Tempo:** se os tempos de resposta das transações estão dentro do intervalo esperado, inclusive os desvios, conforme a especificação;
- **Utilização de Recursos:** quantos recursos o software consome para oferecer seus resultados e se este consumo está uniforme e de acordo com a especificação.

Só esta qualidade já justifica a contratação de um arquiteto de software para o

projeto. O desenho de uma arquitetura que gere um sistema eficiente é um trabalho muito complexo, que exige grande conhecimento técnico do arquiteto de software, além de muita experiência. Porém, o principal insumo para um projeto eficiente é a coleta e aceitação dos requisitos não funcionais.

Tudo tem um preço, inclusive a eficiência de um sistema. Para que não haja surpresas, é necessário levantar bem o grau de eficiência esperado, fazer estimativas e simulações e negociar o investimento necessário para isto. O Cliente tem que saber o que está pagando e por quê.

## **Comportamento em relação ao tempo**

O comportamento em relação ao tempo pode ser dividido em:

- Tempo de resposta de transações: o tempo que demora entre a solicitação de uma transação pelo usuário e a apresentação do seu resultado;
- Tempo de processamento: o tempo que o sistema demora para realizar processamentos batch.

O comportamento em relação ao tempo depende de vários fatores, tanto internos como externos ao sistema, que podem ser identificados previamente com base em testes de carga.

Um teste de carga é realizado quando se estima a quantidade de transações (simultâneas ou sequenciais) que um sistema tem que processar de forma típica. Isto pode demonstrar se ele está ou não em conformidade com os requisitos.

O ideal é que o desenvolvedor atue proativamente, desde a elaboração do modelo arquitetural até a construção do sistema, de modo a evitar possíveis inconformidades com os requisitos de eficiência. Para isto, vamos analisar algumas das principais causas de ineficiências dos sistemas:

- Consultas e atualizações lentas no banco de dados;
- Mapeamento O/R ineficiente;
- Problemas com “garbage collector” e alocação ineficiente de memória;
- Algoritmos ineficientes;

- Má estruturação do código, evitando a distribuição de carga;
- Componentes externos mal desenvolvidos.

Vamos ver cada uma destas causas em detalhe.

### **Consultas e atualizações lentas no banco de dados**

Não basta usar “Hibernate” para produzir um banco de dados eficiente. Um dos assuntos que mais causam problemas é a falta de cobertura de índices, forçando a leitura total de tabelas (“table scan”). Operações de “join” mal elaboradas também podem causar problemas. Um arquiteto deve contar com o apoio de um DBA (DataBase Administrator) experiente, que pode fazer um trabalho de “profiling” no banco e ver o que está impactando a performance. Um trabalho proativo de exame das principais consultas pode ajudar bastante.

Evite gerar bancos pelo seu software de mapeamento O/R (“Hibernate”, por exemplo), deixando o projeto do banco a cargo de um DBA experiente.

### **Mapeamento O/R ineficiente**

O mapeamento Objeto / Relacional (O/R) serve para associar classes e instâncias a tabelas e tuplas. Isto auxilia o desenvolvimento e permite a independência do software gerenciador de banco de dados. Porém, existem algumas “armadilhas” neste processo, por exemplo: o uso de carga tardia (“lazy fetch”), a escolha da coleção adequada para representar relacionamentos, a parcimônia na escolha de relacionamentos bidirecionais, a falta de limpeza do cache interno, entre outros problemas. O estudo cuidadoso do mapeamento deve ser feito por um desenvolvedor experiente, pelo DBA e pelo arquiteto de software.

### **Problemas com “garbage collector” e alocação ineficiente de memória**

É minha opinião que o “garbage collector” traz mais problemas do que vantagens para o Java. Ao assumir a tarefa de liberar área de memória de objetos descartados, ele também gerou um problema, retirando do programador a possibilidade de controlar o processo. Isto tudo acrescenta problemas de “memory leak” ao Java, algo jamais pensando antes. A alocação e liberação ineficientes (especialmente em “loops”) podem deixar seu programa extremamente lento. É necessário que o arquiteto de software atue junto aos

desenvolvedores promovendo o uso de boas práticas de programação Java e configurando os servidores para serem mais eficientes na execução do “garbage collector”.

## **Algoritmos ineficientes**

Entre as várias causas de construção de algoritmos ineficientes está a “mania” de “reinventar a roda”. Por exemplo, eu canso de ver código-fonte ordenando vetores com “bubble sort”, ao invés de usar o método “sort()” da classe “java.util. Arrays”, ou então pesquisando vastas coleções, ao invés de usar o método “binarySearch()”, da classe “java.util. Collections”. Tudo está relacionado com a capacitação da equipe, logo, a tarefa de validação e inspeção, realizada pelo arquiteto de software, passa a ser fundamental, assim como o acompanhamento durante a codificação.

## **Má estruturação do código, evitando a distribuição de carga**

Duplicação e replicação de código-fonte podem causar problemas de estrutura, impedindo a criação de componentes fracamente acoplados, que possam ser executados em nós diferentes, seja utilizando a técnica de EJB (Enterprise JavaBeans) ou Web Services. Outro fator importante é a dependência indesejada entre camadas, o que também quebra a estrutura do código e impede sua distribuição. São fatores aos quais o arquiteto de software deve estar atento.

## **Utilização de recursos**

Todo sistema informatizado necessita e consome recursos para ser operado. É o que chamamos de “ambiente de produção”. Os recursos podem ser: ocupação de CPU, acesso a disco, quantidade de memória RAM etc.

A utilização de recursos pode ser dividida em:

- Quantidade de recursos consumidos: quanto o sistema está consumindo e qual o desempenho que está oferecendo;
- Capacidade de consumir os recursos disponíveis: se o sistema consegue consumir os recursos disponíveis, de modo a oferecer o desempenho esperado



- **Conformidade:** se o consumo está compatível com o desempenho proposto no projeto arquitetural.

Isto envolve a quantificação de consumo de recursos e de desempenho do software. Está muito relacionado também à sua arquitetura. Por exemplo, há aplicações que são bem escaláveis e podem aumentar seu desempenho consumindo recursos de maneira proporcional. Por outro lado, há aplicações que sequer conseguem aumentar seus recursos ou, quando o fazem, consomem desproporcionalmente ao aumento de desempenho.

## Manutenibilidade

É um outro neologismo que descreve a facilidade de alterar o código-fonte de um sistema, seja para correções, adaptações ou evoluções. Pode ser dividida em:

- **Analisabilidade:** facilidade de calcular impacto ou identificar causas de problemas, deficiências ou falhas;
- **Modificabilidade:** facilidade de alterar o código-fonte;
- **Estabilidade:** possibilidade de comportamentos anômalos introduzidos pelas modificações;
- **Testabilidade:** facilidade de testar o sistema e as modificações;
- **Conformidade:** o quanto a manutenibilidade está de acordo com o que foi pedido pelo cliente.

## Portabilidade

É a capacidade de um sistema ser executado em ambientes de diferentes plataformas. Não somente para servidores, como também para a parte cliente. Por exemplo, temos que considerar diferentes tipos de navegadores web. Seus componentes são:

- **Adaptabilidade:** se o software é capaz de ser processado em plataformas diferentes, sem necessidade de configurações ou modificação da programação;

- **Capacidade para ser instalado:** facilidade de instalação em plataformas diferentes;
- **Coexistência:** facilidade de funcionar em conjunto com outros componentes instalados na plataforma;
- **Capacidade para substituir (Replaceability):** características do software que façam valer a pena um esforço para usá-lo no lugar de outro software que exista no mesmo ambiente;
- **Conformidade:** se as características de portabilidade exibidas pelo software estão de acordo com o que foi especificado.

Quando se fala em Web, portabilidade é uma palavra-chave! Hoje, temos três grandes plataformas de navegadores: Firefox, Microsoft Internet Explorer e Safari, porém, existem diversos tipos de dispositivos e navegadores diferentes, como: Smartphones, Tabs, Pads etc. Logo, a questão da portabilidade se torna crítica, pois um web site deve poder ser navegado por uma audiência o mais ampla possível.

Também temos que considerar a portabilidade de plataforma servidora. Por exemplo, se desenvolvermos nosso aplicativo em Microsoft .NET, estaremos restringindo a plataforma ao ambiente Microsoft Windows. Porém, se quisermos ficar independentes de sistema operacional, podemos optar por outras linguagens, como: PHP, Java ou Ruby.

Porém, mesmo em linguagens multiplataforma, temos que prestar atenção à portabilidade. Por exemplo, quem desenvolve aplicações Java EE sabe muito bem que existem diferenças fundamentais entre os Containers. A melhor solução para aumentar a portabilidade é ficar atento ao uso de padrões, evitando componentes e soluções proprietárias.

A coexistência também é um item fundamental! No mesmo ambiente, podem existir outras aplicações e outros componentes que sejam incompatíveis com o software que estamos desenvolvendo. É fundamental um projeto arquitetural que reduza os riscos, promovendo o uso de componentes e tecnologias padrões de mercado e da empresa do Cliente.

### 3.

## Os suspeitos de sempre

Quais são as causas dos problemas de qualidade de software? Muitas vezes, presenciei discussões com os tradicionais “dedos apontados” entre os desenvolvedores, os analistas de requisitos e o gerente de projeto, todos tentando evitar a culpa. Só que, quando se chega a este ponto, não há mais o que fazer, ou seja, o projeto atrasou e falhou.

Um software de qualidade é aquele que atende em alto grau as expectativas do cliente, inclusive quanto ao prazo e ao custo.

É claro que uma gestão ineficaz, seja de projeto ou de recursos, afeta o sucesso de um projeto e, conseqüentemente, sua qualidade percebida, mas, na maioria das vezes, são os pequenos erros, oriundos da análise, projeto e construção, que formam a “bola de neve” final.

E isto acontece com qualquer modelo de desenvolvimento, seja ele: cascata, ágil ou iterativo, e também com qualquer paradigma, por exemplo: procedural, orientado a objetos e em tempo real. Se eu tivesse que apontar possíveis causas para problemas de qualidade, escolheria:

- a. Negligência com os requisitos;
- b. Negligência com os riscos;
- c. Negligência com os testes;
- d. Má gestão.

Detalhando:

### **Negligência com os requisitos**

- Falha (ou falta) de validação e verificação;
- Falha de levantamento de requisitos funcionais;
- Falha de levantamento de requisitos não funcionais.

Normalmente, os desenvolvedores culpam os analistas de requisitos e vice-versa. Porém, a equipe toda será responsabilizada. Estas falhas poderiam ser evitadas com maior atenção aos testes (criar roteiros de teste aderentes aos requisitos e não à implementação) e também com inspeções V&V (Validação e Verificação) mais frequentes.

E quanto aos requisitos não funcionais, devemos nos lembrar que, muitas vezes, eles se enquadram na categoria de “Requisitos esperados” (QFD); logo, é papel da equipe de requisitos, em conjunto com os desenvolvedores, tentar obter o máximo de informações quanto às expectativas do cliente.

### **Negligência com os riscos**

- Falta de um levantamento de riscos adequado;
- Falta (ou falha) no planejamento de riscos;
- Falta (ou falha) das resoluções dos riscos (aceitar, transferir);
- Falta (ou falha) de estratégias de mitigação e de priorização.

O que é um risco para um projeto de software? Vamos analisar alguns riscos típicos:

- Mudanças na estratégia corporativa que impactem o cronograma de recursos;
- Mudanças na legislação que afetem negativamente alguns requisitos do sistema;
- Mudanças na estratégia da concorrência que impliquem em alteração nos requisitos do sistema.

Sim, sem dúvida são riscos. Agora, veja estes:

- Dificuldade de conectar com o Web Service por conta de bloqueios (firewalls) na rede;
- Atrasos no desenvolvimento devido à inadequação em componentes desenvolvidos por outras equipes;
- Determinada solução crítica do sistema está com tempo de resposta muito alto para ser aprovado pelo cliente;
- Rejeição nos testes de aceitação porque determinada regra complexa não foi aprovada;
- A capacitação inadequada da equipe afeta o ritmo do desenvolvimento;
- Determinado componente “open-source”, utilizado largamente no sistema, possui restrições de licença.

Podem ser riscos do seu projeto e, normalmente, são ignorados pelos arquitetos de software, desenvolvedores e gerentes de projeto. Vou dar alguns exemplos, nos quais vivi algumas destas situações:

### ***Dificuldade de conectar com o Web Service***

*Certa vez, fui dar suporte a uma equipe que precisava entregar um caso de uso que envolvia consumir alguns web services. Porém, o acesso ultrapassava duas ou três redes diferentes, dependendo do caso, e todas possuíam firewalls, com regras e administradores diferentes. Em certos casos, os firewalls bloqueavam apenas parte do acesso, o que dificultava o diagnóstico. Resultado: mais de trinta dias de atraso na homologação, devido à investigação destes problemas.*

*Poderia ter sido evitado? Sim. Desde o início as pessoas envolvidas sabiam do nível de segurança exigido e estavam cientes de que a comunicação seria feita através de vários firewalls. Poderiam ter trabalhado para mitigar este risco, com uma versão menor para teste de conectividade, antes mesmo do desenvolvimento terminar.*

### ***Atrasos no desenvolvimento devido à inadequação em componentes desenvolvidos por outras equipes***

*A palavra da “moda” em TI é “reuso”! E parece que os administradores e*

*gestores de TI descobriram o significado da palavra “componente” apenas agora! Todos querem divulgar seus trabalhos, através do uso de seus componentes no projeto dos outros. O problema é que nem sempre as equipes estão preparadas para isto.*

*Há algum tempo, uma equipe em que trabalhei utilizava um (digamos assim...) “componente” feito (orgulhosamente) por outra equipe. Havia um belo site Maven com a interface do componente publicada, incluindo a execução de testes. O gerente insistia que deveríamos usar o tal “componente”, devido à economia que proporcionaria. Bem, o problema é que o tal “componente” não tinha todas as funções implementadas da maneira esperada. Os testes não eram abrangentes e pior: a interface estava desatualizada.*

*Isto só foi descoberto no final da implementação, quando fomos utilizar outra função do mesmo componente.*

*Tem como evitar? Sim, é claro! Primeiramente, usar um componente provido por outra equipe é um risco que deve ser mitigado. Uma das maneiras é criar testes para cada funcionalidade que vamos necessitar e executá-los ANTES que o desenvolvimento avance muito.*

***Determinada solução crítica do sistema está com tempo de resposta muito alto para ser aprovado pelo cliente***

*A equipe deu uma solução para um problema crítico do sistema que, a princípio, funcionou bem nos testes iniciais, portanto, resolveu adotá-la e seguiu em frente. Ao iniciarem os testes de sistema (carga e “stress”), verificou-se que a solução não escala de maneira adequada. E agora?*

*Também tem como evitar! Se há um problema crítico, então deve-se avaliar cuidadosamente cada possível solução ANTES que o desenvolvimento chegue ao ponto de ter que utilizá-la.*

Poucas metodologias e modelos são claros com relação aos riscos técnicos e arquiteturais de um sistema. Ver projetos que usam modelo ágil, como Scrum, é muito bonito. Usar “planning poker” é muito legal, mas quando será que estes problemas serão discutidos? O Rational Unified Process (IBM) cita que o arquiteto de software deve separar os casos de uso “mais significativos”, criando

o modelo arquitetural. Porém, não é claro sobre os critérios de significância.

### **Negligência com os testes**

- Falta de testes de compatibilidade retroativa;
- Testes unitários incompletos;
- Falta de testes de integração;
- Testes de aceitação “guiados” pelos desenvolvedores;
- Cobertura insuficiente de testes.

Cada vez mais eu concluo que muitos desenvolvedores são aversos a testes. De certa forma, é natural, afinal de contas, ninguém quer ver seu projeto falhar. E, se não gostam de testar, quanto mais guardar os testes! Você já pegou um programa que, em sua nova versão, trazia problemas antigos, que já haviam sido corrigidos? Isto é a falta de testes de regressão (ou compatibilidade retroativa).

E quanto aos “testes viciados”? Sim, você cria alguns casos “JUnit” para testar seu módulo e fica satisfeito com eles, levando-os para a integração. Se entrar alguma coisa diferente do que está no teste, seu módulo poderá falhar.

Também é muito comum a situação na qual o desenvolvedor “guia” o usuário durante os testes de aceitação. É claro que ele vai ensinar a fazer da maneira “correta”, de modo a evitar problemas. Devemos nos lembrar que o usuário poderá fazer QUALQUER COISA e o sistema, se tiver um mínimo de qualidade, deverá se precaver quanto aos usos inadequados.

Finalmente surge a pergunta: será que você testou TODO o seu código? Qual o percentual de linhas de código efetivamente testadas? Quantos métodos foram testados? Quantos caminhos básicos?

### **Má gestão**

Caro amigo ou amiga, eu já vi (e participei) de projetos bem gerenciados que falharam vexaminosamente. Projetos que tinham gerente certificado pelo PMI, que tinham belos cronogramas gerados no Microsoft Project ® e que seguiam todas as boas práticas apregoadas nos bons livros do mercado.

Então, o que faltou? Gestão técnica! É o que vivo pregando: gerente de projeto de software TEM que ser engenheiro de software, ou então tem que compartilhar a gerência com um deles. Quando isso não acontece, detalhes fundamentais de gestão relacionados com a parte técnica podem ser ignorados, em prol de fazer um cronograma “bonitinho”.

E cabe a nós, engenheiros de software, gerirmos o projeto de verdade, utilizando métodos, técnicas e ferramentas que realmente garantam o resultado. Como? Organizando o ambiente e os componentes a serem utilizados, o repositório de código-fonte, controlando o versionamento e a distribuição, promovendo as boas práticas, ou seja, tudo o que um bom programador deve fazer.



## 4.

# Software deve ser fácil de entender e manter

*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*

(“Há duas maneiras de construir um projeto de software: Uma maneira é fazê-lo tão simples que obviamente não há deficiências, e a outra maneira é fazê-lo tão complicado que não existem deficiências óbvias. O primeiro método é muito mais difícil.” – Traduzido pelo autor)

C. A. R. Hoare (o inventor do algoritmo “Quicksort”)

Certamente, os “suspeitos” representam a maior parte dos problemas de baixa qualidade, mas podem existir outros, dependendo do aspecto da qualidade que queiramos observar. Eu já vi sistemas corretos (atendem bem aos requisitos), com boa performance e bem testados apresentarem problemas em outras áreas.

Só para citar um exemplo, vou contar mais uma “historinha” que vivi:

*Há alguns anos, eu fui chamado para ajudar a resolver um problema em um sistema da empresa onde eu trabalhava. Quando cheguei lá, constatei que era um problema realmente cabeludo. Havia um programa, feito em C++, que estava apresentando um problema muito complicado de resolver.*

*O tal sistema era muito elogiado, pois foi desenvolvido dentro do prazo, com testes bem feitos e riscos bem administrados. Era um exemplo dentro da empresa e funcionou bem por algum tempo. Um dia, foi necessário fazer uma alteração, e os responsáveis pelo projeto não trabalhavam mais na mesma empresa, sendo designada outra equipe. Não conseguiram alterar e fazer o sistema funcionar e começaram a pedir ajuda.*

*O código era extremamente complicado, descendo até o nível de gerenciamento de Threads e concorrência.*

*Foram necessários quase dois meses para resolver o problema.*

Não precisa dizer que este “incidente” acabou com a reputação do sistema (e do gerente também). Qual o motivo? Simplesmente a solução adotada não era a mais adequada, pois era de baixa manutenibilidade e apresentava pouca flexibilidade.

Como disse “Sir” Hoare na citação que abre esta seção, tornar coisas complexas simples e, ao mesmo tempo, eliminar deficiências é uma tarefa difícil. As pessoas, em geral, consideram que complexidade é sinal de inteligência ou competência. Quer um exemplo? Lembra das aulas de cálculo da faculdade? Aquelas em que o professor passava uma equação “cabeluda” e você não conseguia resolver? Normalmente é porque a solução é mais simples do que você está pensando.

## **A única constante é a mudança**

Eu ouvi esta frase de um professor meu e, a princípio, parece um paradoxo, mas é verdade. Cada vez mais, a única constante é a mudança. A velocidade com que as coisas mudam força as empresas a viverem uma renovação constante, e os sistemas aplicativos devem acomodar esta característica.

Quando eu entrei na faculdade, nos idos de 1980, dizia-se que a vida útil máxima de um sistema aplicativo seria em torno de cinco anos, pois, depois deste prazo, era mais barato escrever outro sistema do que alterar o original.

Com o surgimento do microcomputador e das redes, este prazo caiu rapidamente para dois anos.

E agora, com o “boom” dos smartphones e tablets, além da ubiquidade das informações, este prazo é muito curto. É claro que a tecnologia, os métodos e a plataforma de desenvolvimento evoluíram bastante, mas, mesmo assim, é necessário que os desenvolvedores assegurem que o software possa evoluir sem ter que ser reconstruído.

De acordo com a norma ISO/IEC 9126 (ISO 9126), isto se reflete na “manutenibilidade” do software, que mede a facilidade de manutenção do código-fonte, seja para corrigir problemas, adicionar ou adaptar nova funcionalidade. Vamos ver resumidamente algumas maneiras de melhorar a manutenibilidade do código-fonte.

## Código autodocumentado

Uma possível solução está em criar código autodocumentado, conforme pregam alguns ícones do desenvolvimento, como Martin Fowler (FOWLER), que afirmam que esta é a melhor maneira de documentar uma implementação. Uma boa definição de código autodocumentado é:

(...) aquele que é legível e facilmente compreendido por um desenvolvedor observador, de modo que suas intenções, propósitos e técnicas são claros, sem necessidade de comentários suplementares.  
– Traduzido pelo autor)

(John Elm, IBM DeveloperWorks - <http://www.ibm.com/developerworks/rational/library/edge/09/jun09/designdebteconomics/>)

Ao conversar com outros desenvolvedores, especialmente os mais antigos, noto uma confusão muito grande entre “código autodocumentado” e “código comentado”, que são conceitos completamente diferentes. Comentários, além de não documentar o código, podem atrapalhar seu correto entendimento. A melhor frase sobre comentários que eu li foi da Vovó (Granny) do site Java Ranch (<http://www.javaranch.com/granny.jsp>):

*Debug only code - comments can lie.*

(Depure apenas o código – comentários podem mentir. – Traduzido pelo autor).

Comentários representam um grande problema no código-fonte pelos seguintes motivos:

- a. Representam uma visão pessoal de quem escreveu e não necessariamente a realidade;
- b. Nem sempre são atualizados quando o código-fonte é alterado;
- c. Dão a falsa ilusão de que o código é facilmente inteligível;
- d. Podem enganar os outros!

Isto quer dizer que não devemos comentar o código-fonte? Não! Existem os comentários bons e os malévolos. Por exemplo, comentários que explicam classes e métodos, como os do estilo JavaDoc (Oracle JavaDoc), devem sempre ser escritos, assim, fica fácil descobrir o uso de uma classe sem ter que analisar seu código-fonte. Porém, comentários com intenção de explicar o código-fonte devem ser evitados.

## Simplificar a implementação

Martin Fowler (FOWLER) faz uma pergunta interessante: se você sente necessidade de comentar um algoritmo complexo, por que não o reescreve para torná-lo mais inteligível? Ele introduz os refactorings, técnicas de rearranjo de código-fonte, que podem facilitar o seu entendimento. Entre eles:

- Extract Method (extrair método) – <http://martinfowler.com/refactoring/catalog/extractMethod.html>;
- Introduce Explaining Variable (introduzir variável explicativa) – <http://martinfowler.com/refactoring/catalog/introduceExplainingVariable.html>;
- Replace Conditional with Polymorphism (substituir condicionais por polimorfismo) – <http://martinfowler.com/refactoring/catalog/replaceConditionalWithPolymorphism.html>.

## Utilizar padrões de projeto

Não tem jeito! O livro da Gangue dos Quatro (GAMMA et al) ainda é ignorado muitas vezes. É claro que muitos padrões já são oferecidos pelas próprias linguagens de programação e frameworks, por exemplo: “Iterator”, “Chain of responsibility”, “Composite entity”, mas nem sempre os desenvolvedores os utilizam, por exemplo: “Flyweight”, “Memento”, “Strategy”, “Observer”, “Visitor”, entre outros, poderiam ser mais empregados.

Se você usa um padrão de projeto, fica mais fácil comunicar sua intenção a outros desenvolvedores.

## Empregar mecanismos e interfaces padrões de mercado

Empregar interfaces padronizadas é sempre uma boa maneira de aumentar a manutenibilidade do software. Em vez de se comunicar via arquivos “TXT”, use XML, com um esquema próprio e disponível para consulta.

Evite criar seus próprios protocolos, utilize os padrões! Hoje em dia, para invocar remotamente funções, é indicado utilizar XML-SOAP, em vez de criar protocolos de aplicação próprios.

Em vez de ficar inventando mecanismos para processamento assíncrono (usar crontab, sincronizar através do banco de dados etc.), utilize um sistema de mensagens assíncronas, baseado em JMS, ou seja, sempre existe um padrão de mercado para atender às suas necessidades.

## Evitar dependências desnecessárias

Um dos maiores sintomas de código-fonte ruim é a propagação de alterações, que é por si uma das causas de “Brittleness” ([http://en.wikipedia.org/wiki/Software\\_brittleness](http://en.wikipedia.org/wiki/Software_brittleness)). Quando um software possui dependências rígidas e distribuídas, fica difícil alterar uma parte sem ter que alterar outras.

Devemos, sempre que possível, separar interface de implementação, seguindo o princípio de Segregação de Interfaces (MARTIN et al):

*The dependency of one class to another one should depend on the smallest possible interface.*

(“A dependência de uma classe para outra deve depender da menor interface possível.” – Traduzido pelo autor).

(Principles of Object-Oriented Design (MARTIN et al)).

Desta forma, se você alterar a implementação (e não a interface), evitará a propagação de alterações.

Outra causa das dependências indesejáveis é trafegar estruturas específicas de uma camada para outra do software. Por exemplo, tentar passar uma instância de “[javax.servlet.http.HttpServletRequest](#)” para uma classe de negócio, ou então

passar um “java.sql.**ResultSet**” para uma classe de apresentação. Os únicos tipos de dados que podem trafegar entre camadas diferentes são as classes do tipo DTO – Data Transfer Objects (Objetos de Transferência de Dados), que somente transportam dados, ou Domain Model (Modelo de Domínio), que também incorporam comportamento.

## **Procurar alcançar baixo acoplamento e alta coesão interna**

Meilir Page-Jones, em seu livro “Projeto Estruturado de Sistemas” (Page-Jones), difundiu os conceitos de acoplamento e coesão, para identificar o grau de interdependência entre os módulos de um software.

Normalmente, módulos (ou classes, ou métodos) com baixa coesão apresentam maior dificuldade de compreensão e manutenção, além de aumentarem as chances de existir alto acoplamento (maior dependência) com outros módulos.

E módulos com alto acoplamento possuem dependências indesejáveis com outros módulos, aumentando a propagação de alterações. Normalmente acontecem porque possuem baixa coesão interna.

## **Fazer a coisa certa**

Deixei para o final porque é a atitude mais difícil de tomar e também a mais eficaz. Todos queremos fazer a coisa certa quando desenvolvemos software, mas o que isto significa? Significa pensar, analisar, pesquisar e usar a solução adequada para um problema.

Você não é o primeiro a criar um programa, logo, alguém já deve ter passado por problema semelhante e tem uma solução mais adequada. O que quero dizer é que não devemos ser presunçosos, assumindo que resolvemos qualquer problema “do nosso jeito”. Sempre vale a pena ler um pouco e ver o que o pessoal andou fazendo, antes de escolher uma maneira de fazer as coisas.

Os desenvolvedores, especialmente os mais antigos, costumam ter “calos”, ou seja, vícios em programação, aplicando sempre as mesmas soluções para

problemas que julgam serem semelhantes. Até parece “Padrão de Projeto”, mas não é! Aliás, uma boa olhada nos Antipadrões de projeto é uma boa maneira de evitar cair nessa “tentação”. O “Anti-Pattern Catalog” (<http://c2.com/cgi/wiki?AntiPatternsCatalog>) é uma excelente referência sobre soluções inadequadas.

Eu chamo estas soluções de “Gambiarrras da Informática”, e existe até um site o POG – Programação Orientada a Gambiarrras ([http://desciclopedia.ws/wiki/Programação\\_Orientada\\_a\\_Gambiarrras](http://desciclopedia.ws/wiki/Programação_Orientada_a_Gambiarrras)) que trata deste assunto. Quer alguns exemplos?

### ***Emaranhado de substrings***

- Problema: você precisa ler informações passadas de um sistema externo;
- Solução: mande criar um arquivo texto e vá analisando as informações com base em posição e tamanho, inserindo “números mágicos” dentro do seu código-fonte (o índice da posição e o tamanho de cada dado). Se quiser ser mais “profissional”, use a versão binária, com um mapa de bits (emaranhado de bits)!

### ***Sincronização por file-system***

- Problema: você precisa executar algum processamento caso receba uma solicitação, ou precisa processar alguns registros, de forma pontual;
- Solução: crie um programa que fique monitorando um diretório na rede. Quando um novo arquivo é gravado lá, você o abre e processa. Normalmente, é encadeado com o “emaranhado de substrings”;

### ***Tabelão***

- Problema: você precisa ler dados de 450 filiais, classificar e fazer análises sobre eles;
- Solução: crie um vetor bem grande, leia tudo, faça um “bubble-sort” no vetor e o processe. Lembre-se de alocar bastante memória para caberem com folga os dados. Use o “emaranhado de substrings” ou o “emaranhado de bits” para diferenciar os dados dentro de cada filial.

### ***Mamãe, criei meu primeiro servidor!***

- Problema: você precisa calcular valores sob demanda para pedidos vindos pela rede, mas se cansou de usar a “sincronização por file-system”;
- Solução: leia um livro e aprenda a criar um servidor baseado em sockets! Mostre para o seu chefe, sua namorada (ou namorado) e para a mamãe! Aproveite e use também o “tabelão” e o “emaranhado de substrings” em conjunto...

É claro que, se feitas com bom senso, nem todas estas alternativas são ruins, não é?

**ERRADO!** São todas ruins! Em uma plataforma moderna de programação, existem mecanismos testados e padronizados para cada um destes cenários; logo, você deveria estudá-los antes de tomar uma decisão. As pessoas que elogiam ou defendem estas soluções “alternativas” são as mesmas que dizem que “o bom programador é aquele que usa o bom senso”.

### **Bom senso???**

Muita gente boa alega que usa o “bom senso”, e são as mesmas pessoas que dizem que seu código é bom porque é bem comentado, ou que suas soluções são melhores que as da Microsoft (ou da Oracle)... Caro leitor, cara leitora: cuidado com o “bom senso”. A engenharia existe para acabar com o “bom senso”, pois onde são aplicados métodos e técnicas, não é necessário utilizar subjetividade, afinal de contas o que é bom senso para uns pode não ser para outros.



## 5.

# Soluções

*Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.*

(“A simplicidade é uma grande virtude, mas requer trabalho duro para alcançá-la e educação para apreciá-la. E para piorar as coisas: a complexidade vende melhor.” – Tradução do autor)

Edsger W. Dijkstra (EWD896 - <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD896.html>)

Até agora, só falamos dos problemas com o desenvolvimento de software que afetam a sua qualidade. Porém, a ideia deste livro é ser um “guia de campo”, logo, deve oferecer soluções também.

Para começar, ninguém melhor que o Prof. Dr. Dijkstra para dar a primeira solução: simplicidade! A melhor solução é, na maioria das vezes, a mais simples. Quantas vezes nós erramos problemas de álgebra por não acreditarmos na solução mais simples?

Devemos sempre buscar a simplicidade em nossas implementações de código, o que, na maioria das vezes, não é fácil de fazer. Uma das causas é a falta de foco, característica de todos os desenvolvedores. Se nos concentrarmos no problema real, em vez de ficar imaginando coisas, vamos chegar a uma solução simples. Quer um exemplo? Suponha o seguinte problema:

*Todos os dias chegam relatórios de movimentação financeira via e-mail, de cada um de nossos representantes. O relatório precisa ser conferido para saber se realmente é autêntico, logo, devem ser acessadas duas tabelas no mainframe: uma de agentes financeiros e outra de relatórios emitidos. A consulta tem tempo de resposta variado, logo, o atendente tem que solicitar um relatório de cada agente para conferir cada relatório. Para agilizar a consulta, o funcionário agrupa de cinquenta em cinquenta relatórios. A consulta deverá registrar se o relatório é autêntico. Não há necessidade de conferir os cálculos no momento, pois, ao chegar no mainframe, será feita uma validação mais completa. O que é necessário é garantir que ele seja autêntico, para rejeitar logo na recepção.*

*Queremos automatizar ao máximo o processo, liberando os funcionários da recepção de e-mails.*

Como resolveria este problema? Certamente, poderia desenvolver um módulo que consultasse as duas tabelas, a partir dos dados contidos na mensagem. Como a consulta tem tempo variado, ele poderia utilizar um sistema de mensagens assíncronas (como um sistema JMS) para isto. A maioria das pessoas para quem expus o problema fez sugestões parecidas com esta.

Porém, você leu atentamente o texto? O problema é **saber se o relatório é autêntico para rejeitá-lo logo na recepção**, liberando o funcionário que verifica manualmente. Não é necessário fazer uma verificação completa. Porém, a maioria das pessoas se “distrai” com os detalhes e fica imaginando situações hipotéticas, perdendo o foco no problema.

Qual é a melhor solução? Por que não usar assinatura digital? A mensagem pode ser assinada digitalmente, simplificando a conferência automatizada e garantindo a autenticidade da mensagem. Acabou! Qualquer verificação posterior será feita no mainframe. Um simples filtro de mensagens, com uma pequena função acoplada, pode fazer essa verificação rapidamente, rejeitando os relatórios com assinaturas inválidas.

Aliás, esta solução também tem uma característica importante: menos código-fonte produzido!

A falta de foco nos faz pensar em detalhes, como: acessar as tabelas no mainframe, dificuldade de sincronizar os resultados etc. Você pode até alegar que o uso de assinatura digital exigiria alterações do lado dos agentes (os emissores do relatório), mas quem disse que é proibido? Será que o benefício não compensaria o custo? Quem disse que fazer as consultas e usar um sistema de mensagem assíncrona seria mais barato? Usar assinatura digital, neste caso, é a solução mais simples e pronto.

## **As três regras de ouro**

Ao longo do tempo, fui desenvolvendo e aprimorando regras que me auxiliavam a desenvolver software com menor “dor de cabeça”. E elas foram se aglutinando até chegarem a três regras simples:

1. **Programar o menos possível;**
2. **Chamar os “universitários”;**
3. **Trabalhar com transparência.**

Eu acredito que, se seguir estas três regras à risca, produzirei software com qualidade. Mas vamos esclarecer melhor...

## **Programar o menos possível**

Já que este é um livro para desenvolvedores, esta regra parece um paradoxo! E é mesmo. Para criar software de qualidade, você deve programar o menos possível. Quanto menos linhas de código criadas, menor o risco e, quanto menor o risco, maiores as chances de sucesso.

Hoje em dia, temos componentes prontos e testados para muitas funcionalidades, logo, é melhor procurar se já existe um componente que realize todo o trabalho necessário. Se não tiver, pode ser possível integrar dois ou três componentes prontos e obter o resultado, sem a necessidade de escrever todo o código-fonte necessário.

Isto também significa criar soluções simples. Em outras palavras, procurar sempre simplificar as coisas, o que nem sempre é fácil (como disse o Professor Dijkstra, no início deste capítulo). Uma das causas de complexidade é tentar “reinventar a roda”. Hoje em dia, em toda plataforma de desenvolvimento, a maioria dos problemas de infraestrutura está resolvida. Por exemplo: processamento de requisitos, mensagens assíncronas, inversão de controle, controle de concorrência, criptografia, entre diversos outros problemas de computação; logo, não faz sentido criar mecanismos e algoritmos próprios.

## **Chamar os “universitários”**

O apresentador de TV Sílvio Santos popularizou esta frase, que era uma das opções em seu programa de perguntas e respostas. Na verdade, a ideia é sempre procurar outras pessoas, de modo a buscar a melhor solução. Muitas vezes, nós, desenvolvedores, deixamos nosso orgulho atrapalhar o trabalho e evitamos discutir nossas ideias com outras pessoas, ou mesmo consultar o que já existe sobre o assunto. É igual aquele casal dentro do carro: o marido, visivelmente

perdido, se recusa a atender aos pedidos da esposa e parar para perguntar o caminho.

Mostrar suas ideias e discuti-las com outras pessoas sempre vai gerar um resultado melhor do que você virar duas noites programando uma solução e depois descobrir que já existia algo mais simples. Antes de sentar e sair codificando feito um louco, escreva o que acha e envie para alguns colegas, de modo a obter opiniões diferentes sobre o assunto. Procure também pesquisar o que já existe, seja na Internet ou na literatura. Assim, quando se sentar para programar, terá a certeza de que refinou bastante a ideia original, reduzindo a possibilidade de problemas futuros.

Isto também se aplica a padrões de codificação, de projeto e a frameworks e técnicas. Procure sempre saber o que os outros já fizeram, e como fizeram, antes de partir para o desenho da sua solução.

## **Trabalhar com transparência**

Parece bobagem, mas uma regra de ouro para mim é poder comprovar os resultados. Para mim, é necessário que todos os passos e componentes envolvidos em uma solução sejam passíveis de testes. E eu insisto nisso! Sempre que possível, prefiro escrever os casos de teste ANTES mesmo de iniciar a codificação e vou produzindo os scripts de teste em paralelo com a solução em si.

O ideal é manter um registro dos testes realizados em todos os passos da solução, guardando os scripts para realização de testes regressivos, caso haja alguma alteração.

Mas nem só os testes precisam ser comprovados... a implementação dos requisitos também! Você deve prover meios para que o Cliente consiga rastrear seus requisitos até os produtos de trabalho, de modo que possa saber se o que pagou foi realmente implementado e onde está.

E, finalmente, temos a questão dos riscos. É comum os desenvolvedores “engravidarem” prazos e orçamentos para cobrirem eventuais problemas, os chamados “riscos”. Lembro-me de uma conversa com meu primeiro chefe na área de programação sobre um cronograma que estávamos elaborando (era na

época do COBOL e do cartão perfurado):

***Chefe:*** *E aí? Qual é o prazo do seu programa?*

***Eu:*** *Bem, eu acho que uns dois dias... sei lá...*

***Chefe:*** *O quê!? “Dois dias” é o que a digitadora vai levar para perfurar os cartões do seu programa. Além disso, tem a compilação, o teste...*

***Eu:*** *Mas é um programa muito simples. É só uma cópia adaptada de outro relatório. Que tal cinco dias?*

***Chefe:*** *E se o arquivo não estiver gerado da maneira que esperamos? E se a Produção não puder colocar o Job no schedule? E se o programa gerador do arquivo demorar a ficar pronto? Bota aí dois meses!*

Isso é “engravidar” prazo, ou seja, inflacionar números para poder acomodar eventuais problemas. E quem “engravida” prazo geralmente “engravida” custo também, por medo de ser mais complexo e demandar mais trabalho do que o esperado.

É claro que podemos nos precaver com relação aos riscos. Podemos até cobrar mais ou requisitar um prazo maior em função de riscos, só que o Cliente tem que saber disto. E tem que ter o desconto caso o risco não aconteça.

A metodologia de gerenciamento de projeto do PMI ([www.pmi.org](http://www.pmi.org)) nos auxilia a lidar com riscos. Primeiramente, temos que identificar os riscos e depois quantificá-los. Isto feito, nós decidiremos como vamos lidar com os riscos: se vamos transferir (fazer seguro), aceitar (guardar uma grana) ou criar uma estratégia de mitigação (tentar evitar). Então, já reservamos uma parte do tempo e do custo para isto.

Se restarem riscos residuais, podemos criar uma “Reserva de contingência”, seja de prazo ou de custo, para ser distribuída entre estes riscos (Known-Unknowns). Finalmente, podemos também criar uma “Reserva gerencial” para os riscos não identificados (Unknown-Unknowns).

Desta forma, estamos nos precavendo dos riscos e o Cliente sabe exatamente o quanto está esperando ou pagando a mais, e o motivo.

Transparência é isto: não ter o que esconder! O Cliente consegue saber exatamente onde está o código-fonte pelo qual pagou, como foi testado (quantidade e qualidade dos testes) e também sabe o motivo de aumentos de prazo e custo.

## Os monstros sagrados

Existem alguns luminares da ciência da computação cujas obras nos ajudam a criar código-fonte com mais qualidade. É claro que você já deve ter ouvido falar sobre alguns deles, mas devemos sempre citá-los como fonte de inspiração e conhecimento. Na verdade, a leitura de textos destes dois autores sempre me inspirou.

### Edsger Dijkstra

De acordo com a Wikipédia ([http://pt.wikipedia.org/wiki/Edsger\\_Dijkstra](http://pt.wikipedia.org/wiki/Edsger_Dijkstra)):

*Edsger Wybe Dijkstra (Roterdã, 11 de Maio de 1930 — Nuenen, 6 de Agosto de 2002) foi um cientista da computação holandês conhecido por suas contribuições nas áreas de desenvolvimento de algoritmos e programas, de linguagens de programação (pelo qual recebeu o Prêmio Turing de 1972 por suas contribuições fundamentais), sistemas operacionais e processamento distribuído.*

O meu primeiro contato com os escritos de Dijkstra foi através do texto: “The humble programmer” [Dijkstra, THP] (“O programador humilde” - traduzido pelo autor), literatura obrigatória na faculdade. Este texto é a palestra que ele deu ao ganhar o prêmio Turing, da Association for Computing Machinery (ACM), de 1972.

Só na leitura deste texto encontramos algumas citações famosas, como:

- *If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start*

*with. (Se quiser programadores mais efetivos, descobrirá que eles não deveriam perder seu tempo depurando (o software), eles não deveriam introduzir bugs (problemas), para começar. - Tradução do autor);*

- *Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (O teste de programas é um meio eficaz de mostrar a presença de bugs (erros), mas é inadequado para mostrar sua ausência. - Tradução do autor).*

O professor Dijkstra foi um dos defensores pioneiros da melhoria do processo de construção de software. No início da década de 70 ele já falava de programação estruturada, TDD (embora com outro nome) e dos princípios do que hoje conhecemos como Engenharia de Software.

## **Donald Knuth**

De acordo com a Wikipédia ([http://pt.wikipedia.org/wiki/Donald\\_Knuth](http://pt.wikipedia.org/wiki/Donald_Knuth)):

*Donald Ervin Knuth (Milwaukee, 10 de Janeiro de 1938) é um cientista computacional de renome e professor emérito da Universidade de Stanford. É o autor do livro The Art of Computer Programming, uma das principais referências da ciência da computação. Ele praticamente criou o campo análise de algoritmos e fez muitas das principais contribuições a vários ramos da teoria da computação.*

Sua obra mais conhecida é o compêndio: “The Art of Computer Programming” [Knuth, TACP], onde ele fala sobre construção e análise de algoritmos. Tudo o que sabemos sobre a notação “Big O” é, em grande parte, oriundo dos seus trabalhos. É dele a famosa citação:

*Premature optimization is the root of all evil (or at least most of it) in programming.*

(“A otimização prematura é a raiz de todo o Mal (pelo menos a maior parte dele) na programação.”)

## **A base teórica**

É claro que o trabalho dos “monstros sagrados” influenciou tudo o que chamamos de “Engenharia de Software”, porém, existe uma gama de livros que

eu chamo de “base teórica” de todo desenvolvedor, ou seja, o que ele ser lido ANTES de começar a programar (ou mesmo depois, caso não tenha lido).

Mesmo que você já seja um desenvolvedor experiente, vale a pena conferir os conhecimentos listados nestes livros e sites, pois muita coisa mudou na arte da programação e pode ser necessário fazer uma “reciclagem”.

Peço desculpas porque alguns estão em inglês, mas é que eu tive que me basear na minha própria biblioteca. Você pode utilizar no idioma que preferir, pois muitos já estão traduzidos. No caso dos websites fica mais difícil, embora você possa utilizar o tradutor do Google.

## **Meilir Page-Jones: Projeto Estruturado de Sistemas**

(Page-Jones)

É um livro da década de 80, no qual são explicados os conceitos de projeto modular, como acoplamento e coesão. Uma leitura importante (mesmo nesta época de “programação orientada a objetos”) que não deve ser menosprezada.

A construção modular pressupõe a criação de componentes altamente coesos e fracamente acoplados entre si, o que reduz a propagação de alterações e aumenta a manutenibilidade do código-fonte. Page-Jones introduz os dois importantes conceitos, com exemplos bem ilustrativos.

## **GoF: Design Patterns – Elements of Reusable Object-Oriented Software**

(GAMMA et al)

Gamma, Helm, Johnson e Vlissides (a gangue dos quatro – GoF) escreveram este livro em 1994 para tratar de soluções para problemas recorrentes, os chamados padrões de projeto.

A maioria das linguagens de programação e frameworks modernos já traz componentes com alguns dos padrões GoF, como: Iterator, Composite Entity etc. Mas todo engenheiro de software que se preza deve ler este livro e tentar



entender o uso dos principais padrões.

## **Robert C. Martin: Principles Of Object Oriented Design**

(MARTIN et al)

É um website que traz uma compilação de vários trabalhos sobre projeto Orientado a Objetos. Ele fala sobre:

- **Single Responsibility Principle** (Princípio da responsabilidade única): cada responsabilidade deve estar em uma classe separada;
- **Open/Closed Principle** (Princípio Aberta/Fechada): toda classe deve estar aberta para extensão, mas fechada para modificação;
- **Liskov Substitution Principle** (Princípio da substituição de Liskov): se “S” é um subtipo de “T”, então os objetos do tipo “T”, em um programa, podem ser substituídos pelos objetos de tipo “S” sem que seja necessário alterar as propriedades deste programa;
- **Interface Segregation Principle** (Princípio da segregação de interfaces): a dependência de uma classe para outra deve ser baseada na menor interface possível;
- **Dependency Inversion Principle** (Princípio da inversão de dependência): módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Aborda também vários outros princípios consagrados do projeto orientado a objetos. É também leitura obrigatória para o moderno projeto de software.

## **Alur, D.; Crupi, J.; Malks, D.: Core J2EE Patterns**

(ALUR et al)

É um guia fundamental sobre os padrões de projetos recomendados para desenvolvimento com a plataforma Java Enterprise Edition. Também pode ser acessado pelo website: <http://www.corej2eepatterns.com/index.htm>.

É fundamental entender o que são “Dispatcher view”, “Front controller”, “Service to worker” e outros padrões utilizados no mundo Java EE, pois formam o “idioma” comum, através do qual você pode se comunicar com outros engenheiros de software pelo mundo afora.

É claro que cada plataforma de desenvolvimento adota um conjunto de padrões diferente. No mundo Java, este é o mais recomendado. Já no .NET temos o tópico MSDN “Patterns & Practices” (<http://msdn.microsoft.com/en-us/library/ff921345.aspx>), que contém informações semelhantes.

## Joshua Bloch: Effective Java (Java Efetivo)

(BLOCH)

No site da Sun (agora Oracle), onde este livro está citado (<http://java.sun.com/docs/books/effective/>), tem um comentário de James Gosling (criador do Java):

*I sure wish I had had this book ten years ago. Some might think that I don't need any Java books, but I need this one.*

*(“Eu com certeza gostaria de ter tido este livro há dez anos. Alguns podem pensar que eu não preciso de nenhum livro de Java, mas eu preciso deste.” – Tradução do autor).*

James Gosling, Fellow and Vice President, Sun Microsystems, Inc.

O meu comentário a respeito deste livro é: “como é que eu pude aprender Java sem ler este livro?”

Ele contém vários princípios a serem adotados no desenvolvimento de programa utilizando a plataforma Java. Alguns deles você já deve saber, mas com certeza existem outros que podem lhe surpreender.

Alguns exemplos:

- Item 1: **Consider static factory methods instead of constructors** (considere factory methods estáticos em vez de construtores);
- Item 2: **Consider a builder when faced with many constructor parameters** (considere o uso de um “builder” quando se deparar com muitos parâmetros no construtor);

- Item 11: **Override clone judiciously** (Sobrescreva o método “clone” com sensatez);
- Item 16: **Favor composition over inheritance** (favoreça composição em vez herança);
- Item 18: **Prefer interfaces to abstract classes** (prefira interfaces a classes abstratas);
- Item 74: **Implement Serializable judiciously** (implemente “Serializable” com sensatez);
- Item 75: **Consider using a custom serialized form** (considere a utilização de uma forma de serialização personalizada).

Eu sugiro que você visite o site do livro e leia o seu índice (<http://java.sun.com/docs/books/effective/toc.html>). Depois me diga vale ou não a pena comprá-lo.

## **Martin Fowler: Refatoração – Aperfeiçoando o projeto de código existente**

(FOWLER)

Este livro é a “cereja do bolo”! Depois de lê-lo, você vai pensar: “Como eu pude programar tanto tempo sem ler este livro!”

Fowler é um dos “monstros sagrados” modernos da engenharia de software. Neste livro, ele explica o que é refatoração e como implementá-la de maneira simples e prática. Por exemplo, o conceito de código autodocumentado é muito bem explicado, com exemplos simples e práticos.

Eu acho que muito desenvolvedor experiente deveria ter humildade e ler este livro, pois ajudaria muito em suas carreiras. Muitas crenças e mitos são simplesmente “vaporizados” pela lógica simples e contundente de Fowler.

Ele possui citações excelentes, que derrubam mitos, como esta:

*When you feel the need to write a comment, first try to refactor the code so*

*that any comment becomes superfluous. (Se você sentir a necessidade de escrever um comentário, tente primeiramente refatorar o código, de modo que qualquer comentário se torne supérfluo.).*

O capítulo “Bad smells in code” (maus cheiros no código) é excelente e leva-nos a refletir sobre o que fazemos no dia a dia.

Outro dia estava conversando com dois colegas mais antigos de profissão sobre comentários. Eu estava falando sobre “código autodocumentado” e eles entenderam “código comentado”, um engano muito comum. Quando eu comecei a explicar por que não gosto de comentários no código, eles explodiram em revolta! Acharam um absurdo! Bem, é difícil mudar velhos hábitos... para estas pessoas eu prefiro repetir a “Granny” do Javaranch (<http://www.javaranch.com/granny.jsp>):

*Debug only code - comments can lie. (Depure apenas o código – os comentários podem mentir.)*

## **Maurice Naftalin, Philip Wadler: Java Generics and Collections**

[Java Collections]

Este livro é bem prático e explica muito bem o uso de Generics, Reflection, padrões de projeto e coleções. Ele faz comparações entre os vários tipos de implementações de coleções utilizando a notação “Big O” de análise de algoritmos. Uma ferramenta fundamental para subsidiar suas decisões de implementação.

Se você utiliza .NET, existem livros semelhantes, como o de Michael McMillan: “Data Structures and Algorithms Using C#”.

## **Derek C. Ashmore: The J2EE Architect’s Handbook**

[Ashmore]

Não é um livro muito conhecido, mas é excelente. É necessário ler algum livro sobre arquitetura de aplicações Java EE ANTES de começar a trabalhar em um

projeto corporativo, e este é um excelente começo! O livro é bem interessante porque aborda assuntos como:

- Planejamento da arquitetura;
- Estruturação da aplicação em camadas;
- Planejamento e execução de testes.

Ele ensina até mesmo a utilizar “profilers” para detectar gargalos de desempenho.

Se você utiliza plataforma .NET, tem um guia no MSDN que é equivalente a este livro: “Microsoft Application Architecture Guide, 2nd Edition”:

(<http://msdn.microsoft.com/en-us/library/dd673617.aspx>).

## **Roger S. Pressman: Engenharia de software**

(PRESSMAN)

Um verdadeiro compêndio sobre engenharia de software, este livro é o resumo de um curso de graduação em engenharia de software. Ele contempla os vários aspectos do projeto, da construção e do teste de software, com uma abordagem bastante didática.

Se você quer conhecer os modelos de desenvolvimento (ágil, iterativo, formal etc.), a engenharia de requisitos e de teste, este é o livro mais adequado para estudar, além de ser um excelente preparo para concursos públicos!

## 6.

# Práticas e técnicas

Além dos princípios e da teoria, existem práticas e técnicas (incluindo ferramentas) que podemos utilizar para melhorar a qualidade do software que produzimos. O propósito deste “Guia de campo” é exatamente oferecer uma abordagem prática para a execução de projetos de software.

Como já deve ter ficado claro, não é propósito deste livro ensinar gerenciamento de projetos. Igualmente, não é um compêndio sobre arquitetura ou engenharia de software. O que desejamos é fornecer um guia de atitudes proativas que, se tomadas no momento adequado, podem reduzir custo, prazo e aumentar a qualidade do código-fonte produzido.

Entre os vários exemplos de práticas e técnicas importantes estão:

- **SCM – Software Configuration Management** (Gestão de Configuração de Software): um processo para controle de mudanças em um projeto de software, com manutenção de versões e gestão de liberação (release) do produto;
- **Peer reviews** (Revisões por pares): promover sempre revisões técnicas, feita por outros desenvolvedores, de cada produto de trabalho em um projeto de software;
- **TDD – Test Driven Development** (Desenvolvimento orientado a testes): uma técnica de desenvolvimento na qual os testes guiam a implementação de funcionalidades;
- **Gestão de montagem e componentes**: controlar o processo de montagem (“build”) e as versões de componentes é uma prática que garante a qualidade do software, evitando a “síndrome da máquina do desenvolvedor

(\*)”;

- **Integração contínua:** integrar as unidades ao sistema continuamente, recuperando e montando o projeto com base no repositório. Isto evita “surpresas” na véspera da demonstração para o Cliente;
- **O princípio das soluções:** muitos problemas podem ser evitados se utilizarmos os princípios adequados no momento de escolha das soluções. Pouco adianta tentar resolver problemas depois que o software foi desenvolvido. É preciso atuar ANTES de programar, ou seja, no momento em que as soluções “nascem”.

(\*) Tudo sempre funciona na estação de trabalho do desenvolvedor, porém, quando precisamos executar o software em outra máquina, um servidor, por exemplo, começamos a ter problemas com versões de componentes e outras incompatibilidades.

## Momentos

Tomar a atitude certa no momento certo evita aquela sensação de estar “construindo o avião durante o voo”, que acontece quando tentamos, atabalhoadamente, fazer as coisas fora de hora. Muita gente simplesmente começa a programar, normalmente uma ou duas pessoas, mais preocupadas em mostrar resultados do que em organizar as coisas. Vão atropelando e esquecendo de fazer coisas simples, como criar um ambiente de programação padronizado, ou escrever um documento de arquitetura abrangente. Aí quando entra um programador novo na equipe o caos se instala.

É muito importante deixar claro em que momentos devemos tomar quais atitudes. Inicialmente, demos o momento anterior à construção do código-fonte, quando os requisitos ainda não estão muito claros ou ainda não temos todos os recursos ao nosso dispor. Há muito o que fazer antes de “meter a mão na massa”, coisas que muita gente deixa para fazer só quando o projeto está em andamento.

## Antes

Antes de começar a construir código-fonte e artefatos técnicos, é fundamental preparar as ferramentas, por exemplo: o ambiente de desenvolvimento,

processos e ferramentas de gestão (de fontes, de componentes), o desenho geral da arquitetura, selecionar componentes, priorizar as tarefas de maior risco, estimar atividades etc.

Outras “ferramentas” que devem ser preparadas são os programadores! Sim, os que vão trabalhar no projeto. É necessário fazer um levantamento dos conhecimentos necessários para concluir o projeto e criar um plano de capacitação, ou então terceirizar parte da construção, transferindo as atividades de maior risco para outros. Temos que ser humildes e reconhecer nossas limitações, ao contrário do que muitos desenvolvedores experientes costumam fazer. Nada é “moleza” quando se tem um cliente no calcanhar!

## **Durante**

Temos que pensar nas soluções que estamos criando, selecionando as que melhor se adequem aos requisitos do projeto, precisamos programar com cuidado, criando código-fonte simples de entender e manter, e precisamos testar continuamente, com efetividade. Também é importante analisar métricas e indicadores durante a construção, pois podem apontar necessidades de “refactorings”.

Mesmo que não utilizemos TDD (Test Driven Development), precisamos sempre criar casos de teste para cada comportamento (\*) que criamos. Eles devem ser testes unitários com boa cobertura (daí a importância de métricas), de modo a garantir a qualidade do software.

(\*) Comportamentos são relacionados a responsabilidades que as classes do projeto devem assumir. Para cada comportamento, deve haver uma Interface e, para cada interface, um teste apropriado. Pense sempre em “comportamentos” do que em classes concretas.

## **Depois**

O código-fonte já está pronto, seja todo ele ou uma parte, logo, precisamos integrar ao conjunto do sistema executando testes. Além disto precisamos avaliar o código, de modo a saber se melhoramos ou pioramos sua qualidade geral.

Temos que revisar os produtos de software, para validar implementação de



requisitos, avaliar adequação as normas e padrões, além de verificar anomalias.

Ao final de tudo, precisamos executar testes de sistema para avaliar se o sistema construído atenderá aos requisitos não funcionais especificados no documento de arquitetura.

## Ferramentas

Nos próximos capítulos, passaremos a discutir as atitudes que devem ser tomadas, de acordo com o momento do projeto. Também vamos instalar e utilizar várias ferramentas, todas Open Source.

Para começar, vamos utilizar o sistema operacional Linux Ubuntu, com a implementação Java Open JDK 6. É claro que você pode utilizar outro sistema operacional, como: Microsoft Windows, Apple MacOS X, Oracle Solaris ou Red Hat. Para os baseados em arquitetura Unix (MacOS, Solaris ou Red Hat), sempre há versões específicas. Para o Microsoft Windows, nem sempre. Porém, existem ferramentas alternativas.

Outras ferramentas que vamos utilizar e que você já pode ir baixando:

- IDE “eclipse” para Java EE, versão “Indigo”: (<http://eclipse.org/downloads/>);
- Apache Maven (<http://maven.apache.org/download.html>) – atenção: algumas distribuições de software já incluem o Maven, ou então já possuem pacotes prontos para instalação. Verifique antes de baixar o Maven diretamente;
- Apache Archiva (<http://archiva.apache.org/>);
- Apache Continuum (<http://continuum.apache.org/download.html>);
- Apache Subversion – Cliente e Servidor (<http://subversion.apache.org/packages.html>) – atenção: algumas distribuições de software já incluem o SVN ou possuem pacotes prontos, como é o caso do Ubuntu (“apt-get install subversion”/“apt-get install libapache2-svn”);

- Plugin Maven2 para eclipse. Use o “update site” para instalar via Eclipse: <http://eclipse.org/downloads/>;
- Plugin Subversion para eclipse. Use o que já vem no update site da versão “Indigo”: <http://download.eclipse.org/releases/indigo>, selecione “Collaboration tools”;
- Apache Tomcat (<http://tomcat.apache.org/download-60.cgi>) – pode vir pré-instalado ou possuir um pacote específico em sua distribuição. Verifique;
- Apache Derby ([http://db.apache.org/derby/derby\\_downloads.html](http://db.apache.org/derby/derby_downloads.html)).

## Sobre o projeto exemplo

Vamos utilizar um projeto para exemplificar muitas coisas neste livro. É um projeto simples, que já está pronto, sendo desenvolvido em Java EE com JSF, Facelets e Hibernate. Este projeto pode ser baixado da Internet (veja na introdução como baixar os complementos) e utilizado livremente.

Trata-se de um microblog, como o Twitter, muito simples. Nós não vamos entrar em detalhes sobre como ele foi construído. Se for necessário, mostraremos outros exemplos (também disponíveis para download). Este aplicativo foi criado para um curso rápido de JSF / Hibernate, portanto, não tem pretensões de ser um software com alta qualidade. Eu não me preocupei com isto, pois apenas queria ter algo pronto para poder usar Maven, Archiva e outros softwares. Sugiro que o utilize como um rascunho para exercitar o que vamos estudar no livro.

## 7.

# Antes da construção do código-fonte

*Poor management can increase software costs more rapidly than any other factor.*

(“Má gestão pode aumentar os custos de software mais rapidamente do que qualquer outro fator.” – Traduzido pelo autor)

Barry Bohem, “Software Engineering Economics”, Prentice-Hall, Inc., 1981.

Imagine-se arrumando as malas para uma viagem à África. Você vai para um safári no meio do Quênia e tem que se preparar para diversas situações. É muita coisa para lembrar e pouco tempo para arrumar tudo. Bem, a preparação anterior à programação de um sistema tem o nível de complexidade semelhante.

Quanto melhor você planejar, mais preparado estará para o desenvolvimento. Quando o trabalho “braçal” começar, você terá pouco tempo para lidar com tarefas que já deveriam ter sido feitas.

Imaginemos que você esteja naquela fase inicial, na qual os primeiros requisitos estão chegando, a arquitetura sendo desenhada, mas ainda não tem a equipe completa, ou mesmo o nível de detalhe suficiente para começar. É neste momento que você deve se preparar para o desenvolvimento do projeto. A primeira coisa a fazer é dar uma olhada nos grandes requisitos, ou macrorrequisitos, ou seja, ainda sem o nível de detalhamento necessário. Participar das reuniões junto com os analistas de requisitos é muito importante, mesmo que seja só em algumas delas. A ideia é capturar os requisitos mais importantes, aqueles que “delineiam” a aplicação.

O *Rational Unified Process* (PRESSMAN) tem a denominação “Requisitos arquiteturalmente significativos” porque a solução é desenhada para atendê-los. Eu acrescentaria que também devem ser considerados os de maior risco. Uma análise preliminar de riscos deve ser efetuada para poder selecioná-los. É claro

que este não é um livro sobre arquitetura de software, logo, não nos concentraremos nisto, mas é necessário falar um pouco sobre a importância do estudo arquitetural para a qualidade do software.

Estas atitudes devem ser tomadas ANTES do início do projeto, de modo a garantir um trabalho tranquilo e sem “solavancos”. A visão geral da arquitetura nos dá uma ideia do que esperar, de quais componentes serão necessários e da complexidade geral do desenvolvimento. Então, podemos preparar um ambiente de desenvolvimento controlado e organizado, evitando problemas no período mais crítico (quando o “taxímetro” estiver rodando).

## **O que “arquitetura” tem a ver com “ambiente de desenvolvimento”?**

Excelente pergunta! Na verdade, um colega meu a formulou quando mostrei o livro a ele. A melhor resposta é: **TUDO!** Como vamos selecionar a “IDE”, os componentes e montar o ambiente, se não sabemos o tipo de projeto que vamos desenvolver? Além disto, os critérios para seleção e montagem do ambiente também podem variar. Por exemplo, se vamos criar um pequeno sistema de uso interno, com pouca funcionalidade, então podemos relaxar um pouco e usar ferramentas mais simples, com menor requisito de segurança e robustez.

## **O moderno engenheiro de software tem que gerenciar**

A tendência é, cada vez mais, montarmos software a partir de componentes prontos, escrevendo apenas código-fonte de interface entre eles e o usuário. Desta forma, o engenheiro de software tem que saber integrar e gerenciar toda essa variedade de componentes, tanto internos como externos. Como disse Barry Bohem na citação reproduzida no início deste capítulo, a má gestão é o que mais encarece o custo do software.

Para dar um exemplo, vamos ver três situações comuns, as quais, tenho certeza, já aconteceram com você ou com algum conhecido:

- *Um desenvolvedor tem problemas para compilar e testar seu projeto. Ele alega que o problema está no ambiente ou nos componentes compartilhados, e não em sua parte do código-fonte (ou da configuração). Os outros desenvolvedores não têm problema algum, e você passa três dias*

*até descobrir o problema com um plug-in que estava sendo usado no Eclipse da máquina original;*

- *O projeto está para entrar em testes de aceitação. Será feita uma demonstração para os usuários. Você pediu a última versão dos arquivos dos desenvolvedores, integrou e montou o pacote, enviando para o pessoal de infraestrutura. No momento da demonstração, uma das funcionalidades mais importantes apresentou um problema (que havia sido resolvido anteriormente). Você vai investigar o que aconteceu, enquanto o Cliente, revoltado, adia o teste em mais uma semana;*
- *O projeto utiliza três componentes internos, feitos por outras equipes. Cada cópia de cada componente é obtida de maneira diferente: via anexo de e-mail, via pen drive ou então de dentro do CVS de outra equipe. Tudo vai bem, até que os testes de integração do projeto começam a falhar sem explicação. Você perde uns dois ou três dias até descobrir que as versões dos componentes utilizados estão diferentes em várias partes do seu projeto. Então perde mais alguns dias tentando localizar as versões corretas.*

Essas três situações demonstram como os problemas de gestão podem aumentar o custo do software. Em todos os casos houve perda de tempo, além de problemas de relacionamento com o Cliente, com outras equipes ou entre os desenvolvedores.

## **A intenção desta parte do livro**

Antes de entrar em maiores detalhes, gostaria de salientar que este não é um livro de referência das ferramentas e técnicas aqui mencionadas. Ele não esgota o assunto. Na verdade, trata-se de um roteiro para elaboração de software com qualidade. As ferramentas e técnicas são apresentadas como sugestões de atitudes proativas, não tendo a intenção de encerrar as discussões.

## **Visão geral da arquitetura**

Você já deve ter uma noção de qual será a arquitetura do futuro sistema. Ainda não dá para saber todos os detalhes, pois o levantamento de requisitos mal começou, certo? Mas você já deve saber algumas coisas, por exemplo: trata-se

de uma aplicação web? Haverá interfaces via web services? Serão acessados sistemas externos? Qual a interface? Haverá características especiais, como geração de PDF ou processamento de XML?

Bem, essas características podem representar riscos técnicos para o futuro sistema, devendo ser tratados antecipadamente. Outra situação que pode acontecer é a exigência de utilizar plataforma específica, demandada pelo Cliente.

É bom fazer um desenho geral da arquitetura, sem se preocupar com formalismo neste momento. Nada de usar ferramenta “Case” ou de criar artefatos complexos. Este desenho geral deve ser claro o suficiente para transmitir a ideia para os outros desenvolvedores, sem que seja gasto muito tempo em sua elaboração. Neste resumo, o que deveria constar:

- As camadas da aplicação (apresentação, lógica, persistência e integração);
- Os tipos de componentes a serem desenvolvidos e seu relacionamento com os principais casos de uso;
- Qual é o SGBD, o servidor de aplicação etc;
- Os componentes externos necessários para complementar as características da linguagem, por exemplo: “parsers” de XML, geradores de relatório, utilitários etc;
- Tipos de interfaces: internas e externas, síncronas ou assíncronas, tipos de protocolos etc.

Muitas alternativas de soluções arquiteturais podem existir; logo, para basear suas futuras decisões arquiteturais, você deve estabelecer uma lista de critérios. Depois, pode usar algum tipo de sistema de pontuação (como o AHP – Analytic Hierarchy Process – [http://pt.wikipedia.org/wiki/Analytic\\_Hierarchy\\_Process](http://pt.wikipedia.org/wiki/Analytic_Hierarchy_Process)) para justificar sua decisão. Os critérios podem ser:

- Desempenho;
- Custo;
- Conhecimento da equipe;

- Prazo;
- Manutenibilidade.

Vamos a um pequeno exemplo:

*O sistema de controle logístico da loja XPTO deverá receber os pedidos conciliados (aqueles cujo pagamento já foi processado e que a mercadoria já está em estoque) e preparar o conhecimento de carga, selecionando o transportador e preparando as informações de entrega, alterando o status do pedido. O cliente terá acesso ao status do serviço através do web site da loja. O sistema deverá contatar a transportadora (via interface a ser definida – hoje é por e-mail) e dar informações ao agente de logística sobre o andamento do processo.*

*Ele deverá ter a opção de imprimir e/ou enviar o conhecimento em formato PDF.*

Uma boa ideia é fazer uma reunião de “brainstorm” para separar os principais módulos e integrações necessários.

Parece claro que o sistema terá as seguintes interfaces:

- Com o sistema de pedidos da loja;
- Com o website da loja;
- Com as transportadoras.

E também deverá ter uma interface de usuário para que o agente de logística possa acompanhar o processo.

Estas interfaces devem ser discutidas com o usuário, e isto é feito pelos analistas de requisitos; logo, é interessante que você (ou outro técnico) participe das sessões de levantamento, de modo a garantir que a preferência do usuário seja capturada de maneira adequada.

## **Mecanismos de interface entre sistemas**

Integrar diferentes sistemas é sempre uma tarefa de alto risco. Vamos ver as alternativas:

- **Mensagens assíncronas** (JMS – Java Message Service): um tipo de interface de baixo acoplamento, ideal para processamento sob demanda;
- **Web service** (JAX-RS / JAX-WS ou então qualquer camada compatível com WS-I): interessante, mas o acoplamento (e controle) são um pouco maiores. Ou o sistema de pedidos deverá hospedar o Web Service ou então o de logística, mas pode ser uma solução, caso os dois sistemas sejam feitos em plataformas diferentes (por exemplo: Microsoft .NET x Java EE);
- **Via banco de dados**: muitos engenheiros de software adoram este tipo de interface. Eu não gosto, pois gera maior acoplamento, já que uma das aplicações precisa conhecer o modelo de dados da outra. Além disto, exige algum tipo de temporização, pois uma das aplicações deverá ficar “vigilando” o banco da outra;
- **Via Socket server**: uma interface de alto risco! Criar um Socket server não é tarefa trivial, pois, além do controle de concorrência e de consumo de recursos, é necessário desenhar um protocolo para troca de mensagens. Além disto, você está se arriscando a criar vulnerabilidades de protocolo e de servidor (Buffer Overrun);
- **Via File system**: você ficaria impressionado(a) com a quantidade de engenheiros de software que adotam este tipo de “gambiarra”! Uma aplicação grava um arquivo em uma determinada pasta e a outra consulta, utilizando algum tipo de temporização (Cron etc.). Pode ser a solução mais adequada em certos casos, dependendo do prazo ou custo, mas deve ser aplicada com muito cuidado.

## Interface com o usuário

A princípio, a interface com o usuário (agente de logística) poderia ser web. Utilizando as recomendações do framework Java EE, seria uma aplicação do tipo Dynamic Web Site, como uma aplicação JSF – JavaServer Faces. Poderia utilizar “JPA” ou “Hibernate” para persistência.

## Interface com o sistema de pedidos



Existe um sistema de pedidos na empresa e o controle logístico deve obter os pedidos conciliados, alterando seu status depois do processamento. Normalmente são poucas interações. Aparentemente, o sistema de pedidos e o de controle logístico rodarão no mesmo ambiente, logo, temos maior controle sobre a interface. Eu recomendaria o uso de Mensagens assíncronas transacionais, a não ser que as plataformas sejam diferentes.

## **Interface com a loja virtual**

Como vimos, o Cliente da loja poderá acessar seu pedido e saber o status da entrega; logo, é necessária uma interface com o website da loja. O importante é que haverá várias interações entre os dois sistemas, já que a entrega pode mudar de status várias vezes. Bem, uma coisa é certa: o sistema da loja roda em uma DMZ, que pode estar dentro da empresa ou não. E precisamos atualizar as informações. Talvez a exposição do status via um Web service seja a melhor opção.

## **Interface com a transportadora**

O sistema precisa enviar os conhecimentos para a transportadora selecionada. Certamente será necessário saber a situação das entregas. São empresas diferentes, logo, é melhor adotarmos uma solução de baixo acoplamento e multiplataforma, como os Web services.

Notaram que o sistema também terá que gerar PDF? Bem, esta é uma característica especial importante, que devemos destacar. Podemos utilizar alguns componentes para gerar PDF, como o iText (<http://itextpdf.com/>) ou mesmo o JaperReports (<http://jasperforge.org/projects/jasperreports>), mas temos que considerar a questão da instalação desta característica, pois geração de PDF é uma atividade que costuma consumir muitos recursos de um servidor.

## **Mitigando riscos técnicos**

Analisar antecipadamente a arquitetura da aplicação é uma maneira de aumentar a qualidade do código-fonte, pois evitamos “sustos” durante o projeto. Estes “sustos” normalmente provocam alterações de última hora, implementadas de maneira ruim e, normalmente, mal testadas.

Devemos mitigar os riscos técnicos ANTES de começar a desenvolver a aplicação propriamente dita. Podemos criar pequenas “provas de conceito” que permitam estudar as possíveis soluções enquanto temos tempo para isso.

No caso do exemplo dado, temos as interfaces todas. Podemos ir conversando com os responsáveis dos outros sistemas, de modo a obter o máximo de dados antecipadamente, como os arquivos WSDL e esquemas XML dos Web services. Desta forma, podemos criar clientes e serviços simulados, utilizando os esquemas e padrões fornecidos. Podemos até tentar enviar e receber arquivos, validar opções etc. Isto é essencial, especialmente se a equipe não tiver muita experiência com Web services.

Temos outras duas características que merecem ser estudadas: a interface com o sistema de pedidos, que será feita via mensagens assíncronas, e a geração de PDF. É preciso saber o que existe no ambiente de produção e se já existe algum padrão para uso de JMS na instalação onde o sistema será processado. E, quanto à geração de PDF, precisamos fazer alguns testes, inclusive de desempenho, com monitoração de recursos.

Faça uma lista dos componentes que você precisa utilizar e pesquise se já existe alguma coisa no catálogo de software da sua empresa (ou do Cliente). Lembre-se das regras de ouro e só desenvolva o que você precisa realmente fazer.

Finalmente, construa uma lista de implementações necessárias, em ordem decrescente de risco para o projeto. Use esta lista como um guia de onde concentrar mais esforços e atenção. Veja quais são os requisitos para o ambiente de desenvolvimento, para os componentes comuns e como vai gerenciar a integração e o uso disso tudo.

## **O ambiente de desenvolvimento**

Finalmente! Estamos começando a falar de coisas práticas, não? O ambiente de desenvolvimento precisa estar pronto ANTES de você iniciar. Vamos selecionar e configurar ferramentas que auxiliarão a manter a qualidade do projeto.

Para começar, precisamos ter um bom ambiente de desenvolvimento configurado, o que já é uma tarefa bem complexa por si só.

## **Mas qual é a vantagem? Qual a contribuição para a melhoria da qualidade?**

Eu vejo as vantagens:

- Com um ambiente organizado, a curva de aprendizado é menor, facilitando a entrada de novos desenvolvedores ou a migração do projeto para outras equipes;
- Um bom ambiente evita que problemas de configuração atrapalhem o desenvolvimento. Podem existir problemas que “mascarem” problemas reais no código-fonte, fazendo com que os desenvolvedores percam tempo precioso.

Quanto à contribuição para melhoria da qualidade, temos um impacto direto na manutenibilidade do futuro sistema, já que o ambiente organizado auxilia a entender melhor o projeto.

## **O sistema operacional**

Sim! Bem lembrado! Qual será o sistema operacional que a equipe vai usar? E para quantos bits (32 / 64)? Muito importante determinar isto. É claro que alguns vão querer usar Microsoft Windows, outros vão usar Linux e haverá um grupo que teimará em usar Mac OS. Como Java é multiplataforma, as pessoas não prestam muita atenção a este detalhe, que pode causar diversos problemas ao projeto.

A escolha do sistema operacional é um assunto complicado. Normalmente, o cliente ou o patrocinador do projeto determinam qual será o sistema operacional. Se for um aplicativo distribuído, como um web site ou web service, o sistema operacional do ambiente de “produção” poderá ser diferente do sistema operacional dos desenvolvedores. Porém, se for um aplicativo pessoal (“desktop”), então o sistema operacional deverá ser o mesmo para toda a equipe.

Mesmo sendo um aplicativo “desktop”, se for desenvolvido em Java, não há necessidade que o sistema operacional seja o mesmo do Cliente. Porém, existem algumas coisas que podem variar conforme o sistema operacional da equipe, como por exemplo:

- Nomes de caminhos de arquivos e diretórios;

- Versões da plataforma Java e da IDE;
- Disponibilidade dos componentes a serem utilizados.

A solução que eu proponho está no final deste capítulo (virtualização), mas, a princípio, o melhor é baixar uma “lei” obrigando todo mundo a usar o sistema operacional mais comum. Se a empresa (ou a equipe) está mais acostumada com o Microsoft Windows, então será em Windows! Se algum “cabeça de bagre” teimar em usar o Linux ou o seu MacBook Pro, então ele deve ser advertido de que será responsável por resolver quaisquer problemas de compatibilidade, seja no servidor ou na máquina dos colegas.

Para evitar esses problemas, recomendo que você prefira ambientes virtuais. Se as máquinas físicas tiverem pelo menos 8 GB de RAM, então não haverá problemas de desempenho.

Sobre o fato de ser 32 ou 64 bits, é importante que todos usem rigorosamente a mesma plataforma, mesmo que seja simulada (máquina virtual).

Neste livro, assumimos que o sistema operacional será Linux Ubuntu, 32 bits.

## O kit de desenvolvimento Java

Todos devem utilizar uma versão Java Development Kit (JDK), e tem que ser na versão que consta no documento de arquitetura, ou a versão que o Cliente deseja. Pode ser a versão homologada para executar no ambiente de produção.

Hoje existe mais de uma opção de JDK a ser instalada:

- **Oracle Java JDK:** <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, oficial e suportada pela Oracle. Uma boa escolha;
- **Oracle JRockit JDK:** <http://www.oracle.com/technetwork/middleware/jrockit/downloads/index.html?ssSourceSiteId=ocomen>, uma das JVMs mais rápidas que eu já vi, agora com um Kit JDK próprio;
- **Open JDK:** <http://openjdk.java.net/>, um projeto open source, da qual a Oracle é um patrocinador, muito utilizada em ambientes Linux;

- **Apache Harmony:** <http://harmony.apache.org/>, uma implementação verdadeiramente Open Source, totalmente desconectada da Sun ou da Oracle, que, infelizmente, foi movida para o “sótão” (attic) do grupo Apache e, provavelmente, será descontinuada no futuro.

Para ambientes Mac, existe o tradicional JDK criado pela Apple, mas, daqui para a frente, a Oracle e a Apple firmaram um acordo para portar o JDK da Apple para o Open JDK; logo, é melhor consultar o site para ver se já está disponível: <http://openjdk.java.net/projects/mac-osx-port/>.

E existem pelo menos três versões ativas (ainda em uso):

- Java SE 5;
- Java SE 6;
- Java SE 7.

Você até pode utilizar uma versão mais recente do Java SE, mas tem que definir qual é o “target”, ou o nível de compatibilidade que TODOS os membros da equipe deverão configurar. Se você definir que o “target” será 1.6, então todos terão que configurar sua IDE e seus builds para gerar código “bytecode” na versão 1.6.

Então, são duas atitudes importantes: qual é o tipo de JDK e qual a versão do “target” que será utilizada.

Neste livro, estamos assumindo que a JDK será a Open JDK com o “target” para Java 1.6.

## O ambiente de desenvolvimento

Se vamos desenvolver em Java, então precisamos de um bom editor. Se for um ambiente integrado (Integrated Development Environment – IDE), com edição, compilação e depuração, melhor ainda! Existem algumas IDEs interessantes no mercado:

- **Netbeans** (<http://netbeans.org/>) é uma das mais antigas IDEs para Java. É gratuita e patrocinada pela Oracle, sendo totalmente compatível com os

padrões Java;

- **JCreator** (<http://www.jcreator.com>) é também bastante antiga, mas muito fácil de usar. Tem menos bugs que as IDEs gratuitas, além de possuir um bom suporte. O preço não é muito alto;
- **IntelliJ** (<http://www.jetbrains.com/idea/>) muito boa, com muitas funcionalidades. Possui versão community gratuita;
- **JDeveloper** (<http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html>) a IDE gratuita da Oracle. Possui total integração com JSF, além de suporte ao ciclo de vida de aplicações;
- **Eclipse** (<http://www.eclipse.org>) é uma das mais populares, sendo gratuita, open source e versátil.

Seja qual for a IDE escolhida, você TEM que obrigar os outros membros da equipe a utilizá-la na configuração que você determinar! O livre arbítrio tem limites! Eu já vivi situações nas quais os programadores decidiam o que e como queriam usar. Imagine a “bagunça” generalizada que era o projeto. Você tem que fazer igual ao ditado:

Sou a favor da democracia, e todos devem dar sua opinião, mas devem fazer o que EU digo!

Antes de iniciar o projeto é o momento certo para se discutir qual IDE será utilizada, mas esta discussão deve ter limites. Por exemplo, existem fatores que são indiscutíveis e que podem decidir por você:

- Qual é o padrão do Cliente ou da sua empresa?
- Qual IDE é a mais conhecida pela equipe?
- Qual a que apresenta menor risco?

Eu detesto estar errado, logo, prefiro ver o que as pessoas estão utilizando. Corri atrás de algumas pesquisas e consegui alguns números:

- **Replay solutions 2010** - [http://info.replaysolutions.com/l/1772/2010-03-08/12KD1/1772/18317/Survey\\_Results.pdf](http://info.replaysolutions.com/l/1772/2010-03-08/12KD1/1772/18317/Survey_Results.pdf) (1.101 respostas): **Eclipse** – 66%, NetBeans – 17%;
- **Zero turn around 2011** - <http://zeroturnaround.com/java-ee-productivity->

[report-2011/](#) (1.027 respostas): **Eclipse** – 65%, IntelliJ IDEA – 22%;

- **Dice.com** (ofertas de emprego pedindo IDE específica) - <http://www.dice.com/> (dezembro de 2011): **Eclipse** – 1592 empregos, NetBeans – 125 empregos;
- **e-panelinha** (ofertas de emprego pedindo IDE específica) – <http://www.e-panelinha.com.br> (dezembro de 2011): **Eclipse** – 5, outras – zero.

Bem, o povo está dizendo que a IDE Eclipse é a mais popular. Provavelmente é mesmo! Nas três últimas empresas em que trabalhei ela era a IDE padrão. Embora não seja exatamente a melhor, é a mais utilizada e versátil de todas. Você não estará errando se escolher o Eclipse, que é a minha escolha para este trabalho.

Você deve orientar os desenvolvedores para configurar o Eclipse de modo a utilizar o Open JDK, com o “target” na versão 1.6.

Para adicionar o JDK:

1. Abra o menu “Window / Preferences”;
2. Expanda “Java” na lista à esquerda e escolha “Installed JREs”;
3. Adicione a “java-6-openjdk”, que no Ubuntu fica em: “/usr/lib/jvm/java-6-openjdk”.

Para configurar o “target” ou o nível de compatibilidade Java:

1. Abra o menu “Window / Preferences”;
2. Expanda “Java” na lista à esquerda e escolha “Compiler”;
3. Na caixa “Use default compliance level”, selecione “1.6”;

Neste livro, estamos utilizando o “Eclipse” versão “Indigo”, para Java EE.

## Plug-ins

Ok. O “**Eclipse**” ganhou, mas e quanto aos plugins? É muito importante limitar o número de plugins que serão utilizados. Por exemplo:

- SubEclipse para acessar SVN;
- m2e para usar o Maven;
- JBoss tools.

Você deve delimitar criteriosamente o conjunto de plugins permitidos. Na verdade, eu empacotaria o Eclipse (crie um arquivo ZIP) com o que é necessário e mande usarem exatamente como você disse.

Voltaremos a falar com maiores detalhes sobre plugins mais adiante. Por enquanto, levante qual é a necessidade de utilizar plugins e liste quais são importantes, separando daqueles que são pura preferência pessoal.

### **Configuração da codificação de caracteres**

Esta é outra origem de dores de cabeça! O Eclipse varia a configuração de codificação de caracteres (character encoding) em alguns sistemas operacionais, logo, arquivos com acentos criados em um sistema operacional podem aparecer incorretos em outro.

É melhor fixar um padrão a ser configurado em todas as IDEs do projeto. Eu recomendaria fortemente o uso de UTF-8, que é um padrão internacional, assim com o ISO, só que mais popular, devido à World Wide Web.

O “Eclipse” permite configurar qual será a codificação padrão para cada tipo de arquivo que ele cria ou edita. Selecione: “Window / Preferences” e expanda “General”, na lista à esquerda. Selecione “Content Types” e expanda “Text”. Vá selecionando cada tipo de arquivo (“CSS”, “DTD”, “HTML”, “Java properties”, “Java source” etc.). Para cada tipo de arquivo, vá para a parte inferior da tela e procure o campo “Default encoding”, mudando seu conteúdo para “UTF-8” e clique no botão “Update”.

O Eclipse mantém alguns arquivos em ISO-8859-1 e outros em UTF-8. Eu prefiro TUDO em UTF-8, em todas as plataformas, pois isto evita que artefatos gerados em uma estação de desenvolvedor apresentem problemas em outra, ou mesmo no servidor.

Todos devem utilizar a mesma codificação de caracteres.



## **Outros itens**

Ainda existem diversos itens relacionados com o ambiente de desenvolvimento, os quais falaremos ao longo do livro. Por exemplo: uso de repositórios de pacotes, SCM, integração contínua, avaliação contínua etc.

## **Crie um ambiente virtual**

Eu recomendo fortemente que você aproveite o tempo e crie um ambiente virtual ideal para o desenvolvimento do seu projeto. Hoje em dia, com a popularização dos softwares gerenciadores de máquinas virtuais (Paralels, Virtual-Box, VMWare, VirtualPC...), fica muito fácil criar e exportar máquinas inteiras.

## **Open Virtualization Format**

Várias empresas fornecedoras de soluções de virtualização se uniram e propuseram o formato único, conhecido como OVF. Com ele, é possível criar e exportar máquinas virtuais que possam ser executadas por qualquer software compatível. Existe o formato OVA (Open Virtual Appliance), que junta toda a configuração em um só arquivo, facilitando o transporte.

## **Por que eu faria isso? Qual a vantagem?**

Certa vez, trabalhei com uma colega muito, mas muito atrapalhada mesmo. Ela era tão ansiosa que ficava logo na defensiva caso acontecesse algum problema. Ela sempre dizia: “não é culpa minha” ou “meu programa está certo, o ambiente é que está errado”. E geralmente era uma bobagem que ela tinha feito. Só que, para provar seu erro, era necessário provar que o ambiente estava correto, e isto exigia instalar o projeto em outra máquina.

Parece incrível, mas problemas deste tipo (falsos positivos) acontecem frequentemente em equipes de desenvolvimento de software, consumindo tempo precioso, apenas para constatar que o erro é em outro lugar.

Então, existe pelo menos uma vantagem clara: ter um ambiente de referência para facilitar a resolução de problemas e a instalação de novas estações de desenvolvimento.

Porém, com o barateamento do hardware, é comum termos máquinas com 8 ou

16 gigabytes de memória e multicore (Intel i5, i7 ou AMD Phenom ou Opteron). Com um sistema operacional de 64 bits (Linux, Windows 7 ou Mac OS X) praticamente não existe limite de memória, logo, é possível instalar um sistema virtualizador e desenvolver diretamente na máquina virtual.

Veja só as vantagens:

1. Todos os desenvolvedores utilizarão o mesmo ambiente, configurado de maneira correta;
2. É fácil instalar novas máquinas, bastando importar o arquivo “OVA”;
3. O backup fica muito mais fácil de fazer.

### **Softwares de virtualização**

Existem muitos softwares de virtualização no mercado, tanto gratuitos como pagos. Eu utilizo muito o VirtualBox (<https://www.virtualbox.org/>), um produto open source, atualmente mantido pela Oracle. Ele é compatível com o padrão OVF / OVA e roda muito bem em sistemas operacionais Linux, Windows e Mac, além de suportar todos eles (\*).

(\*) Mac OS X não pode ser executado em ambientes virtuais, pois viola o acordo de licenciamento da Apple.

### **O que colocar na VM**

Você deve criar pelo menos uma máquina virtual (VM) com o ambiente de desenvolvimento completo, incluindo:

- Sistema operacional (Linux / Windows);
- Versão do Java (JDK);
- Drivers e outros componentes (JDBC etc.);
- IDE (Eclipse) com todos os seus plugins necessários para o desenvolvimento, e com a configuração do repositório do projeto (“SVN Repositories”);
- Editores de query e SQL (Squirrel, Oracle SQL Plus etc.), e com os arquivos de configuração (no caso da Oracle: TNSNAMES);

- Softwares acessórios: Maven, Subversion etc.

Eu criaria também máquinas para os principais servidores do projeto, por exemplo: o Servidor Web, o Servidor de Aplicação, de Banco de Dados etc. Com isto, fica fácil comunicar-se com o pessoal de infraestrutura, além de permitir a rápida instalação de servidores para teste.

## Gestão de configuração de software

De acordo com o Software Engineering Body of Knowledge (“Corpo de conhecimento da engenharia de software”) (IEEE, 2004), Software Configuration Management – SCM (“Gestão de configuração de software”) é um processo de suporte do ciclo de vida de software cujo propósito é controlar os itens de configuração de um software, por exemplo:

- Artefatos de código-fonte (classes e interfaces);
- Artefatos de controle de montagem (arquivos de controle da IDE, de ambiente, etc.);
- Artefatos de configuração do software (arquivos de propriedades, XML etc.);
- Artefatos de documentação do projeto (arquivos texto, diagramas etc.);
- Artefatos de persistência (arquivos SQL, stored procedures, triggers etc.).

Chamamos o conjunto de artefatos de um projeto de “base de configuração do projeto” e os próprios artefatos de “itens de configuração do projeto”.

Muita gente confunde SCM com gerenciamento de versões, o que é um tipo de “metonímia” (pegar parte para representar o todo). Afinal de contas, gerenciamento de versões é uma parte do processo de SCM.



As atividades do SCM são:

- **Gestão e planejamento:** determinação de papéis e responsabilidades, seleção de ferramentas, desenho do processo, mecanismos de controle e acompanhamento etc;
- **Identificação da configuração:** identificação de itens de configuração, relacionamento entre artefatos, controle de versão, “baselines” etc;
- **Controle de configuração:** o processo de aprovar, controlar e realizar mudanças em itens de configuração;
- **Contabilização e estado:** acompanhamento de atividades, geração de relatórios de estado e gerenciais sobre a base de configuração e as atividades;
- **Auditoria:** avaliação da conformidade dos itens de configuração com as normas e padrões, verificação de acessos e avaliação de “baselines”;
- **Liberação e entrega:** montagem (“building”) do software, geração de pacotes, processos de liberação de versões.

Nosso objetivo é dar a você uma ideia geral do que é o SCM e quais são as melhores práticas a serem adotadas para melhoria da qualidade. Assim, vamos examinar um pouco como deveria funcionar o processo de SCM e depois ver como implementar a parte prática.

### **Por que SCM? Qual é a contribuição para a qualidade?**

Eu trabalhei com um chefe que achava tudo isso uma bobagem. Ele tinha um sistema próprio de marcação de versões, que mantinha em seu computador, copiando tudo para uns disquetes (sim, faz muuuuuiito tempo...) com etiquetas coloridas. Ele se gabava de saber exatamente onde estava cada versão e o que estava em desenvolvimento. Bem, um dia ele esteve muito doente, precisando tirar licença médica. Adivinhe o que aconteceu? É óbvio: caos generalizado! Ninguém sabia mais quais versões deveriam utilizar, ou como gerar um pacote para a produção.

Se esta historinha não adiantou, então vou tentar ser mais claro. Um processo de

SCM:

- Organiza e mantém versões diferentes dos artefatos do projeto;
- Ajuda a manter o “roadmap” do software;
- Evita problemas com lançamento de novas versões.

E isto, obviamente, melhora a qualidade geral do produto de software.

## Gestão e planejamento

A primeira coisa que devemos fazer é planejar. Talvez sua empresa já tenha todo o processo de SCM pronto, logo, é melhor você estudá-lo. Mas, na maioria das vezes, o que existe é um repositório de código-fonte, cujo controle é feito pelos próprios desenvolvedores. De qualquer forma, vale a pena conhecer um pouco do processo de SCM e seu planejamento.

*Itens de configuração de software são patrimônio?*

Se respondeu “não” a esta pergunta, então responda a esta outra pergunta: “foi gasto dinheiro para construí-los (os itens de configuração)?” É claro que foi. O custo para construir um item de configuração engloba:

- Custos diretos: horas (homem/hora) gastas para codificar (ou escrever), testar e documentar;
- Custos indiretos: percentuais de horas de máquina, licenças de software, ar-condicionado, energia elétrica, custos administrativos, custos de folha de pagamento, impostos etc;
- Custo de oportunidade: difícil de mensurar, mas é o que poderia ser feito em vez daquele item específico, ou seja, quanto dinheiro poderíamos ganhar, caso não estivéssemos produzindo aquele item de configuração específico?

Então, se houve despesa, podemos considerar um item de configuração de software como patrimônio ou investimento. Na verdade, a tendência atual, segundo o **CoBIT** (<https://www.isaca.org/Pages/default.aspx>), é tratar os itens de configuração como investimentos, agrupando-os em um “portfólio de TI”, de

modo a avaliar seu retorno.

Para começar, é necessário planejar o processo de controle e solicitação de mudança (“change request”), que governará a manutenção dos itens de configuração. Podemos estabelecer alguns papéis para isto:

- **Comitê de controle de mudanças:** responsável por examinar, aprovar ou reprovar manutenções nos itens controlados pelo SCM. Normalmente, fazem parte o gerente do projeto, o gerente da área da equipe, o gerente de configuração e um representante do Cliente;
- **Gerente de configuração:** responsável pela manutenção do “baseline” do projeto, pela liberação de acessos, pelo controle de “checkouts” e “check-ins”, pelo acompanhamento dos “backups” etc;
- **Gerente do projeto:** responsável por designar pessoas para trabalharem nas solicitações de mudança e solicitar acessos;
- **Mantenedor:** responsável por realizar as mudanças solicitadas nos itens de configuração. Podem ser: desenvolvedores, analistas de requisitos, documentadores etc;
- **Gerente de liberação:** responsável por promover a montagem (“build”) e a liberação de versões do projeto. Pode ser a própria pessoa que gerencia a configuração.

O processo deve começar com uma “solicitação de mudança” (SM), na qual devem constar os itens a serem alterados ou adicionados. O Comitê de controle de mudanças deverá examinar e aprovar (ou rejeitar) a SM, e o Gerente de configuração deverá conceder acesso ao Mantenedor para que possa baixar o(s) item(ns) afetado(s).

Mas não termina assim... o Gerente do projeto deve acompanhar o trabalho do Mantenedor, e o Gerente de configuração deve auditar os prazos, de modo a evitar que fiquem “trabalhos em aberto” na base de configuração. Se for uma manutenção corretiva longa, talvez seja melhor criar um “branch” (ou cópia temporária) do projeto. Assim, caso haja outra SM em paralelo, não será afetada pela primeira.

Parece burocrático, mas é eficaz!

## Identificação da configuração

A função mais importante da identificação é estabelecer critérios para “branches”, “tags” e “baselines”, além de estudar o relacionamento entre os itens de configuração de um projeto.

### Baseline

Toda mudança implica em alguns estados dos itens de configuração: inicial, intermediário(s) e final. Uma “baseline” é uma imagem (pode ser cópia pura ou diferencial) dos itens de configuração em determinado instante do tempo.

O estado inicial é importante, pois, caso a SM precise ser removida (ou desfeita), os itens afetados deverão ser recuperados em seu estado inicial. Os estados intermediários podem ocorrer durante a realização das alterações propostas pela SM e podem ser associados a ações de “guarda” (ou “commit”) de itens de configuração. E, finalmente, o estado final é a situação dos itens de configuração após a execução da SM.

O conceito de “baseline” não é o “estado atual dos itens de configuração”, mas o estado deles em algum momento importante no tempo. Na verdade, os itens de configuração em um projeto podem existir em vários estados (ou versões) simultaneamente:

- **Cópia de trabalho** (“working copy”): é o estado dos itens na máquina de um desenvolvedor. Normalmente não são confiáveis, pois podem conter bugs e alterações ainda não testadas. Além disto, a cópia de trabalho é montada (“build”) com o ambiente de trabalho do desenvolvedor, podendo apresentar diferenças para o ambiente de execução final;
- **Trunk** (“tronco principal”): é a versão atual dos itens de configuração de um projeto. A política de uso do “trunk” é muito importante, pois direciona como os Mantenedores deverão trabalhar. Se você não especificar uma versão, é do “trunk” que obterá os itens de configuração;
- **Branch** (“ramo, galho”): é uma versão específica e em desenvolvimento dos itens de configuração. Pode ser devido a uma manutenção evolutiva ou corretiva (“bug fix”). Normalmente, contém versões em manutenção e está em uso por um ou mais Mantenedores. Podem existir diversos “branches” simultaneamente. Após a conclusão da SM que o motivou, o “branch” é

reintegrado ao “Trunk” através do processo de “Merge” (fusão ou integração). Um “branch” pode ser também integrado a um “Tag”, caso seja uma manutenção corretiva de uma determinada versão do software;

- **Tag / Label** (“rótulo”): é uma versão do projeto em um instante importante no tempo, também conhecida como “baseline” ou “snapshot”. Os “branches” podem ser criados a partir dos “tags” também.

Uma “baseline” é uma versão (ou estado) dos itens de configuração de um projeto em um determinado instante (“tag”). Alguns autores associam ao termo “baseline” o estado ATUAL dos itens de configuração, ou seja, o “trunk”, mas, na verdade, uma “baseline” é uma foto (“snapshot”) dos itens de configuração em um instante importante do projeto.

### Uso do “trunk” e rotina de “commit”

É necessário determinar como o “trunk” será utilizado e se todas as SMs serão executadas em “branches” separados ou não. Isto tem um impacto grande no projeto. Se o “trunk” for a versão ATUAL (ou em desenvolvimento) do software, então a maioria das ações será feita nele; logo, ele conterá uma versão não estável do software. Eis algumas sugestões:

- **“Trunk” estável:** o “trunk” sempre contém a última versão estável e aprovada dos itens de configuração; logo, todo o desenvolvimento e a manutenção ocorrerá em um “branch” à parte; o qual deverá ser reintegrado ao “trunk” somente quando testado e aprovado pelo Comitê;
- **“Trunk” não estável:** o “trunk” contém a última versão dos itens de configuração, ou a versão em desenvolvimento. Os Mantenedores sempre adicionam novos itens ao “trunk”; logo, a versão ainda não está aprovada, sendo considerada “não estável”. Neste modo, evita-se criar “branches”, a não ser que seja para “bug fix” de alguma versão anterior. Igualmente, a versão estável deverá ser algum “tag” anterior;
- **“Trunk” de integração:** o “trunk” contém a última versão que passou em testes de integração, ou seja, pode não ser estável e aprovada, mas foi totalmente integrada sem problemas. Neste caso, usa-se um “branch” para desenvolvimento e seu conteúdo é integrado ao “trunk” quando o teste de integração passar sem problemas.



Além da utilização do “trunk”, uma política de “commit” deve prever a periodicidade e a rotina de trabalho de cada desenvolvedor. Exemplos:

- “Commit” diário: todos devem fazer “commit” pelo menos uma vez, ANTES do final do expediente;
- “Commit” de unidade: todos devem fazer “commit” só quando terminarem sua unidade.

O que não pode acontecer é bagunça! Muita gente evita criar “tags” e “branches”, usando o “trunk” como única versão dos itens de configuração. A falta de uma rotina de “commit” bem estudada e aceita por todos é a causa de todos os conflitos de versões, o que leva a operações custosas de “merge” e até gera desentendimentos entre a equipe.

Martin Fowler, em seu artigo sobre Integração Contínua (<http://martinfowler.com/articles/continuousIntegration.html>), sugere que todos os desenvolvedores devam fazer “commit” uma vez ao dia (“trunk não estável”), e que o processo de integração de todas as unidades deve ser diário. A rotina sugerida por ele (e por vários outros especialistas) é:

1. No início do dia, todos devem fazer um “update” em suas cópias de trabalho, pois todas foram salvas (“comitadas”) no dia anterior;
2. Ao completar seu trabalho (e antes de terminar o dia), cada desenvolvedor deve fazer um outro “update” em sua cópia de trabalho, resolvendo conflitos e testando tudo;
3. Depois de todas as divergências locais terem sido resolvidas, cada desenvolvedor faz um “commit” de suas alterações;
4. É executada a integração contínua com os testes.

Outra política é a do “trunk de integração”, na qual o desenvolvimento (cada “sprint” ou “build version”) é feito em um “branch” específico. O “trunk” só é atualizado quando os testes de integração passaram naquele “branch”. Muita gente prega que o “commit” não deve se prender a unidade de tempo, mas de trabalho.

Porém, algumas atitudes devem ser sempre tomadas, independentemente da

política de “commit” escolhida:

- **Nunca deixar “commits” incompletos:** após um “commit”, deve ser executado o teste de integração. Se houver problemas (“build quebrado”), devem ser resolvidos ANTES dos desenvolvedores irem para casa;
- **Exigir que os desenvolvedores sigam a rotina:** eles devem manter suas cópias de trabalho atualizadas e sempre fazer “commit” dentro do período esperado;
- **Após o “commit”, integrar o software:** faça um “build” (montagem) no servidor de integração e rode os testes.

Você deve estudar qual política vai funcionar melhor, lembrando que ela impacta diretamente a integração e a qualidade do produto final.

## Controle da configuração

No planejamento nós devemos estabelecer o processo de controle da configuração, de modo que todas as manutenções ocorram de forma controlada e com qualidade.

Por exemplo, podemos criar um documento de “Solicitação de Mudança”, o qual deverá conter:

- Tipo de solicitação: desenvolvimento (de nova versão), manutenção corretiva (“bug fix”) ou manutenção evolutiva;
- Quais itens serão adicionados ou alterados;
- Descrição das alterações;
- Casos de teste;
- Critérios de aprovação;
- Estimativa de tempo e recursos;
- Versão-alvo: qual é o “tag” que receberá as alterações, ou se serão feitas no “trunk” ou se vai ser criado um “branch” de desenvolvimento.

O Comitê poderá aprovar, reprovar ou reagendar a solicitação de mudança para uma ocasião mais apropriada. O Gerente de configuração estudará a versão-alvo e poderá propor mudanças, liberando os acessos necessários. Também deverá inspecionar os artefatos da SM, de modo a verificar se está faltando alguma coisa.

A versão-alvo é extremamente importante, pois é a partir dela que o Mantenedor obterá os itens necessários (“Checkout”). Se ele já tiver os itens em sua cópia de trabalho, então poderá atualizá-los (“update”).

### **Política de “commit”**

Você deverá ter uma política de controle de revisão e de “commit” (conclusão) dos itens de configuração. Esta política estabelece o uso do “trunk” (como vimos anteriormente) e os critérios para “commit”.

Quando um Mantenedor quer salvar o trabalho que está em sua cópia de trabalho, ele faz uma operação de “commit”, que é copiar os itens da sua cópia de trabalho para a base de itens de configuração do projeto. Quando ele faz isto, todos os que têm acesso à mesma versão dos itens de configuração poderão fazer um “update”, recebendo a última versão dos itens que ele alterou.

Como já vimos antes, NUNCA se deve fazer “commit” incompleto, ou seja, de itens que estão incompletos, que contêm erros de compilação ou que não passaram por testes de unidade. Mas este critério tem um problema: como o Mantenedor pode salvar o estado dos itens em sua cópia de trabalho? Muita gente prefere fazer “commit” ao final do expediente (ou antes de sair para almoçar), mesmo que o trabalho esteja incompleto. Se for um “branch” específico de desenvolvimento, não há problemas. Porém, se for o próprio “trunk” ou um “branch” compartilhado, poderá gerar o que se chama de “quebra de build”, ou seja, a versão daquele “branch” não compila (ou não roda).

***Mas minha classe depende de outras classes, que ainda não estão prontas. Eu preciso fazer “commit” para poder testar tudo junto.***

Esta é uma das razões mais comuns para “commits” incompletos. Os desenvolvedores querem testar “tudo junto”, mesmo que ainda não tenham testado (ou terminado) sua unidade de trabalho. Na verdade, é um problema de conceituação: sua unidade só pode ser integrada quando passar nos testes

unitários!

Vamos falar sobre testes mais adiante no livro, mas, adiantando, quero dizer que existem técnicas e ferramentas para permitir o teste unitário “desacoplado”, de modo que cada unidade possa ser testada e completada ANTES de ser integrada (o que acontece no momento do “commit”). Por exemplo, podemos usar “Mocks” ou “Method Stubs” para simular os comportamentos dos quais dependemos. Um comportamento deve ser representado por uma Interface (Princípio da Segregação de Interfaces), podendo ser “sintetizado” por ferramentas como o “JMock” (<http://www.jmock.org/>). Portanto, não há necessidade real de fazer “commit” incompleto só para testar.

Eu recomendo que o “commit” seja apenas de itens completos, ou seja, estão prontos, não contêm erros de compilação e passaram pelos testes unitários do desenvolvedor. No meio termo, para salvar os arquivos que estão na cópia de trabalho, é melhor fazer o checkout do projeto em uma pasta remota, em um servidor de arquivos, na conta do Mantenedor.

Para fazer isto usando o Eclipse, no momento do “checkout”, desmarque a caixa “Use default location” e indique uma pasta no servidor de rede. Com isto, a “workspace” ficará na máquina local, mas o projeto ficará em um servidor de rede, do qual, supostamente, é feito backup regular.

De qualquer forma, é sempre bom manter uma cópia em um pen drive.

## **Aprovação e integração**

Uma vez que a SM tenha sido realizada e aprovada (os critérios devem fazer parte da própria SM), o Gerente de configuração, o Gerente do projeto e o Gerente de liberação decidirão onde serão integradas as alterações da SM. Depois, o Gerente de configuração e o responsável pela SM deverão fazer o “merge” (integração) do “branch” da SM (se for o caso) e também deverão gerar o “tag” da nova versão. Então, o Gerente de liberação vai coordenar a sua distribuição.

Dependendo do software de gestão de configuração, o processo de integração “merge” pode ser simples ou complexo. Se ele dispuser de boas ferramentas de “merge”, então é apenas um trabalho burocrático.

## **Conflitos e merge**

No momento de fazer o “commit”, o Mantenedor pode se deparar com uma situação de conflito: o item de configuração na base (dentro do “branch” ou “trunk”) é mais recente do que o original que ele obteve, logo, as duas versões (a dele e a da base) precisam ser integradas. Este processo é conhecido como “merge” e consiste em analisar caso a caso as alterações e incluí-las ou não no código final de cada artefato.

As ferramentas modernas de SCM podem até fazer “merge” automático, que pode ser “2 way”, quando são considerados apenas os dois artefatos que estão sendo comparados, ou “3 way”, quando são considerados os dois artefatos e o “pai” deles, ou aquele de onde ambos foram copiados. Isto só é possível quando estamos integrando “branches” que tiveram uma única origem.

## **Liberação e entrega**

A gerência de liberação e entrega é extremamente importante no desenvolvimento de software, especialmente se for para fins comerciais. Sua principal ocupação é:

- Decidir quando serão feitos os “releases”;
- Decidir o que será incluído no próximo “release”;
- Montar o pacote do software com as características que serão incluídas no “release”;
- Distribuir (ou entregar) o pacote para os Clientes (ou para o ambiente de produção).

## **Seleção de características**

Em um grande projeto de software podemos ter várias solicitações de mudanças sendo concluídas em determinado momento. Logo, não podemos assumir que TODAS serão incluídas na próxima versão. Questões comerciais, de recursos e de cronograma podem estar envolvidas, logo, precisa haver uma pessoa que seja o “guarda de trânsito” da liberação de versões. Esta pessoa é o Gerente de liberação e entrega.

Não cabe ao Gerente de liberação determinar se as SMs serão ou não incluídas no software, pois isto é da alçada do Comitê de Controle de Mudanças, mas ele pode decidir quando serão incluídas. Normalmente, há um cronograma de previsões de versões, com datas limite para aprovação de SMs. Ele precisa saber quais SMs precisam entrar juntas e quais são independentes, também quais estão prontas ou não. Além disto, fatores comerciais podem influenciar a entrada ou não de características em uma nova versão.

O Gerente de liberação e o Gerente de configuração criam um “tag” para a versão que entrará no ar. Isto gera uma “baseline” de versão.

### **Montagem e distribuição**

Uma vez decidido o que entrará em uma versão, o Gerente de liberação determina o número da versão / release, coordena a montagem (“build”) e o teste final, antes da liberação (são geradas versões do tipo: “Release candidate” - candidata à liberação).

Assim que aprovada no teste final, são gerados os pacotes da versão (pode haver mais de um: PC, Mac, Windows, Linux, Android, iOS etc.), o Gerente de liberação coordenará a sua distribuição.

É muito importante ressaltar um detalhe: quem faz a montagem (“build”) final é o Gerente de liberação (ou alguém designado por ele), e não a equipe de desenvolvimento. Isto garante que o software inclua apenas os componentes aprovados, sejam eles itens de configuração ou componentes externos. Usar “build” de desenvolvedor para gerar versão final é prova de desorganização, além de expor toda a equipe a riscos desnecessários.

O processo de “build” deve ser automatizado e sem necessidade de ambiente de desenvolvimento. Além disto, devem ser utilizados apenas os itens de configuração pertencentes ao “tag” determinado para a versão (a “baseline” da versão). Os componentes prontos, aqueles que não foram desenvolvidos, devem vir do repositório de componentes da empresa, não podendo ser os que o Mantenedor tem em seu ambiente de trabalho.

### **Contabilização e estado**

O estado da base de configuração do projeto deve ser analisado periodicamente,

sendo relatado ao Comitê de Controle de Mudanças, ao Gerente do projeto e a todos os interessados em geral. Devem ser capturadas informações sobre os diversos estados dos itens (“tags” e “branches”), sobre as versões liberadas e sobre os desenvolvimentos atuais. Igualmente importante é saber quais SMs estão aguardando liberação, quais estão em processo de avaliação e quais estão em desenvolvimento.

Isto pode ser feito pelo Gerente de configuração, com apoio de ferramentas automatizadas. Com base nestas informações, ele pode fazer cobranças (SMs paradas, fora do prazo, versões com “build” quebrado) e informar ao Gerente do projeto.

## **Auditoria**

Periodicamente, é necessário fazer auditorias na base de configuração do projeto, com o objetivo de garantir que o processo de SCM esteja sendo seguido. Estas auditorias, geralmente, são feitas por pessoas de fora do projeto, normalmente ligadas ao processo de Garantia de Qualidade de Software (SQA).

Elas podem auditar as especificações e os artefatos para saber se estão de acordo, podem verificar se o que está documentado é compatível com o que está disponível (versões liberadas) e podem auditar “baselines” específicas para determinar se a documentação está sendo atualizada e se há algum descontrole sobre os itens de configuração.

## **Como implementar o SCM**

O processo de Gestão de Configuração de Software é muito complexo e envolve diversos papéis e atividades diferentes, mas aumenta a qualidade do software através do controle de sua construção e liberação. Diversos problemas podem ser evitados se tivermos um SCM eficaz em prática, entre eles:

- **Falha de “build”:** quando uma versão foi distribuída e contém uma falha catastrófica, normalmente ocasionada pelo uso de um componente diferente do que foi aprovado. Isto acontece quando o “build” é feito na máquina do desenvolvedor, que pode utilizar componentes (ou arquivos de configuração do software) diferentes dos que foram aprovados;
- **Falha de versão:** quando determinada funcionalidade esperada em uma

versão distribuída apresenta falha. Normalmente, é ocasionado pelo uso de versão incorreta de item de configuração;

- **Mudanças conflitantes:** quando os desenvolvedores implementam mudanças que conflitam entre si e são incluídas em uma versão distribuída.

Nem sempre é prático implementar um SCM em equipes pequenas. Porém, existe um conjunto mínimo de atitudes que podem ser tomadas, de modo a garantir pelo menos uma parte dos benefícios de um SCM:

1. **Utilize um software de controle de versões** (ou de revisões), como o Subversion, CVS, Clear Case ou outros. Tem uma página na Wikipédia que lista vários deles: [http://en.wikipedia.org/wiki/List\\_of\\_revision\\_control\\_software](http://en.wikipedia.org/wiki/List_of_revision_control_software). Prefira sistemas Cliente-Servidor, somente utilizando sistemas distribuídos se for necessário (equipes remotas);
2. **Designue um Gerente de configuração**, de preferência que não seja o Gerente de projeto ou o arquiteto, para tomar conta da base de configuração. Ele poderá atuar também como Gerente de liberação. Esta pessoa tem que ter autoridade para dar a última palavra nestes assuntos;
3. **Escreva um processo mínimo de controle de configuração** que determine o uso do “trunk” e uma política de “commit”. Eu recomendo o “trunk” de integração e a política de só fazer “commits” completos. Crie “tags” para as versões distribuídas, mantendo-as de acordo com o que diz em seu manual de usuário – por exemplo, se você dá manutenção em até três versões passadas, então tem que manter três “tags”: “stable”, prévia e anterior. Manutenções corretivas devem ser feitas utilizando-se “branches”;
4. **Guarde a cópia de trabalho dos programadores na rede** dentro de um servidor de arquivos, protegida pela conta de cada um deles e com backup rotineiro. Já mostramos como fazer isto no Eclipse: é só alterar o local de armazenamento do projeto no momento do “Checkout”. Não faça isso com as “workspaces”, pois elas devem sempre ficar nas máquinas locais;
5. **Só o Gerente de configuração faz a montagem final** (“build”) e a distribuição, pois assim a responsabilidade ficará concentrada em uma só pessoa, evitando “builds” feitos pelos desenvolvedores. E este “build” deve ser feito recuperando-se o “tag” da versão diretamente da base de



configuração. Não devem ser utilizados artefatos de cópias locais para isto, mesmo que sejam do Gerente de configuração. Existem ferramentas para automatizar o processo, como: Maven e “ant”;

6. **Toda integração (“merge”) deve ser feita pelo Gerente de configuração**, acompanhado pelo Gerente de projeto e pelo desenvolvedor responsável.

## **Escolha de um sistema de controle de versão**

Em nosso projeto imaginário, vamos utilizar um sistema de controle de revisões. Devemos utilizar um software open source. Como existem muitos candidatos, procurei uma pesquisa sobre o assunto e encontrei uma ótima fonte: a pesquisa da comunidade Eclipse de 2010, com 1.696 respostas completas. Ela pode ser acessada em: [http://www.eclipse.org/org/press-release/20100604\\_survey2010.php](http://www.eclipse.org/org/press-release/20100604_survey2010.php). Eis um resumo dos resultados para software de controle de versões:

- **Subversion (SVN):** 58,3%;
- **Concurrent Versions System (CVS):** 12,6%;
- **Git / GitHub:** 6,8%;
- **Nenhum:** 5,6%;
- **Mercurial:** 3,0%.

Note que uma parte dos desenvolvedores (5,6%) não usa um sistema de controle de versões.

Não satisfeito, procurei outra pesquisa para basear minhas conclusões e, por incrível que pareça, achei uma feita por ninguém menos do que Martin Fowler (<http://martinfowler.com/bliki/VcsSurvey.html>). Ele conduziu uma pesquisa via Google Docs, entre fevereiro e março de 2010, obtendo 99 respostas. Eis os três primeiros colocados:

1. **Git;**
2. **Mercurial;**
3. **Subversion.**

Já temos uma boa indicação para usar o SVN, mas vamos comparar algumas características dos três primeiros colocados, de modo a saber se mudamos esta opinião ou não.

### **Subversion:**

É um sistema cliente-servidor e open source, sendo o sucessor do CVS e o mais utilizado atualmente. Originalmente criado pela CollabNet ([www.collab.net](http://www.collab.net)), hoje é parte do grupo Apache.

Pontos positivos:

- Faz tudo o que o CVS faz;
- Suporta “commits” atômicos, sem deixar inconsistências devido a problemas (servidor, rede etc.);
- Pode rodar usando o servidor Apache;
- Pode utilizar também WebDav;
- Permite “file-lock” ou “Checkout” reservado;
- Retém o histórico de arquivos removidos ou renomeados.

Pontos negativos:

- Guarda arquivos e pastas adicionais na cópia de trabalho (.svn) que podem ser corrompidos pelo usuário;
- Não armazena a data/hora de modificação na base de configuração, utiliza sempre a data do último “Check-in”.

### **CVS:**

É um sistema cliente-servidor, ainda muito utilizado, sendo um software simples, robusto e maduro e uma boa opção para controle de versões.

Pontos positivos:

- Controla múltiplas versões;

- Impede o “Check-in” de arquivos desatualizados;
- Suporta a criação de “branches”.

Pontos negativos:

- Não versiona arquivos e diretórios renomeados ou removidos;
- Não suporta “commits” atômicos, ou seja, em caso de problemas com o servidor ou a rede durante um “commit”, a base pode ficar inconsistente.

### **Git:**

É um sistema distribuído (não é cliente-servidor) e colaborativo, muito popular atualmente. Foi criado por Linus Torvalds e é utilizado para controle do Linux Kernel.

Pontos positivos:

- É descentralizado, logo, a cópia inteira do repositório pode residir em mais de uma máquina;
- Funciona bem com grandes projetos;
- Histórico com autenticidade garantida por criptografia;
- Estratégias de integração (“merge”) “plugáveis”.

Pontos negativos:

- O sincronismo entre repositórios Git dos desenvolvedores não é muito simples;
- Depende muito da experiência e prática dos desenvolvedores envolvidos.

### **Mercurial:**

Eu não tenho experiência com o Mercurial, ao contrário do CVS, SVN e Git, mas é um sistema distribuído que vem se tornando mais popular a cada dia.

Pontos positivos:

- Alto desempenho;
- Ferramentas de “merge” e “branch” avançadas.

Pontos negativos:

- O sincronismo entre repositórios dos desenvolvedores não é muito simples;
- Depende muito da experiência e prática dos desenvolvedores envolvidos.

Realmente, eu não tenho muita experiência com o Mercurial, mas, como ele é um sistema distribuído, eu tenho a tendência de “torcer o nariz”. Sistemas de controle de versão distribuídos são para desenvolvimentos Open Source, nos quais existe uma comunidade enorme e espalhada de desenvolvedores, cuja boa vontade e motivação são altas. Tenho minhas dúvidas quanto ao seu uso em sistemas comerciais.

Além disto, me lembra muito o antigo “Source Safe”, da Microsoft.

Minha experiência com o Git foi péssima, mas não foi culpa dele. Eu acredito que sistemas de controle de versão do tipo cliente-servidor são mais indicados para projetos de software comercial, nos quais existem vários desenvolvedores e múltiplas versões. Imagine-se em uma conferência via Skype tentando descobrir quem está com qual versão do código-fonte... Ou tente fazer um “merge” com outros três desenvolvedores (não existe repositório central).

Logo, é importante que o software de controle de versões seja cliente-servidor. Outra característica importante é que ele suporte “commits” atômicos; logo, o SVN parece ser uma boa escolha.

## **Instalação do Subversion no Linux e no Windows**

Como eu disse anteriormente, minha preferência é utilizar Linux nas estações de desenvolvimento. Porém, nada impede que você utilize Microsoft Windows, tanto nas estações como nos servidores. Vamos mostrar como instalar o Subversion (cliente e servidor) em ambas as plataformas.

### **Subversion**

O Subversion administra um repositório que contém os itens de configuração do seu projeto. Podemos criar um único repositório, com uma pasta por projeto, ou podemos criar repositórios separados. Para facilitar a configuração, vamos criar um único repositório.

Ele pode utilizar dois mecanismos para guardar os dados: um banco de dados Berkeley DB (BDB) ou arquivos no File System (FSFS). Embora pareça interessante, utilizar BDB não é uma escolha muito popular. Vamos utilizar o que todo mundo usa: FSFS.

O repositório será uma pasta, simples assim. Cada usuário terá acesso ao repositório somente via rede e utilizando o cliente SVN. É muito importante que ninguém (a não ser o Gerente de configuração) tenha permissão direta de acesso ao repositório, seja via sessão interativa (“shell”) ou via rede (“Samba”, “NFS” etc.). Eu não vou entrar em detalhes de configuração de segurança, mas recomendo que você crie um grupo de administração do SVN e adicione apenas os usuários específicos, mudando a propriedade da pasta.

Ao fazer um “Checkout”, o desenvolvedor obtém uma “Cópia de trabalho”, que ficará armazenada em sua máquina local, a não ser que ele queira guardar a cópia de trabalho em uma pasta na rede, hospedada em um servidor com backup regular. Não vamos entrar em detalhes dos comandos SVN porque vamos utilizar Eclipse, mas se quiser saber os principais comandos:

<http://svnbook.red-bean.com/en/1.4/index.html>.

Para que os desenvolvedores possam utilizar o Subversion, três elementos são necessários:

- Um repositório, localizado em um servidor na rede;
- Uma porta TCP aberta neste servidor, onde um programa está escutando os pedidos (Daemon);
- Um programa cliente para enviarem comandos ao Daemon do SVN.

Existem duas opções de hospedar o servidor SVN: uma delas é rodar o “SVN Daemon”, também conhecido como “svnserve”, e a outra é utilizar um Servidor Apache. É claro que esta última é uma excelente opção, afinal o Apache é um

servidor bem testado e com várias funcionalidades. Porém, não é só isto: ele permitirá acesso utilizando HTTP e HTTPS, além de WebDAV!

Mas a instalação no Apache é mais complexa e pode não ser necessária para projetos pequenos e médios, portanto, vamos usar o “svnserve” mesmo. Em Linux ele é um programa “daemon” que fica escutando a porta TCP 3690 (atenção, administradores de rede!). Em Windows é a mesma coisa, só que ele pode ser instalado como um serviço.

Para instalar e testar, crie uma pasta com a seguinte estrutura:

```
/testesvn  
  /branches  
  /tags  
  /trunk
```

E coloque alguns arquivos dentro da subpasta “trunk”. Se fosse para “valer”, nós colocaríamos a pasta do projeto dentro da pasta “trunk”, pois é isso que ele vai trazer quando fizermos “checkout”.

## **Instalação no Linux (Ubuntu)**

Toda distribuição Linux tem um pacote pronto com o Subversion (se é que ele já não está instalado!). No Ubuntu, use o comando:

```
sudo apt-get install subversion
```

Este comando instalará o cliente e o servidor SVN. Depois, crie uma pasta para servir como repositório. No meu caso será: “/home/cleuton/projetos/svnrepo”. Se quiser, mude a propriedade e a segurança da pasta. Então, transforme esta pasta em um repositório com o comando “svnadmin create”:

```
svnadmin create ~/projetos/svnrepo
```

Se listar o conteúdo desta pasta, verá a estrutura normal de um repositório SVN:

```
$ ls svnrepo  
conf db format hooks locks README.txt
```

No momento, apenas a pasta “conf” nos interessa, pois ela contém configurações importantes do servidor SVN. Agora, importe o projeto de teste lá para dentro (você deve ter criado uma pasta para isto: “testesvn”). Fazemos isto com o

comando “svn import”:

```
$ svn import testesvn file:///home/cleuton/projetos/svnrepo/testesvn -m  
“initial import”
```

```
Adicionando testesvn/trunk  
Adicionando testesvn/trunk/virtualbase.cpp  
Adicionando testesvn/trunk/virtualbase2.cpp  
Adicionando testesvn/trunk/naovirtual.cpp  
Adicionando testesvn/trunk/virtual.cpp  
Adicionando testesvn/trunk/naovirtual2.cpp  
Adicionando testesvn/trunk/estatico.cpp  
Adicionando testesvn/trunk/purovirtual.cpp  
Adicionando testesvn/branches  
Adicionando testesvn/tags
```

Vamos testar para saber se está tudo ok. Podemos listar o projeto com o comando “svn list”:

```
svn list file:///home/cleuton/projetos/svnrepo/testesvn  
branches/  
tags/  
trunk/
```

A URL do repositório é importante! Neste caso, estamos acessando diretamente, mas os desenvolvedores não deverão fazer isto. Temos mais uma coisa a fazer: configurar a segurança de acesso. O SVN permite usar um esquema simples, baseado no arquivo “<repositório>/conf/passwd”, que serve para a maioria dos projetos. Para começar, temos que alterar as configurações de acesso. Edite o arquivo “svnserve.conf”, que fica dentro da pasta “conf” no seu repositório. Faça o seguinte:

- Descomente as linhas que começam com: “password-db”, “anon-access” e “auth-access”. É só retirar o caractere “#”, que fica no início destas linhas. Como praxe, eu costumo retirar quaisquer espaços do início também;
- Altere o valor de “anon-access” para “none”, pois não queremos permitir acesso anônimo nem de leitura!

As linhas deverão ficar assim:

```
password-db = passwd  
anon-access = none  
auth-access = write
```

Os usuários autenticados deverão poder ler e gravar, e os anônimos sequer poderão acessar. Depois disto, devemos criar os usuários que terão acesso ao nosso repositório, dentro do arquivo “passwd”, que fica dentro da pasta “conf”, no repositório. A sintaxe é “nome = senha”.

Agora, vamos iniciar o “SVN Daemon” com o comando “svnserve”:

```
svnserve -d -r /home/cleuton/projetos/svnrepo
```

E vamos testar. Lembre-se de que, como eliminou acesso anônimo, ele vai pedir a sua senha:

```
$ svn list svn://localhost/testesvn --username cleuton
```

```
branches/  
tags/  
trunk/
```

Agora, vamos testar um “Checkout”. Primeiramente, vá para uma pasta vazia, depois dê o comando:

```
$ svn checkout svn://localhost/testesvn --username cleuton
```

```
A testesvn/trunk  
A testesvn/trunk/virtualbase.cpp  
A testesvn/trunk/virtualbase2.cpp  
A testesvn/trunk/naovirtual.cpp  
A testesvn/trunk/naovirtual2.cpp  
A testesvn/trunk/virtual.cpp  
A testesvn/trunk/estatico.cpp  
A testesvn/trunk/purovirtual.cpp  
A testesvn/branches  
A testesvn/tags
```

Gerado cópia de trabalho para revisão 1.

Agora você tem uma pasta com a cópia de trabalho do projeto. Dentro dela, haverá algumas pastas ocultas “.svn”, que servem para controle de alterações. Não mexa nestas pastas e nem em seus arquivos.

Até agora tudo bem, mas note que apenas mandamos executar o “svnserve”. O que acontecerá quando o servidor for desligado? Será que quando for religado o “svnserve” estará no ar? É claro que não. Para isto, temos que configurar um “script” de inicialização, dentro de “/etc/init.d”. Eu procurei na internet e achei



um “script” pronto:

```
#!/bin/sh
#
# start/stop subversion daemon.
EXECUTABLE=/usr/bin/svnserve

# Test existence of the executable
test -f $EXECUTABLE || exit 0

# Command line options for starting the service
OPTIONS="-d -r /home/cleuton/projetos/svnrepo"

case $1 in
start)
echo -n "Starting subversion daemon: $EXECUTABLE $OPTIONS\n"
start-stop-daemon -vo -x $EXECUTABLE -S -- $OPTIONS
echo -n "."
;;

stop)
echo -n "Stopping subversion daemon: $EXECUTABLE $OPTIONS\n"
start-stop-daemon -K -qo -x $EXECUTABLE
echo -n "."
;;

force-reload)
$0 restart
;;

restart)
$0 stop
$0 start
;;

*)
echo 'Usage: /etc/init.d/subversion {start|stop|restart}'
exit 1
;;
esac

exit 0
```

Você precisa alterar o caminho do seu repositório (em **negrito**). Depois, faça o seguinte:

1. Copie este “script” para dentro de “/etc/init.d” (use “sudo”), criando com o nome “svnserve” (sem extensão);

2. Dê permissão de execução com o comando: “sudo chmod +x svnserve”;
3. Use o comando para alterar os “scripts” dos vários “run levels”: “sudo update-rc.d svnserve defaults”.

Agora o “svnserve” será iniciado junto com o sistema.

## Instalação no Windows

O processo é basicamente o mesmo. Para começar, vamos obter o pacote binário para instalação. Dentro do site <http://subversion.apache.org/packages.html> há vários pacotes para Windows. Uns com cliente e servidor, outros só com cliente, e ainda outros gráficos. Eu recomendo o “win32svn”, que pode ser baixado aqui: <http://sourceforge.net/projects/win32svn/>. Depois, rode o instalador.

Crie uma pasta para servir como repositório (eu usei: “c:\svnrepo”) e use o comando:

```
svnadmin create c:\svnrepo
```

Importe o projeto de teste:

```
C:\>svn import c:\projetos\testesvn file:///c:/svnrepo/testesvn -m “initial  
imp ort”  
Adicionando  projetos\testesvn\trunk  
Adicionando  projetos\testesvn\trunk\virtualbase.cpp  
Pulou ‘projetos\testesvn\trunk\.svn’  
Adicionando  projetos\testesvn\trunk\virtualbase2.cpp  
Adicionando  projetos\testesvn\trunk\naovirtual.cpp  
Adicionando  projetos\testesvn\trunk\naovirtual2.cpp  
Adicionando  projetos\testesvn\trunk\virtual.cpp  
Adicionando  projetos\testesvn\trunk\estatico.cpp  
Adicionando  projetos\testesvn\trunk\purovirtual.cpp  
Pulou ‘projetos\testesvn\.svn’  
Adicionando  projetos\testesvn\branches  
Pulou ‘projetos\testesvn\branches\.svn’  
Adicionando  projetos\testesvn\tags  
Pulou ‘projetos\testesvn\tags\.svn’  
  
Committed revision 1.
```

Agora, teste se está tudo ok:

```
C:\>svn list file:///c:/svnrepo/testesvn  
branches/
```

tags/  
trunk/

Note que estamos utilizando uma URL diferente para o repositório. Todos os comandos SVN (exceto os de administração – svnadmin) utilizam URL em vez de path. Em Windows, temos que utilizar a sintaxe:

```
file:///<drive>:/<path>
```

Porém, temos o mesmo problema do Linux, ou seja, queremos que o SVN sempre esteja no ar, mesmo que o servidor tenha sido reiniciado. Podemos adicionar o “svnserve” como um serviço do Windows. Isto pode ser feito via “Ferramentas Administrativas”, ou com o comando “sc”:

```
C:\>sc create svnserve binpath= "C:\Arquivos de programas\Subversion\bin\
svnserv e.exe --service -r C:\svnrepo" displayname= "Subversion Server" de-
pend= Tcpip st art= auto
```

```
[SC] CreateService SUCCESS
```

A sintaxe é assim:

```
sc create <nome do serviço> binpath= "<caminho do executável> --service"
displayname= "<nome descritivo>" depend= "<nome do serviço do qual ele
depende>"
start= "<tipo de inicialização: auto = inicia automaticamente>"
```

Só que este comando é muito chato, pois não aceita continuação em outra linha (terminar cada linha com “\” para poder quebrar linhas). Outras características são:

- O “binpath” é o comando para execução do “svnserve”, incluindo os argumentos (opções e path do repositório);
- Dentro do “binpath”, após o path do executável, acrescente a opção “--service” (dois hífen e a palavra “service”);
- Cada opção do comando “sc” (como: “binpath”, “displayname” etc.), deve ser seguida de um sinal “=” e um (somente um) espaço.

Depois de acrescentarmos o serviço, podemos iniciá-lo de três formas: reiniciando a máquina, via painel de controle ou então com o comando:

```
net start svnserve
```

Agora, é só fazer backups regulares do seu repositório e pronto.

## **Organizando o repositório**

Você pode criar um repositório por projeto, mas, neste caso, precisará de um servidor para cada um deles. Outra opção é criar um repositório único, com pastas para projetos. O SVN permite criar permissões de acesso por pasta.

Para começar, voltemos ao arquivo “svnserve.conf”, que fica dentro da pasta “conf” do repositório. Descomente a linha abaixo (retire o caractere “#” do início da linha:

```
authz-db = authz
```

Agora, edite o arquivo “authz” dentro do mesmo diretório. Ele serve para dar níveis de autorização para cada usuário identificado dentro do “passwd”. Funciona de modo hierárquico de acordo com as pastas do repositório:

```
[groups]
desenv_projeto1 = cleuton,fulano,beltrano
qualidade = jose,pedro

[/projeto1]
desenv_projeto1 = rw
qualidade = r
```

A configuração é bem simples: criamos um grupo “desenv\_projeto1” e colocamos três usuários dentro dele. Também criamos um segundo grupo, chamado “qualidade”, com dois usuários. Depois configuramos o acesso ao caminho “/projeto1” dando acesso de leitura e escrita (incluindo “Check-in”) para o grupo de desenvolvedores, mas acesso somente de leitura para o grupo de qualidade.

Para maiores detalhes, consulte o manual do SVN: <http://svnbook.red-bean.com/en/1.4/svn.serverconfig.pathbasedauthz.html>.

## **Configuração do Eclipse para usar o SVN**

O Eclipse precisa de um “plug in” para poder acessar projetos armazenados em

repositórios Subversion. Há duas opções interessantes:

- SubEclipse (update site: [http://subclipse.tigris.org/update\\_1.8.x](http://subclipse.tigris.org/update_1.8.x));
- SubVersive (update site: <http://download.eclipse.org/releases/indigo>), selecione “Collaboration tools”.

Eu sempre usei o SubEclipse sem problemas, mas, para este livro, preferi instalar o “SubVersive”, que já vem no update site da própria distribuição Eclipse.

Antes de mais nada, crie um projeto Java simples, só para testar, dentro de uma workspace sua.

Vamos supor que você vá instalar o SubEclipse. Então, abra o menu “Help / Install new Software”, depois clique no botão “Add” para adicionar o update site ([http://subclipse.tigris.org/update\\_1.8.x](http://subclipse.tigris.org/update_1.8.x)). Após adicionar o “plug-in”, você vai ganhar algumas novas “Views”, entre elas a “SVN Repositories” (Menu “Window / Show View / Other / SVN”).

Bem, o seu SVN Server está rodando?

- Linux:

```
$ ps -e | grep svn  
5584 ? 00:00:00 svnserve
```

- Windows:

```
> tasklist /scv
```

(procure por “svnserve.exe”)

Então vamos adicionar o seu repositório na sua workspace:

1. Abra a view “SVN Repositories” (“Window / Show View / Other / SVN” – expanda “SVN”);
2. Clique dentro da view “SVN Repositories” com o botão direito e selecione “New / Repository location”;

3. Digite a URL do seu servidor SVN: “svn://<nome ou ip>”. Clique “OK”;
4. Depois, informe seu código de usuário e senha (conforme o arquivo “passwd”).

Você verá algo como a imagem a seguir.



Ilustração 4: Nosso repositório SVN

Note que temos o nosso projeto anterior aparecendo. Isto é porque eu dei acesso ao meu usuário dentro do arquivo de autorização (veja capítulo anterior).

Antes de colocarmos nosso projeto lá, vamos criar a estrutura de pastas para ele. Lembre-se que queremos algo assim:

```
/projeto
  /trunk
    /projeto trunk
  /branches
  /tags
```

A boa notícia é que você não precisa mais utilizar o cliente em modo terminal. Simplesmente selecione o repositório e, com o botão direito do mouse, escolha “new” e “Remote folder”, criando a estrutura como a da ilustração seguinte.

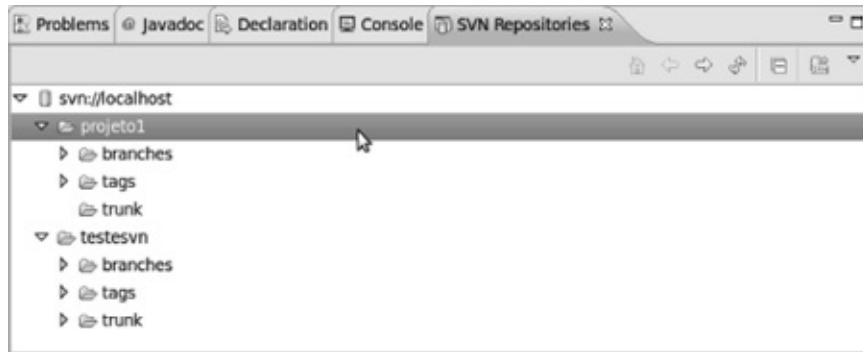


Ilustração 5: Estrutura padrão

A pasta principal do nosso projeto se chamará “projeto1” e ela deverá estar mencionada dentro do arquivo de autorizações; caso contrário, ninguém terá acesso a ela. Dentro da pasta “trunk” vamos criar a primeira “baseline” do projeto. Para isto, selecione o seu projeto (dentro do “Package explorer”) e, com o botão direito do mouse, escolha: “Team / Share project...”.

Depois, selecione “SVN” e qual é o repositório onde deseja colocar o projeto. Não use “Use project name as folder name”, pois já criamos uma estrutura de pastas e queremos colocá-lo dentro de “/projeto1/trunk”. Marque “Use specified folder name” e clique no botão “Select”, selecionando a subpasta “trunk”. Seu caminho deve ser algo assim:

```
svn://localhost/projeto1/trunk/testesvnplugin
```

O nome do meu projeto de teste é “testesvnplugin”. Clique em “Finish”. Ele vai mostrar o comentário inicial e depois vai pedir para mostrar a perspectiva “Team synchronizing”.

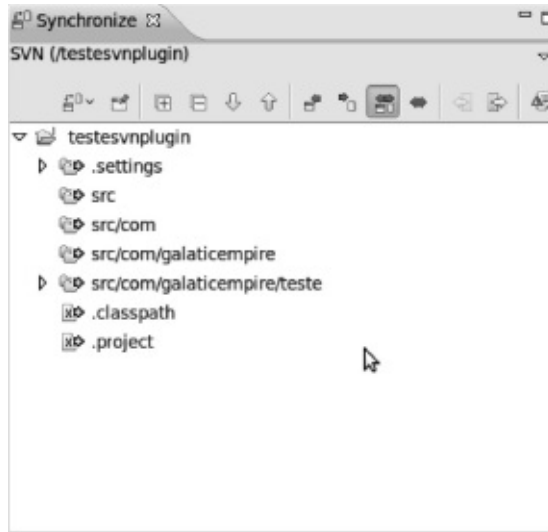


Ilustração 6: Team synchronizing

Ao expandir o projeto na janela “Synchronize”, veremos quais artefatos precisam ser sincronizados com o repositório. Neste caso, todos. Há setas em direção à direita em todos eles, e todas estão só com o contorno, indicando que são novos itens. Para efetuar o “Commit”, basta clicar sobre o projeto com o botão direito do mouse e selecionar “Commit”.

Pronto! Seu projeto está pronto para ser utilizado. Os outros desenvolvedores podem fazer “Checkout”:

1. Cadastrar o repositório dentro da view “SVN Repositories”;
2. Selecionar a “baseline” desejada (svn://localhost/projeto1/trunk/<nome do projeto>);
3. Marque “Checkout as a project in the Workspace”.

### **Como manter o controle**

De tempos em tempos é bom que o desenvolvedor faça updates em sua cópia de trabalho. Para isto, pode tentar sincronizar com o repositório (“Team / Synchronize with repository”). Na view “Team Synchronizing” ele pode ver o que foi atualizado e se existe algum conflito. Conflitos são resolvidos através de “Merge”.

Se você escolheu a estratégia “Trunk” não estável, então os desenvolvedores vão



fazendo “Commits” diretamente nele. Neste caso, é melhor fazer uma sincronização com o repositório no início de cada dia de trabalho, fazendo “Commit” ao final. Se você selecionou “Trunk de integração” ou “Trunk estável”, então o “Trunk” conterá um projeto estável, logo, você não pode fazer “Commit” a todo momento.

Não se deve confundir “Repositório” com “Backup”, ou seja, não podemos usar o repositório apenas para preservar o trabalho individual dos programadores. Isto pode ser feito com um “Checkout” do projeto para uma pasta de rede, que deve ser sujeita a backup.

### **Criação de branches e tags**

Branches são versões de desenvolvimento diferentes. Podem ser duas versões diferentes do software (por exemplo: Linux e Windows), ou podem ser características que ainda serão adicionadas. Normalmente, um “branch” é reintegrado ao “Trunk” através da opção de “merge” (“Team / merge”).

Tags são “baselines” completas do projeto, ou seja, são imagens funcionais do projeto. A principal diferença para um “branch” é que um “tag”, a princípio, contém código completo e pronto.

No Subversion, “branches” e “tags” são pastas que contêm cópias do software. Podemos até mesmo dar permissões diferenciadas para elas, por exemplo, um grupo de desenvolvedores tem acesso a um determinado “branch”, mas não a outro.

Para criar um “branch” ou “tag” selecione o projeto e o menu de contexto: “team / branch/tag”. Selecione a pasta destino “/branches/<nome do branch>” ou “/tags/<nome do tag>”. Você pode criar um “branch / tag” a partir da versão mais nova (“HEAD”) do “trunk”, ou a partir de uma revisão específica (número de versão), ou mesmo a partir da sua cópia de trabalho.

Ele vai criar uma pasta com o nome que você escolheu e vai colocar os arquivos lá. Você pode mudar sua cópia de trabalho para o novo “branch/tag” com o menu “Team / switch to another branch, tag, revision”.

### **Merge**

O “merge” pode ser de um artefato ou de um branch inteiro. Sempre que fazemos uma sincronização e há conflitos, também temos que fazer “merge”. O menu “team / merge” nos dá várias opções, dependendo do que queremos fazer. Se quisermos reintegrar um “branch”:

1. Vá para o “trunk”: “Team / Switch to another branch...” e selecione “trunk”;
2. Inicie o “merge”: “Team / merge” e selecione “Reintegrate a branch”;
3. Analise os conflitos que tenham acontecido.

Sempre que ocorrer um conflito, você verá os artefatos marcados na view “Team Synchronizing”.

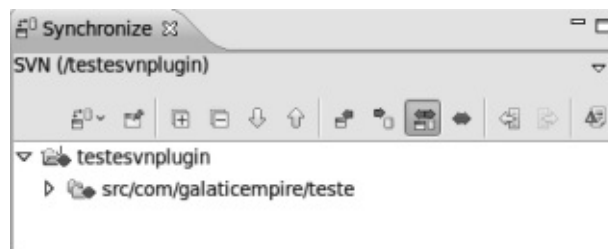


Ilustração 7: Conflito

Se dermos um duplo clique no arquivo conflitante, veremos o “SVN Compare editor”, que compara a versão local (da cópia de trabalho) com a versão do repositório.

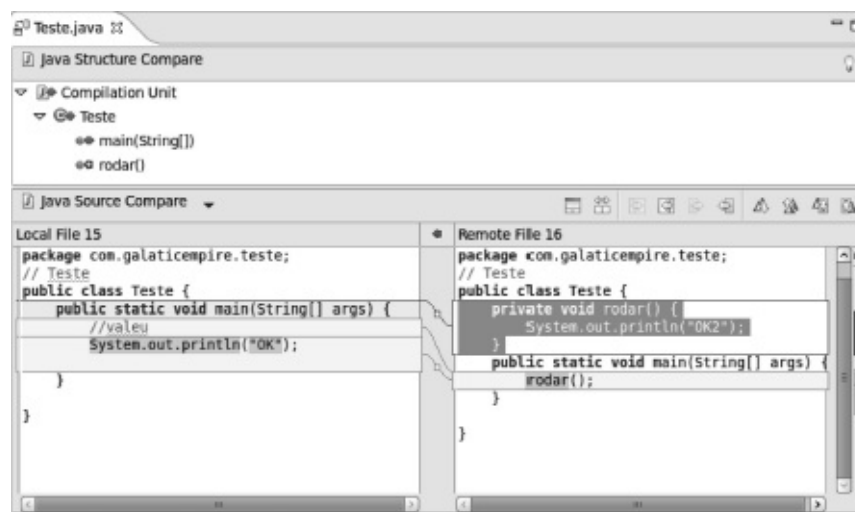


Ilustração 8: SVN Compare editor

Podemos escolher quais alterações serão aceitas definitivamente no arquivo utilizando os botões da barra “Java source compare”.

No final do “merge” a sua cópia local deverá estar atualizada.

## Gestão de montagem (build) e de componentes

Eu estava em um evento para desenvolvedores quando ouvi esta pérola:

*Escrever código é a parte mais fácil e divertida da programação, só que não basta para criar uma aplicação.*

(Autor desconhecido).

Realmente, em nosso mundo moderno, o desenvolvimento de aplicações tornou-se extremamente complexo e caro. Hoje em dia, uma aplicação é feita com diversos componentes, sejam criados por terceiros ou componentes próprios. E a tendência é cada vez mais reutilizar software, aumentando mais ainda a “colcha de retalhos”. O mais importante é saber como integrar todos estes componentes para produzir o resultado que esperamos.

O que quero dizer com isto é que um projeto típico envolve muitos componentes, e que todos representam possíveis pontos de falha. Quer um exemplo? O modesto projeto Web que vamos utilizar para configurar nosso ambiente tem pelo menos 26 bibliotecas:

- antlr-2.7.6.jar
- commons-beanutils-1.7.0.jar
- commons-codec-1.3.jar
- commons-collections-3.1.jar
- commons-collections-3.2.jar
- commons-digester-1.8.jar
- commons-discovery-0.4.jar
- commons-fileupload-1.2.1.jar

- commons-io-1.3.2.jar
- commons-logging-1.1.1.jar
- derbyclient.jar
- dom4j-1.6.1.jar
- freemarker-2.3.15.jar
- hibernate3.jar
- javassist-3.9.0.GA.jar
- jsf-facelets.jar
- jta-1.1.jar
- log4j.xml
- log4j-1.2.16.jar
- myfaces-api-1.2.9.jar
- myfaces-impl-1.2.9.jar
- ognl-2.7.3.jar
- slf4j-api-1.5.8.jar
- slf4j-log4j12-1.6.1.jar
- struts2-core-2.1.8.1.jar
- xwork-core-2.1.6.jar

Sem contar as bibliotecas Java EE, que são fornecidas pelo container. Porém, é comum existirem mais algumas bibliotecas, sejam como projetos associados ou componentes pré-compilados. O que todos estes “retalhos” têm em comum? Versão! Cada componente e biblioteca tem um número de versão e revisão, podendo também dependerem uns dos outros. Se você substituir um componente por uma versão mais moderna, pode ser que o seu projeto “quebre o build”, ou pior: apresente erro de lógica.

Certamente você já ouviu aquela famosa frase: “na máquina do desenvolvedor tudo funciona!” E deve ter sido em uma demonstração para a gerência (ou para o Cliente). Veja, escrever o software é só uma parte do trabalho, já que é necessário instalá-lo em seu ambiente definitivo (fazer o “deploy”). Normalmente os desenvolvedores já possuem todas as “porcarias” e “rebimbocas” necessárias para que o software rode “macio” em suas estações. O problema é quando tentamos promovê-lo para outro ambiente.

## **Roteiros de montagem**

Alguns desenvolvedores procuram suavizar este problema através de roteiros de montagem, também conhecidos como “build scripts”. Desde o antigo “makefile” até o Apache Ant, vários softwares tentaram resolver o problema de compilar e montar um projeto para distribuição.

Eu trabalhei em um projeto grande cujo roteiro “build.xml” tinha mais de quinhentas linhas! Imagine só! Ele fazia tudo, mas tudo mesmo. Era tão completo que precisava de especialistas para sua manutenção. É o cúmulo: um projeto dentro de um projeto!

Projetos não podem depender do ambiente do desenvolvedor. De acordo com a filosofia SCM (discutida anteriormente), a montagem final deve ser feita na gerência de liberação, e não na máquina do desenvolvedor. Neste caso, roteiros de montagem devem ser simples e devem buscar os artefatos da versão específica, diretamente na base de configuração do projeto.

## **Onde guardar e de onde pegar componentes**

O problema pode piorar! Vamos a mais uma historinha... certa vez, em uma empresa onde trabalhei, havia uma biblioteca de componentes que era compartilhada por vários projetos. Legal, não? Só que ninguém sabia onde estava a versão “oficial”! Alguns projetos colocavam uma cópia do JAR em seu repositório (CVS, Clear Case etc), outros mantinham cópias locais, em CDs e até em “pen drives”. E pior: havia versões diferentes de interface e alguns projetos eram extremamente dependentes disto.

Repositórios de controle de versão não devem ser utilizados para armazenar bibliotecas compiladas. Na verdade, deve haver um repositório centralizado e um modo de obter exatamente a versão necessária, seja de componentes próprios

ou de terceiros.

É necessário organizar o processo de montagem (build) e o armazenamento dos componentes de cada projeto, de modo independente da configuração de cada desenvolvedor.

### **Qual a vantagem disto?**

Organização! Assim como criamos uma base de configuração no SCM, precisamos ter uma base de configuração de componentes prontos, de modo que cada projeto possa obter de uma só fonte tudo o que é necessário para sua montagem (build) e execução, além de tornar o processo de montagem totalmente independente e transparente.

## **Apache Maven**

*O Maven! Ah, o Maven...*

Você vai **odiar** imediatamente o Maven, assim como eu. Não adianta! A maioria dos meus alunos e colegas, quando apresentados ao Maven, reage de maneira negativa. Somente com o tempo as vantagens de sua utilização aparecem. Por quê? Realmente, eu não sei... acredito que seja porque ele tira um pouco do controle que os desenvolvedores acham que mantêm sobre o projeto. É um pouco de ciúme e desconfiança, sei lá... todos os problemas que eu tive com o Maven não foram culpa dele. Em sua maioria, eram problemas causados pelo meu desconhecimento e por “plug-ins” do Eclipse.

Mas existem muitas críticas ao Maven. Procure no Google: “I hate maven” e você verá vários sites e artigos em blogs. Porém, o Maven é uma excelente alternativa para manter a gestão de montagem e de componentes (juntamente com o Archiva, que veremos mais adiante).

O Maven é simples e genial! Uma ferramenta simplesmente fantástica, que colabora muito para gerenciamento de montagem e de componentes. Ele organiza e controla o “build” evitando que você utilize componentes de versões diferentes das que necessita, além de ter várias funções auxiliares, como: geração de sites, integração contínua e execução de testes e análises. E, caso ainda não esteja convencido, o grupo Apache usa o Maven para gerir seus produtos de software.

Hoje em dia, o Maven já está na versão 3, porém, a maioria dos desenvolvedores utiliza a versão 2 – logo, vamos trabalhar com ela.

A intenção deste capítulo é mostrar o Maven em linhas gerais, ressaltando sua contribuição para melhoria da qualidade do software. Se você quiser saber mais detalhes sobre ele, sugiro que vá ao site do projeto: <http://maven.apache.org/>.

## Repositório

Todos os componentes que seu projeto irá utilizar ficarão armazenados em um repositório. Eles serão identificados por três dados:

- **group id** (identificação de grupo): identifica a empresa que criou o artefato. Usamos o nome de domínio ao contrário, por exemplo: “org.apache.derby”. Em alguns casos, pode incluir o nome do macroprojeto (ou programa), como: “com.sun.facelets”. Os componentes criados pela sua empresa também poderão ser adicionados, em nosso caso: “com.galaticempire”;
- **artifact id** (identificação do artefato): identifica cada artefato controlado pelo Maven. Lembre-se que o Maven lida com módulos “executáveis” ou montagens, e não com código-fonte. Um artefato típico pode ser um arquivo JAR ou WAR. Neste caso, usamos o nome do componente (ou do War file);
- **version** (versão): identifica a versão do componente, geralmente usada assim: <major>.<minor>.<revision>[-SNAPSHOT], onde:
  - ❑ major: número da versão “vertical”, somente alterado quando houver grande alteração (ou adição) de funcionalidades. Pode ocorrer quebra de compatibilidade com versões anteriores;
  - ❑ minor: número da versão “horizontal”, ou seja, alterado quando houver algumas melhorias ou correção de grandes problemas. Geralmente compatível com versões horizontais anteriores;
  - ❑ revision: quando houver apenas correção de problemas;
  - ❑ SNAPSHOT: sufixo utilizado para indicar que a versão ainda está em desenvolvimento.

Cada máquina que utiliza Maven tem uma cópia local do repositório de componentes, que, geralmente, fica dentro de uma pasta oculta chamada “.m2”:

- Linux: “~/.m2”;
- Windows XP: “C:\Documents and Settings\<usuário>\.m2”;
- Windows 7: “C:\Users\<usuário>\.m2”;
- Mac OS X (Lion): “~/.m2”.

Na primeira vez em que você faz um “build”, o Maven baixa as cópias dos componentes para este repositório local, de modo a criar um cache para você. Ele baixa do repositório padrão Maven, que fica em <http://repo1.maven.org/maven2/>.

Os artefatos ficam dentro da pasta “.m2/repository”, de acordo com o “group id”, o “artifact id” e a “version”. Por exemplo:

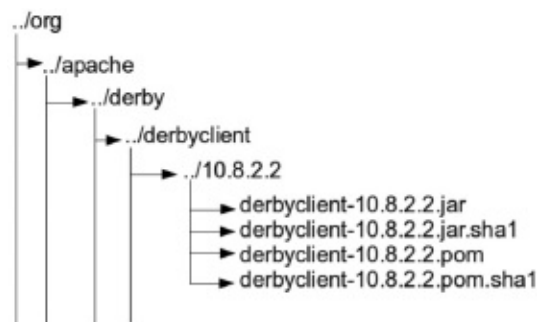


Ilustração 9: Organização do repositório

O arquivo “jar” é o próprio artefato deste componente (“derbyclient”), e o arquivo “pom” é o “Project Object Model” ou o arquivo de descrição do projeto deste componente para o Maven. Os arquivos com extensão “.sha1” servem para controle de integridade (“checksum”) dos arquivos que compõem o artefato. (SHA-1 significa “Secure Hash Algorithm e serve para gerar “hashcodes” de arquivos).

Os componentes criados por você também ficam armazenados no repositório. Isto acontece quando você utiliza o comando “mvn install”. Por exemplo:





Ilustração 10: Projeto interno

Note que a versão deste projeto é “1.0.0-SNAPSHOT”, designando um projeto ainda em desenvolvimento.

## POM

Todo projeto Maven é configurado em um arquivo XML com nome “pom.xml”. Este arquivo deve ficar na pasta raiz do projeto. Ele descreve o projeto, seu tipo de montagem (“build”), sua versão, quais componentes ele necessita, entre diversas outras informações.

Eis um exemplo simples de arquivo “pom.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/maven-v4_0_0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>com.teste</groupId>
```

```
  <artifactId>projeto</artifactId>
```

```
  <version>0.0.1-SNAPSHOT</version>
```

```
  <packaging>war</packaging>
```

```
  <name></name>
```

```
  <description></description>
```

```

<url></url>

<properties>
    <encoding.default>UTF-8</encoding.default>
</properties>

<repositories>

    <repository>
        <id>jboss.org</id>
        <url>http://repository.jboss.org/nexus/content/groups/
public-jboss</url>
    </repository>
</repositories>

<profiles>

</profiles>

<dependencies>

    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>

    <!-- ...Um monte de dependências... -->

</dependencies>

<build>
    <finalName>projeto</finalName>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
            <includes>
                <include>**/log4j.*</include>
            </includes>
        </resource>
    </resources>
    <testResources>
        <testResource>
            <directory>src/test/resources</directory>
            <filtering>true</filtering>
        </testResource>
    </testResources>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>

```

```

        <artifactId>maven-resources-
        plugin</artifactId>
        <version>2.4</version>
        <configuration>
            <encoding>${encoding.default}
            </encoding>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-
        plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
            <encoding>${encoding.default}
            </encoding>
            <source>1.6</source>
            <target>1.6</target>
        </configuration>
    </plugin>
</plugins>
</build>

</project>

```

Como o “pom.xml” é muito grande, cortamos algumas dependências. Vamos examinar as principais informações:

```
<modelVersion>4.0.0</modelVersion>
```

Deve ser sempre “4.0.0”.

```

<groupId>com.teste</groupId>
<artifactId>projeto</artifactId>
<version>0.0.1-SNAPSHOT</version>

```

Identificam o projeto (grupo, artefato e versão).

```
<repositories>
```

Se você quiser utilizar um repositório adicional de componentes.

```
<dependencies>
```

Todos os componentes do qual o seu próprio componente depende.

<build>

A configuração de montagem (“build”) para o seu projeto, incluindo quais “plug-ins” do Maven devem ser executados.

## Fases do Maven

O Maven possui várias “fases” (ou “goals”) do ciclo de vida de montagem de um projeto:

- **validate:** executa uma verificação geral no projeto;
- **compile:** compila o projeto;
- **test:** testa o projeto utilizando os scripts de teste (de acordo com o “pom.xml”);
- **package:** empacota o projeto compilado para distribuição (JAR, WAR);
- **integration-test:** instala o pacote no ambiente para testes de integração;
- **verify:** verifica se o pacote é válido;
- **install:** instala o pacote no repositório local da máquina;
- **deploy:** instala o pacote no repositório compartilhado;
- **clean:** limpa os arquivos gerados em montagens anteriores;
- **site:** gera o website de documentação do projeto.

Podemos executar as fases com o comando “mvn”, por exemplo, dentro da pasta de um projeto podemos executar os seguintes comandos no terminal:

```
mvn clean  
mvn install
```

O “mvn install” forçará a geração do pacote novamente (“mvn package”).

## Criação de projetos

É muito fácil criar um projeto com o Maven, bastando usar o comando:

```
mvn archetype:generate \
```

```
-DarchetypeGroupId=<grupo do arquétipo> \
-DarchetypeArtifactId=<nome do arquétipo> \
-DarchetypeVersion=<versão do arquétipo> \
-DgroupId=<grupo do projeto> \
-DartifactId=<identificação do projeto>
```

Um arquétipo é um modelo para geração de projetos. O Maven vem com vários arquétipos para você usar, por exemplo:

```
mvn archetype:generate \
-DarchetypeArtifactId=maven-archetype-webapp \
-DgroupId=com.teste \
-DartifactId=projetoweb
```

Se formos utilizar os arquétipos padrões do Maven, não precisamos especificar o “groupId”, “artifactId” e “version” do arquétipo.

Na verdade, o mecanismo de arquétipo é excelente para padronizar a geração de projetos e eu recomendo que você crie um ou mais arquétipos padrões para sua equipe ou empresa.

## **Plug-in para a IDE**

Podemos integrar o Maven com a nossa IDE, Eclipse, através de um “plug-in”, o que veremos mais adiante. No momento, vamos ver a outra ferramenta importante para gestão de montagem e componentes.

## **Apache Archiva**

O Archiva é o companheiro ideal do Maven, sendo um repositório de componentes que os armazena de acordo com seu fornecedor (pacote) e número de versão. O Maven pode ser configurado tanto para obter componentes do Archiva como fazer “deploy” para ele, ou seja, ele pode instalar seus componentes internos diretamente no repositório corporativo, disponibilizando imediatamente aquela versão para todos os consumidores.

## **Mas o que podemos fazer com o Archiva?**

Ele é um servidor de repositório de componentes que serve para guardar pacotes de componentes, ou seja: código compilado. Ele controla versões e utiliza a mesma estrutura de armazenamento que o Maven, facilitando a integração entre os dois produtos.

Podemos utilizar o servidor Archiva como um “proxy” para o repositório central do Maven (ou de outros fornecedores), além de podermos também instalar nossos próprios componentes nele, para que outras equipes possam reusar nosso código.

A vantagem de usar o Archiva é a centralização do repositório, com cópias confiáveis e evitando a busca de componentes na Internet, o que aumenta a segurança. Além disto, ele permite criar vários repositórios, que ficam armazenados no servidor onde está sendo executado. No início, ele cria dois repositórios padrões: “internal”, para componentes prontos, e “snapshots”, para componentes em desenvolvimento. O Maven dirá em qual repositório o Archiva deverá guardar um componente.

Como nós utilizamos o Archiva sempre junto com o Maven, nós vamos configurar os dois juntos.

## Relacionamento do Maven com o Archiva

Se ainda não ficou claro, então vamos explicitar o relacionamento das duas ferramentas, o que deve lhe dar uma visão mais objetiva dos benefícios de seu uso em conjunto.

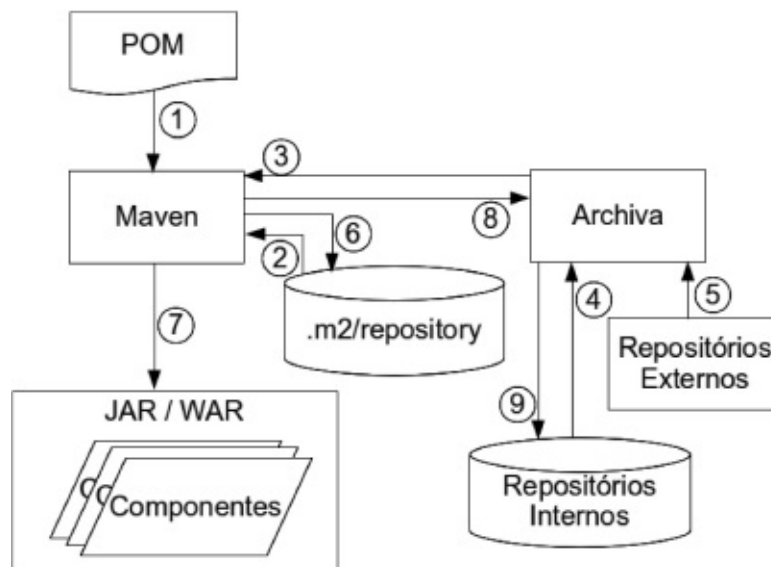


Ilustração 11: Relacionamento do Maven com o Archiva

1. **Empacotamento:** o usuário invoca o Maven para montar e empacotar o

projeto (mvn package). O Maven lê o “pom.xml”, identifica as dependências e verifica o que necessita ser compilado;

2. **Dependências:** o Maven vai ao repositório interno buscar os componentes dos quais o projeto depende (arquivos JAR / WAR etc). Em primeiro lugar, ele vai ao repositório local, buscando pelo “groupId”, “artifactId” e “version”;
3. **Download:** o Maven busca no Archiva todas as dependências não encontradas em seu repositório local. Ele buscará sempre no repositório apropriado (“internal” ou “snapshots”). Uma cópia será armazenada no repositório local, para usos futuros;
4. **Entrega:** o Archiva recebe um pedido do Maven e vai ao repositório interno (dele) para procurar o(s) componente(s). Se encontrar, envia para o Maven;
5. **Download externo:** caso não encontre uma dependência, o Archiva busca nos repositórios externos (que estejam configurados) todos os componentes que estejam faltando em seu repositório interno, copiando para ele;
6. **Install:** se o usuário invocar o Maven com o objetivo “install” (mvn install), o próprio projeto será armazenado no repositório local da máquina (.m2/repositories);
7. **Final:** o pacote do projeto (JAR / WAR) é gerado e inclui todas as dependências necessárias, dentro da pasta apropriada (WEB-INF/lib, por exemplo);
8. **Deploy:** se o usuário invocar o Maven com o objetivo “deploy” (mvn deploy), ele quer instalar o projeto no repositório compartilhado, então o Maven o enviará para o Archiva;
9. **Compartilhamento:** o Archiva recebe um pedido de “deploy” do Maven e, caso as permissões estejam corretas, ele armazena o novo componente no repositório apropriado (“internal” ou “snapshots”).

Qualquer projeto que necessite de um componente, seja ele desenvolvido internamente ou entregue por terceiros, irá buscar no Archiva, que é o ponto central de distribuição de componentes. Além de controlar o acesso, o Archiva nos fornece informações úteis sobre cada componente, por exemplo, de quais

componentes ele depende, quem depende dele etc.

## Configuração do Maven e do Archiva

### Instalação do Maven

A instalação do Maven é muito simples. Se o sistema operacional for **Linux**, provavelmente já existe um pacote pronto. Por exemplo, no Ubuntu basta usar o comando:

```
sudo apt-get install maven2
```

Para outros sistemas, basta baixar o pacote do Maven 2 a partir do site: <http://maven.apache.org/download.html>.

Certifique-se de que a variável de ambiente `JAVA_HOME` esteja criada. Também é importante criar as variáveis:

- `M2_HOME=< diretório onde o Maven foi instalado>`
- `M2= < diretório onde o Maven foi instalado>/bin`

### Instalação do Archiva

O Archiva é um projeto Java EE, logo, precisa de um servidor de aplicação Java. Ele pode ser instalado de duas maneiras diferentes: “standalone” ou como um projeto War. Eu prefiro a segunda opção, pois tenho maior controle sobre a instalação (por exemplo: escolher número da porta TCP, pastas etc.) e evito o uso de servidores desconhecidos. Sugiro instalar o Archiva em um servidor Apache Tomcat 6.

A instalação é bem simples, porém, como o Archiva utiliza alguns recursos especiais, é necessário fazer configurações no Tomcat. As instruções estão todas na documentação (<http://archiva.apache.org/docs/1.3.5/adminguide/webapp.html>), mas vou resumir e traduzir para você. Primeiramente, baixe o arquivo War do Archiva (<http://archiva.apache.org/download.html>) e depois:

1. Crie uma pasta “archiva” dentro da pasta do Tomcat (como subdiretório, no mesmo nível que as outras pastas: “bin”, “logs” etc);



2. Copie o arquivo War para dentro da pasta “archiva”, que você acabou de criar;
3. Crie o arquivo de configuração de contexto para o site do Archiva, no caminho “/conf/Catalina/localhost/archiva.xml”, dentro da pasta do Tomcat, conforme mostrarei a seguir;
4. Copie as dependências do Archiva para a pasta “lib” do Tomcat: “derby.jar” (driver JDBC), “activation-1.1.1.jar” (JavaBeans Activation Framework) e “mail-1.4.jar” (JavaMail). Mostrarei como obter estes arquivos mais adiante;
5. Altere o script de inicialização do Tomcat (“catalina.sh” ou “catalina.bat”, dependendo do sistema operacional), para configurar a variável de ambiente “CATALINA\_OPTS”;
6. Inicie o Tomcat e teste: “http://localhost:8080/archiva”.

O arquivo de configuração de contexto do Archiva deve ser assim:

```
<Context path="/archiva"
    docBase="pasta do tomcat/archiva/apache-archiva-<versao>.war">

<Resource name="jdbc/users" auth="Container" type="javax.sql.DataSource"
    username="sa"
    password=""
    driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
    url="jdbc:derby:<path onde quer gravar o banco "users">;create=true" />

<Resource name="jdbc/archiva" auth="Container" type="javax.sql.DataSource"
    username="sa"
    password=""
    driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
    url="jdbc:derby:<path onde quer gravar o banco "archiva">;create=true"
/>

<Resource name="mail/Session" auth="Container"
    type="javax.mail.Session"
    mail.smtp.host="localhost"/>
</Context>
```

Substitua os textos em negrito e salve o arquivo. Os dois “paths” de banco de dados são para o banco de contas de usuário (“jdbc/users”) e o banco auxiliar do Archiva (“jdbc/archiva”). Escolha onde quer gravá-los e informe uma pasta para cada um.

Quanto às dependências do Archiva:

- Driver JDBC do Apache Derby: instale o Derby ([http://db.apache.org/derby/derby\\_downloads.html](http://db.apache.org/derby/derby_downloads.html)) e copie o arquivo “derby.jar” da subpasta “lib”;
- JavaBeans Activation Framework: baixe de: <http://www.oracle.com/technetwork/java/javase/downloads/index-135046.html>.
- JavaMail: baixe de: <http://www.oracle.com/technetwork/java/javamail/index-138643.html>;

Finalmente, edite o arquivo “catalina.sh” (ou “catalina.bat”) e acrescente a linha:

```
export CATALINA_OPTS="$CATALINA_OPTS -Dappserver.home=$CATALINA_HOME -Dappserver.base=$CATALINA_HOME"
```

ou, para MS Windows:

```
set CATALINA_OPTS="-Dappserver.home=%CATALINA_HOME% -Dappserver.base=%CATALINA_HOME%"
```

Esta linha deve ser acrescentada logo no início, depois das linhas:

```
# Copy CATALINA_BASE from CATALINA_HOME if not already set  
[ -z "$CATALINA_BASE" ] && CATALINA_BASE="$CATALINA_HOME"
```

Ou, no MS Windows, depois da linha:

```
rem ----- Execute The Requested Command -----
```

## Configuração de segurança do Archiva

Na primeira vez em que você acessar o Archiva, ele pedirá para que cadastre a senha do usuário “admin”. Faça isto e anote em lugar seguro.

Depois, temos que criar um ou mais usuários ou grupos. Comece criando um usuário para uma equipe inteira e depois mude conforme sua preferência. Dê a este usuário os papéis de: “Repository Manager” para ambos os repositórios: “internal” e “snapshots” ou então separe os papéis, deixando apenas alguns usuários publicarem no repositório “internal”.

Nota importante: depois de criar um usuário, peça para fazerem login (via website: <http://<servidor>:8080/archiva>) e troquem a senha, caso contrário o Archiva rejeitará as tentativas de fazer “deploy”, pois a primeira senha dada deve ser imediatamente trocada pelo usuário.

## Utilizando o Archiva

Para entender melhor as funcionalidades e opções do Archiva, consulte sua documentação: <http://archiva.apache.org/docs/1.3.5/userguide/index.html> ou <http://archiva.apache.org/docs/1.3.5/adminguide/index.html>.

Você pode pesquisar componentes, criar repositórios, gerenciar usuários e papéis, adicionar componentes, apagar componentes e até mesmo listar componentes.

## Configuração do Maven para utilizar o Archiva como fonte de componentes

Por padrão, o Maven sempre busca componentes no repositório local e, caso não encontre, baixa do repositório central (<http://repo1.maven.org/maven2/>), a não ser que seja informado outro repositório dentro do POM.

Vamos alterar a configuração do Maven para que busque sempre os componentes do Archiva, caso não encontre no repositório local. Dentro da pasta “.m2” crie um arquivo chamado “settings.xml” e adicione as linhas a seguir:

```
<settings>
  <mirrors>
    <mirror>
      <id><nome do repositório></id>
      <url>http://<nome ou ip>:8080/archiva/repository/internal/</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
</settings>
```

Substitua o texto em negrito de acordo com a sua instalação. O meu exemplo é assim:

```
<settings>
  <mirrors>
    <mirror>
      <id>galaticempire.default</id>
      <url>http://localhost:8080/archiva/repository/internal/</url>
```

```
<mirrorOf>*</mirrorOf>
</mirror>
</mirrors>
</settings>
```

Você está informando ao Maven qual será o repositório padrão, caso não seja configurado dentro do “pom.xml” dos projetos. O nome do repositório (<id>) é muito importante, pois ele deve ser mencionado diversas vezes em outras configurações.

Agora, faça um “mvn clean package” no projeto Maven que criamos anteriormente e observe as mensagens. Cada dependência que ele baixar para o repositório local virá do nosso Archiva e não do repositório central, por exemplo:

```
Downloading: http://localhost:8080/archiva/repository/internal//junit/
junit/3.8.1/junit-3.8.1.pom
```

É claro que, na primeira vez que você der um “build”, vai haver uma pequena demora, pois o Archiva vai baixar cada dependência diretamente do repositório central Maven, depois enviando para o Maven local da máquina de desenvolvimento.

### **Configuração do Maven para fazer “deploy” para o Archiva**

O Archiva tem uma boa documentação sobre isto: <http://archiva.apache.org/docs/1.3.5/userguide/deploy.html>. Vamos resumir, traduzir e dar alguns exemplos melhores.

Uma das vantagens de usar o Archiva é poder instalar nossos componentes internos em um único local, de maneira organizada, separada por versão e em um ambiente controlado. Isto é feito através do Maven, com o comando “mvn deploy” (lembre-se: “mvn install” só instala no repositório local).

Aqui tem uma decisão importante: quem terá direito de fazer “deploy” e em qual repositório? O Archiva cria dois repositórios: “internal”, para componentes prontos, e “snapshots”, para componentes em desenvolvimento. Normalmente, os desenvolvedores publicam novas versões provisórias dentro do repositório “snapshots”, mas quem as utiliza sabe do risco que está correndo.

Dentro do repositório “internal” devem ficar apenas versões estáveis, logo, é uma boa ideia separar quem pode e quem não pode fazer “deploy”, especificando quais repositórios podem ser utilizados. Você deverá configurar as máquinas da equipe de maneira separada.

Podemos instalar componentes no Archiva de maneiras diferentes, por exemplo: usando HTTP ou então WebDAV. Usar HTTP é o mais simples e é o que recomendo, caso você não tenha experiência anterior com WebDAV.

Porém, como o Maven vai publicar no Archiva? Será anônimo? Não. Ele terá que ser configurado para utilizar uma conta criada no Archiva. Para isto, temos que identificar o servidor Archiva dentro do “settings.xml”, que fica na máquina local de cada desenvolvedor. Altere o arquivo “.m2/settings.xml” e acrescente as linhas:

```
<servers>
  <server>
    <id>galaticempire.snapshots</id>
    <username>projeto1</username>
    <password>teste123</password>
  </server>
</servers>
```

Podemos configurar mais de um servidor, que será identificado pelo tag “<id>”. Neste caso, estamos apenas configurando o servidor de “snapshots”, mas você pode configurar o “internal” na máquina de quem será o responsável pelo “deploy” das versões finais. O “id” tem que ser o mesmo utilizado no “pom.xml” do projeto.

Mais um detalhe: notou que a senha está em texto simples? Que coisa mal cheirosa, não? Felizmente, o Maven tem alguns recursos para utilizar senhas criptadas: <http://maven.apache.org/guides/mini/guide-encryption.html> mas, para efeito deste livro, podemos usar texto puro mesmo.

Agora, para testar, salve o “settings.xml” e vamos fazer um “deploy” daquele nosso projeto de teste do Maven. Vá para a pasta do projeto e edite o “pom.xml” dele, acrescentando as seguintes linhas:

```
<distributionManagement>
  <snapshotRepository>
    <id>galaticempire.snapshots</id>
```

```
<url>http://localhost:8080/archiva/repository/snapshots/</url>  
</snapshotRepository>  
</distributionManagement>
```

Finalmente, digite “mvn deploy” no terminal (dentro da pasta do projeto) e observe o resultado. Ele deverá indicar que o projeto foi instalado no repositório “snapshots” do Archiva:

[INFO] repository metadata for: ‘snapshot com.teste:projetoweb:1.0-SNAPSHOT’ could not be found on repository: galaticempire.snapshots, so will be created

Uploading: **http://localhost:8080/archiva/repository/snapshots//com/teste/projetoweb/1.0-SNAPSHOT/projetoweb-1.0-20111228.120200-1.war**  
2K uploaded (projetoweb-1.0-20111228.120200-1.war)

Agora, vamos ver no site do Archiva como está o nosso componente. Abra um navegador e digite: “http://localhost:8080/archiva”, entrando com o usuário “admin” e a senha que você escolheu para ele. Selecione “browse” e vá navegando, começando por “com” (“com” / “teste” / “projetoweb” / “1.0-SNAPSHOT”).



Ilustração 12: Nosso projeto dentro do Archiva

Ele até fornece a configuração de dependência para colocar no “pom.xml”.

## Criação de um arquétipo Maven

O site do Maven tem um guia sobre criação de arquétipos: <http://maven.apache.org/guides/mini/guide-creating-archetypes.html>. Vamos

traduzir, resumir e melhorar os exemplos aqui.

Como já vimos, o Maven cria projetos baseado em modelos, conhecidos como “arquétipos” (“archetypes”). Um arquétipo contém o modelo para criar um novo projeto e pode ser muito útil para ajudar a criar um projeto bem organizado desde o princípio.

Usando um arquétipo, podemos indicar o tipo de projeto que vamos criar, quais são as pastas principais (fontes, recursos, testes), qual é a estrutura de pacotes que queremos usar para as classes e quais são as dependências principais a serem utilizadas. Assim, quando criarmos um projeto baseado neste arquétipo, muita coisa já estará configurada.

É claro que podemos utilizar os arquétipos padrões do Maven, mas eles carecem da nossa organização própria. O arquétipo que usamos para criar um projeto de exemplo gerou a seguinte estrutura:

```
projeto / src / main / webapp
```

Certamente não é só isso que queremos, pois precisamos criar uma estrutura de pacotes, outra de recursos (arquivos de configuração) e outra para os arquivos Web. E queremos criar um modelo de “pom.xml” também.

Para isto, precisamos criar nosso próprio arquétipo. Vamos criar um arquétipo para aplicações Java EE Web com JSF e Hibernate (para usarmos em nosso projeto de teste). Um arquétipo Maven é um projeto Maven e tem seu próprio “pom.xml”.

## **Estrutura de um projeto de arquétipo**

Este arquétipo está disponível para download (veja na introdução como baixar os complementos).

Crie uma pasta com o nome do arquétipo e com a seguinte estrutura:

```
../Pasta do arquétipo
../Pasta do arquétipo/src
../Pasta do arquétipo/src/main
../Pasta do arquétipo/src/main/resources
../Pasta do arquétipo/src/main/resources/META-INF
../Pasta do arquétipo/src/main/resources/META-INF/maven
```

```
../Pasta do arquétipo/src/main/resources/archetype-resources/src
../Pasta do arquétipo/src/main/resources/archetype-resources/src/main
../Pasta do arquétipo/src/main/resources/archetype-resources/src/main/java
../Pasta do arquétipo/src/main/resources/archetype-resources/src/main/test
```

Agora, vamos criar um arquivo “pom.xml” para o projeto do arquétipo, dentro de sua pasta raiz (“../Pasta do arquétipo”).

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.galaticempire.guia.arquetipos</groupId>
  <artifactId>galaticempire-archetype-web-jsf-hib</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Arquetipo padrao para aplicacoes web com JSF e Hibernate</name>
  <description>Cria uma aplicacao modelo para desenvolvimento na empre-
sa. Usa JSF com Hibernate.</description>
  <url>http://www.galaticempire.com/guia/</url>
  <organization>
    <name>Galatic Empire</name>
    <url>http://www.galaticempire.com</url>
  </organization>

  <licenses>
    <license>
      <name>Apache 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.html</
url>
    </license>
  </licenses>

  <distributionManagement>
    <snapshotRepository>
      <id>galaticempire.snapshots</id>
      <url>http://localhost:8080/archiva/repository/snapshots</url>
    </snapshotRepository>
  </distributionManagement>

</project>
```

A última parte (<distributionManagement>) não é necessária, servindo apenas



para fazer “deploy” para o repositório do Archiva.

Agora precisamos criar um arquivo descritor do arquétipo. Sua função é indicar quais pastas do projeto correspondem às pastas normais que o Maven espera. Porém, antes de criarmos este arquivo, temos que explicar dois detalhes:

1. Não podemos criar pastas vazias; logo, é melhor criar com arquivos de exemplo ou de texto (“leiam.txt”);
2. O Maven dá problema com alguns tipos de arquivo na criação do arquétipo (imagens e folhas de estilo CSS), logo, use sempre que possível arquivos de texto simples.

Também é bom estudar a estrutura de pacotes que vamos querer utilizar. Eu recomendo separar código-fonte por camada e em pacotes diferentes. Uma boa estrutura é:

- “entidades”: classes DTO e de modelo;
- “managedbean”: classes helper para JSF;
- “negocio”: classes com lógica de negócio e funcionalidade da aplicação;
- “persistencia”: classes com lógica de persistência.

O arquivo descritor de arquétipo deve ser colocado dentro da pasta: “../Pasta do arquétipo/src/main/resources/META-INF/maven” e deve se chamar: “archetype.xml”. Eis um exemplo:

```
<archetype>
  <id>galaticempire-archetype-web-jsf-hib</id>
  <sources>
    <source>src/main/java/package-info.java</source>
    <source>src/main/java/managedbean/package-info.java</source>
    <source>src/main/java/entidades/package-info.java</source>
    <source>src/main/java/negocio/package-info.java</source>
    <source>src/main/java/persistencia/package-info.java</source>
  </sources>
  <resources>
    <resource>src/main/resources/app.properties</resource>
    <resource>src/main/resources/hibernate.cfg.xml</resource>
    <resource>src/main/resources/log4j.properties</resource>
    <resource>src/main/webapp/index.jsp</resource>
    <resource>src/main/webapp/WEB-INF/web.xml</resource>
  </resources>
</archetype>
```

```

    <resource>src/main/webapp/WEB-INF/faces-config.xml</resource>
    <resource>src/main/webapp/WEB-INF/facelet-taglib_1_0.dtd</resource>
    <resource>src/main/webapp/WEB-INF/taglib/component-tags.tld</resource>
    <resource>src/main/webapp/paginas/default.xhtml</resource>
    <resource>src/main/webapp/images/leiametext</resource>
    <resource>src/main/webapp/styles/leiametext</resource>
    <resource>src/main/webapp/templates/gabarito.xhtml</resource>
  </resources>
  <testSources>
    <source>src/test/java/leiametext</source>
  </testSources>
</archetype>

```

Todos os caminhos especificados dentro dos tags “<sources>” e “<test-sources>” conterão código-fonte em Java. Criamos a estrutura de acordo com o que queremos que o Maven gere. Ele vai criar uma pasta “src/main/java” com todos os pacotes da aplicação e uma pasta “src/test/java” com as classes de teste da aplicação. Em cada pasta, ele vai criar também a pasta do pacote da aplicação (“groupid”).

Todas as pastas e arquivos mencionados no “archetype.xml” devem ser criados dentro da pasta “../Pasta do arquétipo/src/main/resources/archetype-resources”, por exemplo:

```

../Pasta do arquétipo/src/main/resources/archetype-resources/src/main/java/
package-info.java
../Pasta do arquétipo/src/main/resources/archetype-resources/src/main/webapp/
images/leiametext
../Pasta do arquétipo/src/main/resources/archetype-resources/src/test/java/
leiametext

```

Os arquivos “package-info.java” servem para criarmos anotações de nível de pacote ([http://java.sun.com/docs/books/jls/third\\_edition/html/packages.html](http://java.sun.com/docs/books/jls/third_edition/html/packages.html)), porém, pode substituir por arquivos “leiametext”, se quiser. O importante é que nenhum diretório pode ficar vazio.

Com esta estrutura, o Maven saberá onde estão os arquivos de código-fonte que devem ser compilados, onde estão os recursos que devem ser preservados e onde estão os casos de teste que devem ser executados pelo JUnit. Finalmente, crie o “pom.xml” protótipo, que será gerado pelo Maven nos novos projetos baseados neste arquétipo. Ele deve ficar na pasta “../Pasta do arquétipo/src/main/resources/archetype-resources”. Deve ser um POM completo, com tudo o que é necessário para compilar projetos do tipo que desejamos.

Uma vez criado o arquétipo, você pode instalá-lo em seu repositório local com “mvn install” (dentro da pasta do projeto). Se quiser instalar no repositório do Archiva, deve incluir o tag “<distributionManagement>”, conforme já falamos, e usar o comando “mvn deploy”.

Eu sugiro que você adicione o arquétipo ao repositório “snapshots”, do Archiva. Assim, você vai exercitar a administração e o uso de componentes.

## Como utilizar o Maven dentro do Eclipse

Para que o Maven possa ser utilizado com o Eclipse, é necessário instalar um “plug-in”. Você pode optar por usar o Eclipse apenas como editor, fazendo o “build” pelo Maven diretamente, mas é muito trabalho. O “plug-in” faz com que o Eclipse crie e mantenha um projeto Maven, sem necessidade de utilizar (ou mesmo instalar) o Maven “standalone”.

O “Plug-in” mais usado era o “m2eclipse”, porém, para versões mais novas, como o Eclipse Indigo, temos o projeto “m2e” (<http://www.eclipse.org/m2e/>), que fornece melhor integração com o Maven.

Para começar, instale o “m2e” diretamente do “update site”: <http://eclipse.org/m2e/download/>. Adicione este site ao seu Eclipse (“help / install new software / Add site”) e marque para instalar tudo.

O “m2e” é o velho “m2eclipse”, criado pelo grupo CodeHaus (e depois passado para a organização Eclipse), só que a versão para o Indigo tem alguns problemas. O primeiro é a integração com o WTP (a plataforma para desenvolvimento Java-Web do Eclipse). Para que os projetos Maven sejam integrados com o ambiente WTP (possam ser reconhecidos como projetos Web), é necessário instalar o componente “m2e-wtp”. Aponte para o “update site”: <http://download.jboss.org/jbosstools/updates/m2eclipse-wtp/> e marque os seguintes itens para instalação:

- “m2e connector for mavenarchiver pom properties”;
- “Maven Integration for WTP”.

Só marque os dois itens mencionados, não instale o “m2e” que está neste site.

## Tenha certeza de usar uma JDK e não uma JRE

Alguns “goals” (ou “mojos”, conforme o jargão do Maven) necessitam utilizar o “javac”, que só está disponível em um pacote Java Development Kit (e não em um JRE). O ideal é que você execute o Eclipse usando uma JDK. Para isto, encerre o Eclipse e edite o arquivo “eclipse.ini” (fica dentro da pasta “eclipse”). Acrescente o parâmetro:

```
-vm  
<path do “javaws”>
```

Lembre-se de colocar o nome em uma linha e o valor em outra. No meu caso (Ubuntu) está assim:

```
-vm  
/usr/lib/jvm/java-6-openjdk/bin/java
```

Crie uma variável de ambiente JAVA\_HOME e aponte para o diretório onde a JDK está instalada. No meu caso é:

```
/usr/lib/jvm/java-6-openjdk
```

E, finalmente, configure sua “Workspace” para usar a JDK (em vez da JRE) para compilar os programas. Abra o menu “Window / Preferences”, localize e expanda o item “Java” e selecione “Installed JREs”, apontando para a JDK e marcando-a como padrão.

## O que você ganha?

Se tudo correr bem, você terá novas opções de projeto. Por exemplo, escolha o menu: “File / New / Project...” e, no diálogo “New project”, note que há um grupo “Maven” e dentro dele existe a opção: “Maven project”.

Você também ganha várias ações no menu de contexto, por exemplo:

- Maven / Add Dependency: adiciona nova dependência ao POM do projeto;
- Maven / Add Plug-in: adiciona novos plug-ins do Maven ao POM do projeto;
- Maven / Update dependencies: atualiza as dependências do projeto, baixando do repositório original;

- Maven / Update project configuration: use com cuidado! Se você alterar as configurações do Maven, como pastas de recursos, de fontes e outros itens, esta opção reconfigura todo o projeto;
- Run as / Maven clean: apaga todos os arquivos gerados anteriormente (dentro da pasta “target”);
- Run as / Maven build: executa os “goals” que você definir. Na primeira vez, ele vai abrir o diálogo de configuração e você pode informar os “goals”, como “clean compile”, por exemplo.

Você pode usar o “Run as / Maven Build...” para configurar um lançador padrão. Por exemplo: “clean package”.

## **Colocando o projeto exemplo dentro do ambiente**

Agora, com tudo configurado, sugiro que você coloque o projeto exemplo dentro do ambiente (veja na introdução como baixar os complementos). Ele está em uma forma padrão do Eclipse, mas nós vamos criar um projeto Maven para ele.

Para começar, temos que criar um novo projeto Maven. Comece criando uma “Workspace” e selecione “File / New / Project...” e depois “Maven / Maven project...”. Clique em “Next” para selecionar o arquétipo.

Precisamos adicionar nosso arquétipo à nossa Workspace (e ao nosso repositório local).

Se você adicionou o arquétipo ao Archiva, então deve poder vê-lo.



Ilustração 13: O arquétipo dentro do Archiva

Clique no botão “Add Archetype...”, da caixa de diálogo “New Maven Project”, no Eclipse. Outra caixa vai abrir (“Add Archetype”) e você deverá informar os campos:

- Archetype Group Id: “com.galaticempire.guia.arquetipos”;
- Archetype Artifact Id: “galaticempire-archetype-web-jsf-hib”;
- Archetype Version: “1.0.0-20111229.121625-1” (preste atenção na versão do seu Archiva);
- Repository URL: “http://localhost:8080/archiva/repository/snapshots/” (a URL do seu repositório).

Clique em “OK” e depois, no diálogo “New Maven Project”, marque a caixa “Include snapshot archetypes”, pois o nosso ainda está com versão “Snapshot”. Pronto! Você deverá ver o seu arquétipo aparecendo.

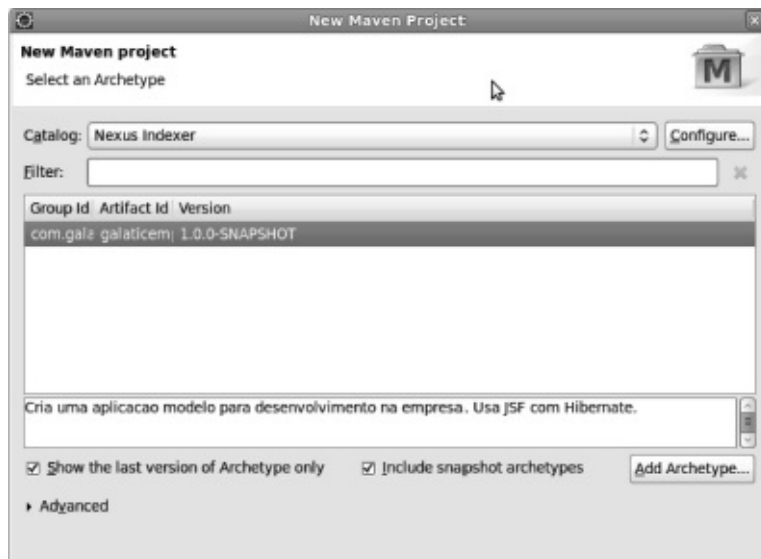


Ilustração 14: Seleção do arquétipo

Só será necessário adicionar o arquétipo se ele não existir no seu repositório local. Depois disto, ele já será copiado do Archiva. Agora, selecione o arquétipo e clique em “Next”. Informe o “Group ID”, “Artifact ID” e “Version” do seu novo projeto, clicando em “Finish”. Agora seu projeto está criado de acordo com o arquétipo. Para o nosso projeto exemplo, use:

- Group ID: “com.galaticempire.guia.microblog”;
- Artifact ID: “microblog”;
- Version: “1.0.0-SNAPSHOT”.

Se você fizer um “Run as / Maven build” com o “goal” “package”, ele vai baixar uma série de componentes do Archiva. Depois, adicione o projeto ao servidor (configure um Tomcat) e rode.

### Altere a porta TCP do servidor do projeto

Atenção: cuidado com as portas do Tomcat! Se você estiver rodando o Archiva e desenvolvendo o projeto na mesma máquina, as portas que o Tomcat vai precisar podem estar ocupadas. A recomendação é que use um servidor separado para o Archiva, mas, para estudo, eu sei que o mais comum é usar uma máquina só, logo, estou rodando o Archiva na porta 8080.

Eu estou usando uma VM com Ubuntu para tudo. Então, edite o arquivo

“server.xml” dentro da pasta “servers”, na Workspace, e mude todas as portas. O Tomcat abre vários conectores: Http, Https, Shutdown etc. Edite o arquivo no Eclipse e procure por “port:”, mudando todos os números. Não se preocupe, pois isto não vai alterar a instalação Tomcat da sua máquina, mas apenas o “server.xml” que fica dentro da Workspace.

### **Copie os arquivos para dentro do projeto Maven**

Abra o pacote do projeto exemplo e copie os arquivos para as pastas correspondentes no novo projeto.

Suba o servidor de banco de dados Derby:

1. Abra a pasta onde instalou o Derby no terminal e vá para a subpasta “bin”;
2. Execute “./setNetworkServerCP” ou “setNetworkServerCP.bat”, dependendo do seu sistema operacional;
3. Execute: “./startNetworkServer” ou “startNetworkServer.bat” (\*);
4. Abra outra janela terminal e vá para a mesma pasta do passo 1;
5. Execute: “./ij” ou “ij.bat”;
6. Digite o comando para criar o banco: connect ‘jdbc:derby://localhost:1527/microblog;create=true’;
7. Mantenha ambas as janelas abertas.

(\*) O Derby sempre usa a porta 1527 por padrão. Alguns softwares, como o SONAR, também usam Derby nessa porta. Se este for o caso, sugiro que você altere a porta padrão. Veja como fazer em:

<http://db.apache.org/derby/docs/dev/adminguide/tadminappssettingportnumbers.html>

Depois, execute: “Run as / Maven build...” e crie uma configuração para executar os “goals”: “clean” e “package”.

O Maven vai criar o pacote e executar os testes. O resultado deve mostrar algo assim:



Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.159 sec

Results :

Tests run: 11, Failures: 0, Errors: 0, Skipped: 0

...

[INFO] -----

[INFO] BUILD SUCCESS

Isto significa que você conseguiu instalar o software e ele está funcionando. Todos os testes JUnit dentro da pasta “src/main/test” foram executados automaticamente pelo Maven. Agora vamos testar para saber se o site está realmente funcionando:



Ilustração 15: O projeto funcionando

Para finalizar, vamos fazer um “deploy” para o Archiva. Para isto, configure o “pom.xml” do projeto, acrescentando as seguintes linhas (antes do “</project>”):

```
<distributionManagement>
  <snapshotRepository>
    <id>galaticempire.snapshots</id>
    <url>http://localhost:8080/archiva/repository/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

Agora, selecione “Run As / Maven build...” e informe o “goal”: “deploy”. Ele vai fazer tudo novamente, inclusive executar os testes. Eis o resultado:

Results :

Tests run: 11, Failures: 0, Errors: 0, Skipped: 0

[INFO]

[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ microblog ---

[INFO] Packaging webapp

[INFO] Assembling webapp [microblog] in [/home/55018335734/projetos/Guia de campo/wsarquetipodemo/microblog/target/microblog]

[INFO] Processing war project

[INFO] Copying webapp resources [/home/55018335734/projetos/Guia de campo/wsarquetipodemo/microblog/src/main/webapp]

[INFO] Webapp assembled in [99 msecs]

[INFO] Building war: /home/55018335734/projetos/Guia de campo/wsarquetipodemo/microblog/target/microblog.war

[INFO] WEB-INF/web.xml already added, skipping

[INFO]

[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ microblog---

[INFO] Installing /home/55018335734/projetos/Guia de campo/wsarquetipodemo/microblog/target/microblog.war to /home/55018335734/.m2/repository/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/microblog-0.0.1-SNAPSHOT.war

[INFO] Installing /home/55018335734/projetos/Guia de campo/wsarquetipodemo/microblog/pom.xml to /home/55018335734/.m2/repository/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/microblog-0.0.1-SNAPSHOT.pom

[INFO]

[INFO] --- maven-deploy-plugin:2.5:deploy (default-deploy) @ microblog ---

Downloading: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/maven-metadata.xml>

Uploading: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/microblog-0.0.1-20111229.165305-1.war>

Uploading: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/microblog-0.0.1-20111229.165305-1.pom>

Uploaded: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/microblog-0.0.1-20111229.165305-1.pom> (4 KB at 67.9 KB/sec)

Uploaded: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/microblog-0.0.1-20111229.165305-1.war> (15832 KB at 70050.6 KB/sec)

Downloading: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/maven-metadata.xml>

Uploading: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/maven-metadata.xml>

Uploading: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/maven-metadata.xml>

Uploaded: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/0.0.1-SNAPSHOT/maven-metadata.xml> (791 B at 24.9 KB/sec)

Uploaded: <http://localhost:8080/archiva/repository/snapshots/com/galaticempire/guia/microblog/microblog/maven-metadata.xml> (301 B at 5.1 KB/sec)

[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 10.301s  
[INFO] Finished at: Thu Dec 29 14:53:05 BRST 2011  
[INFO] Final Memory: 13M/107M  
[INFO] -----

E podemos verificar no próprio Archiva:



Ilustração 16: Nosso projeto no Archiva

O arquivo WAR do nosso projeto está no repositório Archiva, junto com algumas informações importantes. É claro que isto não garante o processo SCM, mas pelo menos organiza as coisas.

Para fechar tudo, acrescente o projeto ao seu repositório Subversion, gerando um tag para ele. Assim, você cumpriu todo o ciclo de atitudes a serem tomadas ANTES do projeto de software iniciar.

## Conclusão

Ainda restam outras atitudes que deveriam ser tomadas ANTES do início do projeto, porém, como estão relacionadas à construção do código-fonte em si, deixaremos para a próxima parte.

Essas atitudes que sugeri (elaborar uma visão geral da arquitetura, priorizar de acordo com o risco, implementar SCM e gestão de montagem e componentes) auxiliam a aumentar a qualidade do projeto de software e de seu produto (o

software em si), pois organizam o trabalho, os papéis e os produtos do projeto.

Sei que não esgotei o assunto apenas nestas páginas, mas espero ter mostrado um caminho para que o desenvolvimento do seu software seja mais suave e organizado. Recomendo que você instale, teste e experimente as ferramentas, além de discutir as técnicas com seus colegas.

## 8.

# Durante a construção do projeto de software

*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.*

C.A.R. Hoare, Palestra do prêmio Turing, da ACM, de 1980

(“Há duas maneiras de construir um projeto de software: uma maneira é fazê-lo tão simples que obviamente não há deficiências e a outra maneira é fazê-lo tão complicado que não existem deficiências óbvias.” – Traduzido pelo autor).

Chegamos à fase de construção do software. O cronograma foi aprovado, os recursos assinalados e você já está programando ou distribuindo as tarefas para os outros. O que mais deve ser feito nesta fase, de modo a aumentar a qualidade do projeto e do software a ser construído? Será que já fizemos tudo o que poderíamos ter feito?

É claro que muitas das atitudes sugeridas nesta parte deveriam ser tomadas ANTES do início da construção, porém, para facilitar o entendimento, resolvi adiá-las para quando fossem mais necessárias.

A simplicidade é o princípio que deve guiá-lo ao longo da jornada de construção do software, como disse o mestre Hoare, na citação que abre esta parte do livro. Outros “monstros sagrados” também defendem este princípio, por exemplo:

*Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better. – Dijkstra.*

(“A simplicidade é uma grande virtude, mas requer trabalho duro para alcançá-la e educação para apreciá-la. E para piorar as coisas: a complexidade vende melhor.” – Traduzido pelo autor)

*do the simplest thing that could possibly work – eXtreme programming*

(“faça a coisa mais simples que possa funcionar” – Traduzido pelo autor).

O que é simplicidade em programação? Como alcançá-la? Humildemente, gostaria de relembrar as minhas três regras de ouro, já citadas no início do livro:

1. Programar o menos possível;
2. Chamar os “universitários”;
3. Trabalhar com transparência.

Ao seguir estas três regras, você acabará produzindo um código simples, adequado e eficaz para o problema, pois terá mitigado o risco de produzir código novo, além de reusar soluções de terceiros e ser absolutamente transparente para seus usuários.

Porém, como disse o prof. Dijkstra, “a complexidade vende melhor”, logo, muita gente acaba enveredando por soluções mais complexas do que deveriam ser, seja por desconhecimento, ansiedade ou mesmo medo de se expor. Existem muitos ditados e regras para construir código-fonte de maneira simples e clara, e a Wikipédia tem uma lista excelente

([http://en.wikipedia.org/wiki/List\\_of\\_software\\_development\\_philosophies](http://en.wikipedia.org/wiki/List_of_software_development_philosophies)).

Vamos ver algumas:

*Faça a coisa mais simples que possa funcionar.*

Esta regra de ouro vem do método ágil XP (eXtreme Programming) e significa que você deve programar o mínimo possível que seja suficiente para atender ao que está sendo pedido. Sempre que vejo alguém complicando as coisas, é por uma das duas razões:

1. Teimosia. A pessoa se recusa a ver soluções ou componentes feitos por outros, que poderiam ser utilizados ou adaptados para resolver o problema;
2. Ansiedade. A pessoa fica tão ansiosa para resolver o problema que fica imaginando diversas situações de uso, tentando fazer com que o código-fonte esteja preparado para elas. Ficam pensando nas inúmeras exceções, em vez de se concentrarem na regra geral.

*You ain't gonna need it - YAGNI*

(“você não vai precisar disto” – Traduzido pelo autor)

Outra regra do método XP, que diz para não acrescentarmos funcionalidades até que sejam realmente necessárias. É comum durante a programação prevermos que necessitaremos de determinada funcionalidade. A pessoa ansiosa sai implementando código-fonte, mesmo que ainda não precise dele.

A consequência é óbvia: aumentamos o custo, o prazo e os riscos do projeto. Lembre-se: “programar o menos possível”.

***Don't repeat yourself – DRY.***

(“Não se repita” – Traduzido pelo autor)

Esta regra diz que não devemos repetir coisas, sejam elas dados ou funções. Este princípio é um dos alavancadores da tecnologia MDA (Model Driven Architecture), na qual o software é gerado a partir de modelos abstratos. Esta regra é muito semelhante ao Princípio de Abstração ([http://en.wikipedia.org/wiki/Abstraction\\_principle\\_\(programming\)](http://en.wikipedia.org/wiki/Abstraction_principle_(programming))).

É claro que, em alguns casos, uma repetição controlada é interessante, ainda mais em bancos de dados, mas devemos tomar muito cuidado, pois toda redundância, mais cedo ou mais tarde, acaba cobrando seu preço.

É comum vermos informações replicadas em bancos de dados, classes e pacotes diferentes. Assim como o próprio código-fonte. A consequência disto é a diminuição da manutenibilidade do software, aumentando muito os custos, prazos e riscos.

***Principle of good enough – POGE.***

(“Princípio do suficientemente bom” – Traduzido pelo autor).

Você deve preferir projetos simples, que possam ser estendidos posteriormente, em vez de projetos complexos, geralmente criados por várias pessoas, com vários modelos em várias camadas de abstração. Lembre-se do famoso ditado popular: “o Bom é inimigo do Ótimo”.

***Codifique de maneira simples e clara***

Se estes argumentos ainda não lhe convenceram, talvez este ditado consiga:

*Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. – Martin Golding.*

(“Sempre programe como se o cara que vai manter o seu código fosse um psicopata violento que sabe onde você mora.” – Traduzido pelo autor).

## Antipatterns: o que deve ser evitado

Já discutimos como criar código simples e claro. Agora vamos ver algumas razões por que este objetivo falha: os famosos antipadrões (antipatterns). Uma lista não conclusiva pode ser encontrada na Wikipédia, mas eu recomendo a leitura do livro “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”, de Brown, Malveau, McCormick e Mowbray. Eis uma breve lista:

### *Analysis paralysis*

(Paralisia de análise – Traduzido pelo autor).

Quando a equipe de projeto demora tempo demais analisando uma situação, sem tomar atitude alguma. Estão focados em demasiados detalhes ou exceções, em vez de produzir alguma coisa e depois acrescentar o que for necessário. É o contrário do **YAGNI** e do **POGE**.

### *Design by committee*

(Projeto por comitê – Traduzido pelo autor).

Normalmente é aplicado quando um grupo de entidades quer criar um padrão de alguma coisa, mas, em projeto de software, acontece quando o projeto (e até mesmo a construção) é feito por um grupo de pessoas onde cada decisão precisa ser discutida e aprovada por todos. Como falta uma liderança forte, não existe uma única visão, logo, são incluídas funcionalidades e complicações desnecessárias. É o contrário de **POGE**.

Esta situação é uma prova clara de falta de liderança. O gerente do projeto não conhece a tecnologia ou não confia nas pessoas (individualmente), logo, abre para discussão cada aspecto do projeto. Já participei de várias equipes assim e de nenhuma delas vi sair alguma coisa boa.

### *Gold plating*

(Folheado a ouro – Traduzido pelo autor).



Quando aprimoramos um software além do que é interessante, na expectativa de agradar ao usuário (ou cobrar mais caro por algo que ele não pediu). Às vezes, entregar funcionalidade adicional ou melhorar um atributo de qualidade é agregar valor e pode gerar maior disposição do cliente (requisitos excitantes – IFQ). Porém, passar deste ponto só gera insatisfação, pois o que pensamos ser agradável pode gerar insatisfação no cliente, além de gerar código-fonte mais complexo, caro e difícil de manter.

É o contrário de “faça a coisa mais simples que possa funcionar”.

### ***Accidental complexity***

(Complexidade acidental – Traduzido pelo autor).

É a complexidade introduzida no código-fonte, que não é essencial para resolver o problema. Existem dois conceitos sobre complexidade: complexidade acidental e complexidade essencial. A Wikipédia tem ótima definição para ambos:

*Accidental complexity is complexity that arises in computer programs or their development process which is non-essential to the problem to be solved. While essential complexity is inherent and unavoidable, accidental complexity is caused by the approach chosen to solve the problem.*

*While sometimes accidental complexity can be due to mistakes such as ineffective planning, or low priority placed on a project, some accidental complexity always occurs as the side effect of solving any problem.*

Wikipédia ([http://en.wikipedia.org/wiki/Accidental\\_complexity](http://en.wikipedia.org/wiki/Accidental_complexity))

(“Complexidade acidental é a complexidade que surge em programas de computador, ou seu processo de desenvolvimento, que não é essencial para o problema a ser resolvido. Enquanto a complexidade essencial é inerente e inevitável, complexidade acidental é causada pela abordagem escolhida para resolver o problema.

Enquanto às vezes a complexidade acidental pode ser devida a erros, como o planejamento ineficaz, ou baixa prioridade colocada em um projeto, alguma complexidade acidental sempre ocorre como o efeito colateral de resolver qualquer problema.” – Traduzido pelo autor).

Como o artigo diz, a complexidade acidental pode ser causada, ou aumentada, pelo efeito colateral da solução aplicada. Como exemplo, vamos imaginar que a empresa precisa de um serviço que receba solicitações, registre no banco de dados e responda com um identificador de “ticket” de serviço. Se a solução adotada for a criação de um servidor “sockets”, a complexidade acidental será muito maior do que se outras soluções padrões fossem utilizadas, como um Web

Service, por exemplo.

Normalmente, o aumento de complexidade accidental é resultado de ansiedade ou teimosia, quando queremos fazer as coisas “do nosso jeito”, ou resolver mais de um problema simultaneamente. Também pode ser resultado de “Gold plating” e “Design by committee”.

### ***Cargo cult programming***

(Cultuar carga, em programação – Traduzido pelo autor)

Quando um programador implementa padrões e soluções que não conhece direito. Na verdade, é o uso ritual de soluções sem pensar como elas funcionam ou se são as mais adequadas.

Hoje em dia, as faculdades “cospem” centenas (ou milhares) de recém-formados no mercado anualmente. Muitos deles saem com um “pacote completo” de soluções na cabeça e, por ter pouca experiência, acreditam que não existam outras maneiras de resolver os problemas. Portanto, saem empregando padrões, frameworks e técnicas sem conhecer ou sem considerar sua adequação ao problema que devem resolver.

Há alguns anos, assisti a uma palestra em uma grande universidade americana, na qual estava sendo debatido o ensino superior de computação. O palestrante disse que, na opinião dele, tecnologias e frameworks como “Java”, “.NET”, “Spring”, “Hibernate”, “Struts” e diversas outras não deveriam fazer parte do currículo universitário. O aluno deveria aprender apenas a utilizar as ferramentas clássicas, como: “C ANSI”, “C++ ANSI”, “STL”, “Unix” e “Shell script”. Todo o resto deveria fazer parte de cursos de especialização, e não do currículo formativo.

Seria excelente, pois o aluno sairia sabendo como construir software da maneira mais “pesada”. Depois aprenderia a otimizar seu esforço com tecnologias e frameworks mais modernos.

## **Não basta ser simples**

Para criar um software de qualidade com um projeto bem-sucedido, não basta a simplicidade na programação. Existem diversas outras atitudes que podemos e

devemos tomar, como bons programadores, as quais vamos explorar nesta parte do livro.

Por exemplo, será que padrões de codificação podem ajudar? Qual é o seu real valor? Também podemos pensar na maneira como estamos modelando os objetos, como estamos organizando o código-fonte, se estamos programando orientados a testes, como vamos integrar o código... enfim, tem muito a fazer.

## O princípio da solução

*If you find a good solution and become attached to it, the solution may become your next problem.*

Dr. Robert Anthony

(“Se você encontrar uma boa solução, e ficar apegado a ela, a solução se tornará seu próximo problema.” – Traduzido pelo autor).

Eu resolvi chamar de “princípio da solução” os critérios, as decisões e as visões que utilizamos para resolver um problema.

Quando o princípio está errado, a solução será inadequada, gerando software de má qualidade. É muito importante prestarmos atenção ao princípio que vamos usar para resolver um problema. Quando a simplicidade, proatividade ou criatividade orientam a escolha da solução, certamente teremos sistemas de boa qualidade e clientes satisfeitos.

O princípio que norteia uma solução está muito relacionado ao aspecto do problema que mais nos preocupa. Vou dar um exemplo: o que você faria se tivesse um vizinho que dá festas de vez em quando e te perturba com o som alto?

- a. Reclamaria com ele. Caso não adiantasse, tentaria processá-lo;
- b. Mudaria para outro local;
- c. Faria um barulho mais alto, até que ele desistisse;
- d. Iria lá e bateria nele;
- e. Sairia para passear nos dias das festas dele;
- f. Daria um jeito dele te convidar para as festas;

g. Mataria ele.

Não há uma solução 100% correta, logo, vamos abstrair os princípios morais da questão. Analisando cada possível solução, vamos entender o princípio que a orientou:

- **Reclamar e processar:** rigorosidade, impor limites, deixar clara sua vontade perante aos outros. O que nos incomoda é a invasão do “nosso espaço”, e não o barulho em si. Uma solução de eficácia duvidosa, pois, dependendo do tipo de vizinho, pode até piorar a situação;
- **Mudar para outro local:** covardia, procrastinação, adiar um problema, evitar aborrecimentos a qualquer custo. Neste caso, o maior incômodo é ter que tomar uma atitude. Esta solução pode resolver o problema atual, mas com alto custo (você vai abrir mão de sua moradia);
- **Fazer mais barulho:** enfrentamento, confronto. O que mais nos incomoda é a petulância do vizinho. Seria uma solução de eficácia duvidosa e que, independentemente do resultado, traria ainda mais desconforto e incômodo para você. Talvez você criasse problemas com outros vizinhos;
- **Bater nele:** enfrentamento extremo, presunção, radicalismo. Mais uma vez, o que nos incomoda é a petulância e audácia do vizinho. Esta solução tem eficácia igualmente duvidosa, já que ele pode acabar te batendo, enquanto aumenta o volume do som. Você também pode ser processado;
- **Sair nos dias de festa:** simplicidade, proatividade. O que nos incomoda é o barulho, então vamos resolvê-lo de forma simples, gastando nosso tempo de forma mais útil. Talvez seja a melhor solução, mesmo que você não consiga evitar todas as festas e gaste um pouco mais de dinheiro para bancar os programas que vai fazer;
- **“Convidar-se” para as festas:** criatividade, inovação, pensamento lateral (“Out-of-the-box”). O que nos incomoda é não estar na festa! Esta solução é eficaz somente se você tiver sucesso. Pode ser uma boa saída para os dias em que não conseguir fazer um passeio diferente, embora possa ter que conviver com pessoas das quais não gosta;
- **Matá-lo:** radicalismo ao extremo. O que nos incomoda é ter que resolver o problema. Embora seja um crime, além de moralmente questionável, resolve o problema imediato, criando diversos outros.

Como vê, o princípio norteia o tipo de solução. Porém, não devemos sempre utilizar os mesmos princípios e ficar repetindo as mesmas soluções, pois isto seria exatamente como o ditado que abre este tópico. Isto acontece quando os princípios que norteiam a solução são: “comodismo” ou “preguiça”.

## **Soluções de problemas de software**

Eu não gosto de simplesmente afirmar que uma solução esteja errada. Prefiro dizer que está inadequada e, geralmente, é porque os seus princípios foram inadequados. Soluções existem para resolver problemas, certo? E quando a solução se torna o problema? Isto geralmente acontece porque foi implementada uma solução inadequada, e o desenvolvimento prosseguiu baseado nela. Logo, outros programas podem ter sido criados como efeito colateral da solução inadequada. Quanto mais tarde descobrimos que uma solução é inadequada, maior é o custo para corrigir a situação. Podemos dizer que o custo de solução é diretamente proporcional ao tempo decorrido desde que o problema foi descoberto.

Vamos ver algumas situações nas quais o princípio de solução inadequado gerou soluções igualmente inadequadas, de baixa eficiência.

### **Tente dar soluções adequadas**

Para começar, siga as três regras de ouro: programar menos, consultar os “universitários” e ser transparente. Vamos ver como fazer:

1. Programar menos: escolha a alternativa de menor risco. Criar código-fonte aumenta o risco, logo, precisamos evitar isto e reduzir a necessidade ao mínimo possível;
2. Consultar: ver como os outros resolveram problemas semelhantes. Para cada problema que você encontre, tenha certeza de que alguém já passou por ele. Às vezes até existe uma ferramenta ou um componente pronto para resolver o problema;
3. Ser transparente: deixe claro os critérios que orientaram a sua solução. Como você quis resolver o problema, quais fatores mais lhe influenciaram, por exemplo: prazo, custo, disponibilidade de recursos etc.

### **Alertando problemas de processamento**

Vamos pensar neste problema, que lhe foi passado pela área de vendas de um cliente:

*A empresa instalou um software de Business Intelligence utilizado para analisar o movimento de vendas diário em um servidor Linux. Ele roda um processamento noturno que é baseado em ETL para atualizar os cubos. O software é Open Source e feito em Java. Funciona bem, porém, não reporta os erros de ETL adequadamente. Ele gera um arquivo XML de cada processamento, que pode conter tags error para cada problema encontrado, com sua descrição e data/hora.*

*Caso ocorra um erro, a equipe tem que ser avisada o mais rapidamente possível, para que entre na interface web e analise o problema. Como as pessoas esquecem de ver se houve problema, muitas vezes a diretoria deixa de receber o relatório pela manhã.*

*Atualmente, um funcionário chega às três horas da manhã para ver os arquivos XML e, caso haja um erro, envia e-mail para os membros da equipe.*

*Precisamos dar uma solução rápida e barata para o problema.*

Como resolveria o problema? Eu fiz esta pergunta a alguns colegas, antigos de profissão, e obtive as seguintes respostas:

Sugestão 1:

*Bem, eu criaria uma classe Java para ler o diretório pegar o último XML. Depois, leria o arquivo e pegaria o string da mensagem de erro, enviando para a equipe. Eu usaria o JavaMail. Depois, colocaria esse script para rodar toda noite, às três da manhã.*

Sugestão 2:

*Eu obteria o código-fonte do projeto, já que é Open Source, e faria uma alteração para que ele enviasse os erros para uma fila transacional, tipo um JMS. Depois, criaria um MessageDriven Bean para ler as mensagens e enviar por e-mail. Depois eu ainda publicaria a nova versão no site do software Open Source.*

### Sugestão 3:

*Bem, eu faria um script para monitorar o diretório e enviaria todos os novos arquivos via e-mail para a equipe. Botaria para rodar no Crontab.*

A primeira solução resolve o problema, porém requer a criação de um programa Java que vai ler diretórios, processar arquivos XML utilizando substring e vai ter que enviar e-mail. Só tem um problema: *eles só querem saber se houve erro e não qual foi a mensagem*. Note que ele vai enviar TODOS os novos arquivos para a equipe inteira, mas será que é isso que foi pedido?

A segunda solução resolve o problema, porém com alta complexidade accidental. Apesar de corrigir a aparente falha do software, ela requer o uso de JMS (para mensagens transacionais), o que não foi solicitado, sendo bem mais cara de implementar. Além disto, seu risco é muito alto, pois não conhecemos a complexidade da alteração a ser efetuada e podemos introduzir erros em um software que já funciona. Neste caso, o programador quis fazer uma solução sofisticada, resolvendo vários problemas que não constam dos requisitos originais. Eu diria que houve uma certa presunção nesta solução, pois ele pensou que poderia resolver vários problemas de uma só vez.

A terceira solução é interessante, mas resolve parcialmente o problema. Apesar de ser simples de implementar, ela envia todos os novos arquivos para os usuários, enchendo suas caixas postais com informações desnecessárias. Com isto, corremos o risco deles ignorarem as mensagens de erro, devido ao grande número de “falsos positivos”. Aqui houve uma certa preguiça para pensar melhor na solução.

A ideia é: “nem tanto ao mar, nem tanto à terra”. Temos que resolver o problema sem extrapolar, mas não devemos solucionar pela metade. Temos que atender a todos os requisitos do cliente.

Então coloquei a questão para um ex-aluno meu, recém-formado, que deu uma solução perfeita:

*Eu criaria um shell script como esse, que peguei da Internet:*

```
#!/bin/sh
arq=""`find . -cmin -30 -iname '*.xml'`
```

```

erro="`cat $arq | grep error`"
if [ -z "$erro" ]
then
    echo "analisei $arq e nao encontrei erro" > ../logs/log.txt
else
    echo "*****encontrei erro em $arq" > ../logs/log.txt
    echo "*****encontrei erro em $arq" | mail -s erro equipe@galati-
cepire.com
fi

```

*Depois colocaria para rodar usando o “crontab”, provavelmente às três da manhã.*

Esta solução resolve o problema, é simples, fácil de implementar e não requer recursos especiais no servidor Linux. O aluno foi inovador porque se concentrou em resolver o problema da maneira mais simples e barata possível.

Este não seria o caso da “sincronização por file system”, considerada uma “gambiarra” anteriormente? Não. É diferente, pois não há necessidade de controle transacional, e o processamento é bem simples: o que se deseja saber é se houve ou não um erro. Só isso.

Por que um recém-formado deu uma solução melhor que a de outros profissionais experientes? Porque foi criativo e não deixou seus “calos” e “manias” influenciarem o seu trabalho. Ele seguiu as três regras de ouro:

- Programar o mínimo possível: usou a solução que iria requerer menos esforço de programação;
- Consultar os “Universitários”: procurou na Internet uma solução simples e pronta;
- Foi transparente: mostrou de onde tirou a solução claramente, sem tentar bancar o sabido.

## **Análise de XML por substring**

Agora eu gostaria de comentar um exemplo real, que vivi há pouco tempo. Fui chamado para fazer análise estática do código-fonte de um sistema grande. Rodei uma ferramenta de análise estática e, como foram apontados vários problemas, resolvi inspecionar visualmente para ver se eram falsos positivos. Um dos módulos com mais problemas era uma classe que lia e analisava um



arquivo XML. Pelo que entendi, ela lia o XML e gerava outro, ou seja: convertia um XML em outro!

O código do módulo tinha centenas de linhas e, em sua maioria, era algo assim:

```
if (arquivo.substring(107,72).equals("opt")) then ...
```

Eram centenas de “ifs” semelhantes a este. Só de olhar, dois “pecados” são aparentes. Você sabe quais são? O primeiro é o uso de “números mágicos” para indicar a posição e o tamanho e o outro é o uso de literal string (“string mágico”), o que dificulta a manutenção do software. Uma implementação melhor seria criar constantes ou arquivos de propriedades, evitando alterar o código caso haja uma alteração no arquivo.

Você vê mais algum problema nessa solução? Não? Ah, fala sério!

O cara está lendo um XML e gerando outro! E gasta centenas de linhas de código só para isto! E se houver alguma mudança nos esquemas dos arquivos, por exemplo, a adição de um novo atributo? E imagine a complexidade de criar casos de teste para um código destes? Fora o tempo gasto na codificação...

Existem várias soluções melhores que a adotada, por exemplo:

- a. Utilizar DOM Parser e ler a coleção de nós e atributos;
- b. Utilizar um framework de “binding”, como o JIBX (<http://jibx.sourceforge.net/>), para transformar em um modelo de domínio, e navegar nessa coleção;
- c. Utilizar XSLT para fazer a transformação.

Se a função é apenas transformar um XML em outro, podemos usar XSLT para fazer isto, dentro do Java ou não. Até os navegadores fazem isto, se o documento tiver um tag “<?xml-stylesheet type=“text/xsl” href=...?>“.

Agora, se você precisa ler um XML, fazer cálculos ou complementar os dados, talvez seja melhor utilizar um framework de “binding”, ou então usar um DOM parser.

O autor do programa foi comodista, preferindo usar apenas o que sabe, talvez

achando que isto lhe traria notoriedade. Ele não quis ver como as pessoas processavam XML ou se havia algo pronto para isto (consulta aos “universitários”). Ao contrário: criou um monstro de código, cuja manutenibilidade é muito reduzida, pois, com o passar do tempo, nem ele vai entender o funcionamento...

## **Sincronização por file system**

Eu já falei sobre isto neste livro, mas é um problema tão recorrente que vale a pena citar um exemplo. File system tem sido utilizado para sincronização e até mesmo IPC (Inter-Process Communications) desde os primórdios da programação. Existe uma versão mais moderna, que é utilizar banco de dados para isto, o que pode ser ainda pior.

O problema é que dois programas precisam trocar informações, mas não estão preparados para se comunicarem diretamente, ou as informações não são dadas de maneira síncrona. Neste caso, um deles grava arquivos em um diretório e o outro lê, gravando as respostas (no mesmo ou em outro diretório). Devem existir subrotinas em ambos os sistemas, cuja função é monitorar os diretórios usados para comunicação, lendo arquivos, processando as tarefas pedidas e gravando arquivos de resposta. Na versão do banco de dados são utilizadas tabelas com este propósito.

Qual é o problema desta solução? Afinal de contas, no início deste capítulo nós mostramos uma solução semelhante (a do script para monitorar o XML). O problema é que existe um limite para o uso de soluções simplistas. No exemplo do sistema de BI, o problema era muito simples e a solução, perfeitamente adequada. Quando a comunicação ou o processamento são mais complexos, este tipo de solução deixa de ser adequado. Os problemas são:

- File system não é transacional;
- A segurança não é das melhores.

E existe até um Antipattern para o uso de banco de dados neste caso: “Database-as-IPC” (<http://en.wikipedia.org/wiki/Database-as-IPC>). A sincronização via banco de dados é mais segura, além de ser transacional; porém, é mais cara, complexa e menos eficiente que outros mecanismos.

Eu dei manutenção em um sistema que era exatamente isto: um processo era acionado pelo crontab e gravava um arquivo. Outro processo lia os últimos arquivos e disparava processamentos internos. No final, gravava outro arquivo para que o primeiro processo lesse. Não precisa dizer que dava problema a todo momento, ou precisa? Então um grupo de “notáveis” se reuniu para tentar “modernizar” o sistema.

A solução que deram para “modernizar o sistema”: criar um servidor socket e um protocolo de aplicação (sobre TCP). “Piraram” de vez! Por que fazer isto? Por que complicar as coisas? Só para poderem dizer que criaram um servidor proprietário? Desta vez, o princípio utilizado foi a presunção, ou seja: pensar que é melhor do que os outros. Imagine dar manutenção em uma porcaria destas...

Existem mecanismos mais simples, práticos e conhecidos de sincronização ou IPC. Um deles é o uso de um servidor de mensagens assíncronas (JMS), que permite o envio de mensagens assíncronas inclusive dentro de transações. O outro mecanismo é o uso de Web services. Através deles, dois sistemas diferentes, em plataformas diferentes, podem se comunicar de maneira segura e até mesmo atravessando firewalls.

## **Conclusão**

Muitas vezes as pessoas ficam tão apegadas a velhas soluções que passam a utilizá-las como princípio para quase todos os problemas. Neste momento, temos que meditar sobre o que disse o grande escritor Pedro Nava:

*A experiência é um carro com os faróis voltados para trás.*

## **Padrões e princípios de projeto**

Esse termo “padrão” é muito sobrecarregado (ops!) atualmente. Ele serve para designar padrões de projeto, padrões arquiteturais e padrões de codificação, entre outros usos. Neste caso, quero me referir ao último significado, que está relacionado à manutenibilidade do código-fonte.

Adotar padrões de codificação é muito importante para melhoria de qualidade de um software, pois facilita a compreensão do código. É claro que não é suficiente, mas é um bom começo. Porém, muita gente resolve impor padrões com motivos

menos nobres, como favorecer o “culto à personalidade”, por exemplo. Duvida? Pois observe bem e verá, muitas vezes, programadores tentando criar padrões para impor aos outros, com o intuito dissimulado de ganhar notoriedade.

As pessoas percebem isto e se revoltam. Aliás, programadores são seres muito vaidosos e difíceis de convencer. Porém, ao se depararem com um “spaghetti code” (código macarrônico), todos reclamam. Logo alguém tenta criar normas de programação e geralmente comete alguns “pecados” que acabam fazendo com que os programadores em geral as rejeitem.

Eu peguei uma frase que ouvi em uma palestra da Sun Microsystems (não lembro quem disse), que era mais ou menos assim:

*Ao pensar em padrões e normas, precisamos nos concentrar apenas no objetivo, que é produzir código-fonte fácil de entender, deixando as exceções para serem tratadas quando for necessário.*

Não seguir este princípio é o grande “pecado” dos “legisladores informáticos” de plantão. Lembro-me de uma discussão que tive com um grupo de colegas há alguns anos. Estávamos tentando criar regras de codificação para nosso projeto e eu defendia que deveriam ser poucas e simples. Um dos programadores mais exaltados defendia que nenhuma variável poderia ter menos de cinco letras em seu nome, e eu dizia que, se o código for claro, isto não é importante. Então, seguiu-se uma discussão ferrenha, com exemplos nos quais seria aceitável ter variáveis com nomes curtos. Ao final, tantas exceções eram criadas que a regra deixava de fazer sentido.

Em vez de ficar se prendendo a “bobagens” como essa (o nome tem que ter cinco letras), por que não olhar o que os outros recomendam? Para começar, eu gosto muito do site Java Ranch ([www.javaranch.com](http://www.javaranch.com)), uma ótima fonte de conselhos para desenvolvedores Java. Uma das áreas mais legais é o “Chicken coop” (galinheiro: <http://www.javaranch.com/style.jsp>), onde eles falam sobre normas básicas de programação. Eu recomendaria que todas elas fossem adotadas e acrescentaria pouca coisa.

Bem, assumindo que você leu e concordou com o “galinheiro” do Java Ranch, vamos então sugerir algumas normas para serem incluídas em seu padrão de codificação. Veja bem: todas as normas sugeridas pelo Java Ranch devem fazer parte de qualquer padrão! As normas que vou citar adiante são complementares a elas.

## Sugestões adicionais de normas de codificação

### Sempre sobrescreva “toString()” (BLOCH)

Não há coisa mais irritante do que, ao depurar um código, ver aquela mensagem padrão de “java.lang.Object”: <classe>@120df4.

Se a sua classe representa uma Entidade, então retorne o identificador desta Entidade, ou então algo que a represente bem para o usuário. Porém, cuidado ao incluir formatação excessiva no literal de retorno, pois pode não atender a todos os usos de sua classe.

Se você retornar o “toString()” adequado, sua classe poderá ser utilizada diretamente em expressões, sem necessidade de invocar métodos específicos. Veja um exemplo:

```
public class Telefone {  
    private String ddd;  
    private String numero;  
    public String getDdd () {  
        return this.ddd;  
    }  
    ...  
    public String getNumero() {  
        return this.numero;  
    }  
}
```

Se eu estiver usando esta classe e precisar ver o telefone, vou ter que fazer isto:

```
System.out.println(telefone.getDdd() + “-”  
                    + telefone.getNumero());
```

Agora, imagine que ela esteja dentro de uma lista (alguma classe que implemente “java.util.List”):

```
for (Telefone t : listaTelefonica) {  
    System.out.println(t.getDdd() + “-”  
                       + t.getNumero());  
}
```

Se sobrescrevermos o “toString()”, podemos facilitar a vida dos usuários. Por exemplo:

```
@Override
public String toString() {
    return this.getDdd()
        + "_"
        + this.getNumero();
}
```

Agora, se eu quiser listar um determinado telefone:

```
System.out.println(telefone);
```

E, melhor ainda, se eu quiser listar uma coleção de telefones, bastaria isto:

```
List<Telefone> lista = (List<Telefone>) getTelefones(usuario.id);
System.out.println(lista);
```

Não preciso fazer um “loop” para pegar os telefones! Todas as implementações de “java.util.List” descendem de “AbstractList”, que sobrescreve o “toString()” invocando o “toString()” de cada objeto contido nelas.

### **Sempre que possível, sobrescreva “equals” e “hashCode” (BLOCH)**

Se você criou uma classe para representar uma Entidade, muito provavelmente precisará saber se uma instância é igual a outra. Isto é feito com o método “equals”. Se você não sobrescrever o método “equals”, a implementação padrão (java.lang.Object) apenas testa se duas instâncias se referem ao mesmo objeto (a == b). Só faz sentido sobrescrever “equals” se a classe possui o conceito de “identidade”.

A implementação de “equals” deve ser feita se a instância lógica é diferente da instância física. Difícil? Então vamos entender melhor:

```
public class Pessoa {
    private String cpf;
    private String nome;
    // suponha que existam os getters e setters apropriados,
    // e que exista um construtor com os campos, ok?
    ...
}
```

Agora, vamos analisar o seguinte código-fonte:

```
List<Pessoa> pessoas = new ArrayList<Pessoa>();
pessoas.add(new Pessoa("123456", "Fulano de Tal");
```

```

pessoas.add(new Pessoa("789123", "Beltrano de Tal");
...
// em outra parte do sistema:
Pessoa p = new Pessoa("123456", "Fulano de Tal");
if (p.equals(pessoas.get(0))) {
    ...
}

```

Se não sobrescrevermos o método “equals” da classe “Pessoa”, a comparação vai ser falsa, embora as duas instâncias da classe se refiram à mesma pessoa. Isto pode acontecer se a camada de apresentação enviar uma instância de “Pessoa” para a camada de negócios, que pode ter sua própria coleção de pessoas.

O método “equals” serve também para encontrarmos instâncias dentro de coleções, por exemplo, se tivéssemos uma lista de pessoas, poderíamos testar a existência assim:

```

if (pessoas.contains(p)) {...}

```

Existem várias exigências para sobrescrever “equals”:

- Deve garantir unicidade: “a” não pode ser igual a “b” e “c” (a não ser que “b” seja igual a “c”);
- Deve garantir simetria: se a.equals(b) então b.equals(a);
- Deve garantir reflexividade: a.equals(a) deve ser verdadeiro;
- Deve garantir transitividade: se a.equals(b) e b.equals(c), então a.equals(c) deve ser verdadeiro.

O critério para sobrescrever o “equals” deve ser o mesmo para se escolher uma chave primária de tabela relacional: deve identificar univocamente uma tupla – neste caso, a instância.

De acordo com o Java Efetivo (BLOCH), o método “equals” não deve retornar “ClassCastException”, caso o objeto passado não seja do mesmo tipo. Ele sugere testar e retornar “false”:

```

@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Pessoa)) {
        return false;
    }
}

```

```

    }
    return this.getCpf().equals(((Pessoa) obj).getCpf())
        && this.getNome()
            .equalsIgnoreCase(((Pessoa) obj).getNome());
}

```

Note que a classe “Pessoa” tem duas propriedades: cpf e nome. Como mais de uma pessoa pode ter o mesmo CPF (marido e mulher), então testamos o nome também.

Se você sobrescreveu “equals”, então deve sobrescrever “hashCode”. Um dos erros mais comuns que vejo é a falta do método “hashCode”, quando o “equals” foi sobrescrito. Se uma instância da sua classe for adicionada a uma coleção baseada em Hash (“HashMap”, “HashTable” e “HashSet”), poderá haver problemas de desempenho ou mesmo de funcionamento.

O “hashCode” deve retornar um número de “particionamento” de sua instância, que serve para auxiliar seu armazenamento em estruturas que utilizem esta técnica (“hashing”). Duas instâncias consideradas iguais pelo método “equals” devem retornar o mesmo valor para o método “hashCode”, porém, instâncias diferentes também podem retornar o mesmo valor.

O “hashCode” deve “espalhar” bem as nossas instâncias, de modo a separá-las em faixas distintas. Se você não conhece o conceito de “espalhamento” ou “tabela de dispersão”, a Wikipédia tem um ótimo artigo: <http://pt.wikipedia.org/wiki/Hashing>.

O principal requisito é: duas classes consideradas iguais pelo “equals” devem retornar o mesmo valor para o “hashCode”. Se a sua classe compara o “equals” baseado em um único campo, então é fácil: basta retornar o “hashCode” deste campo (ou seu valor, se for um inteiro). Porém, se a chave para igualdade for uma combinação de campos, a melhor sugestão (do Java Efetivo) é seguir o algoritmo:

```

@Override
public int hashCode() {
    int resultado = 17;
    resultado = 37 * resultado + <hash da primeira chave>;
    resultado = 37 * resultado + <hash da segunda chave>;
    ...
    resultado = 37 * resultado + <hash da última chave>;
}

```



```

        return resultado;
    }

```

Os números 17 e 37 são apenas dois números primos, tradicionalmente utilizados neste tipo de cálculo. No caso da classe “Pessoa”, que mostramos anteriormente, temos o “CPF”, que é um número “travestido” de String, e o próprio nome da pessoa. Uma implementação básica seria:

```

@Override
public int hashCode() {
    int resultado = 17;
    resultado = 37 * resultado + this.getCpf().hashCode();
    resultado = 37 * resultado + this.getNome().hashCode();
    return resultado;
}

```

Eis um exemplo completo da classe, com os principais métodos de “Object” sobrescritos:

```

package com.galaticempire.teste;
public class Pessoa {
    private String cpf;
    private String nome;
    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public String getNome() {
        return nome;-
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public Pessoa(String cpf, String nome) {
        super();
        this.cpf = cpf;
        this.nome = nome;
    }
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Pessoa))
        {
            return false;
        }
        return this.getCpf().equals(((Pessoa) obj).getCpf())

```

```

        && this.getNome()
            .equalsIgnoreCase(((Pessoa) obj).getNome()));
    }
    @Override
    public int hashCode() {
        int resultado = 17;
        resultado = 37 * resultado + this.getCpf().hashCode(); resultado = 37 * resultado +
        this.getNome().hashCode();
        return resultado;
    }
    @Override
    public String toString() {
        return this.getNome() + " (" + this.getCpf() + ")";
    }
}

```

## Implemente “java.lang.Comparable” se quiser ordenar as instâncias (BLOCH)

Quase sempre nossas instâncias possuem uma “ordem natural” de classificação. Pode ser por ordem numérica, de acordo com uma chave, ou por ordem alfabética. Existem algoritmos e coleções que fazem uso desta “ordem natural” para ordenar as instâncias que armazenam. Eles fazem isto através da interface “Comparable”.

Esta interface possui o método “compareTo()”, que retorna um valor inteiro:

- Zero, se ambas as instâncias forem iguais;
- Negativo, se a instância onde o método está for “menor” que a instância passada para ele;
- Positivo, se a instância onde o método está for “maior” que a passada no argumento.

É muito importante considerar com cuidado a “ordem natural”. É claro que podemos alterá-la com o uso de um comparador externo (java.util.Comparator), mas é mais fácil escolhermos uma ordem que seja aceita por todos. Por exemplo, podemos ordenar nossa classe “Pessoa” por “CPF”, mas isto não significaria coisa alguma, já que o “CPF” é um número arbitrário. Porém, se usarmos ordem alfabética, teremos muito mais aceitação. Então vamos usar a ordem alfabética:

```

public class Pessoa implements Comparable<Pessoa> {
    private String cpf;

```

```

        private String nome;
        ...
        @Override
        public int compareTo(Pessoa pessoa) {
            return this.getNome().compareToIgnoreCase(pessoa.getNome());
        }
    }
}

```

E podemos ver o resultado facilmente. Vamos usar nossa classe colocando as instâncias dentro de um “TreeSet”, que usa a ordem natural para armazenar e recuperar os elementos:

```

Pessoa p = new Pessoa("123456", "Fulano de Tal");
Pessoa p2 = new Pessoa("789123", "Beltrano de Tal");
...
Set<Pessoa> pessoas = new TreeSet<Pessoa>();
pessoas.add(p);
pessoas.add(p2);
System.out.println(pessoas);

```

Veja o resultado que saiu na console:

```
[Beltrano de Tal (789123), Fulano de Tal (123456)]
```

Já classificado em ordem alfabética (naturalmente crescente). Como sobrescrevemos o “toString()”, então não temos o menor trabalho. Poderíamos ter incluído um <CR>+<LF>no “toString()” para separar os elementos em linhas distintas.

Eis nossa classe com todos os métodos discutidos até agora:

```

package com.galaticempire.teste;

public class Pessoa implements Comparable<Pessoa> {
    private String cpf;
    private String nome;
    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {

```

```

        this.nome = nome;
    }
    public Pessoa(String cpf, String nome) {
        super();
        this.cpf = cpf;
        this.nome = nome;
    }
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Pessoa)) {
            return false;
        }
        return this.getCpf().equals(((Pessoa) obj).getCpf())
            && this.getNome()
                .equalsIgnoreCase(((Pessoa) obj).getNome());
    }
    @Override
    public int hashCode() {
        int resultado = 17;
        resultado = 37 * resultado + this.getCpf().hashCode();
        resultado = 37 * resultado + this.getNome().hashCode();
        return resultado;
    }
    @Override
    public String toString() {
        return this.getNome() + " (" + this.getCpf() + ")";
    }

    @Override
    public int compareTo(Pessoa pessoa) {
        return this.getNome().compareToIgnoreCase(pessoa.getNome());
    }
}

```

## Evite o uso de literais numéricos ou textuais

Já ouviu falar em “números mágicos”? Eis um exemplo:

```

if (codigo == 50 or situacao < 4) {
    ...
}

```

Enquanto você estiver dando manutenção no código (e tiver menos que setenta anos de idade), tudo bem! Mas lembre-se daquele ditado:

*Sempre programe como se o cara que vai manter o seu código fosse um psicopata violento que sabe*

*onde você mora.*

Passados três meses, nem você e nem o psicopata saberão o que significa “código 50” ou “situação 4”. São exemplos claros de “números mágicos”, que pioram a manutenibilidade do código-fonte. Um dos exemplos que citei aqui (a análise de XML via “substring”) era cheio de “números mágicos” cuja função era indicar a posição inicial e o tamanho dos atributos em um tag (parece piada, não?).

Não existe desculpa aceitável para utilizar literais numéricos dentro de código-fonte. Hoje temos constantes e enumerações que nos ajudam a evitar tal artifício. Podemos até mesmo utilizar o “import static” e importar as constantes de uma classe especialmente criada para isto.

Assim como os “números mágicos”, temos os textos “embutidos”, por exemplo:

```
int temperatura = Integer
    .parseInt(JOptionPane.showInputDialog(null,
        "Informe a temperatura do processador"));
if (temperatura < 35) {
    msg = "Temperatura normal";
}
else {
    msg = "Atenção";
}
JOptionPane.showMessageDialog(msg);
```

Muito inocente, não? Temos um número “mágico” (a temperatura) e vários textos “embutidos” (as mensagens). O que aconteceria se a temperatura fosse convertida em graus Fahrenheit, ou a aplicação tivesse que ser traduzida para inglês (ou japonês)?

Quando eu abro um código-fonte e vejo literais, minha primeira reação é negativa, com pensamentos não recomendados sobre a ascendência (ou o destino) do programador. Se eu penso assim, imagine o psicopata que vai dar manutenção no seu programa...

Literais devem ser substituídos por constantes e, preferencialmente, parametrizáveis. Se a constante não variar com frequência, podemos até colocar em uma classe de constantes. Porém, textos quase sempre variam muito, sendo alvo de internacionalização; logo, criar um arquivo de propriedades é uma boa

opção.

Um recurso que pode ajudar é o “java.util.ResourceBundle” (<http://docs.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html>), que permite carregar arquivos de propriedades de acordo com a “locale” do usuário.

Imagine aquele exemplo da temperatura. Primeiramente, vamos criar uma classe para acomodar as constantes que precisamos:

- O valor-limite da temperatura;
- O nome do arquivo de propriedades;
- O nome do rótulo para o usuário;
- A mensagem de temperatura normal;
- A mensagem de atenção.

Eis um exemplo:

```
package com.galaticempire.teste;

public class Constantes {
    public static final int TEMPERATURA_NORMAL = 35;
    public static final String RECURSOS
        = "com.galaticempire.teste.temperaturas";
    public static final String MSG_ROTULO = "rotulo";
    public static final String MSG_NORMAL = "normal";
    public static final String MSG_ATENCAO = "acima";
}
```

A constante “RECURSOS” tem o nome do pacote e do arquivo de propriedades, onde as mensagens da aplicação serão gravadas.

Agora criamos um arquivo contendo as mensagens que queremos mostrar ao usuário (temperaturas.properties):

```
rotulo=Informe a temperatura do processador
normal=Temperatura normal
acima=Atenção
```

E criamos um arquivo para as mesmas mensagens, só que em Inglês

(temperaturas\_en.properties):

rotulo=Type processor's temperature  
normal=Normal temperature  
acima=Attention

O próprio mecanismo do “ResourceBundle” saberá qual arquivo deverá ser utilizado, de acordo com o idioma do usuário. Agora vamos reescrever a classe que testa a temperatura:

```
package com.galaticempire.teste;

import java.util.ResourceBundle;
import javax.swing.JOptionPane;
import static com.galaticempire.teste.Constantes.*;

public class TempTest {
    public static void main(String[] args) {
        ResourceBundle res = ResourceBundle.getBundle(RECURSOS);
        int temperatura = Integer
            .parseInt(
                JOptionPane.showInputDialog(
                    null, res.getString(MSG_ROTULO)));
        if (temperatura < TEMPERATURA_NORMAL) {
            JOptionPane.showMessageDialog(null,
                res.getString(MSG_NORMAL));
        }
        else {
            JOptionPane.showMessageDialog(null,
                res.getString(MSG_ATENCAO));
        }
    }
}
```

Agora, a aplicação não contém “números mágicos” ou textos “embutidos”. Até mesmo os literais que identificam cada mensagem estão no arquivo de constantes, e a aplicação já está internacionalizada (e localizada para “pt” – Português e “en” – inglês). Se quiser testar com outro idioma, abra o menu “Run Configurations” e acrescente o seguinte argumento da VM:

```
-Duser.language=<idioma>
-Duser.language=en
```

O que ganhamos com isto? Manutenibilidade! Sim, pois agora TODAS as constantes estão dentro de um arquivo e as mensagens, em arquivos externos, o que facilita a localização da aplicação.

## **Características de ambientes operacionais**

Um dos muitos problemas que temos são as características de diferentes ambientes operacionais. Ainda vou falar mais sobre o assunto, porém, como estamos falando de arquivo de propriedades, é interessante começar a pensar sobre características diferentes dos ambientes de: desenvolvimento, testes, homologação e produção.

A maioria das aplicações corporativas utiliza ambientes operacionais separados, de acordo com o estado do software. Se ele estiver estável, então estará em utilização plena, sendo instalado e executado no ambiente de “produção”. Se estiver em desenvolvimento, estará em um ambiente onde os desenvolvedores possam fazer testes unitários, com dados falsos, também conhecido como ambiente de “desenvolvimento”. Os outros dois ambientes muito utilizados são:

- Testes: para execução de testes de integração e de sistema. Um ambiente semicontrolado, utilizado para testar as qualidades do sistema ANTES dos testes de aceitação;
- Homologação: para execução de testes de aceitação pelo cliente.

Dependendo do que você estiver utilizando, existem características que variam de acordo com o ambiente, entre elas:

- Nome do servidor de banco de dados;
- Caminho do componente EJB ou do Web Service;
- Conta de usuário e senha para consumir recursos.

Colocar estas informações dentro do código-fonte, para mim, é demissão por justa causa! Porém, colocar em arquivos de propriedades (ou de configuração – XML) pode criar alguns problemas. Mais adiante veremos alternativas melhores para cada tipo de característica de ambiente.

## **Escreva código autodocumentado**

Como eu já mencionei anteriormente, muitos desenvolvedores confundem “código autodocumentado” com “código comentado”, que são duas coisas completamente diferentes.



Outro dia estava conversando exatamente sobre este assunto com um colega, mas antigo que eu, que me dizia que sempre escreveu código autodocumentado, sendo constantemente elogiado pelos outros. Para provar o que estava dizendo, me mostrou um trecho de código que, segundo ele, deveria ter mais de vinte anos. Era uma quantidade tão grande de asteriscos que o programa parecia até animação de “Star Wars”. Na verdade, ele entendeu que código autodocumentado era código cheio de comentários...

Eu falarei sobre comentários mais adiante, porém, vou repetir algo que já mencionei: comentários podem ser ruins. Na verdade, Martin Fowler (FOWLER) os classifica como “maus cheiros” no código-fonte. Ele tem uma postura extremamente objetiva sobre o assunto:

*When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.*

Martin Fowler (FOWLER)

(“Se você sentir necessidade de escrever um comentário, tente primeiramente refatorar o código, de modo que qualquer comentário se torne supérfluo.” – Traduzido pelo autor).

Vamos ver um pequeno exemplo. Imagine o código para calcular a taxa de juros a ser cobrada em um financiamento. Vamos ver o método que calcula a taxa de juros:

```
package com.galaticempire.teste;
```

```
import java.util.List;
```

```
public class Emprestimo {
```

```
    ...
```

```
    public double calcularTaxa (Cliente cliente, double montante,
                                int meses, int
                                tipo) {
```

```
        double tusar = lerTaxaPadrao();
        if (meses < 6 && (tipo == 2 || tipo == 4)) {
            tusar = tusar * calcPercEcp();
        }
        switch(tipo) {
        case 1:
            tusar = calcTxc();
            if (meses < 2) {
                tusar = tusar * 0.80d;
            }
            break;
```

```

        case 2:
            if (cliente.lcredcli < montante) {
                tusar = tusar * tcalcprcli(cliente);
            }
            break;
        case 3:
            if (cliente.emprestimos.size() > 0) {
                for (Emprestimo emp : cliente.emprestimos) {
                    if (emp.getTipo() == 3) {
                        tusar = tusar * 1.15d;
                    }
                }
            }
        }
        return tusar;
    }
    ...
}

```

Depois, uma outra classe é responsável por criar novos empréstimos. Eis um trecho do seu código-fonte:

```

Emprestimo emprestimo = new Emprestimo (cliente, meses, montante, tipo);
emprestimo.taxa = emprestimo.calcularTaxa(cliente, montante, meses, tipo);

```

O cálculo da taxa está meio difícil de entender, não? Seria “legal” se contivesse alguma explicação sobre os critérios e cálculos. O programador também pensou assim, logo colocou alguns comentários para explicar como o cálculo funciona:

```

public double calcularTaxa (Cliente cliente, double montante,
                                                                    int meses, int tipo) {

    /*
     * Calcula a taxa a ser usada no próprio emprestimo. Tem 4 tipos:
     * 1=consignado com desconto em folha
     * 2=normal
     * 3=emergencial. Se tiver mais de 1, aumenta a taxa em 15%
     * 4=especial
     * ele pega a taxa padrão do banco de dados, mas pode aumentar ou
     * diminuir
     * de acordo com outros parâmetros.
     */
    double tusar = lerTaxaPadrao(); // Lê a taxa padrão do banco de dados
    if (meses < 6 && (tipo == 2 || tipo == 4)) {

```

```

        // empréstimo de curto prazo, adicionar percentual à taxa
padrão
        tusar = tusar * calcPercEcp();
    }
    switch(tipo) {
    case 1:
        tusar = calcTxc(); // é consignado use a taxa consig
        if (meses < 2) {
            tusar = tusar * 0.80d; // Se menor que 2 meses, dê um
desconto
        }
        break;
    case 2:
        // Empréstimo normal
        if (cliente.lcredcli < montante) {
            // cliente pede mais que o limite
            tusar = tusar * tcalcprcli(cliente); // calcula da tx
risco
        }
        break;
    case 3:
        if (cliente.emprestimos.size() > 0) {
            // ver se já tem outro emergencial. Se tiver, aumenta taxa em
15%
            for (Emprestimo emp : cliente.emprestimos) {
                if (emp.getTipo() == 3) {
                    tusar = tusar * 1.15d;
                    break;
                }
            }
        }
        // tipo 4 é comum usar tx padr
    }
    return tusar;
}

```

Melhorou alguma coisa? Bem, acho que não... Agora, além de ler o código, tenho que ler e entender os comentários...

Para demonstrar a ineficácia dos comentários, imagine que o requisito foi alterado e agora o critério para empréstimos emergenciais mudou. Em vez de aplicar multa de 15% (para quem tem mais de um empréstimo deste tipo), o programa deverá verificar a quantidade de empréstimos emergenciais que o cliente tem. Se o cliente tiver mais do que dois empréstimos emergenciais, então a função deverá obter o percentual a ser aplicado à taxa. Eis a alteração efetuada:

case 3:

```

if (cliente.emprestimos.size() > 0) {
    // ver se já tem outro emergencial. Se tiver, aumenta taxa em 15%
    int conta = 0;
    for (Emprestimo emp : cliente.emprestimos) {
        if (emp.getTipo() == 3) {
            conta++;
        }
    }
    if (conta > 2) {
        tular = tular * tcalctxempemadic(conta);
    }
}

```

Legal, não? Testado e funcionando! Só tem um problema: o programador não alterou o comentário!

Isto é exatamente o que a “granny” (<http://www.javaranch.com/granny.jsp>), do Java Ranch, diz: “debug only code: comments can lie!” (depure apenas o código, os comentários podem mentir). Muita gente vai argumentar que sempre revisa os comentários e que isto é muito difícil de acontecer, mas sejamos francos: o código está fácil de alterar? Não está acontecendo nada errado?

### **Mais de uma responsabilidade significa mais de um eixo de mudança**

Para começar, o que o cálculo da taxa está fazendo dentro da classe “Empréstimo”? O cálculo é uma regra de negócio e não uma responsabilidade específica da classe “Empréstimo”, certo? O problema é que a classe “Empréstimo” é considerada como parte do modelo, logo, existem várias classes que dependem dela. Uma alteração na classe “Empréstimo” sujeita todas as outras classes a possíveis problemas.

O princípio da responsabilidade única (“Single Responsibility Principle”) diz que cada responsabilidade que uma classe assume é uma razão para alterações. Logo, uma classe deve ter uma e somente uma razão para ser alterada. Não alterar a classe “Empréstimo” só porque a regra de cálculo da taxa mudou; afinal de contas, só serão afetados os novos empréstimos e não os antigos.

### **Devemos sempre evitar acoplamento de controle em métodos**

Além do mais, isto diminuirá a dependência que outras classes possam ter da classe “Empréstimo”, por utilizá-la para calcular taxas. Então vamos criar uma classe “RegrasDeEmprestimo” e colocar o método dentro dela, que passa a

receber os dados do empréstimo. Porém, este não é o maior problema do código. Na verdade, o método “calcularTaxa” está sendo controlado externamente – logo, há um acoplamento de controle com quem o invoca. Isto é devido ao uso do condicional baseado no parâmetro. Na verdade, existem várias regras dependendo do tipo de empréstimo:

1. Empréstimo consignado que utiliza a taxa de consignação com um possível desconto, se for de curto prazo;
2. Empréstimo normal que usa a taxa padrão, mas pode ter um percentual de risco;
3. Empréstimo emergencial que está sujeito à quantidade que o cliente já possui;
4. Empréstimo comum, que simplesmente usa a taxa padrão com o desconto para curto prazo.

Notamos que a diferença entre o “Empréstimo normal” e o “Empréstimo comum” é a ausência do percentual de risco. Não é difícil perceber que esta parte do sistema é muito sensível à política da empresa, logo, deve sofrer alterações com maior frequência. Poderíamos utilizar o refactoring: “Substituir condicional por polimorfismo” proposto por Fowler (FOWLER). Neste caso, criaríamos uma classe padrão “RegraDeEmprestimo” e classes filhas para cada tipo de empréstimo.

Só que não é exatamente o caso. As regras nada têm em comum. Podemos criar uma interface comum para cada tipo de regra e instanciar de acordo com o tipo de empréstimo. Podemos usar um padrão “Strategy” para escolher qual regra utilizar. Eis a interface comum a todas as regras de cálculo de taxa de empréstimo:

```
public interface RegraDeEmprestimo {  
    public double calcularTaxa(Cliente cliente, Emprestimo emprestimo);  
}
```

Agora, vamos ver como ficaria a classe “EmprestimoNormal”:

```
package com.galaticempire.teste;  
  
import static com.galaticempire.teste.Constantes.*;
```

```

public class EmprestimoNormal implements RegraDeEmprestimo {

    @Override
    public double calcularTaxa(Cliente cliente, Emprestimo emprestimo) {
        double taxaCalculada = lerTaxaPadrao();
        if (emprestimo.meses < CURTO_PRAZO_NORMAL) {
            taxaCalculada *= calcularPercentualCurtoPrazo();
        }
        double limiteDeCreditoDoCliente = cliente.lcredcli;
        if (limiteDeCreditoDoCliente < emprestimo.montante) {
            taxaCalculada *= obterTaxaDeRiscoDoCliente(cliente);
        }
        return taxaCalculada;
    }

    private double obterTaxaDeRiscoDoCliente(Cliente cliente) {
        double taxaDeRisco = 1.0d;
        // obter ou calcular a taxaDeRisco daquele cliente...
        return taxaDeRisco;
    }

    private double calcularPercentualCurtoPrazo() {
        double percentualCurtoPrazo = 1.0d;
        // Obter ou calcular o percentual curto prazo...
        return percentualCurtoPrazo;
    }

    private double lerTaxaPadrao() {
        double taxaPadrao = 0.0d;
        // Ler e/ou calcular a taxa padrão...
        return taxaPadrao;
    }
}

```

Eu empreguei alguns refactorings aqui, como “introduzir variável explicativa”, pois nós temos acesso à classe “Empréstimo”, mas não à classe “Cliente”; logo, não podemos trocar o nome horrível daquela propriedade “lcredcli”, mas trocamos o nome de outros métodos originalmente utilizados na classe “Empréstimo”.

Você pode alegar que, como o método tem um “if”, ainda existe um condicional, logo, o refactoring não foi perfeito. Bem, condicionais sempre vão existir. O problema ocorre quando um método muda radicalmente o seu comportamento dependendo de um parâmetro, como o que ocorria anteriormente, denotando acoplamento “de controle”.

Agora vamos ver a classe para regra de empréstimo consignado:

```
package com.galaticempire.teste;

import static com.galaticempire.teste.Constantes.*;

public class EmprestimoConsignado implements RegraDeEmprestimo {

    @Override
    public double calcularTaxa(Cliente cliente, Emprestimo emprestimo) {
        double taxaCalculada = lerTaxaConsignada();
        if (emprestimo.meses < CURTO_PRAZO_CONSIGNADO) {
            taxaCalculada *= REDUCAO_CURTO_PRAZO_CONSIGNADO;
        }
        return taxaCalculada;
    }
    private double lerTaxaConsignada() {
        double taxaConsignada = 0.0d;
        // Ler e/ou calcular a taxa para empréstimos consignados...
        return taxaConsignada;
    }
}
```

Repare que criamos mais duas constantes, reforçando nosso princípio de evitar literais a qualquer custo. Eis a classe para empréstimos emergenciais:

```
package com.galaticempire.teste;

import static com.galaticempire.teste.Constantes.*;

public class EmprestimoEmergencial implements RegraDeEmprestimo {

    @Override
    public double calcularTaxa(Cliente cliente, Emprestimo emprestimo) {
        double taxaCalculada = lerTaxaPadrao();
        int QtdEmprestimosEmergenciais = 0;
        for (Emprestimo emprestimoCliente : cliente.emprestimos) {
            if (emprestimoCliente.getTipo() == EMPRESTIMO_EMERGEN-
CIAL) {
                QtdEmprestimosEmergenciais++;
            }
        }
        if (QtdEmprestimosEmergenciais > LIMITE_QTD_EMERGENCIAIS) {
            taxaCalculada = taxaCalculada
                * calcularAcrescimoTaxa(QtdEmprestimosEmergenciais);
        }

        return taxaCalculada;
    }
}
```

```

private double lerTaxaPadrao() {
    double taxaPadrao = 0.0d;
    // Ler e/ou calcular a taxa padrão...
    return taxaPadrao;
}

private double calcularAcrescimoTaxa(int qtdEmprestimosEmergenciais)
{
    double acrescimo = 1.0d;
    // obter ou calcular o acréscimo para quem tem mais que o
    // limite
    return acrescimo;
}
}

```

Aqui comecei a notar uma duplicidade de código que está me incomodando: o método “lerTaxaPadrao()” está replicado. Vamos pensar nele mais adiante. Agora veremos a classe para empréstimos comuns:

```

package com.galaticempire.teste;

import static com.galaticempire.teste.Constantes.CURTO_PRAZO_NORMAL;

public class EmprestimoNormal implements RegraDeEmprestimo {

    @Override
    public double calcularTaxa(Cliente cliente, Emprestimo emprestimo) {
        double taxaCalculada = lerTaxaPadrao();
        if (emprestimo.meses < CURTO_PRAZO_NORMAL) {
            taxaCalculada *= calcularPercentualCurtoPrazo();
        }

        return taxaCalculada;
    }

    private double calcularPercentualCurtoPrazo() {
        double percentualCurtoPrazo = 1.0d;
        // Obter ou calcular o percentual curto prazo...
        return percentualCurtoPrazo;
    }

    private double lerTaxaPadrao() {
        double taxaPadrao = 0.0d;
        // Ler e/ou calcular a taxa padrão...
        return taxaPadrao;
    }
}

```



Mais uma vez noto a duplicidade, pois temos os métodos “lerTaxaPadrao()” e “calcularPercentualCurtoPrazo()”. Podemos tomar duas atitudes:

- Substituir a interface comum por uma classe abstrata, implementando os métodos duplicados nela;
- Criar uma outra classe e mover estes métodos para ela.

À primeira vista, substituir a interface por classe abstrata parece uma boa ideia, mas não é... nem todas as regras usariam estes métodos, logo, eles não representam um comportamento comum. Melhor seria criar uma outra classe e utilizar o padrão “Delegate” para delegar a execução para esta classe específica. Esta solução é a melhor. Vamos ver como ficaria esta classe:

```
package com.galaticempire.teste;

public class RegrasAuxiliares {
    public static double calcularPercentualCurtoPrazo() {
        double percentualCurtoPrazo = 1.0d;
        // Obter ou calcular o percentual curto prazo...
        return percentualCurtoPrazo;
    }

    public static double lerTaxaPadrao() {
        double taxaPadrao = 0.0d;
        // Ler e/ou calcular a taxa padrão...
        return taxaPadrao;
    }
}
```

Eu optei por criar os métodos estáticos, mas isto dependerá de sua função. Agora é só delegar em cada classe. Vou mostrar como ficaria na classe “EmprestimoNormal”:

```
package com.galaticempire.teste;

import static com.galaticempire.teste.Constantes.*;
import com.galaticempire.teste.RegrasAuxiliares;;
public class EmprestimoNormal implements RegraDeEmprestimo {

    @Override
    public double calcularTaxa(Cliente cliente, Emprestimo emprestimo) {
        double taxaCalculada = lerTaxaPadrao();
        if (emprestimo.meses < CURTO_PRAZO_NORMAL) {
            taxaCalculada *= calcularPercentualCurtoPrazo();
        }
    }
}
```

```

    }
    double limiteDeCreditoDoCliente = cliente.lcredcli;
    if (limiteDeCreditoDoCliente < emprestimo.montante) {
        taxaCalculada *= obterTaxaDeRiscoDoCliente(cliente);
    }

    return taxaCalculada;
}

private double obterTaxaDeRiscoDoCliente(Cliente cliente) {
    double taxaDeRisco = 1.0d;
    // obter ou calcular a taxaDeRisco daquele cliente...
    return taxaDeRisco;
}

private double calcularPercentualCurtoPrazo() {
    return RegrasAuxiliares.calcularPercentualCurtoPrazo();
}

private double lerTaxaPadrao() {
    return RegrasAuxiliares.lerTaxaPadrao();
}
}

```

Note que eu não removi os métodos locais da classe, apenas deleguei a execução para a classe auxiliar. A diferença pode ser pequena, mas me dá a chance de executar algum procedimento, mesmo que seja para gerar log.

Finalmente, a classe que originalmente gerava o empréstimo será alterada para usar o algoritmo mais apropriado:

```

Emprestimo emprestimo = new Emprestimo (cliente, meses,
                                         montante, tipoEmprestimo);
RegraDeEmprestimo regra = null;
switch (tipoEmprestimo) {
    case EMPRESTIMO_NORMAL:
        regra = new EmprestimoNormal();
        break;
    case
    EMPRESTIMO_COMUM:
        regra = new EmprestimoComum();
        break;
    case EMPRESTIMO_CONSIGNADO:
        regra = new EmprestimoConsignado();
        break;
    case EMPRESTIMO_EMERGENCIAL:
        regra = new EmprestimoEmergencial();

```

```
}  
emprestimo.taxa = regra.calcularTaxa(cliente, emprestimo);
```

Na verdade, subimos o condicional de nível de abstração para um local onde ele é mais aceitável. O código-fonte todo está mais claro e simples de entender e alterar, dispensando comentários. Para falar a verdade, os únicos comentários que deixei foi porque não quis entrar em detalhes do código, como nas funções “lerTaxaPadrao()” e “obterTaxaDeRiscoDoCliente()”, para dar alguns exemplos.

O mais importante é a lição que vai ficar. Eu não tive este trabalho todo só para evitar os comentários. Na verdade, a necessidade de comentários é sinal de mau cheiro no código e precisa ser tratada. O resultado é que o código ganhou em manutenibilidade.

## Comentários

Comentários devem ser utilizados com apenas um propósito: identificar a classe e seus métodos. Para isto, usamos o padrão Javadoc ([http://www.oracle.com/technetwork/java/javase/De\\_acordo\\_com\\_o\\_sitedocumentation/index-137868.html](http://www.oracle.com/technetwork/java/javase/De_acordo_com_o_sitedocumentation/index-137868.html)), de modo a documentar a classe e sua interface pública.

Métodos que não são visíveis publicamente devem ser documentados da mesma maneira, utilizando-se o padrão:

```
/*  
* comentário de métodos “default, protected e private  
*/
```

Algumas pessoas colocam “(non-javadoc)” após a abertura do bloco de comentários, como o Eclipse recomenda.

O que você deve colocar em um comentário Javadoc:

- Descrição do método (em HTML);
- @param: parâmetro do método (pode haver mais de um);
- @return: o que o método retorna;

- @throws: se ele lança alguma exceção;
- @see: referência para outros pontos da documentação.

Para classes temos:

- Descrição da classe (em HTML);
- @author: autor;
- @version: versão atual.

Se quiser, pode utilizar os mesmos tags para os métodos privados, não se esquecendo de abrir o bloco com “/\*” em vez “/\*\*”.

## **Princípios de projeto orientado a objetos**

Muitos dos problemas que levam a refactorings estão relacionados à falta de compreensão e de aplicação dos princípios consagrados de projeto OO (MARTIN et al). Eu já mencionei alguns deles neste livro, mas vale a pena rever os mais importantes.

### **“Single Responsibility Principle” (princípio da responsabilidade única)**

Cada classe deve ter uma única responsabilidade, já que cada responsabilidade é um foco de mudanças. Entendamos como “responsabilidade”. Nós vimos um bom exemplo anteriormente: a classe “Empréstimo”. A sua responsabilidade principal era registrar e representar um empréstimo e seus relacionamentos. Porém, ela também era responsável por calcular a taxa do empréstimo, de acordo com a política da empresa. Isto sempre gera mais alterações do que o necessário e é um exemplo de baixa coesão da classe (veremos mais adiante).

Às vezes é necessário que uma classe implemente mais de uma responsabilidade, o que não é desejável mas pode ser útil. Neste caso, é interessante separar cada responsabilidade em uma interface separada, deixando a classe implementar ambas.

### **“Interface Segregation Principle” (princípio da segregação de interfaces)**

Este princípio diz que a dependência de uma classe para outra deve ser baseada

na menor interface possível. A princípio parece estranho, mas podemos pensar do ponto de vista da classe cliente (a que usa a outra): clientes diferentes, interfaces diferentes. Se uma classe expõe uma única interface para todos os clientes, mesmo para os que não precisam de todos os métodos, também é um sinal de que a classe pode estar violando o princípio da responsabilidade única.

### **“Dependency Inversion Principle” (princípio da inversão de dependência)**

De acordo com o site de princípios de projeto orientado a objetos (MARTIN et al), este princípio diz duas coisas:

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações;
- Abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.

Se você identificou comportamentos nas classes que vai modelar, então deve criar interfaces para cada comportamento em separado, implementando, posteriormente, em uma ou mais classes.

As classes devem depender apenas das abstrações (interfaces, classes abstratas etc.) e nunca das classes concretas. É o caso quando mandamos calcular o empréstimo utilizando a interface “RegraDeEmprestimo”.

Depois de identificar as interfaces, você pode começar a desenvolver seu sistema, mesmo que as classes que implementam as interfaces ainda não estejam prontas. Você pode utilizar algum software de “Mock”, como o jMock (<http://www.jmock.org>) para sintetizar o comportamento desejado.

Você deve tentar evitar instanciar classes concretas em seu código-fonte, e o único mecanismo para isto é a injeção de dependências. Em Java EE 6.0 podemos utilizar o CDI (Contexts and Dependency Injection) para “injetar” as implementações nas interfaces, evitando acoplar o código a alguma implementação específica.

## **Organização do código-fonte**

Bem, vimos resumidamente alguns padrões de codificação que certamente podem ajudar a criar código mais legível e de melhor manutenibilidade. Porém, estamos falando sobre sistemas corporativos, com muitos componentes e com diferentes participações na execução das funções.

É necessário organizarmos o código-fonte criando componentes coesos, de baixo acoplamento e separados em “camadas” lógicas da aplicação, com dependências controladas entre elas.

## **“Layers” (camadas)**

O conceito de “layers” ou camadas de software é muito antigo e se popularizou com o famoso modelo OSI (Open Systems Interconnection) da ISO. Este conceito separa os módulos de um software em camadas, cada uma com sua contribuição para a execução do software.

As camadas separam o código-fonte de forma lógica, podendo gerar pacotes físicos separados. Porém, não implicam necessariamente em separação física. Camadas podem compartilhar CPUs e até mesmo processos.

A ideia da separação é organizar o código-fonte de forma lógica, permitindo que pessoas com diferentes perfis e formações possam trabalhar em conjunto. Isto também possibilita a troca de implementações de camadas.

Para conceituar, as camadas separam o código-fonte de forma lógica. Em um ambiente ideal, seria possível distribuir as camadas por computadores separados, sem maiores esforços de programação. Para que isto seja possível, uma camada deve “conhecer” o mínimo possível sobre a outra, e toda a comunicação entre elas deve ocorrer de forma organizada e limitada. Quando isto acontece, dizemos que existe “baixo acoplamento” entre as camadas.

## **“Layers” e “tiers” (camadas e nós)**

Ainda existe alguma confusão entre estes dois conceitos, mas vamos tentar esclarecer. Camadas (ou “layers”) são divisões lógicas no código-fonte. “Tiers” (ou nós) são divisões físicas, sejam processos ou CPUs separadas.

Esta confusão é reforçada pela popularização dos termos “3-tier” e “n-tier”, que

levam à conclusão de que camadas (ou “layers”) e nós (ou “tiers”) seriam a mesma coisa.

Só para dar um exemplo de como esta confusão pode ser ruim, os modernos sistemas corporativos são tipicamente web, em três camadas:

- Apresentação: código de comunicação com o usuário;
- Lógica: código de lógica de negócios;
- Persistência: lógica de persistência de dados (por exemplo: bancos de dados).

Porém, na prática, os sistemas rodam, em sua maioria, em apenas dois nós, como a antiga arquitetura cliente-servidor. A implementação mais comum é termos uma camada Web instalada em um servidor de aplicação, juntamente com os Jars das classes de negócio, enquanto as tabelas residem no servidor de banco de dados.

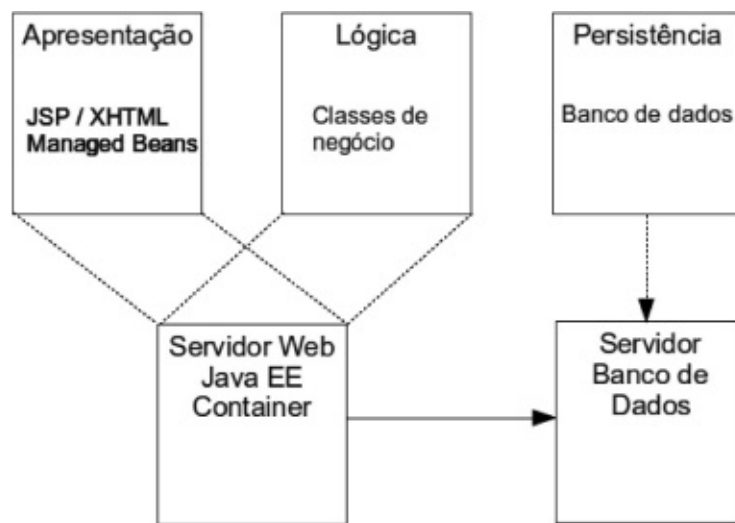


Ilustração 17: Web monolítica

Esta arquitetura é chamada de “web monolítica”. A contrapartida seria termos a lógica de negócios instalada em outro servidor de aplicação, com a comunicação ocorrendo via EJB (Enterprise JavaBeans).

A separação em “Tiers”, ou nós, implica em uma das alternativas:

- Mecanismos de IPC, caso os “tiers” sejam processos separados (em um Servidor ou em um “Cluster”);
- Protocolos de comunicação remota, caso os “tiers” rodem em máquinas separadas.

E, normalmente, a separação dos nós também implica em aumento de custo (instalação, infraestrutura, operação e manutenção).

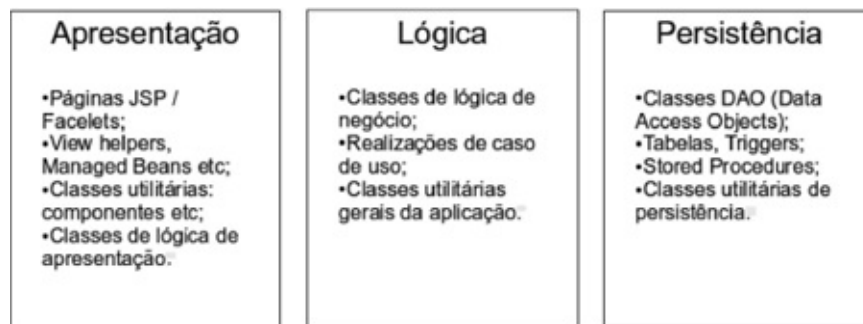
Portanto, a decisão de separar o sistema em nós não depende apenas do desenvolvedor, porém pode ser dificultada por ele. Se você desenvolver o sistema sem separar o código-fonte em camadas, será impossível distribuí-lo entre vários nós sem que seja feito um grande “refactoring”.

## Organização do código-fonte em camadas

Em um grande projeto, sempre devemos organizar nosso código-fonte em camadas. Estas camadas geralmente residem em pacotes de instalação separados. Temos um pacote para o Website, outro para as classes de “lógica de negócio” e um terceiro para a parte de persistência. Porém, nem sempre é assim, pois uma camada pode ser dividida em vários pacotes. Por exemplo, em uma aplicação Web Java temos:

- O arquivo WAR do site;
- Dentro dele, temos um ou mais arquivos JAR contendo componentes utilitários, “helpers” de apresentação e até mesmo classes de lógica de negócio e persistência.

O importante é entender o que deve estar em cada camada:





E como “empacotamos” isso tudo? Eis um arranjo típico:

- Arquivo WAR da aplicação Web;
- Arquivo(s) JAR compartilhados, instalados dentro de ../server/lib (ou equivalente), compartilhados por todos os aplicativos instalados em um container;
- Componentes utilitários do projeto (drivers JDBC, componentes JSF etc.), normalmente distribuídos dentro de WEB-INF/lib (o Maven faz isto para nós);
- Managed Beans, classes de lógica de negócio e classes DAO: dentro de um JAR que é inserido no WAR do projeto pelo Maven;
- Stored procedures, tabelas e triggers: dentro do banco de dados.

Se quisermos separar em nós diferentes, por exemplo, colocar a camada de lógica de negócios em outro servidor de aplicação, temos que “empacotar” as classes de lógica como componentes remotos, neste caso: Enterprise JavaBeans. E temos que providenciar classes para facilitar o acesso, como Service Locators e Business Delegates (<http://java.sun.com/blueprints/corej2eepatterns/Patterns/>) (ALUR et al), que deverão ser empacotados juntamente com as classes usuárias, da camada de apresentação.

### **Porém, como fica isso tudo dentro de um projeto?**

Aí é que entram em campo o Maven e o Archiva, que discutimos anteriormente no livro. Com eles podemos criar projetos separados para cada componente distinto, organizando e promovendo o reuso de código na empresa. Só devemos criar dentro do projeto os componentes de reuso interno ao sistema. Todos os outros componentes devem ser projetos Maven à parte e devem ser obtidos do repositório Archiva, conforme mencionei anteriormente.

Mesmo que vários componentes estejam sendo desenvolvidos simultaneamente, é possível utilizar as versões “Snapshot” diretamente do Archiva. Sempre que possível, você deve evitar criar referências a componentes da mesma workspace no seu “build path”. Deixe o Maven baixar os componentes necessários, mesmo que estejam dentro da workspace, pois assim você exercita o fluxo de integração

do projeto (Maven / Archiva), garantindo que está utilizando a versão “oficial” e não a que está em sua máquina.

## **Pacotes lógicos**

Ao criar projetos separados para componentes distintos, já demos um grande passo para organizar o código-fonte. Outro passo é organizar o código-fonte em pacotes “lógicos” dentro de cada componente. As classes que ficam dentro da pasta “src/main/java” precisam ser organizadas em pacotes, o que você já deve saber muito bem. E é claro que você já sabe que o pacote deve iniciar com o nome de domínio da empresa ao contrário, com todas as letras em minúsculas:

`com.galaticempire.`

A seguir, deve vir o nome do projeto. Opcionalmente, pode-se precedê-lo com o nome do programa ou da organização dentro da empresa, por exemplo:

`com.galaticempire.guia.projetoteste.`

O nome do pacote deve terminar com o nome do seu “estereótipo”, ou seja, o tipo de elementos que ele contém.

No arquétipo que criamos, há uma sugestão de organização em “estereótipos”:

- “entidades”: classes que representam as entidades com as quais o sistema lidará. Se utilizarmos um “modelo anêmico”, com o Hibernate, por exemplo, então nossas classes serão DTOs e entidades do banco;
- “managedbean”: classes “helpers” do framework JSF;
- “negocio”: classes de lógica de negócio, realização de casos de uso etc;
- “persistencia”: classes DAO e utilitários para acesso a dados.

O Maven, orientado pelo Arquétipo, se encarrega de montar a estrutura de pacotes lógicos. Por exemplo:

- `com.galaticempire.guia.projetoteste.entidades;`
- `com.galaticempire.guia.projetoteste.managedbean;`
- `com.galaticempire.guia.projetoteste.negocio;`

- `com.galaticempire.guia.projetoteste.persistencia`.

É claro que você pode imaginar modelos de organização mais detalhados ou complexos. Uma boa ideia é separar as interfaces das implementações. Porém, como eu queria simplificar o projeto de exemplo, não fiz isto.

Você deve sempre separar componentes de reuso mais geral em projetos separados, gerando componentes físicos distintos (dentro do Archiva). Os componentes específicos, de uso interno ao sistema, devem ser criados em pacotes lógicos separados, de acordo com o “estereótipo” dos elementos que eles contêm. Estes pacotes lógicos ajudam a separar as camadas da aplicação.

## Dependência entre camadas e pacotes

Como já dissemos, as classes são organizadas em pacotes “lógicos”. Dependência em UML é quando uma classe ou pacote utiliza elementos que estão em outra classe ou pacote. Entre pacotes é representada assim:



Ilustração 19: Dependência entre pacotes

Se um pacote lógico depende de outro pacote lógico, e ambos estiverem em pacotes físicos separados, haverá dependência entre os pacotes físicos também. Mas, neste caso, estamos nos referindo aos pacotes lógicos da aplicação (onde as camadas são separadas).

Quando um pacote depende de outro, alterações nas classes do segundo pacote podem provocar alterações nas classes do primeiro. Isto é chamado de “propagação” de alterações. O objetivo de controlar a dependência entre as camadas (e pacotes) é exatamente evitar ou minimizar esta “propagação”, aumentando a manutenibilidade do software.

Para controlar a propagação de alterações, devemos separar nosso código-fonte de maneira que as dependências sigam regras claras:

1. Não pode haver dependência “cíclica” entre pacotes;
2. Um pacote só pode depender dos pacotes que lhe prestam serviços diretamente;
3. Controle o “tráfego” de classes entre pacotes.

## Dependência

Um pacote depende de outro porque uma (ou mais) de suas classes utilizam elementos que estão definidos no outro pacote.

É fácil entender dependências do tipo composição ou agregação:

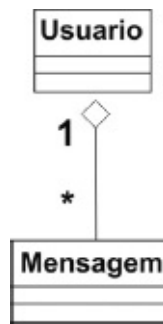


Ilustração 20: Agregação

Mas dependências comuns são mais difíceis de perceber e detectar. Exemplo:

```
package com.galaticempire.guia.projetoteste.managedbean;
```

```
public class ClienteBean {  
    ...  
    public String acaoXpto() {  
        ...  
        Usuario usuario = new Usuario();  
    }  
}
```

```
-----  
package com.galaticempire.guia.projetoteste.entidades;
```

```
public class Usuario {  
    ...  
}
```

Neste exemplo, criamos uma dependência entre o pacote “.managedbean” e o pacote “.entidades”, porque a classe “ClienteBean” está instanciando um objeto de um tipo definido no pacote “.entidades”.

Este tipo de dependência, conhecido como dependência “de uso” (ou “comum”), pode ser mais sutil ainda quando passamos instâncias como argumentos de métodos:

```
package com.galaticempire.guia.projetoteste.negocio;  
public class RegrasDeCalculo {  
    ...  
    public int obterTipoDeTransacao (HttpServletRequest request,  
                                     Cliente cliente) {  
        ...  
    }  
}
```

Esta classe recebe uma instância de `HttpServletRequest` (pacote “`javax.servlet.http`”), logo, criou-se uma dependência entre o pacote “`com.galaticempire.guia.projetoteste.negocio`” e o pacote “`javax.servlet.http`”, e ambos os pacotes físicos (o da aplicação e o “`servlet.jar`”). Imagine o enorme problema caso outra aplicação necessite utilizar a classe “`RegrasDeCalculo`”... teremos que empacotar tudo junto, incluindo o “`servlet.jar`” e todos os outros pacotes físicos, dos quais ele dependa.

E evite associações (composição ou agregação) entre classes de pacotes diferentes.

## Dependência cíclica

É um problema que ocorre quando existe um ciclo de dependências. Por exemplo:

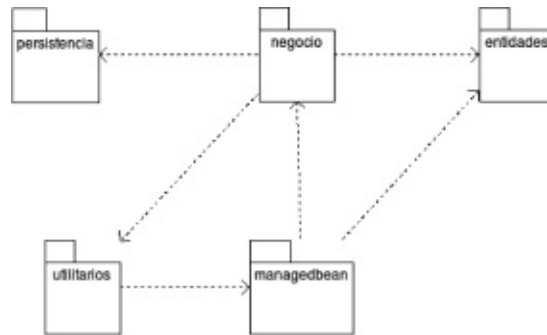


Ilustração 21: Dependência cíclica

O pacote “managedbean” depende do pacote “negocio”, que, por sua vez, depende do pacote “utilitarios”, que depende do pacote “managedbean”. Este é um exemplo típico de dependência cíclica. Isto é um problema porque cria vínculos indesejáveis e, conseqüentemente, propagação de alterações.

Evite dependência cíclica analisando bem o seu modelo de objetos.

## Dependência de serviço

Um pacote somente pode depender de pacotes que lhe prestam serviço diretamente. Não pode haver dependência “transitiva”: o pacote “x” depende do pacote “y”, que depende do pacote “t”; logo, o pacote “x” depende do pacote “t”.

Se um pacote presta serviços a outro, deve ter o menor acoplamento possível (veremos mais adiante), evitando a criação de dependências transitivas, provocadas pelo retorno de objetos cujos tipos estão definidos dentro dele ou de outros pacotes.

Devemos evitar “dependências transitivas” entre pacotes.

## Controle de “tráfego”

Nem sempre podemos contar apenas com tipos primitivos ou classes da API Java e quase sempre devemos utilizar objetos complexos, seja como argumento

ou retorno de métodos.

Dizemos que uma classe “trafega” entre camadas quando suas instâncias são utilizadas, como argumento e/ou retorno de métodos, em outros pacotes de outras camadas. Isto pode acontecer também devido à presença de instâncias da classe em objetos de contexto, como: sessões Http, Singletons etc.

Sempre que uma classe é utilizada em mais de um pacote, criamos uma possível fonte de alterações para ele. Para que possamos minimizar isto, devemos seguir os princípios de “segregação de interfaces” e “inversão de dependência”, diminuindo a interface da classe e fazendo com que as implementações dependam de abstrações.

Um bom exemplo é a dependência entre classes de apresentação (como “Managed Beans”) e classes de negócio. Devemos criar uma (ou mais) interface específica contendo apenas os métodos necessários e com os argumentos e valores de retorno aceitáveis, de modo a evitar transitividade ou dependência cíclica entre as camadas.

E outro exemplo é exatamente o transporte de dados. Por exemplo, como persistir os dados de um novo usuário? Eis a “viagem” dos dados:

1. A classe “Managed Bean” recebe os dados da interface e monta uma instância de “Usuário”;
2. A instância é passada para a camada de negócio, que vai verificar as regras para adição de um novo usuário, invocando as classes de persistência para isto e passando a instância de “Usuário” para ela;
3. A camada de persistência cuida das regras para persistir o novo “Usuário”.

Assim, temos que controlar muito bem quais classes podem “trafegar” entre pacotes de camadas diferentes. Normalmente, apenas as classes DTO fazem isto. Eis algumas regras simples:

1. Crie classes para transportar dados em sua aplicação, com propriedades simples e sem implementar interfaces específicas (“POJO” – Plain Old Java Object). Somente estas classes podem ser utilizadas na comunicação entre camadas diferentes;

2. Quando existe a relação de “serviço”, a classe “cliente” deve conhecer apenas uma interface da classe “prestadora”, evitando ficar presa à implementação. Crie interfaces distintas para clientes distintos;
3. Evite dependências transitivas. Evite expor (ou “encapsular”) tipos que foram definidos fora do pacote. Por exemplo: jamais devolva instâncias de “java.sql.ResultSet” ou “[javax.servlet.http.HttpServletRequest](#)”.

## **Coesão e acoplamento**

Meilir Page-Jones (Page-Jones) foi um dos que ajudaram a popularizar os conceitos de “acoplamento” e “coesão” para programação estruturada.

Posteriormente, outros autores, como Chidamber e Kemerer (CHIDAMBER & KEMERER), criaram outros critérios para medir acoplamento e coesão entre classes.

Devemos sempre criar componentes de baixo acoplamento e alta coesão interna, melhorando a manutenibilidade e robustez do software que vamos produzir. Para isto, as classes, os métodos e os pacotes devem seguir alguns conceitos básicos.

### **Projeto modular**

Page-Jones (Page-Jones) defende a separação do código-fonte em módulos estanques, com ligações simples e bem documentadas entre si. Para isto, ele classifica os módulos de acordo com dois princípios básicos: acoplamento e coesão.

Se você organizar seu código-fonte procurando criar módulos altamente coesos e de baixo acoplamento, certamente aumentará a sua manutenibilidade, que é um dos atributos de qualidade de projetos de software, conforme já mencionamos. Mas não é só isso! Também melhorará a sua testabilidade, já que o código modular permite criar testes mais precisos.

### **Acoplamento**

Segundo a Wikipédia (<http://pt.wikipedia.org/wiki/Acoplamento>):

Acoplamento é o nível de interdependência entre os módulos de um pro-grama



de computador. Quanto maior for o acoplamento menor será o nível de coesão.

Um módulo é um grupo, conjunto de instruções ou sub-rotina, que possui algumas propriedades determinadas, como: função (o que ela faz), entrada (quais são os parâmetros de entrada) e saída (quais são os parâmetros de saída). Podemos pensar em módulos como métodos.

A ligação entre módulos pode ser forte ou fraca, tendo alguns graus de variação entre os limites. Quanto mais forte for a ligação, maior a dependência entre eles, logo, manutenção em um dos módulos poderá exigir manutenção no outro. Chamamos de acoplamento o grau de dependência entre dois módulos. Módulos de baixo acoplamento são ideais e reduzem a “propagação de alterações”, aumentando a manutenibilidade e a testabilidade do código-fonte.

Na definição original, existem cinco tipos de acoplamento, variando do mais fraco para o mais forte:

1. **Acoplamento de dados (melhor):** ocorre quando dois módulos se comunicam através de uma interface bem definida, estável e utilizando apenas os dados necessários para que o módulo chamado execute sua função;
2. **Acoplamento de imagem:** ocorre quando um ou mais módulos recebem como parâmetro uma estrutura de dados ou uma instância de classe. Eles ficam “interligados” por esta instância;
3. **Acoplamento de controle:** acontece quando dois ou mais módulos passam ou recebem parâmetros de controle (indicadores) que modificam o seu funcionamento. Neste caso, dizemos que um módulo está controlando o outro;
4. **Acoplamento comum:** ocorre quando um ou mais métodos se referem à mesma área comum de dados. A diferença para o acoplamento de imagem é que os dados não estão sendo passados como parâmetro, mas são variáveis globais;
5. **Acoplamento de conteúdo (péssimo):** acontece quando dois módulos compartilham código-fonte; por exemplo, um deles desvia o processamento para dentro do código do outro.

## Coesão

De acordo com a Wikipédia (<http://pt.wikipedia.org/wiki/Coes%C3%A3o>):

Coesão é a medida da força relativa de um módulo. Quanto maior for a coesão, menor será o nível de acoplamento de um módulo.

Tenho que fazer uma observação importante: quando se trata de classes, existem outros conceitos para análise de acoplamento e coesão; embora os conceitos clássicos possam ser aplicados com bons resultados, podem não ser suficientes.

Todo módulo possui uma sequência de operações internas. A coesão de um módulo é dada pela força com que estas operações estão unidas e colaboram para executar a sua função. Módulos com alta coesão são mais robustos, mais fáceis de entender e menos sujeitos à propagação de alterações, pois são mais comprometidos com o princípio da responsabilidade única.

Além disto, módulos com alta coesão tendem a apresentar baixo acoplamento com outros módulos, e módulos com baixa coesão tendem a apresentar alto acoplamento. Temos alguns tipos bem definidos de coesão, variando da mais alta para a mais baixa:

1. **Funcional (melhor):** as partes de um módulo estão juntas porque contribuem para a execução de sua única atividade;
2. **Sequencial:** as partes estão agrupadas porque a saída de uma operação é a entrada de outra, em uma ordem sequencial; logo, devem ser executadas dentro do mesmo módulo;
3. **Comunicacional:** quando as partes estão juntas porque operam sobre os mesmos dados;
4. **Procedural:** quando as partes estão juntas porque devem ser executadas em uma certa ordem, pois formam um procedimento;
5. **Temporal:** ocorre quando as partes estão juntas apenas porque devem ser executadas no mesmo momento;
6. **Lógica:** quando as partes estão juntas porque executam tarefas do mesmo grupo lógico, embora sejam completamente diferentes;

7. **Coincidental (péssima)**: quando as partes estão juntas sem nenhum motivo especial.

### **Coesão de classes**

Para criarmos um projeto de boa qualidade, devemos criar classes altamente coesas e de baixo acoplamento. Classes coesas têm uma única responsabilidade e executam apenas ações relacionadas com esta. Isto posto, podemos “sentir” se uma classe é ou não coesa. Porém, a subjetividade sempre leva a diferentes percepções, logo, é necessário criar critérios e métricas mais objetivas para determinação do nível de coesão de classes.

Com o passar do tempo, vários autores procuraram expandir os conceitos originais de acoplamento e coesão, de modo a incluir as Classes.

### **Lack of Cohesion Of Methods - LCOM1 (CHIDAMBER & KEMERER)**

#### **(Falta de coesão dos métodos)**

Esta métrica analisa o número de pares de métodos de uma classe cuja similaridade é zero. Quanto mais pares, menor a coesão da classe. Valores pequenos significam que a classe é altamente coesa, porém valores altos significam que ela está assumindo mais de uma responsabilidade, logo, deveria ser fatorada em classes diferentes.

Para calcular LCOM1:

1. Pegue os métodos de uma classe e analise par-a-par;
2. Se eles acessam conjuntos diferentes de variáveis de instância, incremente o contador “P”;
3. Se eles compartilham pelo menos uma variável de instância em comum, então incremente o contador “Q”;
4. Se “P” > “Q”, então calcule LCOM1 como “P – Q”; caso contrário, atribua zero.

Como foi uma das primeiras medidas de coesão de classes, a LCOM1 sofreu muitas críticas, mas funciona bem para muitos casos.

## LCOM2 (HENDERSON-Sellers)

Este cálculo leva em consideração a quantidade de atributos e métodos da classe. Para calcular LCOM2:

1. Calcule a quantidade de métodos da classe “m”;
2. Calcule a quantidade de atributos da classe “a”;
3. Calcule quantos métodos acessam cada atributo “mA”, para cada atributo da classe;
4. Some os valores de “mA” obtidos no passo anterior;
5. Calcule  $LCOM2 = 1 - \text{soma}(mA) / (m * a)$ .

Se o número de métodos ou de atributos é zero, então a medida, por convenção, é zero. Medidas pequenas são indicativo de classes altamente coesas.

## LCOM3

Este cálculo é um melhoramento do LCOM2 e é mais utilizado. Calcule as variáveis “m”, “a”, “mA” e “soma(mA)”, conforme mostrado em LCOM2. Para calcular LCOM3:

$$\blacksquare LCOM3 = (m - \text{soma}(mA) / a) / (m - 1).$$

Esta medida varia entre alta coesão (zero) e coesão extremamente baixa (maior que 1). Se a classe tem somente um método ou não tem atributos, o valor de LCOM3 é indefinido e zero.

## Um exemplo vale mais que mil palavras

Vamos supor um sistema de controle de empréstimos que tenha uma classe como esta:

```
package com.galaticempire.guia.emprestimo;

import java.util.Date;

public class Emprestimo {
    public Cliente cliente;
    public double montante;
    public Date dataContratacao;
```

```

    public Date dataPagamento;
    public double taxaJuros;
    public int parcelas;
    public Date dataQuitacao;

    public Emprestimo(Cliente cliente, double montante, Date
dataContratacao,
                        Date dataPagamento, double taxaJuros) {
        super();
        this.cliente = cliente;
        this.montante = montante;
        this.dataContratacao = dataContratacao;
        this.dataPagamento = dataPagamento;
        this.taxaJuros = taxaJuros;
    }

    public boolean vencido () {
        boolean vencido = false;
        Date dataHoje = new Date();
        if (this.dataPagamento.compareTo(dataHoje) < 0) {
            vencido = true;
        }
        return vencido;
    }

    public double calcularTaxa (Cliente mcliente, double mmontante,
                                int mprazo) {
        double taxa = 0.0d;
        TaxaDAO taxaDao = new TaxaDAO();
        taxa = taxaDao.obterTaxa();
        switch (mcliente.getSitCredito()) {
            case Cliente.CREDITO_OK:
                break;
            case Cliente.CREDITO_EM_RECUPERACAO:
                taxa *= taxaDao.getPercRec();
                break;
            default:
                taxa *= taxaDao.getPercDesconhecido();
        }
        return taxa;
    }
}

```

Por favor, releve os detalhes neste momento. Vamos calcular LCOM3:

- “a”: número de atributos = 7;
- “m”: número de métodos (incluindo o construtor) = 3;

- `m[cliente] = 1;`
- `m[montante] = 1;`
- `m[dataContratacao] = 1;`
- `m[dataPagamento] = 2;`
- `m[taxaJuros] = 1;`
- `m[parcelas] = 0;`
- `m[dataQuitacao] = 0;`
- `soma(mA) = 6.`

$$\begin{aligned}
 \text{LCOM3} &= (m - \text{soma}(mA) / a) / (m - 1) = \\
 &= (3 - 6 / 7) / (3 - 1) = \\
 &= (3 - 0,857142857) / 2 = \\
 &= 2,142857143 / 2 = \mathbf{1,071428571}
 \end{aligned}$$

A classe tem coesão extremamente baixa, de acordo com a métrica – logo, deveria ser candidata a refactoring. Vamos tentar melhorar a coesão desta classe. Primeiramente, há dois atributos que não são utilizados pelos métodos da classe:

- “parcelas”;
- “dataQuitação”.

Há duas maneiras de lidar com isto: ou são variáveis “mortas” que foram esquecidas, ou então não estão encapsuladas. Se passarmos a receber seu valor no construtor, para sermos coerentes com os outros atributos, o LCOM3 passa a ser aproximadamente 0,93.

Está melhor, porém devemos fazer este valor cair o máximo possível. A classe tem um método “calcularTaxa” que simplesmente não utiliza atributo algum. Poderia muito bem estar em outra classe, mais apropriada. Se removermos este método, o LCOM3 passa a ser 0,86 (aproximadamente).

Se encapsularmos todos os atributos melhoramos muito pouco o LCOM3. Isto é

porque esta é uma classe praticamente “anêmica” (muitos atributos e poucos métodos), o que pode ser normal em nosso sistema. Em classes com mais comportamento, certamente podemos conseguir melhorar ainda mais a coesão.

Porém, com algumas medidas simples, caímos de 1,071 para 0,86, aumentando a coesão da classe e melhorando a qualidade geral do código-fonte.

### **Acoplamento entre classes**

Se nosso objetivo é aumentar a coesão interna das classes, indiretamente queremos diminuir o seu acoplamento, ou a dependência que uma classe tem da outra. Com isto, diminuimos também o acoplamento entre pacotes e componentes criando um sistema mais flexível e adaptável.

### **CBO (Coupling Between Object classes) (CHIDAMBER & KEMERER)**

Mede o acoplamento através do número de classes com as quais uma classe está acoplada. Contamos cada classe apenas uma vez, não importando quantas vezes nós a referenciamos. Alguns autores consideram  $CBO > 14$  como um valor alto, impedindo o reuso da classe por excesso de dependências.

### **Complexidade**

Devemos sempre lutar contra a complexidade, que afeta diretamente as qualidades de um projeto de várias maneiras:

- O software é mais difícil de manter;
- O software é menos flexível;
- O software é mais difícil de testar.

É lógico que um software com estas características corre um risco muito alto de possuir “bugs”; logo, sua qualidade é muito baixa.

Pressman (PRESSMAN) define muito bem a questão da complexidade quando fala sobre testes. Ele explica o conceito de Complexidade Ciclomática, na qual enxergamos o código como um grafo, contando o número de caminhos básicos para criação de testes.

Outras métricas para complexidade, propostas por [CHI 94], são:

- “Weighted Methods Per Class (WMC)” (“Métodos ponderados por classe”): mede a complexidade de uma classe pela complexidade dos seus métodos;
- “Depth of Inheritance Tree (DIT)” (“Profundidade da árvore de herança”): mede a profundidade de uma classe dentro de uma estrutura hierárquica. Quanto maior, mais difícil de manter;
- “Number of children (NOC)” (“Número de filhos”): mede a quantidade de classes descendentes que uma classe possui. Quanto maior, maior é a influência desta classe sobre o sistema;
- “Response For a Class (RFC)” (“Resposta para uma classe”): é a quantidade de métodos que podem ser executados em uma classe, em resposta a uma solicitação (invocação de um método). Quanto maior, mais difícil será de manter e testar a classe.

## Controle automático de métricas

Já vimos que existem diversos critérios para organização do código-fonte, muitos deles difíceis de praticar manualmente. Porém, há uma maneira simples e prática de administrar seu código enquanto está sendo construído: usar um plugin de métricas no Eclipse. Eu costumo usar o “Eclipse Metrics” (<http://www.stateofflow.com/projects/16/eclipsemetrics>). Ele pode ser baixado do “update site”: <http://www.stateofflow.com/UpdateSite> e é muito interessante, pois marca no seu código o que considera incorreto.



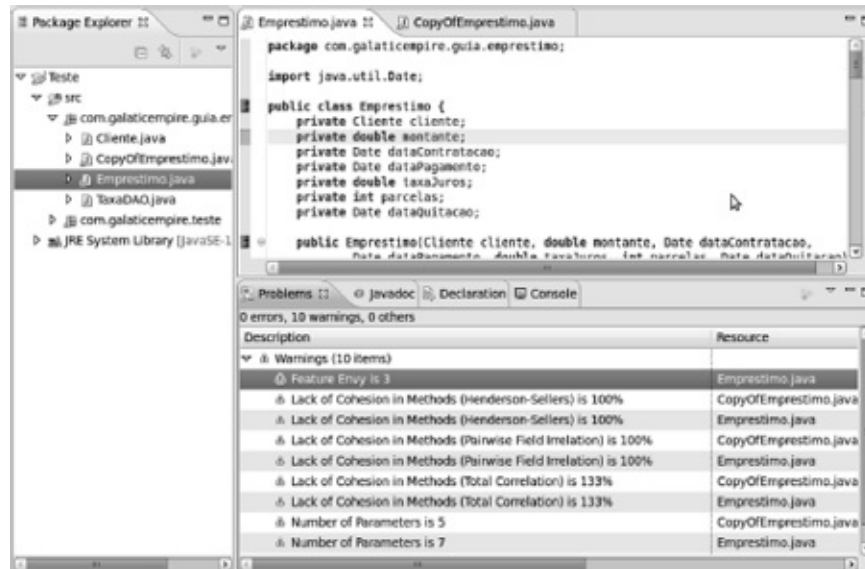


Ilustração 22: Eclipse metrics

Além de marcar cada linha e cada parte onde existem observações, ele informa quais foram as mensagens geradas, conforme pode observar na Ilustração 22. Além disto, podemos exportar um relatório em HTML completo, bastando selecionar o projeto, clicar com o botão direito do mouse e selecionar “Export / Other / Metrics”.

Eis um exemplo do relatório criado pelo plugin:

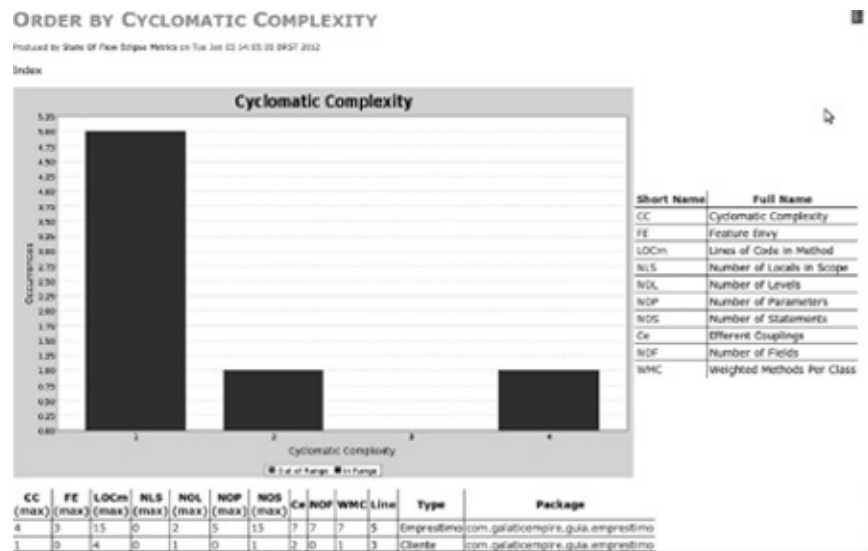


Ilustração 23: Relatório do plugin Eclipse Metrics

Embora ele não utilize todas as métricas mais conhecidas, é bastante útil e pode ajudar a controlar o acoplamento, a coesão e a complexidade do seu projeto enquanto você o programa! Ele mesmo fornece as definições das métricas que utiliza, o que torna muito fácil aprender a usar o plugin e analisar seu código-fonte.

Para resumir, as principais métricas que ele utiliza são:

- “Feature Envy” (inveja de funcionalidades): se um método está utilizando mais características de outras classes do que daquela em que está definido. Um nível maior que 1 deve ser analisado;
- “Number of Locals in Scope” (número de variáveis locais no escopo): quantas variáveis locais estão em escopo em determinado ponto do método. Quanto mais variáveis, mais difícil será de manter o método;
- “Number of Levels” (número de níveis): quantos níveis existem dentro de um método. Considere cada bloco de código como um nível. Vários “if” aninhados aumentam a complexidade do método;
- “Efferent couplings” (acoplamentos eferentes): a quantidade de classes que a classe sendo analisada “conhece”, seja através de herança, implementação, parâmetros, variáveis etc. É semelhante ao CBO (Coupling Between Objects).

Ele também utiliza “Weighted Methods Per Class” e “Cyclomatic Complexity”.

## Testes unitários

*...program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.*

Edsger W. Dijkstra, palestra do prêmio Turing, da ACM, de 1972 (“The Humble Programmer”)

(“...o teste de programas pode ser um meio muito eficaz para mostrar a presença de bugs, mas é inadequado para mostrar sua ausência.”)

*If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.*

Edsger W. Dijkstra, palestra do prêmio Turing, da ACM, de

(“Se quiser programadores mais eficazes, descobrirá que eles não deveriam perder tempo depurando, mas deveriam evitar introduzir bugs, para começar.”)

Realmente, o professor Dijkstra estava inspirado quando escreveu sua palestra “The Humble Programmer” (“O humilde programador”). Estas duas citações mostram bem como é a tríplice relação qualidade-testes-técnicas. A qualidade de um software está relacionada às técnicas aplicadas na sua construção e à efetividade dos testes realizados. Porém, quanto mais cuidado temos na programação, menos bugs introduziremos – logo, menos dependeremos dos testes.

Na verdade, é muito difícil termos um software à prova de bugs. Até mesmo o professor Knuth ([http://en.wikipedia.org/wiki/Knuth\\_reward\\_check](http://en.wikipedia.org/wiki/Knuth_reward_check)), um dos maiores mestres da ciência da computação, tinha bugs em seus programas e livros. Ele oferecia cheques para quem encontrasse bugs em seus trabalhos.

Testes unitários devem ser automatizados e executados continuamente. Uma boa maneira é utilizar o framework JUnit para criar seus casos de teste.

## **Evite introduzir bugs, para começar**

Se você seguir os conselhos aqui apresentados (além das recomendações do tópico “A base teórica”), produzirá código-fonte claro, simples e bem organizado, o que já é um grande passo para evitar bugs. Lembre-se de mitigar os riscos e de programar o menos possível, assim você evita introduzir novos bugs no sistema. Mas isto não é suficiente.

A introdução de bugs muitas vezes é causada por atividades externas:

- Má gestão do “backlog”: quando o gerente de projeto quer atender a muitas pendências simultaneamente, agrupando diversas atividades de programação juntas. Isto aumenta a quantidade de código-fonte produzido e, conseqüentemente, de bugs também;
- Falta de revisões: promova sempre revisões de código (falaremos mais adiante), para deixar claro o que está sendo feito e se está de acordo com os padrões e especificações;
- Requisitos mal levantados: geram dúvidas, problemas e retrabalho

frequentes;

- Estimativas mal feitas: geram pressão na implementação.

O professor Dijkstra dá uma dica de como isso deve ser feito: “the programmer should let correctness proof and program grow hand in hand” (The Humble Programmer). Traduzindo livremente, significa que o programa e sua prova de correção devem ser desenvolvidos juntos, o que é um dos fundamentos da técnica Test Driven Development, desenvolvida por Kent Beck (BECK).

A melhor forma de evitar introduzir bugs é seguir as boas práticas que já mencionei, além de implementar testes unitários detalhados e executá-los frequentemente. Os testes devem ser automatizados, de modo que não seja necessário lembrar de executá-los.

Além disto, os testes são uma parte integrante do produto de software e devem ser mantidos em conjunto com a funcionalidade básica do produto.

## **Testes unitários devem ser limitados**

De acordo com o SWEBOK (IEEE, 2004):

*Unit testing verifies the functioning in isolation of software pieces which are separately testable.*

(“O teste unitário verifica o funcionamento de pedaços do software isoladamente, que são testados separadamente.” – Tradução do autor).

Existe uma abordagem moderna, chamada TDD – Test Driven Development (Desenvolvimento orientado a testes), descrita por Kent Beck em seu livro “Test Driven Development: By Example” (BECK). Resumidamente, a técnica preconiza que, ao receber um requisito, o desenvolvedor deve escrever um caso de teste para ele. Depois, executa o teste (que deverá falhar) e escreve o código-fonte necessário para que o software passe neste novo teste. Depois que o teste passar, ele pode refatorar (ou limpar) o código-fonte, reexecutando os testes.

A maioria dos programadores modernos faz isto, só que ao contrário! Primeiro escrevem o código-fonte da funcionalidade e depois escrevem o caso de teste. O problema com essa abordagem é que, muitas vezes, o teste é incompleto (ou até mesmo ignorado).

Outro “pecado” é testar mais do que deveria. Kent Beck tem uma frase interessante: “Fake it until you make it” (<http://c2.com/cgi/wiki?FakeIt>) ou “falsifique uma funcionalidade até que você possa implementá-la”. De acordo com o TDD, a prioridade é passar no teste, logo, podem existir pedaços do código-fonte que ainda não estejam prontos. A ideia é “falsificar” estas partes para que o teste passe sem problemas. Depois, você implementa a verdadeira funcionalidade.

É comum vermos confusão entre casos de teste unitários e de integração. O próprio aplicativo que serve de exemplo para este livro comete este “pecado”. Um exemplo é criar um caso de teste para uma classe de negócios que também testa o DAO, ou seja, a classe instancia e executa o próprio DAO. Neste caso estão sendo testados: a classe de negócio, o banco de dados e o DAO. É muito difícil criar testes eficazes para a classe de negócio, quando na verdade estamos testando vários itens simultaneamente.

Cada unidade construída deve ter seus próprios testes unitários, e, idealmente, eles devem testar APENAS o código-fonte daquela unidade. Nada mais.

Você somente pode incluir em seu teste unitário os componentes que já estão testados e aprovados – por exemplo, os que já estão no repositório “internal” do Archiva. Incluir componentes que ainda estão em desenvolvimento (“snapshots”) é aumentar o escopo do teste. É claro que nem sempre é possível evitar, mas, se for absolutamente necessário, procure incluir os que estão no repositório “snapshots”, que devem ter passado pelo menos por um teste de integração.

Porém, o que é uma “unidade”? A definição mais simples é: a classe que você está desenvolvendo agora! É claro que pode ser um grupo de classes, mas, de qualquer forma, está tudo sob responsabilidade de um só desenvolvedor. Você deve escrever um caso de teste para os métodos dessa classe tomando cuidado especial com aqueles com maior complexidade (> 3), ou os mais sensíveis.

Alguns autores entendem “unidade” como a unidade de trabalho que está sendo desenvolvida por uma pessoa (ou duas: “pair programming”). Se você está desenvolvendo um caso de uso com umas cinco classes, então poderia considerá-las todas como pertencentes à mesma unidade. Eu prefiro ser mais radical e considerar cada classe uma unidade. Só não escrevo casos de teste para classes

“anêmicas”, embora faça um teste geral do modelo de dados.

## Fingindo

Como eu disse anteriormente, muitos desenvolvedores integram prematuramente suas unidades, pois alegam que precisam testá-las em conjunto com outras, feitas por outros desenvolvedores; logo, fazem “commit” sem testar, ou pior: pegam arquivos “JAR”, ou mesmo código-fonte das cópias de trabalho de outros desenvolvedores, para fazer testes. Isto é negligenciar o teste unitário!

Se você depende de comportamentos a serem providos por outras classes, por que não “finge”? Se você desenvolveu corretamente, deve haver interfaces para cada “comportamento” do sistema, certo? Além disto, estes comportamentos são referenciados pelas interfaces e nunca pelas classes concretas (princípio da inversão de controle). Então, podemos “fingir” os comportamentos dos quais dependemos criando testes unitários que realmente exercitem o código-fonte da nossa unidade de trabalho.

Um teste unitário deve “fingir” o acesso a outros componentes que não tenham sido desenvolvidos. Isto significa que você deve utilizar “method stubs” e “mock objects” para simular o acesso aos componentes verdadeiros.

Por exemplo, suponha que estamos desenvolvendo uma classe que contenha o método a seguir:

```
public double calcularTaxa (Cliente mcliente, double mmontante,
                           int mprazo) {

    double taxa = 0.0d;
    taxa = obterTaxa(mcliente, mmontante, mprazo);
    switch (mcliente.getSitCredito()) {
    case Cliente.CREDITO_OK:
        break;
    case Cliente.CREDITO_EM_RECUPERACAO:
        taxa *= taxaDao.getPercRec();
        break;
    default:
        taxa *= taxaDao.getPercDesconhecido();
    }
    return taxa;
}
```

Queremos testar o nosso método e ainda não sabemos como a taxa vai ser

obtida, então criamos o “method stub” “**obterTaxa**” para simular isto:

```
/* (non-javadoc)
 *
 */
private double obterTaxa(cliente mcliente,
                        double mmontante,
                        int mprazo) {

    return 0.05d;
}
```

Porém, quando se trata de utilizar classes “snapshot”, temos que usar outra solução: “mock objects”. Por exemplo, suponha que estamos desenvolvendo este método:

```
public double calcularTaxa (Cliente mcliente, double mmontante,
                           int mprazo) {

    double taxa = 0.0d;
    TaxaDAO taxaDAO = new TaxaDAO();
    taxa = taxaDAO.obterTaxa(mcliente, mmontante, mprazo);
    switch (mcliente.getSitCredito()) {
    case Cliente.CREDITO_OK:
        break;
    case Cliente.CREDITO_EM_RECUPERACAO:
        taxa *= taxaDao.getPercRec();
        break;
    default:
        taxa *= taxaDao.getPercDesconhecido();
    }
    return taxa;
}
```

Agora precisamos instanciar e acessar um método da classe “TesteDAO” que pertence a outra unidade. Podemos criar um “mock” desta classe contendo o método abaixo:

```
public class TesteDAO {
    public double obterTaxa (Cliente, cliente, double montante, int prazo) {

        return 15.0d;
    }
}
```

Outra maneira mais simples é utilizar um software de simulação, como o jMock (<http://www.jmock.org>). Com ele podemos simular instâncias de objetos dentro de casos de teste, podendo, inclusive, testar como foi a comunicação com eles.

Em primeiro lugar, o jMock simula classes concretas a partir de interfaces, o que é mais recomendado (lembre-se do princípio de inversão de dependências). Logo, transformamos “TesteDAO” em uma interface. Agora vamos alterar nossa classe para utilizar uma variável de instância com getter e setter:

```
public class Emprestimo {
    ...

    private TaxaDAO taxaDAO;
    ...

    public TaxaDAO getTaxaDAO() {
        return taxaDAO;
    }

    ...

    public void setTaxaDAO(TaxaDAO taxaDAO) {
        this.taxaDAO = taxaDAO;
    }

    ...

    public double calcularTaxa (Cliente mcliente, double mmontante, int
mprazo) {

        double taxa = 0.0d;
        taxa = taxaDAO.obterTaxa(mcliente, taxa, mprazo);
        switch (mcliente.getSitCredito()) {
        case Cliente.CREDITO_OK:
            break;
        case Cliente.CREDITO_EM_RECUPERACAO:
            taxa *= taxaDAO.getPercRec();
            break;
        default:
            taxa *= taxaDAO.getPercDesconhecido();
        }
        return taxa;
    }
}
```

Ao criar uma variável de instância para a classe externa, melhoramos a manutenibilidade do código-fonte, pois evidenciamos os componentes externos. Além disso, preparamos o código-fonte para utilizar injeção de dependências. Finalmente, vamos criar um JUnit Test Case usando o jMock para simular o comportamento da interface “TaxaDAO”:

```
package com.galaticempire.guia.emprestimo;
```



```
import static org.junit.Assert.*;
import org.junit.Test;
import org.jmock.Mockery;
import org.jmock.Expectations;
```

```
public class TesteUnitarioEmprestimo {

    @Test
    public void testCalcularTaxa() {

        Mockery contexto = new Mockery();
        final TaxaDAO dao = contexto.mock(TaxaDAO.class);
        final Cliente cliente = new Cliente();
        final double montante = 100.0d;
        final int prazo = 10;
        Emprestimo emprestimo = new Emprestimo();
        emprestimo.setTaxaDAO(dao);
        contexto.checking(new Expectations() {{
            oneOf (dao).obterTaxa(with(any(Cliente.class)),
                                                                    with(any(double.class)),
                                                                    with(any(int.class))));
            will(returnValue(0.05d));
        }});
        double taxa = emprestimo.calcularTaxa(cliente, montante,
prazo);

        contexto.assertIsSatisfied();
        assertTrue(taxa > 0.0d);
    }
}
```

Estou assumindo que você já conhece JUnit, ok? De qualquer forma, falarei sobre JUnit mais adiante. Para usar o jMock, criamos um contexto, que é uma instância de “Mockery”. Depois informamos a ele qual a interface do objeto que queremos sintetizar:

```
final TaxaDAO dao = contexto.mock(TaxaDAO.class);
```

Nós “injetamos” o objeto sintetizado na instância da classe que queremos testar:

```
emprestimo.setTaxaDAO(dao);
```

E declaramos nossa expectativa quanto às invocações dos métodos do objeto sintetizado:

```
emprestimo.setTaxaDAO(dao);
```

```

contexto.checking(new Expectations() {{
    oneOf (dao).obterTaxa(with(any(Cliente.class)),
                           with(any(double.class)),
                           with(any(int.class)));

    will(returnValue(0.05d));
}});

```

Estamos declarando que nossa expectativa é que o método “obterTaxa”, do nosso objeto sintético, seja invocado pelo menos uma única vez, com qualquer instância de “Cliente”, qualquer valor “double” e qualquer valor “inteiro” (“with(any(...))”). Se informarmos valores ou instâncias, o contexto vai verificar se o método foi invocado EXATAMENTE com aqueles valores. E note que especificamos o valor de resposta, que é 5% (0,05).

Depois, invocamos o método que queremos testar:

```
double taxa = emprestimo.calcularTaxa(cliente, montante, prazo);
```

E verificamos se a expectativa foi atendida (o método “obterTaxa” foi invocado uma única vez):

```
contexto.assertIsSatisfied();
```

Finalmente, fazemos um outro teste para saber se o valor da taxa está correto:

```
assertTrue(taxa > 0.0d);
```

É muito legal trabalhar com o jMock, pois é fácil, prático e flexível. Com isto, evitamos testar mais do que devemos. Por exemplo, mesmo que exista uma classe concreta para a interface “TaxaDAO”, não nos interessa testá-la. Porém, note que temos uma outra classe dentro do nosso teste: Cliente. Como é uma classe de entidade (modelo “anêmico”), resolvemos não criar um “mock” para ela, utilizando-a diretamente.

Podemos criar vários tipos de expectativas, por exemplo:

### **Especificar retornos de acordo com os parâmetros:**

```

oneOf (dao).calcularTaxa (clienteRuim,
                           with(any(double.class)),
                           with(any(int.class))); will(returnValue(0.25d));

```

```
oneOf (dao).calcularTaxa (clienteBom,  
                        with(any(double.class)),  
                        with(any(int.class))); will(returnValue(0.05d));
```

### **Especificar um número determinado de invocações de um ou mais métodos:**

```
never (dao).calcularTaxa (clienteRuim,  
                        with(any(double.class)),  
                        with(any(int.class)));  
...  
atLeast(2).of (dao).calcularTaxa (clienteBom,  
                        with(any(double.class)),  
                        with(any(int.class))); will(returnValue(0.25d));
```

### **Lançar exceções:**

```
oneOf (dao).calcularTaxa (null,  
                        with(any(double.class)),  
                        with(any(int.class)));  
will(will(throwException(IllegalArgumentException)));
```

Eu recomendo que você acesse o site do jMock e leia a documentação no site (<http://www.jmock.org/cookbook.html>), pois é um software muito útil para a criação de caso de teste unitário.

### **Como criar um caso de teste unitário**

O JUnit é o framework mais simples (e o mais popular) para criação de testes. Como é integrado ao Eclipse, a criação de casos de teste é muito facilitada. Para criar um “Test Case” no Eclipse:

1. Selecione a classe para a qual quer criar o teste;
2. Com o botão direito do mouse, selecione “new / JUnit Test Case”;
3. No diálogo “New JUnit Test Case” você notará que os dados já foram preenchidos com a classe selecionada. Clique em “Next”;
4. Selecione os métodos para os quais deseja criar testes. Cada teste é um método dentro do “Test Case”.

Para executar um teste, selecione a classe de teste e, com o botão direito do

mouse, escolha “Run as / JUnit Test” (ou então “Debug As / JUnit Test”). Sempre crie seus testes na pasta de testes designada do seu projeto. Se usar Maven, provavelmente será: “src/test/java”. Será exibida a view “JUnit”:



Ilustração 24: Resultado do teste

O mais importante é que o teste será executado também pelo Maven! Se rodarmos os “goals” “package”, “install” ou “deploy”, os testes JUnit serão executados. Como o Maven sabe onde estão os testes? Ele tem uma propriedade, “project.build.testSourceDirectory”, cujo valor padrão é: “src/test/ java”; logo, quaisquer classes (independentemente de pacote) colocadas abaixo desta pasta deverão ser JUnit Tests.

## Anatomia de um “Test Case”

Se você ainda usa uma versão anterior do JUnit, então seu “Test Case” deve ser parecido com isto:

```
package com.galaticempire.guia.microblog.microblog;

import java.util.Date;

import org.hibernate.Session;

import com.galaticempire.guia.microblog.microblog.entidades.Mensagem;
import com.galaticempire.guia.microblog.microblog.entidades.Usuario;
import com.galaticempire.guia.microblog.microblog.HibernateUtil;

import junit.framework.TestCase;
```

```

public class TestCriacaoMensagem extends TestCase {
    public void testMsg() {
        Session session = HibernateUtil.getSessionFactory().
getCurrentSession();
        try {
            session.beginTransaction();
            Usuario usu = new Usuario();
            usu.setCpf("123888");
            usu.setNome("Principal");
            session.save(usu);
            Usuario usu2 = new Usuario();
            usu2.setCpf("456999");
            usu2.setNome("Seguidor");
            usu2.getSeguidos().add(usu);
            usu.getSeguidores().add(usu2);
            session.save(usu2);
            Mensagem msg = new Mensagem();
            msg.setData(new Date());
            msg.setTexto("Bla bla bla");
            msg.setAutor(usu);
            session.save(msg);
            usu.getMensagens().add(msg);
            session.save(msg);
            assertTrue(usu.getSeguidores().contains(usu2));
            assertTrue(usu.getMensagens().contains(msg));
            session.getTransaction().rollback();
        }
        catch (Exception ex) {
            session.getTransaction().rollback();
            fail("Exception> " + ex.getLocalizedMessage());
        }
    }
}

```

É uma classe que estende “TestCase” e que possui vários métodos auxiliares. Na versão mais moderna, a classe de teste não precisa estender “TestCase” e pode utilizar anotações:

```

package com.galaticempire.guia.microblog.microblog;

import java.util.Date;

import org.hibernate.Session;

import com.galaticempire.guia.microblog.microblog.entidades.Mensagem;
import com.galaticempire.guia.microblog.microblog.entidades.Usuario;
import com.galaticempire.guia.microblog.microblog.HibernateUtil;
import org.junit.Test;
import static org.junit.Assert.*;

```

```

public class TestCriacaoMensagem {

    @Test
    public void testMsg() {
        Session session = HibernateUtil.getSessionFactory().
getCurrentSession();
        try {
            session.beginTransaction();
            Usuario usu = new Usuario();
            usu.setCpf("123888");
            usu.setNome("Principal");
            session.save(usu);
            Usuario usu2 = new Usuario();
            usu2.setCpf("456999");
            usu2.setNome("Seguidor");
            usu2.getSeguidos().add(usu);
            usu.getSeguidores().add(usu2);
            session.save(usu2);
            Mensagem msg = new Mensagem();
            msg.setData(new Date());
            msg.setTexto("Bla bla bla");
            msg.setAutor(usu);
            session.save(msg);
            usu.getMensagens().add(msg);
            session.save(msg);
            assertTrue(usu.getSeguidores().contains(usu2));
            assertTrue(usu.getMensagens().contains(msg));
            session.getTransaction().rollback();
        }
        catch (Exception ex) {
            session.getTransaction().rollback();
            fail("Exception> " + ex.getLocalizedMessage());
        }
    }
}

```

Agora, a classe é um “POJO” comum, que não precisa estender “TestCase”. Os métodos “assert...” e “fail” agora são membros estáticos da classe “Assert”. A anotação “@Test” deve marcar cada Teste dentro da classe.

## Fixture

Chamamos de “Text Fixture” as condições necessárias para executar um ou mais testes. Por exemplo, vamos supor que temos alguns objetos (e seus estados) que devem ser instanciados ANTES da execução de um teste. Então, podemos criar um método e anotar com “@Before” ou “@After”, por exemplo:

```

public class TestComponente1 {

    Fixture1 fixture1;
    Fixture2 fixture2;

    @Before
    public void preparaTeste() {
        fixture1 = <algum Objeto>.getFixture(1);
        fixture2 = <algum Objeto>.getFixture(2);
    }

    @After
    public void limpaAposTeste() {
        fixture1.liberarConexao();
        fixture1 = null;
        fixture2.liberarConexao();
        fixture2 = null;
    }

    @Test
    public void teste1() {
        assertTrue(fixture1.equals(1));
    }

    @Test
    public void teste2() {
        assertTrue(fixture2.equals(2));
    }
}

```

Neste caso, temos um método que deve ser invocado ANTES de cada teste, “preparaTeste()”, e outro que deve ser invocado APÓS cada teste: “limpaAposTeste()”. Assim, antes de executar cada um dos métodos de teste (“teste1()” e “teste2()”), o método “preparaTeste()” será invocado e, após a execução de cada teste, o método “limpaAposTeste()” será invocado.

Se você quiser fazer uma inicialização e finalização “globais”, então pode usar as anotações “@BeforeClass” e “@AfterClass”. Elas devem anotar métodos públicos, estáticos, sem valor de retorno e sem argumentos:

```

public class TestComponente1 {

    Fixture1 fixture1;
    Fixture2 fixture2;
    Conexao conexao;

    @BeforeClass
    public static void inicializaTudo() {

```

```

        conexao = Conexao.abrirConexaoCustosa();
    }

    @AfterClass
    public static void fechaTudo() {
        conexao.fecharEliberar();
        conexao = null;
    }

    @Before
    public void preparaTeste() {
        fixture1 = <algum Objeto>.getFixture(conexao,1);
        fixture2 = <algum Objeto>.getFixture(conexao,2);
    }

    @After
    public void limpaAposTeste()
    { fixture1.liberarConexao();
      fixture1 = null;
      fixture2.liberarConexao();
      fixture2 = null;
    }

    @Test
    public void teste1() {
        assertTrue(fixture1.equals(1));
    }

    @Test
    public void teste2() {
        assertTrue(fixture2.equals(2));
    }
}

```

O método “inicializaTudo” será invocado ANTES de qualquer coisa ser executada na classe e o método “fechaTudo”, APÓS a execução de todos os testes. Veja bem, eles serão executados apenas uma única vez, ao contrário dos métodos anotados com “@Before” e “@After”.

## Finalmente

Guarde seus testes no repositório Subversion! Conheço muita gente boa que esquece de fazer isto e acaba perdendo tudo! Qual é a consequência? Bem, ao longo da vida útil de um software podem ocorrer manutenções corretivas e/ou evolutivas, certo? Então, quando acrescentamos (ou corrigimos) alguma funcionalidade, temos que criar novos testes. E os antigos? Toda a funcionalidade que não foi alterada deve se comportar exatamente como



deixamos. É a única maneira de garantir a compatibilidade retroativa.

Você já deve ter visto softwares que, em um release, consertaram problemas, porém estragaram o que estava funcionando. Certo? Isto é causado por propagação de alterações sem o devido teste.

Eu sugiro que você crie um pacote de testes (dentro de “src/test/java”) para cada unidade de implementação e comente (no JavaDoc da classe de teste) qual é a versão do software para a qual o teste foi criado.

Eu recomendo que você exercite o uso de “mocks” e “Test Cases”. Por que não pega o aplicativo exemplo (veja na introdução como baixar os complementos) e melhora os testes que existem? Procure separar os testes unitários em pacotes específicos utilizando “mocks” ao invés das classes concretas. Depois pense nos testes de integração.

## **Cobertura dos testes**

Como saber se o teste é suficiente? Essa é uma dúvida eterna de todo desenvolvedor. Para garantir que o software tenha sido testado de maneira eficaz ANTES de sua liberação para os usuários finais, costumamos dividir os testes em categorias. De acordo com Pressman [Pressman 2006]:

- Testes unitários: os que estamos vendo nesse momento. Visam saber se cada unidade está funcionando de acordo com a especificação;
- Testes de integração: os que veremos mais adiante no livro, cujo objetivo é saber se as unidades funcionam em conjunto de acordo com o esperado;
- Testes de validação: analisam se o software como um todo está de acordo com as especificações. Em algumas empresas, a validação se confunde com “testes de aceitação” e é feita pelo próprio usuário, o que pode gerar um certo nível de estresse para a equipe;
- Testes de sistema: verificam as qualidades do software, ou seja, seus requisitos não funcionais. Existem vários tipos: teste de recuperação, de segurança, de estresse, de desempenho etc.

Além destes tipos de testes, Pressman cita dois outros que são bem conhecidos

dos desenvolvedores:

- Teste “alfa”: realizado por um grupo de usuários, em ambiente controlado pelo desenvolvedor, sob sua supervisão;
- Teste “beta”: realizado sem a supervisão do desenvolvedor e no próprio ambiente do usuário (ou de produção). Espera-se que uma ferramenta em versão “Beta” já seja bastante madura. Os testes “beta” iniciam-se após a conclusão bem-sucedida dos testes “alfa”.

Quando é empregado um processo de SCM, conforme já discutimos anteriormente, há uma gestão de liberação que é responsável por determinar o que será entregue em cada versão do software. Neste caso, pode ser que, após a conclusão dos testes, algumas funcionalidades sejam retiradas, seja para mitigar riscos ou por questões comerciais. Neste caso, é produzida uma versão “Release candidate” (candidata à liberação), que é entregue para testes “beta” novamente. Podem ser criadas versões diferentes e paralelas de RCs (“Release candidates”), de modo a avaliar qual é a melhor configuração.

Ao final, uma das RCs é escolhida para liberação.

Eu pretendo falar mais sobre estratégias a seguir no livro, mas fiz esta introdução para podermos comentar a importância do teste unitário, pois ele é a primeira “defesa” do time, ou seja, o “goleiro”. Desta forma, é muito importante que ele exercite o máximo possível do componente, ou seja, contenha critérios que forcem a passagem do fluxo de execução pelo maior número de instruções possível. A quantidade do código-fonte exercitada por um teste é conhecida como: “cobertura do teste”.

Quando falamos em “cobertura de teste”, estamos falando em testes do tipo “caixa branca”, nos quais a construção interna do software é utilizada para guiar a construção dos testes. Existem algumas técnicas para estudo de cobertura, de modo a construir testes mais eficazes. De acordo com a Wikipédia ([http://en.wikipedia.org/wiki/Code\\_coverage](http://en.wikipedia.org/wiki/Code_coverage)):

- Cobertura de funções: verifica se cada função no código foi invocada pelos testes. Existe uma variante chamada de “cobertura de métodos”, para orientação a objetos;

- Cobertura de comandos: verifica quantos comandos do código-fonte foram executados devido aos testes;
- Cobertura de decisão: verifica se os critérios de cada bloco de decisão (“if”, “case”) foram testados em todas as suas alternativas;
- Cobertura de condição: verifica se cada expressão lógica no código foi avaliada como “verdadeira” e “falsa” pelos testes.

Pressman [Pressman 2006] cita várias outras técnicas para análise de cobertura de testes. Uma das mais interessantes é o teste de caminho básico. Com ele determinamos um conjunto de “caminhos” de execução que permitem exercitar cada comando pelo menos uma vez. Isto feito, construímos um teste para cada caminho básico encontrado.

Para calcular o teste do caminho básico precisamos saber a complexidade ciclomática da unidade que vamos testar. A definição da Wikipédia é bem concisa:

*Cyclomatic complexity (or conditional complexity) is a software metric (measurement). It was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. The concept, although not the method, is somewhat similar to that of general text complexity measured by the Flesch-Kincaid Readability Test.*

[Wikipédia - Cyclomatic Complexity](#)

(“Complexidade ciclomática (ou complexidade condicional) é uma métrica de software (medição). Ela foi desenvolvido por Thomas J. McCabe, Sr. em 1976 e é usada para indicar a complexidade de um programa. Diretamente mede o número de caminhos linearmente independentes através do código de um programa fonte. O conceito, embora não seja o método, é um pouco semelhante ao de complexidade do texto geral medido pelo teste de legibilidade Flesch-Kincaid.” – Traduzido pelo autor).

Vamos mostrar um exemplo de complexidade ciclomática. No sistema exemplo, vamos pegar o método “public int abandonar(Usuario seguidor, Usuario seguido)”, da classe “com.galaticempire.guia.microblog.microblog.-persistencia.UsuarioDAO”. Eis o código-fonte:

```
145     public int abandonar(Usuario seguidor, Usuario seguido) {
146         int returnCode = SEM_ERROS;
147         try {
148             init();
149             session.beginTransaction();
```

```

150         if (seguidor == null || seguido == null) {
151             throw new IllegalArgumentException("Usuários não
ser nulos");
152         }
153         String cpfSeguidor = seguidor.getCpf();
154         String cpfSeguido = seguido.getCpf();
155         seguidor = (Usuario) session.get(Usuario.class, segui
getCpf());
156         if (seguidor == null) {
157             returnCode = DAO_ERRO_ABANDONAR_USUARIO_SEGUIDOR_
TENTE;
158             session.getTransaction().rollback();
159             logger.debug("Abandonar - Usuário seguidor não ex
cpf: " + cpfSeguidor);
160         }
161         else {
162             seguido = (Usuario) session.get(Usuario.class, se
getCpf());
163             if (seguido == null) {
164                 returnCode =
DAO_ERRO_ABANDONAR_USUARIO_SEGUIDO_INEXISTENTE;
165                 session.getTransaction().rollback();
166                 logger.debug("Abandonar - Usuário seguido não
te, cpf: " + cpfSeguido);
167             }
168             else {
169                 seguidor.getSeguidos().remove(seguido);
170                 seguido.getSeguidores().remove(seguidor);
171                 session.save(seguidor);
172                 session.save(seguido);
173                 session.getTransaction().commit();
174                 logger.debug("Usuario: " + cpfSeguidor + " de
seguir o usuário: " + cpfSeguido);
175             }
176         }
177     }
178     catch (Exception ex) {
179         returnCode = DAO_ERRO_ABANDONAR_USUARIO;
180         logger.error("Erro ao tentar Abandonar. Usuário: "
dor.getCpf() + ", seguido: " + seguido.getCpf(), ex);
181     }
182     return returnCode;
183 }

```

Para calcular a complexidade ciclomática, precisamos determinar o número de “nós” e “arestas” do código. Um nó é um comando por onde o fluxo pode passar.

Podemos agrupar nós, exceto se representarem condições. Comandos condicionais como “if”, “switch”, “for”, “while”, operador ternário e “try/catch” representam condicionais. Geralmente utilizamos grafos para representar nós e arestas:

- Um nó é uma ou mais instruções que sejam sempre executadas em conjunto. Podemos agrupar várias instruções em um só nó, de modo a facilitar o cálculo. Se a última instrução for condicional, temos que colocar as seguintes a ela em outros nós;
- Uma aresta é uma ligação entre nós.

Eu marquei na listagem os nós em negrito. Agrupei os nós em torno deles, já que são os nós condicionais.

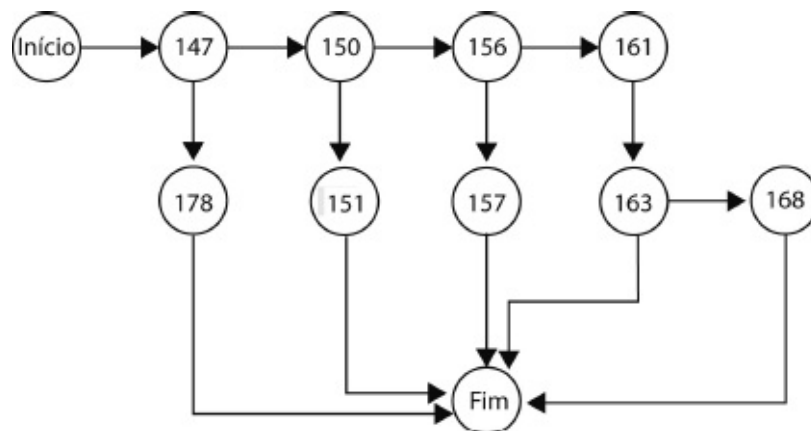


Ilustração 25: Grafo do método “abandonar()”

Na Ilustração 25 eu coloquei os números das linhas dos nós importantes, acrescentando dois outros: “Início” e “Fim”. Depois, é só contar o número de nós e arestas:

- Nós: 11 (Início, 147, 150, 151, 156, 157, 161, 163, 168, 178 e Fim);
- Arestas: 14 (Início-147, 147-150, 150-156, 156-161, 147-178, 150-151, 156-157, 161-163, 163-168, 178-Fim, 151-Fim, 157-Fim, 163-Fim e 168-Fim);

O cálculo da complexidade ciclomática é dado pela fórmula:

- Complexidade = Arestas – Nós + 2;

■  $5 = 14 - 11 + 2$ .

Isto significa que temos cinco caminhos básicos (ou independentes) para os quais precisam ser escritos testes. Eles são:

1. Início --> 147 --> 178 --> Fim
2. Início --> 147 --> 150 --> 151 --> Fim
3. Início --> 147 --> 150 --> 156 --> 157 --> Fim
4. Início --> 147 --> 150 --> 156 --> 161 --> 163 --> Fim
5. Início --> 147 --> 150 --> 156 --> 161 --> 163 --> 168 --> Fim

O “plugin” “Eclipse Metrics”, que discutimos anteriormente, já calcula a complexidade ciclômática de cada método; logo, não é necessário ter esse trabalhão todo.

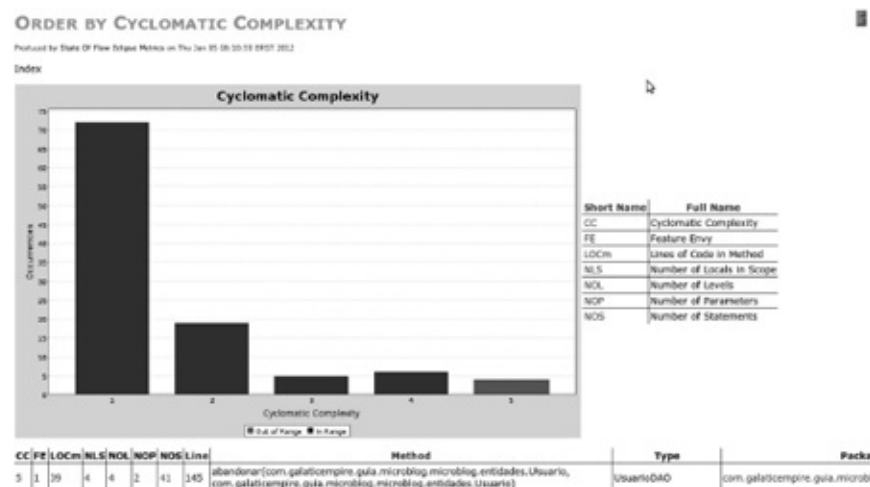


Ilustração 26: Métricas do plugin “Eclipse Metrics”

O relatório do plugin aponta os métodos de maior complexidade, como os de nível 5, logo, são bons candidatos a testes mais apurados. Porém, como gerar um teste que cubra todos os caminhos básicos? Temos que simular as condições necessárias. Por exemplo:

1. Teste 1: podemos provocar um problema tirando o servidor de banco de dados do ar, de modo a provocar uma exception. Se a exception não acontecer, algum código interno está interceptando e fazendo alguma

bobagem. O resultado aceitável é a exception;

2. Teste 2: vamos invocar com um (ou ambos) os argumentos como “null”;
3. Teste 3: esta é uma regra de negócios que diz que, para abandonar um usuário, a chave do “USUARIO\_SEGUIDOR” deve existir. Podemos inserir manualmente registros inválidos no banco;
4. Teste 4: é semelhante ao Teste 3. Insira um registro cuja chave estrangeira é nula;
5. Teste 5: é o “happy-path” (se tudo der certo). Insira registros totalmente regulares e teste.

Você pode argumentar que são testes em demasia para um só método; porém, com eles, garantimos que todos os caminhos básicos foram testados.

Uma maneira de melhorar este método seria desacoplar a classe do Hibernate. Isto pode ser feito através de adaptadores ou métodos “stub”. Se fizéssemos isto, poderíamos diminuir o escopo do teste. Neste caso, para que a classe “UsuarioDAO” fosse testada, precisaríamos ter o banco e o Hibernate configurados, o que aumenta o escopo (e a dificuldade) do teste.

## **Teste de componentes altamente acoplados**

Existem alguns componentes de alto acoplamento em qualquer aplicação, especialmente se for uma aplicação Java Web. Eles dependem de instâncias fornecidas pelo ambiente de processamento e que não podem ser facilmente simuladas.

Por exemplo, como testar os seguintes componentes:

- Uma página JSF ou JSP?
- Um Servlet?
- Um Managed Bean do JSF?
- Uma classe DAO que usa o Hibernate?

Estes componentes dependem tanto do ambiente que fica difícil testá-los adequadamente. Logo, criamos testes manuais para eles, o que é muitas vezes fonte de problemas. Certa vez, tínhamos uma alteração a fazer em uma aplicação JSF que seria bastante simples, requerendo apenas poucas mudanças em uma página, um Managed Bean e em uma classe de negócio. Bem, escrevemos testes unitários abrangentes para a classe de negócio e deixamos para testar os outros componentes manualmente. Adivinha onde os problemas aconteceram? Exatamente nos componentes para os quais não tínhamos testes unitários automatizados, ou seja: a página e o Managed Bean.

Páginas e navegação JSF só podem ser adequadamente testadas em um ambiente completo. Existe um framework para isto, o “JSFUnit” (<http://www.jboss.org/jsfunit/>) que é bem abrangente. Podemos escrever casos de teste bem semelhantes ao JUnit, porém necessitamos instalar a aplicação em um Container Java EE ANTES de executar os testes.

Se quisermos testar Servlets, o ServletUnit (<http://httpunit.sourceforge.net/doc/servletunit-intro.html>) é uma boa opção, pois simula todo o ambiente para execução de Servlets, que pode ser utilizado dentro de testes JUnit.

Managed Beans são um problema! O grupo Apache tinha um software interessante, o Shale (<http://shale.apache.org/>), que fazia “mock” do ambiente JSF, permitindo a criação de testes JUnit para Managed Beans. O problema é que o projeto Shale foi “aposentado”, ou seja, movido para o “sótão” (attic) de projetos do Apache. Você pode até usar, mas não espere suporte ou correções.

Então, como testar Managed Beans? Bem, você pode usar o JSFUnit... o problema é que ele requer o “deploy” e não pode ser executado em certos “goals” do Maven, por exemplo, “package”. Outra maneira é alterar seu Managed Bean para permitir a injeção de FacesContext, por exemplo. Então você poderá criar um “mock” para ele (usando o JMock). Há mais uma opção, que é delegar todos os métodos do Managed Bean para uma classe “helper” sua, concentrando os testes nela.

Classes DAO podem ser testadas com o uso de JMock.



## Organize sua rotina de trabalho

Como último conselho deste capítulo, eu gostaria de falar um pouco sobre organização do trabalho, algo que influencia diretamente a sua qualidade.

Vamos começar com a pergunta: como você trabalha? Sim, como inicia seu dia, como desenvolve suas tarefas? Pense um pouco...

Muitos desenvolvedores trabalham “ad-hoc”, ou seja, sem planejamento ou organização. Simplesmente pegam a tarefa que devem fazer, trabalham algumas horas e largam tudo para ir embora. Isto não é necessariamente ruim, mas implica em desorganização pessoal. Por exemplo, como planejar “commit” diário se você não sabe o que vai fazer durante o dia?

Quando eu estava cursando o Mestrado, havia uma cadeira de “Didática do Ensino Superior”, a qual eu achava “um saco”. Porém, com o passar das aulas, aprendi algumas coisas interessantes. Para começar, as pessoas médias só conseguem prestar atenção contínua por um período de aproximadamente 20 minutos. Depois disto, sentem cansaço e frequentemente desviam a atenção. Logo, devemos dividir nossa carga de trabalho em “módulos” de concentração.

### Divida seu tempo

Francesco Cirillo criou a técnica “**pomodoro**” na década de 1980 (<http://www.pomodorotechnique.com/>), com a intenção de diminuir a ansiedade e melhorar a concentração das pessoas. Segundo esta técnica, você deve usar um cronômetro (“pomodoro” de cozinha, um pequeno relógio para cozinhar, frequentemente imitando um tomate), ajustado em cerca de 25 minutos, trabalhar neste tempo e, ao acabar, fazer anotações sobre tudo o que foi feito, tirando 5 minutos de descanso. A cada quatro pomodoros, você deve tirar uma pausa maior, digamos uns 10 ou 15 minutos.

Pode parecer estranho, mas eu consegui encaixar o trabalho de desenvolvimento nesta técnica! Posso inclusive utilizar o “pomodoro” como medida de esforço: por exemplo, quantos pomodoros eu espero gastar com determinada atividade. É claro que é uma medida totalmente pessoal, mas serve para que eu organize meu trabalho.

Você teve ter um relógio pomodoro, seja um despertador ou um software (<http://www.focusboosterapp.com/live>) para te avisar que um pomodoro se passou. O importante é respeitar as pausas entre os pomodoros (5 minutos entre cada um e 10 a 15 minutos a cada quatro pomodoros).

Eu sei que parece piada, mas não é!

## **Divida o trabalho de acordo com o tempo**

Agora que você tem uma referência, vamos saber quantos pomodoros temos ao nosso dispor. Dentro de uma rotina padrão, de 9 horas (4h pela manhã, 1h para almoço e 4h à tarde), temos:

- 240 minutos de manhã;
- 240 minutos à tarde.

O tempo gasto com cada pomodoro é de 25 minutos + 5 de descanso, ou seja: 30 minutos. Pela manhã, temos oito pomodoros. Só que precisamos tirar um descanso maior a cada quatro pomodoros, além do tempo para um cafezinho, ler e-mail e jogar conversa fora. Então, vamos contar com seis pomodoros pela manhã e seis à tarde.

Portanto, a cada dia temos no máximo doze pomodoros. Então precisamos dividir nosso trabalho por estes módulos de tempo.

## **Não deixe tarefas incompletas**

A pior coisa que pode acontecer é você sair para almoçar deixando o trabalho incompleto. Quando voltar, vai perder tempo analisando onde parou... mas o pior mesmo é quando fazemos “commit” de trabalho incompleto, prejudicando as outras pessoas. Um planejamento cuidadoso pode evitar “commits” incompletos.

Ao final de cada dia, procure planejar o dia seguinte. Por exemplo, você está desenvolvendo um caso de uso do sistema de empréstimos. Falta criar a classe de cálculo de taxa de empréstimos. Se você trabalhou corretamente, já tem a interface e um diagrama UML com todos os seus relacionamentos, além de um

caso de uso bem explicado. Logo, você tem que estimar o tempo necessário para:

- Criar os métodos de teste;
- Criar os “mocks” necessários;
- Codificar cada método da classe de negócio e testá-lo.

Divida esse tempo em “pomodoros” e planeje cada dia, de modo a não deixar tarefas incompletas. Se for necessário, “empurre” tarefas para outro dia, substituindo-as por outras que caibam na quantidade de “pomodoros” disponível.

Uma tarefa incompleta é:

- Uma classe com erros de compilação ou exceções;
- Um trabalho que não foi salvo;
- Um erro investigado pela metade.

Se você utilizar interfaces, então deverá criar pelo menos os “stubs” dos métodos de implementação, o que evita um grande problema. Planeje para sempre ter uma versão “executável” de cada classe ao fim de cada turno de trabalho. Não deixe classes com erros ou exceções, mesmo que tenha que desfazer alguma coisa.

E anote sempre (ao fim de cada pomodoro) o que foi feito e o que falta fazer.

O mais importante disso tudo é: planeje e organize seu trabalho, incluindo a confecção e execução dos testes!

Se a estratégia de “commit” for “trunk não estável” e sua classe estiver compilando e rodando, então faça “commit” da classe e do teste a cada quatro pomodoros. Se estiver usando um “branch” de desenvolvimento, faça o “commit” da mesma forma. Caso contrário, faça “backup” a cada pomodoro! Por isto é que eu recomendo armazenar o projeto em uma pasta de rede (o projeto, não a workspace!).

## Conclusão

Esta parte do livro está muito relacionada à forma como o Engenheiro de software constrói o código-fonte, que é um constante fruto de problemas em equipes de desenvolvimento.

Mais uma vez, volto a salientar a importância da liderança técnica em projetos de software. Somente um líder técnico, que conheça a tecnologia e esteja atualizado, pode gerenciar o desenvolvimento de maneira adequada, certificando-se de que métodos, técnicas e ferramentas de apoio estejam sendo utilizados, assim como os padrões.

Certa vez, em uma grande empresa brasileira, eu estava conversando com algumas pessoas sobre os problemas de qualidade enfrentados por alguns dos softwares produzidos por ela. Havia um processo de desenvolvimento bem definido, com métricas e analistas de qualidade, assim como ferramentas de apoio. As equipes eram treinadas nas ferramentas e no framework utilizado, porém, os resultados ainda eram insatisfatórios. Por quê? A resposta é simples: os líderes eram o ponto fraco! Os projetos eram liderados por pessoas despreparadas...

Não tem jeito! Engenharia de software é uma disciplina com um grande corpo de conhecimento (IEEE, 2004), logo, para liderar projetos de software é necessário conhecer e ter experiência com desenvolvimento. É a mesma coisa que tentar gerenciar a construção de um edifício sem ser Engenheiro Civil.

## 9.

# Depois do código **pronto**

Bem, terminamos uma unidade, um subsistema ou todo o trabalho. Ainda existem diversas atitudes que podemos tomar, de modo a garantir a qualidade do software. Para começar, o trabalho está realmente pronto? Vamos supor que uma unidade ficou pronta. Temos que saber se:

- Ela passou nos testes unitários?
- Ela está em conformidade com a especificação (requisitos)?
- Ela está funcionando sem problemas no conjunto do sistema (integração)?
- Ela foi codificada de acordo com as normas e os padrões estabelecidos?

Hoje em dia, com equipes de desenvolvimento grandes, heterogêneas e dispersas geograficamente, o trabalho de desenvolvimento de um software é muito parecido com a fabricação de um automóvel, ou seja, recebemos módulos pré-montados, os quais devemos integrar para formar o produto final.

Quando todas as unidades estiverem integradas temos que fazer mais testes! Precisamos saber se tudo o que foi especificado foi atendido e se o usuário está gostando do produto final.

## Revisões

Para evitar a introdução de bugs, precisamos evoluir nosso processo de desenvolvimento. Uma das formas de fazer isto é promover **revisões de código**. Todos os modelos de processo de desenvolvimento, inclusive o CMMI, pregam a revisão por pares (“Perform Peer Reviews”), mas o SWEBOK (IEEE, 2004), em seu capítulo 11 (“Software Quality”), indica alguns tipos de revisões:

- **“Management reviews”** (revisões gerenciais): realizadas sobre os artefatos de controle do projeto, de modo a monitorar o progresso e a eficácia da gestão do projeto de software;
- **“Technical reviews”** (revisões técnicas): seu objetivo é avaliar um produto de software para determinar sua adequação para o uso pretendido, identificando as discrepâncias das especificações aprovadas e dos padrões;
- **“Inspections”** (inspeções): a finalidade de uma inspeção é detectar e identificar anomalias do produto de software. São revisões mais focadas, geralmente em uma parte do software, com o objetivo de encontrar anomalias;
- **“Walk-throughs”** (travessia do código-fonte): o propósito de um walk-through é avaliar um produto de software de maneira informal, normalmente para explicar à equipe como ele foi desenvolvido;
- **“Audits”** (auditorias): o objetivo de uma auditoria de software é fornecer uma avaliação independente da conformidade de produtos de software e processos para com a regulamentação aplicável: normas, diretrizes, planos e procedimentos.

A revisão por pares é uma técnica utilizada “ad-hoc” pelos desenvolvedores há muito tempo e é um tipo de revisão do software em que um produto de trabalho (normalmente, código-fonte) é examinado pelo seu autor e um ou mais colegas, a fim de avaliar o seu conteúdo técnico e de qualidade.

Para mim, o sucesso de uma revisão está na aplicação das seguintes práticas:

1. **Escolher os colegas por critérios técnicos**, excluindo: chefes, “encostados” e “puxa-sacos”. Eles não são bons participantes porque não seguem critérios objetivos e não são comprometidos com o resultado técnico. Evite transformar uma revisão em um “paredão de fuzilamento” ou em um “palanque eleitoral”;
2. **Deixar claros os critérios que devem ser utilizados para a revisão**. Pode ser um documento simples, que indique os padrões utilizados e o que o código-fonte deveria seguir. Deixe claro também o que é importante e o que não deve ser alvo de revisão, por exemplo: estilo de programação;
3. **Enviar os critérios e os produtos a serem revistos antecipadamente**.

Tem uma frase atribuída à Tancredo Neves que deixa claro o motivo: “Reunião só depois de tudo combinado”. Ao enviar antecipadamente, damos tempo aos participantes para se prepararem e tirarem dúvidas, tornando a reunião mais objetiva;

4. **Preparar uma agenda e segui-la rigorosamente.** A reunião de revisão deve seguir uma agenda, com tempo determinado para terminar. Você pode designar um facilitador, de preferência externo, que controle tudo, incluindo se os assuntos estão dentro do escopo.

Se você foi convidado para participar como revisor, deve ler antecipadamente os critérios e o produto a ser revisado, criando uma lista de “não conformidades”. Para cada elemento no produto que apresente discrepância (com os critérios), anote:

- a. O nome do módulo (pacote, classe, método...);
- b. A linha de código (muito importante). Se recebeu o código-fonte impresso, então indique a página e a linha;
- c. O título da discrepância. Por exemplo, poderíamos ter algo assim: “Método com condicional controlada por parâmetro”;
- d. Se estiver nos critérios, indique a solução recomendada, por exemplo: “substituir condicional por polimorfismo”. Evite colocar soluções pessoais. O objetivo não é dar soluções, mas encontrar discrepâncias;
- e. A importância. Se a lista de critérios não trazer esta classificação, procure adotar algo simples, tipo: “alta”, “média” e “baixa”.

Sua lista deve conter apenas estas entradas. Se você tiver algo mais a falar, que esteja fora dos critérios, então prepare observações e fale ao final da revisão. Envie sua lista e suas observações por escrito.

Durante a revisão, é melhor seguir uma programação, que pode começar por ordem alfabética de elementos (classes), ou por número de página, se o código-fonte foi enviado em forma de documento. Esta ordem deve ser seguida à risca, sem desvios. Para cada item (página ou classe), o facilitador pergunta se algum dos revisores tem alguma observação. É o momento de falar o que colocou em sua lista, apenas para o item que está sendo tratado, sendo claro e sucinto.

Evite falar mais do que o solicitado e evite opiniões pessoais. Naturalmente, se o autor ou qualquer outro revisor tiver dúvidas, poderá lhe perguntar a sua opinião.

### **Quando a revisão deve ser feita?**

Precisamos revisar cada unidade de produto de trabalho ANTES que seja integrada ao sistema, pois pode ser necessário efetuar correções no código-fonte. O ideal é que cada unidade tenha sido aprovada no teste de integração ANTES de fazer a sua revisão. Então, se ela passou no teste de integração, por que não estaria aprovada? Porque a adequação às normas e aos padrões é condição necessária para considerar uma unidade como “pronta”.

### **Promover a gestão do conhecimento**

Um objetivo secundário importante das revisões é melhorar a “gestão do conhecimento”. Existem dois tipos básicos de conhecimento: o tácito e o explícito. Conhecimento explícito é aquilo que está escrito e que pode ser facilmente praticado, mensurado e ensinado. Já, o conhecimento tácito ([http://pt.wikipedia.org/wiki/Conhecimento\\_tácito](http://pt.wikipedia.org/wiki/Conhecimento_tácito)) é o acumulado por uma pessoa (ou equipe) ao longo do tempo e que não pode ser facilmente formalizado.

Uma das maneiras de tentar “explicitar” o conhecimento é através da troca de experiências. As revisões são excelentes para promover a integração e divulgação de conhecimento na equipe. Analisando as implementações (walk-throughs) os participantes entendem o que deu certo e o que deu errado, guardando as lições aprendidas.

### **Otimização do processo**

Finalmente, com tudo isto que falamos, podemos otimizar continuamente o processo de desenvolvimento da equipe (nível 5 do CMMI), de modo a melhorar sua eficiência. A análise de problemas comuns, a análise de causa e efeito e a revisão de lições aprendidas formam um conjunto essencial para melhorar qualquer processo.

Vou dar mais um exemplo... lembra-se daquela regra idiota que mencionei? Aquela que dizia que o nome de uma variável não pode ter menos de cinco letras? Pois bem, eu fui voto vencido e a regra foi implementada. Porém, algum



tempo depois, começamos a notar muitos problemas de revisão relacionados a esta regra. Os desenvolvedores não conseguiam aceitar a regra e, portanto, acabavam infringindo-a sem perceberem. O custo de alterar um programa apenas para aumentar o nome de variáveis é muito alto, ainda mais quando o problema se repete.

Quando vemos um problema recorrente, temos que tomar uma atitude para resolvê-lo. E, neste caso, a melhor atitude foi alterar a regra. A equipe inteira aprendeu uma lição e melhorou o processo como um todo.

Otimização é uma prática obrigatória em processos de desenvolvimento com CMMI nível 5 (PRESMANN). Independentemente do nível de CMMI em que sua equipe trabalhe, é bom sempre olhar para o processo “do alto”, tentando identificar pontos fracos e oportunidades de melhoria, pois melhorando o processo melhoramos o produto de software que ele gera.

## Integração

Integração é a atividade de integrar uma (ou mais) unidade ao conjunto de unidades que compõem o software. As principais mudanças de estado são:

- Passa a fazer parte do repositório de fontes. Se a política de “commit” for “Trunk de integração”, então ela é salva no tronco principal do projeto;
- Passa a fazer parte do repositório de componentes “snapshot”, do Archiva. Ela passa a estar disponível para uso, junto com as outras unidades do sistema.

Normalmente, a integração é uma atividade arriscada, pois podem acontecer diversas incompatibilidades entre a unidade e o todo.

### Por que a integração é problemática?

Pressman [Pressman 2006] tem uma frase interessante:

*Se todos eles (os componentes) funcionam individualmente, por que você duvida que vão funcionar quando colocados em conjunto?*

Quando falamos em unidades, há alguns limites: quantidade de código-fonte, quantidade de configurações e quantidade de responsáveis; logo, temos maior controle sobre os elementos componentes. Porém, ao integrarmos software estaremos fazendo uma ou mais das seguintes tarefas:

- Substituir “mocks” por classes concretas reais;
- Substituir “method stub” por métodos funcionais;
- Substituir dados falsos por dados reais (pelo menos de teste);
- Utilizar interfaces e protocolos reais;
- Utilizar ambientes reais, sejam de desenvolvimento, teste ou produção.

Desta forma, quase certamente uma coisa pode falhar, certo? Se você e os outros desenvolvedores seguiram as especificações, e construíram testes unitários com grande cobertura, então as três primeiras tarefas podem transcorrer de forma tranquila. Mesmo assim, as duas últimas, quase sempre, causam “chabu”.

Normalmente, na maioria dos projetos, as integrações são realizadas apenas em determinados marcos, por exemplo: no início da construção, com classes “mock”, no meio, quando alguns componentes críticos foram concluídos, antes de um teste de carga ou de aceitação e no final de tudo. E, geralmente, dá problema! Por exemplo, em uma sexta-feira a equipe está estressada, integrando tudo e colocando no ambiente de testes, pois na segunda-feira o Cliente virá para avaliar a aplicação. Então, quase sempre o pessoal vira a noite porque descobriram que o “build” integrado está quebrado.

Pressman [Pressman 2006] diz que existem alguns tipos de integração:

- “Big-bang”: todo os componentes são agregados, compilados e testados de uma só vez. É a abordagem utilizada no exemplo que dei anteriormente;
- Descendente: integramos os módulos de mais alta ordem, na hierarquia de controle, com “mocks” para os de mais baixa ordem;
- Ascendente: integramos os módulos de mais baixa ordem em primeiro lugar.

## **Integração contínua**

Martin Fowler ([http://pt.wikipedia.org/wiki/Martin\\_Fowler](http://pt.wikipedia.org/wiki/Martin_Fowler)) tem um excelente artigo sobre integração contínua: [http://martinfowler.com/articles/continuousIntegration.html#Practices\\_OfContinuousIntegration](http://martinfowler.com/articles/continuousIntegration.html#Practices_OfContinuousIntegration). No artigo, ele expõe diversas práticas recomendadas para integrar o software continuamente e periodicamente.

Podemos escolher a frequência de integração e, a cada intervalo, produzir uma versão completa do software, compilando, executando testes e mostrando os resultados. Tem gente que faz “build” a cada dez minutos, a cada hora, duas vezes por dia ou então sempre que algo for “comitado” (salvo) no repositório.

O processo de integração deve ser repetitivo e preciso, informando aos interessados qual foi o resultado. Daí a importância de ser automatizado.

O princípio defendido por Fowler é que o “Mainline”, a linha principal (“baseline”) no repositório SCM, sempre deve ser estável, sendo renovada diariamente. Podemos usar o “trunk” ou um “branch” de desenvolvimento no SVN, não importa... o que importa é que ninguém pode deixar a “baseline” com “build quebrado”, ou seja, sem compilar ou com falha nos testes. O ideal é que, caso ocorra uma quebra em um “build”, as pessoas sejam avisadas rapidamente.

Outra recomendação importante é que os desenvolvedores façam “commit” diariamente, nunca quebrando o “baseline”. Para fazer “commit” diário, algumas coisas são importantes:

1. Antes de começar o trabalho, atualizar sua cópia de trabalho com um “svn update”;
2. Verificar se houve algum conflito (o que nunca deveria acontecer);
3. Ao concluir o trabalho, ou antes de terminar o dia, faça outro “update” e “commit”, analisando os possíveis conflitos;
4. Espere pelo próximo relatório da integração contínua.

## **Configurações de ambiente**

Configurações de ambiente sempre incluem:

- URL de serviços (bancos de dados, web services etc.);
- Credenciais de acesso (nome de usuário, senha).

Conforme já falei anteriormente, alguns desenvolvedores gostam de colocar estas informações dentro do código-fonte (argh!). Outros, mais conscientes, colocam em arquivos de propriedades. Porém, elas sempre acabam gerando dor de cabeça quando chega o momento de “promover” um componente (promover é passar um componente de um ambiente menos controlado para um mais controlado). Qual é a solução?

## JNDI (Java Naming and Directory Interface)

JNDI é uma API para consumo de serviços de diretório e é parte integrante do Java EE. Com o JNDI, podemos consumir informações e até mesmo obter instâncias de classe armazenadas em um Servidor de Diretórios. Todo container Java EE deve oferecer serviços de diretório para as aplicações, representadas por um objeto de contexto de nomes (InitialContext). Usando JNDI, podemos configurar entradas no Container e consultá-las dentro da aplicação. Por exemplo:

```
try {  
    contexto = new InitialContext();  
    url = (URL)contexto.lookup("java:comp/env/url/servidorcontas");  
    ...  
}  
catch (NamingException nex) {  
    ...  
}
```

Neste caso, o código está obtendo uma URL que está armazenada no diretório, recebendo uma instância de “java.net.URL”.

Para cada tipo de objeto, o Container fornece um contexto para armazenamento e recuperação. Eis os principais tipos utilizados:

Tipo de recurso	Contexto	Objeto retornado
DataSources	“java:comp/env/jdbc”	“javax.sql.DataSource”
Filas de mensagens	“java:comp/env/jms”	“javax.jms.QueueConnectionFactory”
URL	“java:comp/env/url”	“java.net.URL”

Outros                   |“java:comp/env/”       |“java.lang.Object”

Podemos consultar outros tipos de recursos, como Strings, por exemplo:

```
Context ctx = new InitialContext();
String parametro = (String) ctx.lookup("java:comp/env/parametro");
```

A configuração de objetos dentro do JNDI depende do servidor de aplicação (ou Container) que você está utilizando. A maioria deles permite fazer via arquivo de configuração ou via console de gerenciamento. No Tomcat, podemos adicionar uma entrada no arquivo de contexto da aplicação “../conf/catalina/localhost/<aplicacao>.xml”:

```
<Context ...>
...
<Environment name="parametro" value="meu parametro 1"
              type="java.lang.String" override="false"/>
...
<Resource name="jdbc/meuBanco" auth="Container"
          type="javax.sql.DataSource"
          description="Meu banco de dados"/>
...
<ResourceParams name="jdbc/meuBanco">
  <parameter>
    <name>driverClassName</name>
    <value>org.apache.derby.jdbc.ClientDriver</value>
  </parameter>
  <parameter>
    <name>url</name>
    <value>jdbc:derby://localhost:1527/microblog</value>
  </parameter>
  <parameter>
    <name>user</name>
    <value>APP</value>
  </parameter>
  <parameter>
    <name>password</name>
    <value>APP</value>
  </parameter>
</ResourceParams>
...
</Context>
```

Criamos duas entradas no JNDI: um string e uma Datasource. Note o atributo “override” na entrada “<environment>” significando que este atributo “parametro” não pode ser substituído por entradas cadastradas no arquivo

“web.xml” da aplicação.

## Para bancos de dados, use Datasources

Uma Datasource é uma instância de “javax.sql.DataSource” e é uma fábrica de conexões físicas para um servidor. Como já vimos, é possível armazenar instâncias de Datasources no Container acessando via JNDI:

```
<Context ...>
...
<Resource name="jdbc/meuBanco" auth="Container"
    type="javax.sql.DataSource"
    description="Meu banco de dados"/>
...
<ResourceParams name="jdbc/meuBanco">
    <parameter>
        <name>driverClassName</name>
        <value>org.apache.derby.jdbc.ClientDriver</value>
    </parameter>
    <parameter>
        <name>url</name>
        <value>jdbc:derby://localhost:1527/microblog</value>
    </parameter>
    <parameter>
        <name>user</name>
        <value>APP</value>
    </parameter>
    <parameter>
        <name>password</name>
        <value>APP</value>
    </parameter>
</ResourceParams>
...
</Context>
```

Note que determinamos: a classe do “driver” JDBC, a URL de acesso ao Servidor de banco de dados, incluindo o nome do catálogo (banco), e o usuário e a senha a serem utilizados. Isto fica armazenado no Container e não dentro de cada aplicação. A aplicação pode obter a Datasource programaticamente:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/meuBanco");
Connection db = ds.getConnection();
```

## Evite usar parâmetros dentro do pacote

Existem várias maneiras de configurar parâmetros de ambiente. Colocar dentro do código-fonte é a pior delas! Porém, colocar dentro de arquivos implica em mais trabalho, ou seja, alguém tem que desempacotar a aplicação, substituir o arquivo de configuração e empacotar novamente. Existem outras técnicas para isto:

- Usar “profiles” Maven: permitem alterar o POM em tempo de “build” (<http://maven.apache.org/guides/introduction/introduction-to-profiles.html>);
- Usar um software de CI (integração contínua), como o Apache Continuum (<http://continuum.apache.org/>), e configurar variáveis de ambiente para “build”;
- Criar variáveis de ambiente em cada Container.

Eu acredito que a melhor solução seja utilizar parâmetros JNDI, pois fazem parte da especificação Java EE.

## Testes de integração

Chegou o momento de integrar! Então, temos que escrever testes que avaliem a integração das unidades.

Um teste de integração, por definição, testa mais de uma unidade. Ele pode ser bem complexo, incluindo inicialização de servidores, por exemplo. Mas, para efeito deste livro, vamos apenas mostrar como criar e configurar testes de integração bem simples.

### Anatomia de um teste de integração

Considerando nosso sistema exemplo, vamos pegar um dos testes que já existem, o qual é basicamente de integração:

```
package com.galaticempire.guia.microblog.microblog;
```

```
import java.text.DateFormat;  
import java.util.Date;  
import java.util.List;
```

```
import com.galaticempire.guia.microblog.microblog.entidades.Mensagem;
import com.galaticempire.guia.microblog.microblog.entidades.MensagemTO;
import com.galaticempire.guia.microblog.microblog.entidades.Usuario;
import com.galaticempire.guia.microblog.microblog.negocio.UsuarioBC;
import com.galaticempire.guia.microblog.microblog.persistencia.UsuarioDAO;
```

```
import static com.galaticempire.guia.microblog.microblog.GlobalConstants.*;
import junit.framework.TestCase;
```

```
public class TestUsuarioBC extends TestCase {
```

```
    private Usuario usu1;
    private Usuario usu2;
    private UsuarioDAO dao = new UsuarioDAO();
    private UsuarioBC bc = new UsuarioBC();
    private Mensagem mJose1;
    private Mensagem mJose2;
    private Mensagem mSeguidor1;
    private Mensagem mSeguidor2;
```

```
    @Override
```

```
    protected void setUp() throws Exception {
```

```
        super.setUp();
```

```
        try {
```

```
            usu1 = new Usuario();
            usu1.setCpf("121314777");
            usu1.setNome("José da Silva");
            usu1.setSenha("teste");
            usu1.setArquivoFoto("be1.jpg");
            assertTrue(dao.addUsuario(usu1) == SEM_ERROS);
            usu2 = new Usuario();
            usu2.setCpf("151617888");
            usu2.setNome("Seguidor de José da Silva");
            usu2.setSenha("teste");
            usu2.setArquivoFoto("lm1.jpg");
            assertTrue(dao.addUsuario(usu2) == SEM_ERROS);
            assertTrue(dao.seguir(usu2, usu1) == SEM_ERROS);
            mJose1 = new Mensagem();
            mJose1.setData(new Date());
            mJose1.setTexto("Primeira mensagem de José da Silva");
            assertTrue(dao.postarMensagem(usu1, mJose1) == SEM_ERROS);
            mJose2 = new Mensagem();
            mJose2.setData(new Date());
            mJose2.setTexto("Segunda mensagem de José da Silva");
            assertTrue(dao.postarMensagem(usu1, mJose2) == SEM_ERROS);
            mSeguidor1 = new Mensagem();
            mSeguidor1.setData(new Date());
            mSeguidor1.setTexto("Primeira mensagem do seguidor");
            assertTrue(dao.postarMensagem(usu2, mSeguidor1) == SEM_ERROS);
```



```

        mSeguidor2 = new Mensagem();
        mSeguidor2.setData(new Date());
        mSeguidor2.setTexto("Segunda mensagem do seguidor");
        assertTrue(dao.postarMensagem(usu2, mSeguidor2) == SEM_ERROS);
    }
    catch (Exception ex) {
        fail("Exception> " + ex.getLocalizedMessage());
    }
}

public void testAtualizarTelaInicial() {
    try {
        List<MensagemTO> msgs = bc.atualizarTelaInicial(usu2);
        DateFormat df = DateFormat.getDateInstance();
        for (MensagemTO m : msgs) {
            System.out.println("@@@@ " + m.getTexto());
        }
        MensagemTO m1 = new MensagemTO(df.format(mJose1.getData()),
            mJose1.getTexto(), mJose1.getAutor().getNome(),
            "../images/" + mJose1.getAutor().getArquivoFoto());
        MensagemTO m2 = new MensagemTO(df.format(mJose2.getData()),
            mJose2.getTexto(), mJose2.getAutor().getNome(),
            "../images/" + mJose2.getAutor().getArquivoFoto());
        MensagemTO m3 = new MensagemTO(df.format(mSeguidor1.getData()),
            mSeguidor1.getTexto(), mSeguidor1.getAutor().getNome(),
            "../images/" + mSeguidor1.getAutor().getArquivoFoto());
        MensagemTO m4 = new MensagemTO(df.format(mSeguidor2.getData()),
            mSeguidor2.getTexto(), mSeguidor2.getAutor().getNome(),
            "../images/" + mSeguidor2.getAutor().getArquivoFoto());
        assertTrue(msgs.contains(m1) &&
            msgs.contains(m2)
            &&
            msgs.contains(m3) &&
            msgs.contains(m4));
        usu1 = dao.getUsuario("121314777");
        assertTrue(dao.removeUsuario(usu1) == SEM_ERROS);
        usu2 = dao.getUsuario("151617888");
        assertTrue(dao.removeUsuario(usu2) == SEM_ERROS);
    }
    catch (Exception ex) {
        fail("Exception> " + ex.getLocalizedMessage());
    }
}
}

```

O nosso sistema atualiza a tela inicial de cada usuário, mostrando as mensagens de todos os usuários que ele “segue” (semelhante ao “Twitter”). Este teste verifica se o nosso “Business Controller” (uma classe de negócios) traz a

listagem correta para um determinado usuário. É um teste de integração, que abrange quatro componentes do sistema: entidades, “UsuarioDAO”, “UsuarioBC” e o próprio banco de dados.

No final, ele verifica se a lista de mensagens retornada contém as mensagens que deveria conter.

## **Separação dos testes de unidade**

Por padrão, o Maven invoca o plugin “Surefire” durante a fase “test” do ciclo de montagem. Só para lembrar, as fases são:

- validate – valida o projeto;
- compile – compila o projeto;
- test – invoca os testes;
- package – empacota o projeto;
- integration-test – instala o projeto no ambiente de testes de integração;
- verify – verifica se o pacote é válido e se atendeu aos critérios;
- install – instala o pacote no repositório local (“../.m2”);
- deploy – instala o pacote no repositório definitivo.

Então ele executa TODOS os testes que estejam dentro da pasta “src/test/java”. Se os seus testes estiverem dentro desta pasta (e se incluírem a palavra “test” em seu nome), o “Surefire” vai executá-los. Isto inclui os testes de integração também.

Isto é muito custoso, pois os testes de integração somente devem ser executados quando utilizamos os “goals”: “install”, “verify” e “deploy”. Podemos separar isto através de duas medidas: separação física dos casos de teste de integração e configuração do “pom.xml”.

Para começar, vamos criar mais um pacote dentro de “src/test/java”, onde os casos de teste de integração deverão ficar. Depois, copiamos as classes para ela.

Em nosso caso ficou assim:

```
package com.galaticempire.guia.microblog.microblog.integracao;
```

```
import java.text.DateFormat;
import java.util.Date;
import java.util.List;
```

```
...
```

```
public class TestUsuarioBCIT extends TestCase {
```

No “pom.xml” podemos utilizar outro plugin para executar os testes de integração: o “FailSafe”. Ele é indicado para executar testes de integração e somente executa os testes cujo nome termina em “\*IT.\*”, por isto tivemos que renomear a classe. O Maven tem as seguintes fases adicionais:

- pre-integration-test: para inicializar o ambiente de teste (servidores etc.);
- integration-test: para executar os testes de integração;
- post-integration-test: para finalizar o ambiente de teste.

Ao usar o “FailSafe”, mesmo que ocorram problemas durante o teste de integração, a fase “post-integration-test” sempre será executada. Além disto, você deve sempre utilizar o “goal” “verify” em vez de “integration-test”. Assim, garante que “post-integration-test” seja executada sempre.

Para usar o “FailSafe”, adicione o seguinte “plugin” no seu “pom.xml”, dentro do tag “<build><plugins>”:

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-failsafe-plugin</artifactId>
<version>2.11</version>
<executions>
  <execution>
    <goals>
      <goal>integration-test</goal>
      <goal>verify</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

Estamos informando ao Maven que o plugin “FailSafe” deverá ser invocado nos dois “goals”: “integration-test” e “verify”.

Se você colocar isto e mandar executar “Run As / Maven Test”, notará que os testes de integração foram executados mesmo assim... é porque o “Surefire” os executou. Para evitar isto, adicione a configuração do “Surefire” ao seu “pom.xml”, dentro de “<build><plugins>”:

```
<!-- Ignorar os testes de integração -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.7.1</version>
    <configuration>
        <excludes>
            <exclude>*/integracao/*</exclude>
        </excludes>
    </configuration>
</plugin>
```

Estamos alterando a configuração padrão do “Surefire” para que ele ignore TODOS os testes que estejam no subpacote “integracao”. Ao executar “Run As / Maven Test” você notará que o teste de integração não foi executado. Agora, para executar o teste de integração, basta selecionar “Run As / Maven Build...” e selecionar o “goal” “verify”.

### **Mas ele roda os testes unitários também?**

Sim. O padrão do ciclo de build é executar todos os “goals” configurados até o que você especificou. Porém, se você não quiser rodar os testes unitários, pode criar um “profile” dentro do seu “pom.xml”:

```
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://maven.apache.org/POM/4.0.0"
    xsi:schemaLocation=
        "http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.teste</groupId>
    <artifactId>projeto</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>

    <name></name>
```

```

<description></description>
<url></url>

<properties>
</properties>

<repositories>
</repositories>

<profiles>
  <profile>
    <!-- para ignorar os testes unitarios -->
    <id>integracao</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-
            plugin</artifactId>
          <version>2.7.1</version>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
...

```

Neste “profile” estamos mandando que o plugin “Surefire” pule os testes. Agora, para ativar o profile podemos:

- a. Executar “Run As / Maven Build...” e especificar o “goal”: “verify” e o “profile”: “integracao”;
- b. Ou rodar via linha de comando: “mvn verify -P integracao”.

Como o “profile” não alterou a configuração do “FailSafe”, os testes de integração ainda serão executados sem problemas.

## Processo de liberação

Temos que seguir um processo com várias etapas ANTES de liberarmos o software para uso. Conforme vimos anteriormente, o próprio conceito de SCM

prevê isto.

As etapas de um processo de liberação podem ser resumidas assim:

1. **Construção:** construir e aprovar em testes unitários cada componente do software;
2. **Integração:** integrar cada componente e aprovar em testes de integração. Depois, revisar cada nova unidade integrada;
3. **Qualidade:** testar as qualidades do software (capacidades) e validar contra os requisitos;
4. **Aceitação:** fazer o cliente testar e aprovar o software;
5. **Liberação:** criar uma estratégia para liberação e disponibilizar o software de acordo com ela.

Durante a construção, precisamos criar cada unidade e testá-la adequadamente, conforme já vimos. Devemos também avaliar a qualidade individual, para saber se a manutenibilidade e a cobertura de testes estão adequadas.

Depois é o momento de uma ou mais unidades. Para isto, devem ter sido aprovadas em testes de integração e em revisões formais. Devemos também fazer inspeções automatizadas no código-fonte, para saber se melhoramos ou pioramos a qualidade geral a cada integração.

Uma vez gerada uma versão final, precisamos avaliar sua qualidade, ou seja, se os requisitos não funcionais estão de acordo com a especificação. Fazemos isto através de testes de sistema, como teste de desempenho, de estresse etc. Também precisamos validar o software implementado contra os requisitos, de modo a fazer uma verificação final de tudo.

Depois da versão final ter sido aprovada pela equipe, é necessário promover testes de aceitação. Estes podem ser do tipo “alfa”, ou seja, sob controle do desenvolvedor, mas também podem incluir testes “beta”, no ambiente do usuário.

Finalmente, com a funcionalidade testada, precisamos determinar quais

funcionalidades serão liberadas e quando faremos isto. Parece estranho, mas nem sempre é interessante liberar TUDO o que foi feito no último ciclo de desenvolvimento. Pode haver questões técnicas ou mercadológicas que influenciem nesta decisão. A estratégia deve contemplar também o tipo de liberação, por exemplo:

- **“Upgrade”** (atualização): muito empregada em aplicações móveis. Podemos publicar a nova versão como uma atualização da anterior, e o software do dispositivo se encarrega do resto;
- **“Turn-key”** (virar a chave): usado em sistemas corporativos, significa que vamos disponibilizar a nova versão de uma hora para outra para uso imediato, desativando o sistema antigo;
- **Paralela**: quando a nova versão (ou a nova aplicação) será processada em conjunto com o procedimento anterior, evitando quebra nos negócios;
- **Incremental**: vamos liberar em paralelo, selecionando grupos de usuários para entrar na nova versão. Conforme a confiança vai aumentando, vamos adicionando novos grupos até podermos desativar o sistema antigo.

Até mesmo o tipo de liberação que vamos fazer pode influenciar na composição que vamos liberar – por exemplo, podemos usar liberação incremental, aumentando também a funcionalidade disponível conforme ganhamos confiança. O tipo de liberação depende muito do tipo de aplicação e do público-alvo que vamos atingir. Por exemplo:

- **Novos websites**: não temos nada publicado e queremos atingir o público da Internet. Apesar do risco ser maior, é melhor usar liberação “turn-key”, pois ninguém tem paciência para ficar voltando ao site até que todas as funcionalidades estejam disponíveis;
- **Aplicações corporativas**: é melhor usar liberação incremental, de modo a não causar uma interrupção nos negócios.

## Implementando a integração contínua

Conforme já mencionei, a integração é um processo complexo e, para ser bem-sucedido, é necessário promover integrações frequentes. A Integração Contínua (CI – Continuous Integration) é uma técnica geralmente aceita para mitigar os

problemas.

Para que seja bem-sucedida, é necessário ter um ambiente de “build” (ou montagem) apoiado por ferramentas automatizadas, além de procedimentos a serem seguidos por todos os desenvolvedores. Nós já falamos sobre algumas ferramentas importantes:

- **Maven:** para controle de montagens (ou “builds”);
- **Archiva:** para gerenciamento de componentes;
- **Subversion:** para controle de versões de artefatos do projeto.

Podemos utilizar uma ferramenta para automatizar a integração contínua. As mais populares são:

- **CruiseControl:** <http://cruisecontrol.sourceforge.net/>;
- **Apache Continuum:** <http://continuum.apache.org/>;
- **Go Agile Release Management:** <http://www.thoughtworks-studios.com/-go-agile-release-management>;
- **IBM Rational Team Concert:** <http://www-01.ibm.com/software/rational/products/rtc/>;
- **Hudson:** <http://hudson-ci.org/>.

Eu já usei muito o CruiseControl, porém, atualmente, estou usando o Apache Continuum devido a sua integração com outros produtos, como o Archiva e o Maven.

## Instalação e configuração do Continuum

Baixe o “continuum”: <http://continuum.apache.org/download.html>. O Continuum, assim como o Archiva, vem em dois “sabores”: “standalone” e “WAR”. Eu instalei o Archiva WAR dentro de um Tomcat que tenho configurado aqui. A princípio, você poderia instalar o WAR do Continuum no mesmo servidor, usando a mesma porta, porém dá problemas com o banco de dados. Como ambos usam o Derby Embedded, acontecem conflitos quando são executados juntos. Eu recomendo instalar o Continuum “standalone”, mudando



Note que ela já mostra a URL que configuramos no servidor Jetty. Eu recomendo que você não mexa em nada, especialmente o campo “Diretório para Repositório de Artefatos”. Afinal de contas, o “pom.xml” dos nossos projetos já indica onde devem ser distribuídos (Archiva).

A explicação detalhada de como funciona o Continuum está fora do escopo deste livro. Se quiser saber mais, procure a documentação do projeto no Apache: <http://continuum.apache.org/docs/1.3.8/index.html>.

## O que o Continuum faz? Qual é o seu relacionamento com o Archiva?

Uma imagem vale mais do que mil palavras...

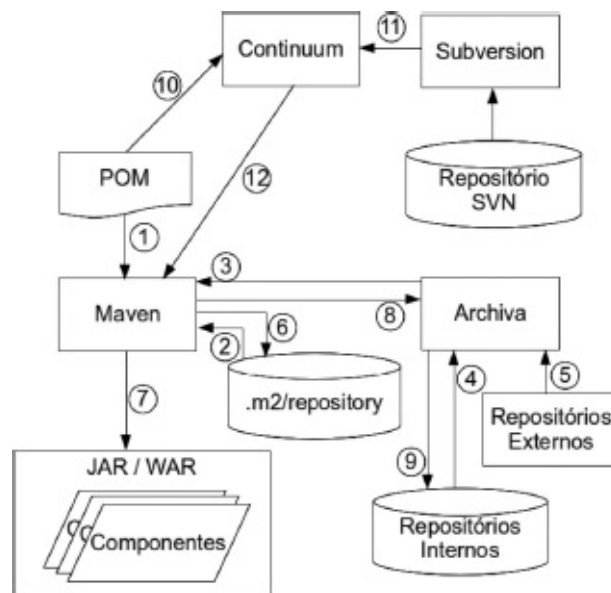


Ilustração 28: Onde o Continuum se encaixa no conjunto...

Lembra-se desta figura original? Bem, acrescentei o Continuum e o Subversion, de modo a apresentar qual é o papel dos dois neste processo.

Um dos princípios do SCM é que o software tem que ser montado (“build”) a partir do repositório de versões, e não do código-fonte presente nas cópias de trabalho dos desenvolvedores. Bem, o Continuum vai ao nosso repositório Subversion, faz “Checkout” (do Tag que desejarmos ou do “trunk”) e aciona o Maven para fazer a montagem de acordo com a configuração do nosso “pom.xml”. Podemos também adicionar parâmetros, incluindo quais “profiles” Maven devem ser ativados.

Com isto, o Continuum também faz “deploy” do pacote do aplicativo, instalando-o de acordo com a configuração do “pom.xml” (tag “<distribution-Management>”), ou seja, o fluxo de número 8 da figura anterior.

Na verdade, ele pode fazer mais do que isto – por exemplo, a liberação final do projeto (Release).

## Gerenciando nosso projeto

O primeiro passo para promover a integração contínua é adicionar o projeto ao Continuum. Para isto, faça “logon” com a conta de administrador e selecione “Adicionar projeto / Projeto Maven 2”. A seguinte tela aparecerá:



Ilustração 29: Cadastramento do projeto

Você pode fazer “upload” do pom diretamente, informando o path do arquivo a partir de sua cópia de trabalho. Se fizer isto, preste bastante atenção para ver se o “pom.xml” está correto. É só selecionar o arquivo clicando no botão correspondente. Você também pode informar a URL onde está o “pom.xml” (normalmente a do SVN), só que esta opção só funciona bem com WebDAV.

Antes de cadastrar seu projeto, vamos dar uma “arrumadinha” na URL do SVN. O formato para URLs no Continuum (e no Maven e no Archiva) é o seguinte:

```
scm:<provedor>:<url nativa do provedor>/<pasta do projeto>
```

Como usamos o protocolo “svn”, nossa URL nativa é: “svn://localhost/projetoMicroblog/trunk/microblog”. É a URL onde fica o “pom.xml”, ou seja, na raiz do projeto. De acordo com o padrão de URL, a nossa deverá ficar assim:

scm:svn:svn://localhost/projetoMicroblog/trunk/microblog

Porém, é necessário informar um código de usuário e uma senha para que o Continuum acesse o repositório. Quando usamos WebDAV, não precisamos alterar a URL do repositório, informando o usuário e a senha apenas no Continuum. Mas, neste caso, é necessário informar na própria URL. Para proteger o projeto, crie um usuário com direito apenas de leitura no Subversion. Eis a URL apropriada:

scm:svn:svn://**cleuton:teste**@localhost/projetoMicroblog/trunk/microblog

Acrescentamos um nome de usuário e senha para acesso ao repositório Subversion. Se estiver em outra máquina da rede, é só fornecer o nome (ou IP) em vez de “localhost”.

Bem, o Continuum exige que nosso “pom.xml” tenha um tag “<scm>”, então vamos alterá-lo para incluir as informações necessárias:

```
<scm>
    <connection>scm:svn:svn://localhost/projetoMicroblog/trunk</
connection>
    <developerConnection>scm:svn:svn://localhost/projetoMicroblog/trunk</
developerConnection>
    <url>svn://localhost/projetoMicroblog/trunk</url>
</scm>
```

Se você não quiser colocar o usuário e a senha dentro do “pom.xml”, então terá que configurar no próprio Continuum.

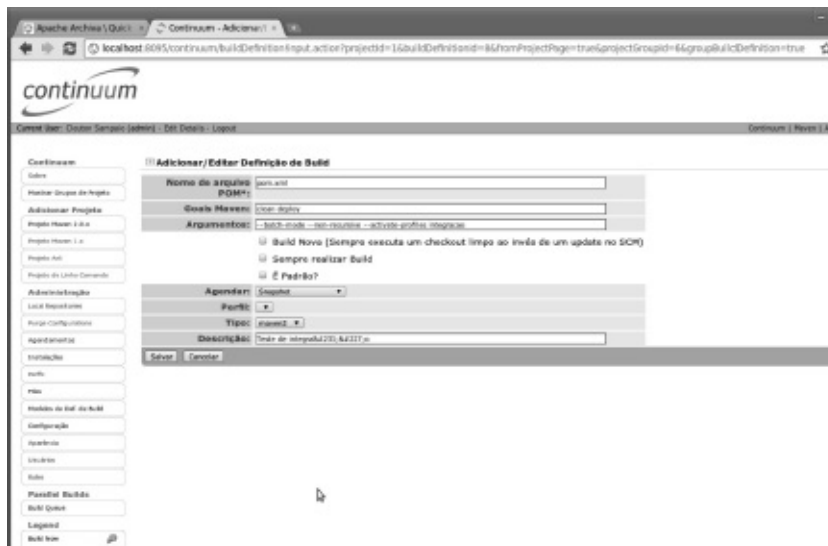
Após adicionar seu “pom.xml”, o Continuum criará um grupo de projetos para ele. Podemos selecionar o menu “Mostrar grupos de projeto”, selecionar nosso grupo e, no quadro “membros do grupo”, selecionar nosso projeto.



Se quisermos alterar a URL do repositório, basta clicar no botão “Editar” e alterar o campo “URL do SCM” para:

```
scm:svn:svn://cleuton:teste@localhost/projetoMicroblog/trunk/microblog
```

Bem, agora temos que definir as configurações de “build”. Edite o projeto, vá até a parte de “Definições de build” e clique no botão “Adicionar”.



Informe os campos:

- “Goals Maven”: o nome dos “goals” que você quer executar, em ordem cronológica e separados por espaços;
- “Argumentos”: os argumentos que deseja passar ao Maven. Note o “--activate-profiles integracao”, que indica qual “profile” Maven queremos que esteja ativo nessa montagem. Neste caso, queremos apenas executar os testes de integração, conforme já mostrei anteriormente;
- “Build nova”: se você quer fazer um “Checkout” toda vez que ela for executada, ou se basta fazer um “update” na cópia de trabalho do Continuum. Eu recomendo que sempre deixe marcada esta opção;
- “Sempre realizar build”: ao marcar esta opção, você diz ao Continuum para sempre realizar o “build”, independentemente de terem sido feitas alterações no projeto (dentro do Subversion);
- “Agendar”: você indicará o agendamento que o Continuum deve utilizar para executar esta montagem.

O agendamento pode ser criado na opção “Agendamentos”.

Expressão Cron:	Segundo:	<input type="text" value="0"/>
	Minuto:	<input type="text" value="0"/>
	Hora:	<input type="text" value="*"/>
	Dia do Mês:	<input type="text" value="*"/>
	Mês:	<input type="text" value="*"/>
	Dia da Semana:	<input type="text" value="?"/>
	Ano [opcional]:	<input type="text"/>

Informe a expressão cron. O formato está descrito aqui: [Sintaxe](#)

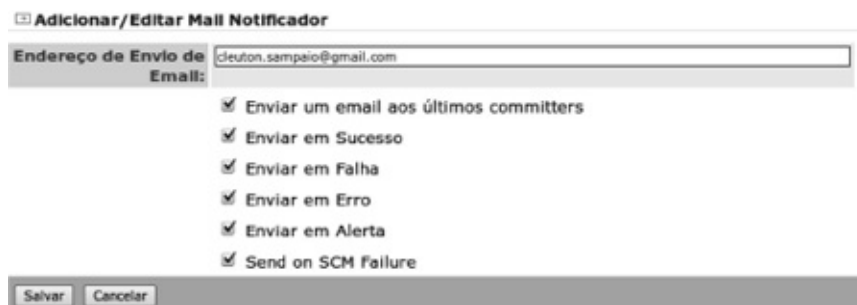
Ilustração 32: Agendamentos no Continuum

Do modo em que está, este agendamento está previsto para rodar a cada hora do dia. É uma expressão “cron”, e a sintaxe é um pouco complexa. Há um problema com a URL informada no link “Sintaxe”, e o endereço correto é: <http://www.quartz-scheduler.org/documentation/quartz-2.1.x/tutorials/crontrigger>.

Para simplificar, um asterisco no campo significa “a cada...” e um zero significa “qualquer”. Neste caso, este agendamento está previsto para ocorrer a cada hora, independentemente do minuto, segundo, dia do mês, da semana etc.

## Adicionando notificadores

O Continuum pode notificar os desenvolvedores utilizando vários meios: Jabber, IRQ, MSN ou e-mail. Para adicionar uma notificação, basta ir até a parte “Notificadores” do projeto e adicionar um notificador:



**Adicionar/Editar Mail Notificador**

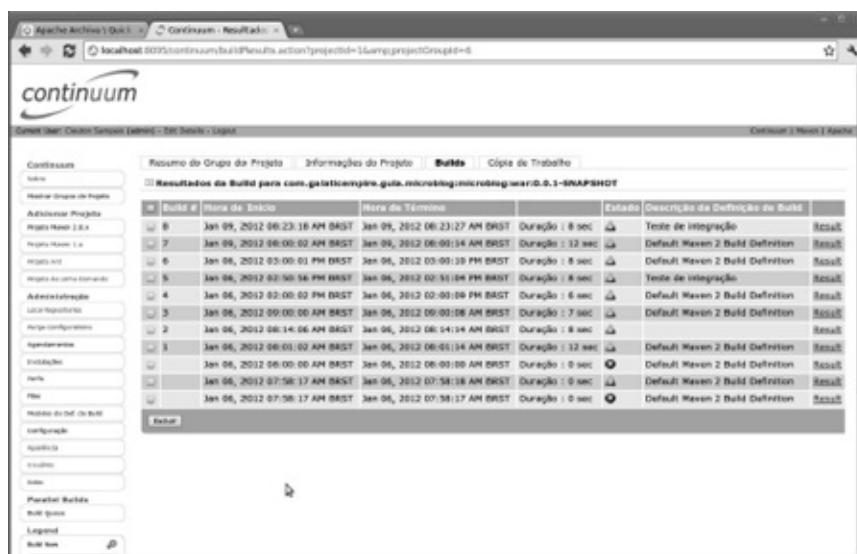
Endereço de Envio de Email:

- ☒ Enviar um email aos últimos committers
- ☒ Enviar em Sucesso
- ☒ Enviar em Falha
- ☒ Enviar em Erro
- ☒ Enviar em Alerta
- ☒ Send on SCM Failure

Ilustração 33: Adicionando notificadores

## Verificando Builds

Podemos acionar manualmente os “builds” ou deixar que eles ocorram de acordo com o agendamento. Para ver os resultados, podemos consultar os “builds” do projeto. Para isto, basta acessar o grupo do projeto e selecionar qual é o projeto membro.



Build #	Name de Início	Hora de Término	Duração	Estado	Descrição da Definição de Build	Result
6	Jan 05, 2012 06:23:18 AM BRST	Jan 05, 2012 06:23:27 AM BRST	Duração : 8 sec	Success	Teste de integração	Result
7	Jan 05, 2012 06:00:02 AM BRST	Jan 05, 2012 06:00:14 AM BRST	Duração : 12 sec	Success	Default Maven 2 Build Definition	Result
6	Jan 05, 2012 05:00:01 PM BRST	Jan 05, 2012 05:00:10 PM BRST	Duração : 8 sec	Success	Default Maven 2 Build Definition	Result
5	Jan 05, 2012 02:50:56 PM BRST	Jan 05, 2012 02:51:04 PM BRST	Duração : 8 sec	Success	Teste de integração	Result
4	Jan 05, 2012 02:00:02 PM BRST	Jan 05, 2012 02:00:09 PM BRST	Duração : 6 sec	Success	Default Maven 2 Build Definition	Result
3	Jan 05, 2012 09:00:00 AM BRST	Jan 05, 2012 09:00:06 AM BRST	Duração : 7 sec	Success	Default Maven 2 Build Definition	Result
2	Jan 05, 2012 08:14:06 AM BRST	Jan 05, 2012 08:14:14 AM BRST	Duração : 8 sec	Success	Default Maven 2 Build Definition	Result
1	Jan 05, 2012 06:01:02 AM BRST	Jan 05, 2012 06:01:14 AM BRST	Duração : 12 sec	Success	Default Maven 2 Build Definition	Result
	Jan 05, 2012 06:00:00 AM BRST	Jan 05, 2012 06:00:00 AM BRST	Duração : 0 sec	Success	Default Maven 2 Build Definition	Result
	Jan 05, 2012 07:58:17 AM BRST	Jan 05, 2012 07:58:18 AM BRST	Duração : 0 sec	Success	Default Maven 2 Build Definition	Result
	Jan 05, 2012 07:58:17 AM BRST	Jan 05, 2012 07:58:17 AM BRST	Duração : 0 sec	Success	Default Maven 2 Build Definition	Result

Ilustração 34: Builds do projeto

Com estes resultados, podemos conferir o que está acontecendo com o projeto ao longo do tempo.

Se quisermos especificar qual é o “branch” ou “tag” que o Continuum deverá

utilizar, então podemos fazer isto na tela de edição do projeto.

## Conclusão

O Continuum é uma ótima opção para implementarmos a integração contínua, compilando, testando e avisando aos desenvolvedores em intervalos regulares. Podemos também distribuir o resultado da montagem em um repositório Archiva.

Ainda existe muito a explorar sobre o Maven e o Continuum; por exemplo, o controle de release. Porém, vamos deixar isto para estudos posteriores, já que foge um pouco ao escopo do livro.

## Analizando a qualidade do código-fonte

Conforme já mostrei a vocês (o plugin “Eclipse Metrics”), é interessante fazermos análises estáticas do código-fonte de forma automatizada. Podemos aproveitar o Maven e o Continuum e executar estas análises periodicamente, permitindo aos desenvolvedores consultar os resultados.

## Geração de informações do projeto

O Maven é capaz de gerar um site para o projeto contendo informações importantes inclusive sobre a integração contínua. Se entrarmos na pasta do projeto e digitarmos “mvn site” (vai demorar um pouco na primeira vez), ele criará um website completo dentro da pasta “target/site” do projeto. Eis o resultado para o nosso projeto exemplo:



Ilustração 35: Site padrão do projeto

Meio decepcionante, não? É claro que ele tem informações úteis, como as



dependências, o repositório de fontes etc. Mas nós não demos informações suficientes ao Maven para que ele criasse um site completo. Para começar, podemos fazer algumas alterações em nosso “pom.xml”.

## Nome e resumo

Vamos preencher os tags abaixo em nosso “pom.xml”:

```
<name>Microblog</name>
<description>Projeto exemplo do livro</description>
```

## Desenvolvedores

Podemos e devemos adicionar os nomes dos desenvolvedores e de quaisquer outras pessoas da equipe do projeto:

```
<developers>
  <developer>
    <id>cleuton</id>
    <name>Cleuton Sampaio</name>
    <email>cleuton.sampaio@galaticempire.com</email>
    <url>http://www.galaticempire.com</url>
    <organization>Galatic Empire Software</organization>
    <organizationUrl>http://www.galaticempire.com</organizationUrl>
    <roles>
      <role>architect</role>
      <role>developer</role>
    </roles>
    <timezone>-3</timezone>
    <properties>
    </properties>
  </developer>
  <developer>
    <id>fulano</id>
    <name>Fulano de Tal</name>
    <email>fulano.de.tal@galaticempire.com</email>
    <url>http://www.galaticempire.com</url>
    <organization>Galatic Empire Software</organization>
    <organizationUrl>http://www.galaticempire.com</organizationUrl>
    <roles>
      <role>developer</role>
    </roles>
    <timezone>-3</timezone>
    <properties>
    </properties>
  </developer>
</developers>
```

## Integração contínua

É interessante apontarmos para nosso servidor de integração contínua, de modo que as informações fiquem interligadas:

```
<ciManagement>
  <system>continuum</system>
  <url>http://127.0.0.1:8095/continuum</url>
  <notifiers>
    <notifier>
      <type>mail</type>
      <sendOnError>true</sendOnError>
      <sendOnFailure>true</sendOnFailure>
      <sendOnSuccess>true</sendOnSuccess>
      <sendOnWarning>true</sendOnWarning>
      <configuration><address>cleuton.sampaio@galaticempire.com</
address></configuration>
    </notifier>
  </notifiers>
</ciManagement>
```

Os notificadores serão adicionados automaticamente pelo Continuum, bastando acrescentar “site” aos “goals” da configuração de “build”.

## Distribuição do website Maven

Por padrão, o Website é sempre gerado dentro da pasta “target/site”, mas você pode mudar isso. Ao executar o “goal” “mvn site:deploy”, o conteúdo do site pode ser distribuído para um servidor web remoto, ficando disponível para acesso geral. Para isto, inclua as configurações no tag “distributionManagement”, do seu “pom.xml”:

```
<distributionManagement>
  <site>
    <id>website</id>
    <url>scp://www.galaticempire.com/project</url>
  </site>
</distributionManagement>
```

Podemos usar o protocolo “scp” ou “file” diretamente, além de podermos configurar o plugin “maven-site-plugin” com vários outros argumentos (<http://maven.apache.org/plugins/maven-site-plugin/deploy-mojo.html>).

Uma configuração simples e funcional é colocar o site no mesmo servidor onde o Continuum está rodando utilizando uma porta diferente.

## Adicionando relatórios de qualidade

A geração de relatórios de inspeção, ou análise estática, é feita na seção “reporting” do nosso “pom.xml”. Existem vários plugins Maven que fazem análises do código, incluindo a cobertura de testes.

Para começar, adicione estes relatórios ao seu “pom.xml”, criando a seção “reporting” logo depois da seção “build”:

```
</build>

<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.5</version>
      <configuration>
        <aggregate>true</aggregate>
      </configuration>
    </plugin>

    <plugin>

      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <targetJdk>1.6</targetJdk>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</
<groupId>
      <version>2.5.1</version>
    </plugin>
```

```

</plugins>
</reporting>

<scm>

```

Ao rodar o “goal” “site” temos mais uma seção no nosso menu:

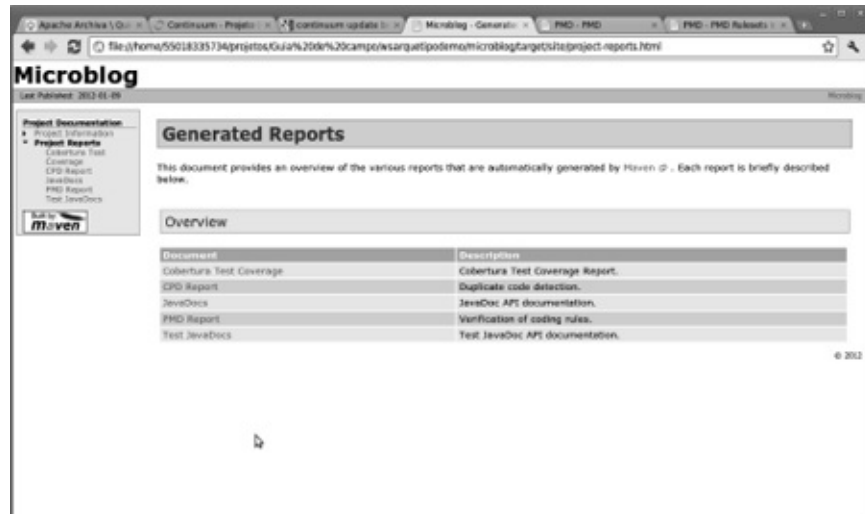


Ilustração 36: Relatórios automáticos

O “JavaDoc” é bem óbvio. O CPD (Code Duplication) e o PMD são gerados pelo plugin “maven-pmd-plugin”. O PMD (<http://pmd.sourceforge.net/>) é uma ferramenta bem conhecida para análise de código-fonte, com um grande banco de regras configuráveis.

O “Cobertura” é um plugin bem interessante que analisa a cobertura dos testes existentes no projeto:

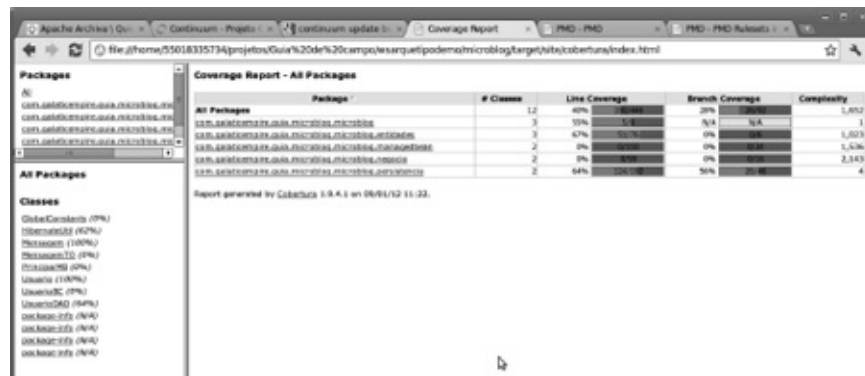


Ilustração 37: Relatório de cobertura

Notamos que o projeto apresenta baixa cobertura de testes em alguns pacotes, logo, é melhor analisar e ver o que está faltando ser testado.

## Rodando o “Cobertura” para testes de integração

Existe uma versão do plugin “Cobertura” para rodar sobre os testes de Integração (<http://code.google.com/p/cobertura-it-maven-plugin/wiki/HowToUse>), mas a documentação é muito ruim e ele sequer faz parte do repositório central do Maven. Logo, recomendo usar o mesmo plugin para estes testes. Para isto, temos acrescentar o “plugin” no “profile” “integracao” em nosso POM:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>cobertura-maven-plugin</artifactId>
  <version>2.5.1</version>
  <configuration>
    <formats>
      <format>xml</format>
    </formats>
  </configuration>
  <executions>
    <execution>
      <id>cobertura-check</id>
      <phase>verify</phase>
      <goals>
        <goal>cobertura</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

## Utilizando o SONAR

Apesar dos plugins individuais serem interessantes, existem softwares que fornecem relatórios mais ricos, como o SONAR (<http://docs.codehaus.org/-display/SONAR/Documentation>), que também possui um plugin Maven (<http://mojo.codehaus.org/sonar-maven-plugin/examples/use-enterprise-database.html>).

O uso do SONAR certamente beneficiará o(s) seu(s) projeto(s), pois é uma ferramenta mais completa e com relatórios mais interessantes.

## Baixe, rode e pronto!

Só um momento! O SONAR usa o banco de dados Apache Derby, por default. Logo, se você usar o SONAR em sua configuração “default”, poderá haver conflito com o banco de dados do sistema exemplo, pois ambos usam a porta 1527. Se quiser evitar problemas, altere a porta padrão do Derby para outro valor, digamos 3527. Veja como fazer isto em:

<http://db.apache.org/derby/docs/dev/-adminguide/tadminappssettingportnumbers.html>

Usar o SONAR é extremamente simples: baixe, inicie o serviço e mande rodar o “goal” Maven. Pronto! Seu software foi analisado e passará a fazer parte do banco de dados do Sonar. Eis o passo a passo, como descrito na documentação do SONAR (<http://docs.codehaus.org/display/SONAR/The+2+minutes+tutorial>):

1. Baixe o SONAR: <http://www.sonarsource.org/downloads/>;
2. Descompacte o zip;
3. Rode o script para iniciar o serviço: “<pasta sonar>/bin/<seu sistema operacional>/sonar.sh console” ou então: “<pasta sonar>\windows-x86-XX\StartSonar.bat”;
4. Na pasta do seu projeto Maven, onde fica o “pom.xml”, rode os dois comandos: “mvn clean install” e depois: “mvn sonar:sonar”;
5. Abra um navegador e digite: “http://localhost:9000”, e selecione o seu projeto.



Ilustração 38: Relatório do SONAR

O relatório possui vários links para diversos tipos de análises. Podemos ver os problemas de acordo com o nível (“Blocker”, “Critical”, “Major”, “Minor” e “Info”) ou podemos ver os detalhes da cobertura de código ou da complexidade das classes e métodos.

O mais “legal” é que o SONAR possui uma grande biblioteca de plugins, com diversos tipos de análises diferentes. Ela pode ser acessada em: <http://docs.codehaus.org/display/SONAR/Sonar+Plugin+Library/>.

O plugin “Quality Index” (<http://docs.codehaus.org/display/SONAR/Quality+Index+Plugin>) é muito interessante, pois avalia a qualidade geral do projeto, com base nas métricas de:

1. “Coding violations”: índice de aderência às regras do PMD;
2. “Complexity”: fator de densidade da complexidade ciclomática dos métodos;
3. “Test Coverage”: cobertura de testes;
4. “Style violations” : índice de aderência às regras do Checkstyle.

Índice de qualidade =  $10 - 4.5 * \text{Fator1} - 2 * \text{Fator2} - 2 * \text{Fator3} - 1.5 * \text{Fator4}$ .

Para usar o “Quality Index” basta fazer o download do JAR, copiá-lo para:

“<sonar>/extensions/plugins/” e reiniciar o SONAR (<sonar>/bin/<sistema>/sonar.sh restart). Depois, mandamos rodar novamente (“mvn sonar:sonar”) e abrimos a console (<http://localhost:9000>).

## Ué? Cadê o “Quality Index”?

Bem, temos que configurar o SONAR para que o resultado do plugin apareça no “Dashboard”:

1. Faça login como “admin / admin”;
2. Visualize o “Dashboard” e clique no link “Configure widgets”;
3. Selecione as informações que deseja adicionar à sua Dashboard;
4. Clique no link “Back to dashboard”.



Ilustração 39: Adicionando informações ao relatório

## Integrando o SONAR ao profile Maven

Podemos utilizar o plugin SONAR do Maven para analisar projetos durante os “builds” da Integração Contínua atualizando o banco de dados do Sonar. Para isto, basta criar uma nova definição de “build” no Continuum com os “goals”: “clean install sonar:sonar”. Se quiser avaliar os testes de integração contínua, use “clean deploy sonar:sonar”.

O plugin Maven vai atualizar o banco de dados do Sonar, conforme a mensagem abaixo (do log de “build”):

[INFO] [08:37:44.043] ANALYSIS SUCCESSFUL, you can browse <http://localhost:9000>



## Como melhorar a qualidade

Já mostrei mais de um meio para analisar métricas do projeto (“Eclipse Metrics”, Plugins do Maven e o SONAR). Agora é hora de saber o que fazer com tantos relatórios... eu gostaria de sugerir algumas mudanças no processo de SCM que você utiliza, de modo a incluir a análise de qualidade como uma das atividades principais.

### Análises em três níveis

Talvez seja a melhor abordagem para análise de qualidade de código. Primeiramente, temos o Engenheiro de software que está criando o código-fonte. Ele deve utilizar ferramentas simples, como o “Eclipse Metrics”, de modo a evitar sobrecarga em sua estação de trabalho, especialmente se estiver usando Máquinas Virtuais (como eu recomendei). Estas métricas servem para fundamentar a decisão de fazer ou não “Commit”, e eu sugiro que os resultados sejam mantidos (pelo menos os da última análise).

Depois, no momento da Integração Contínua, devemos rodar o SONAR dentro de um “build” do Continuum. Assim, estaremos atualizando o relatório geral do projeto de maneira constante e automatizada. Podemos até incluir as métricas dentro do Site Maven, criando uma seção especializada nele (<http://maven.apache.org/guides/mini/guide-site.html>). Este segundo nível é utilizado para análise de evolução do software. O SONAR tem uma visão chamada “Time Machine”, que mostra a evolução do software em várias análises periódicas.

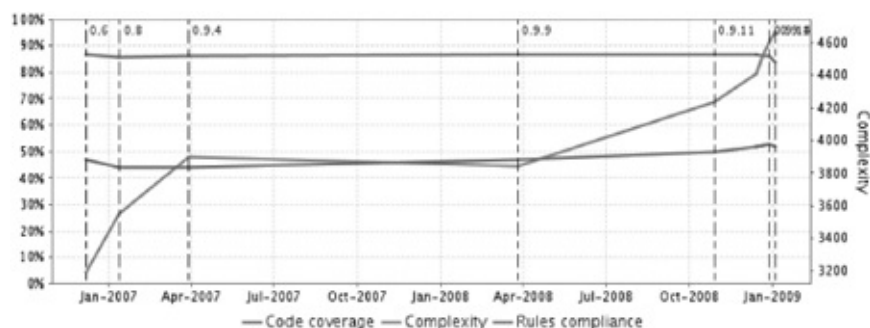


Ilustração 40: O “Time Machine” do SONAR

O objetivo desta análise é verificar a evolução do trabalho de melhoria de qualidade do projeto. Se um indicador ruim não estiver evoluindo, pode ser sinal de que a equipe esteja ignorando a qualidade, ou então um falso-positivo. De

qualquer forma, cabe uma verificação destes índices.

O terceiro e último nível seria a última análise ANTES da revisão técnica. Como eu disse anteriormente, considero a revisão técnica uma ferramenta fundamental para melhoria da qualidade, pois agrega outras visões ao resultado do trabalho. Os revisores deveriam ter acesso ao último relatório, assim como à “Time Machine”, de modo a fundamentar sua opinião técnica.

Este último nível deve subsidiar o Gerente de Liberação na tomada de decisão de quais funcionalidades incluir em quais “releases” do software, criando, desta forma, um “roadmap” mais seguro para a sua evolução.

### **Evolução constante**

Martin Fowler diz que devemos constantemente refatorar o código-fonte. A técnica TDD diz que devemos escrever código para fazer os testes funcionarem, refatorando-o continuamente até que estejamos satisfeitos com sua qualidade.

Uma boa medida é sempre fazer revisões de código, refatorando-o para melhorar a manutenibilidade.

As análises estáticas, fornecidas pelo SONAR, “Eclipse Metrics” ou os plug-ins do Maven, devem ser utilizadas para sabermos quais os pontos de maior prioridade. De cara, eu atacaria dois pontos importantes: cobertura de testes e complexidade.

Se um método apresenta alta complexidade, talvez o problema esteja na própria classe, pois pode ser que esteja assumindo mais de uma responsabilidade. Refatorar pode significar criar mais classes ou mais métodos (substituir condicional por polimorfismo ou por herança), aumentando o risco do código-fonte, mas sempre gera resultados mais positivos.

Cobertura de testes é um assunto mais complicado, já que temos dois níveis (Unitários e Integração) e alguns itens que não podem ser testados diretamente (alto acoplamento com o ambiente). Temos que estabelecer metas alcançáveis e nunca abrir mão do teste manual, que é o verdadeiro “fiel da balança”.

O que não se pode fazer é ignorar problemas. Assim que notar alguma coisa

errada, mesmo que seja na sua própria estação (“Eclipse Metrics”), o desenvolvedor deve anotar para fazer um refactoring posterior. O código só pode ser liberado (“commit”) se estiver em condições para isto.

## 10.

# FCQDC

Para terminar, gostaria de deixar algumas palavras sobre tudo isso.

Eu tinha um chefe que sempre dizia: “F aça C erto Q ue D ará C erto”! Sempre que havia um problema, lá vinha ele com o “**FCQDC**”... Fazer Certo significa: começar certo, continuar certo e terminar certo. Eu não sei o que mais odiava, se era a sigla ou ele... De tanto odiar, passei a tomar cuidado extremo, de modo a evitar ouvir “**FCQDC**” novamente!

Esta é a ideia geral do livro: mostrar o caminho certo, para que você não tenha que ouvir críticas idiotas, até porque críticos de plantão sempre existem, mas pessoas dispostas a ajudar são raras.

Isto não quer dizer que eu faço tudo certo, ou que tudo o que eu digo é certo, longe disso... se você leu com atenção, então deve ter notado a imensa quantidade de referências que eu busquei. Isto é uma prova de que eu não sou o “dono da verdade”. Meu trabalho maior foi compilar o “estado da arte” e as boas práticas, criando um guia prático para você, evitando que apanhe como eu apanhei para aprender tudo isso.

Tempo bom, lê-lê, não volta mais. Saudade... de outros tempos de paz!

(Lilico, comediante. 1937-1998)

Quando eu comecei a desenvolver software, lá pelos idos de 1977, tudo era muito mais simples. Você aprendia uma linguagem de programação, geralmente COBOL, um sistema operacional, geralmente IBM OS (JCL) e pronto! Porém, mesmo assim, sistemas de baixa qualidade davam muito trabalho. Mesmo naquela época havia técnicas para projeto e construção de código visando melhorar a manutenibilidade, como: programação estruturada, por exemplo.

Antigamente havia limitação de tempo até para compilar programas, então recoríamos a boas práticas e testes manuais (computador “chinês”) para diminuir o tempo de CPU gasto para depurar os programas. Antes de rodar um programa fazíamos diversos testes lógicos para antecipar problemas e, quando aconteciam, nós analisávamos “dumps” de memória para descobrir o que teria acontecido. Era comum que os programas compilassem e rodassem sem problemas, gastando pouco tempo de computador, pois o hardware era mais caro que os salários dos programadores.

Depois veio a era dos microcomputadores: baratos e amplamente disponíveis. As linguagens xBase dominaram o mercado (Clipper, dBase etc.), com sua facilidade de uso e relativo poder de programação. A mudança em relação aos sistemas mainframe era enorme, especialmente a facilidade de compilar e testar muitas vezes. Como não existiam as limitações, era mais fácil, bastando compilar, testar, alterar, compilar, testar novamente, alterar novamente, até que ficássemos satisfeitos. Era o reino da “tentativa e erro”, pois os salários começaram a ficar mais caros que o hardware.

Um tempo depois, surgiram os ambientes gráficos (Windows) e as linguagens de programação orientadas a eventos, como Microsoft Visual Basic, Borland Delphi, entre outras. O paradigma mudou novamente e novos complicadores surgiram, como a orientação a objetos. A evolução das IDEs foi enorme, com ambientes gráficos e depuração passo a passo, na qual era possível ver e alterar o conteúdo de variáveis, mudando também o fluxo de execução ao nosso bel-prazer.

Sempre que havia uma melhoria na linguagem ou no ambiente, a consequência era o desleixo dos programadores, pois era mais barato o método da “tentativa e erro” do que perder tempo tentando pensar.

Só que isso vem mudando novamente...

*All is connected... not one thing can change by itself.*

(“Tudo está conectado... nada muda por si só” – Tradução do autor)

Paul Hawken

A antiga Sun Microsystems (agora Oracle) tinha um slogan: ***The network is the computer***, ou seja: “a rede é o computador”. Tudo está conectado mesmo, com

cada elemento influenciando e sendo influenciado por outros. É claro que a frase de Paul Hawken não se referia especificamente aos sistemas aplicativos, mas nos ajuda a entender melhor o “ecossistema” digital atual.

Há exatos dez anos eu dei uma palestra para desenvolvedores na qual disse que nenhum sistema moderno poderia desconsiderar a integração com a Internet em seu projeto. Hoje são as redes sociais! Com isto, a complexidade do software aumentou muito, porém a atenção à qualidade do software não acompanhou a evolução. Eu estimo que um sistema aplicativo moderno é dez vezes mais complexo do que um antigo sistema cliente-servidor, típico dos anos 90. Não dá para acertar só com “tentativa e erro”...

Esta defasagem entre evolução e qualidade não é causada por falta de métodos ou ferramentas. Talvez seja por falta de importância! Todos adoram falar nas últimas novidades de frameworks ou componentes, mas evitam discutir assuntos “chatos”, como: Cobertura de Testes, Acoplamento e Coesão de Classes e outros assuntos relacionados à qualidade. Muita gente gosta de fugir do trabalho chato e repetitivo, procurando ir para atividades mais criativas.

### **20% das críticas podem abalar 80% da funcionalidade**

Desenvolver com qualidade é trabalhoso... muitas regras e práticas a serem seguidas. O legal é arrastar alguns “widgets”, escrever qualquer coisa e mandar rodar. Testar também é chato... ficar criando casos de teste “idiotas”, os quais temos certeza que jamais serão feitos na vida real, é perda de tempo. Pode ser... pode ser que seja certo ao pensar assim, afinal, a vida é muito curta para perder tempo com coisas “chatas”.

O problema é quando o software vai para as “prateleiras”... se tiver tempo, vá ao Android Market e observe as críticas que os usuários fazem aos aplicativos. Você vai se surpreender! Muitos programas bacanas acabam sendo prejudicados por poucos problemas. É um “Pareto” invertido, no qual 20% dos problemas acabam prejudicando 80% da funcionalidade. Bastam algumas pequenas falhas para “manchar” a reputação de um bom programa.

E o que causa essas pequenas falhas? A falta de atenção às coisas “chatas”... uma das coisas mais chatas que existe é o processo de SCM ( Software Configuration Management), especialmente a rotina de liberação de uma versão. No último trabalho que fiz, quase comprei briga com o usuário porque eu insisti

em realizar um teste “beta”, com usuários reais. A pressa de lançar o produto era tão grande que eles não conseguiam entender a importância de fechar o ciclo de testes adequadamente.

Aliás, o mercado “mobile”, representado principalmente pelas plataformas Android e iOS (Apple), tem ensinado a muitos desenvolvedores uma amarga lição: qualidade vende software! Eu leio as críticas no Android Market e no iTunes e vejo o que os usuários dizem sobre os programas. Veja só algumas “pérolas”:

- “...então o programa funciona 3 ou 4 vezes, depois eu tenho que derrubar o aplicativo. Que porcaria de programa! Quero meu dinheiro de volta!” - sobre um programa de compartilhamento de fotos;
- “...muito bom e completo, só que não funciona!” - sobre um jogo;
- “Muito confuso e sem manual. Detestei!” - sobre um utilitário.

Conheço alguns desenvolvedores que começaram a criar aplicações móveis e afirmaram que o controle de qualidade tem que ser ainda maior, pois os consumidores são mais exigentes.

E pode ter certeza de uma coisa: nunca deixe um “...porém...” prejudicar o seu programa. São esses probleminhas pequenos que abalam a reputação de um bom software.

## **O princípio do “tubarão”**

Bem, eu acho que já chega, “né”? Só para terminar mesmo, vou dar um conselho final a você.

Entre as bobagens que eu invento, estão os meus “princípios profissionais”, são práticas que eu costumo seguir na minha carreira, que já me livraram de vários problemas. Entre elas está o que eu chamo de “princípio do tubarão”. Eu li certa vez que os tubarões não podem parar de nadar porque afundam e morrem, logo, fiz uma analogia com o comportamento de um desenvolvedor, que precisa estar constantemente estudando, caso contrário fica desatualizado e “morre” profissionalmente.

Ao contrário de outras profissões, é muito difícil manter-se atualizado dentro da área de TI. As mudanças são constantes e, muitas vezes, conflitantes. Certa vez eu ouvi dizer que um médico não pode ficar mais de dois anos sem ir a um congresso ou se atualizar. Imagine nós, desenvolvedores, ficarmos dois anos sem nos atualizarmos?

Nos últimos dois anos estudei muitas coisas, como: arquitetura ARM, desenvolvimento Android, iOS, entre diversas outras tecnologias, e ainda acho que estudei pouca coisa. Porém, vejo colegas meus que ainda estão no mesmo ponto que estavam há dez ou vinte anos! Tem gente que ainda fica programando em COBOL ou “Clipper”, sem procurar evoluir. O que esperam da vida? Aposentar-se?

Se eu posso lhe dar algum conselho útil seria o seguinte: nunca pare de estudar! Procure se atualizar de vez em quando, faça concursos públicos, leia bastante. Agora com o Kindle for PC (Windows ou Mac), você pode comprar livros na Amazon por um preço muito mais baixo, evitando pagar a taxa de entrega. Leia muito, teste muito e estude muito, pois só isso fará de você um bom profissional.

Lembre-se: “There is no free lunch!” (“Não tem almoço grátis!”, tudo tem um preço).



# Referências bibliográficas

ALUR, D., Crupi, J., Malks, D. (2003). “Core J2EE Patterns”. 2a ed. Prentice Hall.

BECK, Kent (2003) “Test-driven development: by example”. Pearson Education.

BLOCH, Joshua (2008) “Java Efetivo”. 2a ed. Alta Books.

CHIDAMBER, S., R., Kemerer, C., K (1994) “A Metrics Suite for Object Oriented Design”. IEEE Trans. on Software Eng., Vol.20, No.6.

FOWLER, Martin (2004) “Refatoração – Aperfeiçoando o projeto de código existente”. Porto Alegre. Bookman.

GAMMA, E; Helm, R., Johnson, R., Vlissides, J. (1995) “Design Patterns – Elements of Reusable Object-Oriented Software”. Addison Wesley.

HENDERSON-Sellers (1996) “Object-Oriented Metrics: Measures of Complexity”. Prentice Hall.

IBM [Rational Unified Process (RUP)]. Disponível: <http://www-01.ibm.com/software/awdtools/rup/>.

IEEE (2004) [Guide to the Software Engineering Body of Knowledge (SWEBOK)]. Disponível: <http://www.computer.org/portal/web/swbok/htmlformat>.

ISO 9126. [ISO/IEC 9126-1:2001 Software engineering -- Product quality -- Part 1: Quality model]. Disponível: [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=22749](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749)

JCP. [Java Community Process]. Disponível: <http://jcp.org/en/home/index>.

LAPRIE, J. C. (1995) “Dependable Computing and Fault Tolerance: Concepts

and terminology”. Fault-Tolerant Computing, 1995, ‘Highlights from Twenty-Five Years’, Twenty-Fifth International Symposium on. IEEE. Disponível: [http://ieeexplore.ieee.org/search/freesrchabstract.jsp?tp=&arnumber=532603&openedRefinements%3D\\*%26filter%3DAND%28NchField%3DSearch+All%26queryText%3DDependable+Computing+and+Fault+Tolerance%3A+Concepts+and+terminology](http://ieeexplore.ieee.org/search/freesrchabstract.jsp?tp=&arnumber=532603&openedRefinements%3D*%26filter%3DAND%28NchField%3DSearch+All%26queryText%3DDependable+Computing+and+Fault+Tolerance%3A+Concepts+and+terminology)

MARTIN, Robert C. [Principles Of Object Oriented Design]. Disponível: <http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>.

OASIS. [Standards]. Disponível: <http://www.oasis-open.org/standards>.

Oracle JAAS [Java SE Security]. Disponível: <http://www.oracle.com/technetwork/java/javase/jaas/index.html>.

Oracle JavaDoc [How to Write Doc Comments for the Javadoc Tool]. Disponível: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>.

OWASP. [The Open Web Application Security Project]. Disponível: [https://www.owasp.org/index.php/Main\\_Page](https://www.owasp.org/index.php/Main_Page)

Page-Jones, Meilir (1988) “Projeto Estruturado”. McGraw-Hill.

PRESSMAN, Roger S (2006) “ENGENHARIA DE SOFTWARE”. 6a ed. McGraw. Hill/Nacional.

QFD Institute [Frequently Asked Questions about QFD]. Disponível: [http://www.qfdi.org/what\\_is\\_qfd/faqs\\_about\\_qfd.html](http://www.qfdi.org/what_is_qfd/faqs_about_qfd.html).

## Últimos Lançamentos

### Licenças Livres e Direitos Fundamentais

Hellor Medrado de Faria

120 pp. – R\$ 49,00

Este livro apresenta uma profunda pesquisa bibliográfica que, juntamente com estudos de casos, tem por objetivo verificar as relações entre as licenças livres, aquelas que se contrapõem aos Direitos Autorais brasileiros, influenciados pelo Instituto estrangeiro do "copyright", e os Direitos Fundamentais Constitucionais. E demonstra a necessidade da tutela do "copyright" dentro do ordenamento jurídico brasileiro, inclusive ensinando que seja tratado como um Direito Fundamental da 5ª geração.



### Compra Coletiva

Dallan Felipini

184 pp. – R\$ 39,00

(Série e-commerce Melhores Práticas)

Este livro é um guia prático para empreendedores, comerciantes e usuários que desejam explorar e utilizar de forma eficaz o fascinante mundo da compra coletiva. Após a leitura, você terá uma visão panorâmica da que é a compra coletiva, conhecerá o funcionamento do sistema sob a ótica de cada um dos seus personagens e saberá o que a compra coletiva pode trazer de bom para você e sua empresa.



### Clicando com Segurança

Edison Fontes

284 pp. – R\$ 67,00

Este livro apresenta assuntos do dia a dia da segurança da informação em relação às pessoas e em relação às organizações. Foi escrito para o usuário e também para o profissional ligado ao tema segurança da informação. No final do livro são identificados os requisitos de segurança da informação da Norma NBR ISO/IEC 27002 que sustentam ou motivam os textos, possibilitando assim a ligação da prática com a teoria.



### Administração do Windows Server 2008 R2 Server Core

Daniel Danda

452 pp. – R\$ 79,00

O conteúdo deste livro foi elaborado especificamente para profissionais de TI responsáveis por administrar o Windows Server 2008 R2 Server Core ou o Hyper-V Server 2008 R2 e é indicado também para planejadores e analistas de TI, administradores ou gerentes responsáveis pela implementação, segurança e gerenciamento de TI. Aproveite os exemplos do livro, que sempre vão além da proposta, e faça uso dos recursos apresentados.



### Microsoft Project 2010 Standard e Professional

Ricardo Vargas

400 pp. – R\$ 159,00

Como transformar em sucesso os projetos utilizando o Microsoft Project 2010

O livro ensina novas funcionalidades como a falta de opções (Runt user Interface), como planejar e estruturar um projeto no Project 2010, interface de planilha na entrada de dados, agendamento controlado pelo usuário, novas maneiras de formação de dados, múltiplos projetos, análise de valor agregado no Project 2010 e muito mais. Inclui o código de acesso à área do leitor.



## Criação de Sites na era da Web 2.0

Diego Brito

220pp. - R\$ 59,00

Este livro cobre as etapas de Atendimento, Planejamento, Arquitetura de Informação, Redação, Criação, Desenvolvimento e Otimização (SEO) – sete áreas fundamentais para que você tenha uma visão ampla de todo o processo de criação de um site. Este livro é indicado para web designers, estudantes de Comunicação, empresários, diretores de arte, arquitetos de Informação e Gerentes de Marketing.



## Mente Anti-hacker – Proteja-se!

Raulo Moraes

196pp. - R\$ 53,00

Esta obra tem por finalidade ajudar todos os usuários de computador que precisam de alguma forma ter proteção enquanto navegam na Internet, leem seus e-mails e usam seus computadores para outras atividades. Chaga de invasões, chaga de vírus, chaga de cair em golpes virtuais. É preciso ter uma MENTE ANTI-HACKER! PROTEJA-SE! Inclui apêndices sobre ataques a smartphones e tablets.



## Construindo um Blog de sucesso com o WordPress 3

Daniella Borges de Brito

228pp. - R\$ 59,00

Servocê procura entender tecnicamente como criar um blog com o WordPress este livro é para você. Um livro completo que aborda o assunto de forma didática e também prática, com uma leitura de fácil entendimento, exercícios propostos, entrevistas e explicações passo a passo. No CD que acompanha o livro você encontrará vídeos aulas explicativas e diversos aplicativos gratuitos e arquivos que irão auxiliá-lo na criação do seu blog.



## Java Enterprise Edition 6 – Desenvolvendo Aplicações Corporativas

Cleuton Sampaio

280pp. - R\$ 64,00

Neste livro, vamos os alguns aspectos inovadores, como: Profiles de aplicação, RESTful Web Services, JavaServer Faces 2.0, Servlet 3.0 e muitas outras novidades, que agora fazem parte do "cardápio" de soluções corporativas em Java. Além disso, o livro traz um exemplo completo: um sistema de notícias on-line, que utiliza os principais componentes Java EE: Web Services e Session Beans. Tudo simples e prático para que você obtenha resultados rapidamente.



## Virtualização – Componente Central do Datacenter

Manoel Veras

364pp. - R\$ 97,00

O objetivo principal deste livro é contribuir para a formação de arquitetos de DATACENTERS que possuam como elemento central a VIRTUALIZAÇÃO. O livro é modular e detalha diversos aspectos técnicos e práticos da VIRTUALIZAÇÃO. Como não poderia deixar de ser, trata dos conceitos de CLOUD COMPUTING e DATA CENTER, tendo como aspecto central destas estruturas a VIRTUALIZAÇÃO.



## Desenvolvendo Aplicações com UML 2.2 (3ª edição)

Ana Cristina Melo

340pp. - R\$ 79,00

Esta nova edição aborda os principais conceitos de orientação a objetos, a estrutura da UML, por meio de seus elementos, relacionamentos e diagramas, a transformação de modelos criados em UML em códigos escritos nas duas principais linguagens OO do mercado (Java e Delphi), diferenças entre as principais versões da UML (1.4, 2.0 e 2.2) e questões de como usá-la publicamente.



## Bacula – Ferramenta Livre de Backup

Heitor Medrado de Faria

208 pp. – R\$ 55,00

Único livro nacional dedicado à parte teórica de backups, também traz manuais completos de instalação e configuração de um sistema de cópias de segurança baseado na ferramenta mais utilizada no mundo – o Bacula (software livre). Aborda tópicos como estratégias de backup (GRS), restauração, cópia e migração de backups e scripts antes e depois dos trabalhos. O livro é baseado tanto em sistemas Linux como Microsoft.



## ActionScript 3.0 – Interatividade e Multimídia no Adobe Flash CS5

Fábio Flatschart

280 pp. – R\$ 72,00

(Série Web Conceitos & Ferramentas)

Aprenda neste livro os fundamentos da linguagem ActionScript 3.0 e saiba como empregá-la corretamente para adicionar interatividade e multimídia em seus projetos com o Adobe Flash CS5. Explora o workflow (fluxo de trabalho) com o Adobe Flash CS5 e o ActionScript 3.0 através de dicas para otimizar a produção e tornar o seu trabalho mais rápido e eficiente. Acompanha CD-ROM com os arquivos para a construção dos projetos mostrados no livro.



## Investigação e Perícia Forense Computacional

Claudemir Queiroz / Rafael Vargas

156 pp. – R\$ 45,00

A introdução de computadores como uma ferramenta criminal aumentou a capacidade dos criminosos para realizar, ocultar ou auxiliar a atividade legal ou antilegal. Este livro pretende apresentar as principais certificações para a formação de novos profissionais, além de auxiliar com as leis processuais mais utilizadas no mercado, com uma mistura de estudos de caso voltados para coleta e análise de dados tanto físicos como em rede.



## Criando Macros no BrOffice Calc

Cleuton Sampaio

208 pp. – R\$ 52,00

Aprenda a criar macros fantásticas, que podem manipular a planilha de várias formas, além de poder criar diálogos de interface com o usuário, gráficos comerciais: Pizza, Barra, Coluna, Linha, inclusive com efeitos 3D, e macros que criam novos arquivos e acessam bancos de dados. O livro contém vários exercícios resolvidos para você praticar o aprendizado.



## Programando para a Internet com PHP

Odairmir Martinez Bruno / Leandro Farias Estrozi /  
João do E. S. Batista Neto

332 pp. – R\$ 68,00

Este livro foi concebido para facilitar o aprendizado da programação voltada para a Internet. A linguagem PHP é a ferramenta mais popular para a programação de aplicações voltadas para servidores web. Simples e fácil, tem sido escolhida pela vastamania dos desenvolvedores. O livro é voltado para estudantes universitários, profissionais e hobbyistas, tendo como objetivo atender aos níveis iniciante e intermediário.



## Empreendedorismo na Internet

Dalton Felipini

224pp. – R\$ 45,00

(Série E-commerce Melhores Práticas)

Este terceiro título da coleção **E-commerce Melhores Práticas** foi desenvolvido para atender a uma necessidade apresentada por empreendedores e empresários interessados no mercado da Internet: escolher de forma criteriosa o segmento de atuação. O propósito deste livro é duplo: ajudar o empreendedor a encontrar um segmento de negócio compatível com seus interesses e fornecer critérios lógicos de avaliação do negócio.



## Programação Shell Linux 8ª edição

Julio Cezar Neves

588 pp. – R\$ 105,00

Inédito no mercado, este livro apresenta uma abordagem desconhecida da programação Shell dos sistemas operacionais Unix/Linux. Esta **oitava edição** foi atualizada com as novidades que surgiram no Bash 4.0, tais como o coproc (thread), vetores associativos, novas expansões de parâmetros e muito mais. Seu apêndice sobre Expressões Regulares foi bastante aprofundado e agora aborda também as diferenças do seu uso no Bash e no BcOffice.org. O CD-ROM que acompanha o livro contém todos os exercícios do livro resolvidos e alguns scripts úteis.



## Fundamentos do Gerenciamento de Serviços de TI – Preparatório para a certificação ITIL® V3 Foundation

Marcos André dos Santos Freitas

376 pp. – R\$ 92,00

Este livro tem como objetivo ser um material de conscientização, apresentação das práticas propostas pelo ITIL e preparação para a realização da prova de certificação ITIL® V3 Foundation. O livro é voltado para profissionais, Gerentes de TI, Gerentes de Negócio e pessoas que desejam compreender os conceitos, os processos e as funções do Gerenciamento de Serviços de TI baseados no ITIL® V3 e melhorar os Serviços de TI nas empresas.



## BPM & BPMS 2ª edição

Tadeu Cruz

294 pp. – R\$ 69,00

Neste livro você aprenderá sobre a reorganização informacional e as tentativas de organização das informações e conhecimento, como o conceito Computer-Supported Cooperative Work e as ferramentas que foram desenvolvidas com aderência a este conceito. Também vai aprender o que é Business Process Management – BPM e Business Process Management System – BPMS e as diferenças e semelhanças com o software de Workflow.



## Google AdSense

Dalton Felipini

128 pp. – R\$ 34,00

(Série E-commerce Melhores Práticas)

Este livro apresenta o genial sistema de gerenciamento de anúncios por meio do qual qualquer página da Web, desde um simples blog até um extenso site de conteúdo, pode tornar-se rentável. O Google fornece toda a tecnologia para a exposição e o gerenciamento dos anúncios. Com este livro, qualquer um, mesmo com pouca experiência na web, poderá transformar seu site ou blog em uma nova e permanente fonte de renda.



BRASPORT LIVROS E MULTIMÍDIA LTDA.  
RUA PARDAL MALLETT, 23 - TIJUCA - RIO DE JANEIRO - RJ - 20270-280  
Tel./Fax: (21) 2568.1415/2568-1507 - Vendas: vendas@brasport.com.br



Copyright © 2012 por Brasport Livros e Multimídia Ltda.

Todos os direitos reservados. Nenhuma parte deste livro poderá ser reproduzida, sob qualquer meio, especialmente em fotocópia (xerox), sem a permissão, por escrito, da Editora.

Editor: Sergio Martins de Oliveira

Diretora: Rosa Maria Oliveira de Queiroz

Gerente de Produção Editorial: Marina dos Anjos Martins de Oliveira

Revisão: Maria Inês Galvão Editoração

Eletrônica: Abreu's System Ltda.

Capa: Paulo Vermelho

Técnica e muita atenção foram empregadas na produção deste livro. Porém, erros de digitação e/ou impressão podem ocorrer. Qualquer dúvida, inclusive de conceito, solicitamos enviar mensagem para [brasport@brasport.com.br](mailto:brasport@brasport.com.br), para que nossa equipe, juntamente com o autor, possa esclarecer. A Brasport e o(s) autor(es) não assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso deste livro.

**Dados Internacionais de Catalogação na Publicação (CIP)**  
**(Câmara Brasileira do Livro, SP, Brasil)**

Sampaio, Cleuton

Guia de campo do bom programador : como desenvolver software Java EE com qualidade /  
Cleuton Sampaio. — Rio de Janeiro: Brasporte, 2012.

Formato: ePub

Requisitos do sistema: Adobe Digital Editions

Modo de acesso: World Wide Web

Bibliografia

ISBN 978-85-7452-528-0

1. Engenharia de software - Manuais, guias etc I. Título

12-04165

CDD-005.1



Índices para catálogo sistemático:  
1. Engenharia de software: Manuais, guias etc.  
005.1

Edição digital: abril 2012

**BRASPORT Livros e Multimídia Ltda.**

Rua Pardal Mallet, 23 – Tijuca

20270-280 Rio de Janeiro-RJ

Tels. Fax: (21) 2568.1415/2568.1507

e-mails: [brasport@brasport.com.br](mailto:brasport@brasport.com.br)

[vendas@brasport.com.br](mailto:vendas@brasport.com.br)

[editorial@brasport.com.br](mailto:editorial@brasport.com.br)

site: [www.brasport.com.br](http://www.brasport.com.br)

**Filial**

Av. Paulista, 807 – conj. 915

01311-100 – São Paulo-SP

Tel. Fax (11): 3287.1752

e-mail: [filialsp@brasport.com.br](mailto:filialsp@brasport.com.br)

---

---

Arquivo ePub produzido pela [Simplíssimo Livros](#)

---

---