engenharia de software magazine

Requisitos
Trabalhando com casos de uso na prática – Parte 1

Requisitos
Boas práticas na especificação de requisitos

DevMedia group Ano I - Edição 10

PROGRAMAÇÃO EXTREMA-XP

Fique por dentro dos principais conceitos que norteiam uma das mais conhecidas metodologias ágeis

Processo
A importância, processo e boas práticas da gerência de reutilização de software

Validação, Verificação e Testes
Melhore a qualidade do software e otimize recursos com teste baseado em riscos

Desenvolvimento
Introdução à programação orientada a aspectos

Educação
Bacharelado em Engenharia de Software: conheça essa proposta

Aulas desta edição:

- Introdução à Construção de Diagrama de Classes da UML – Parte 18
- Introdução à Construção de Diagrama de Classes da UML – Parte 19
- Introdução à Construção de Diagrama de Classes da UML – Parte 20
- Programação Orientada a Aspectos
- Desempenho em Bancos de Dados como Elemento para Melhoria das Aplicações
- Desenvolvimento Baseado em Componentes

ISSN 1983127-7

Conhecimento faz diferença!

Mais de 60 mil downloads na primeira edição!

Faça já sua assinatura digital! | *www.devmedia.com.br/es*

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

engenharia de software magazine

Ano 1 - 10ª Edição - 2009 **Impresso no Brasil**

EDITORIAL

"Nos últimos anos, os métodos ágeis de desenvolvimento de software ganharam importância em diversos segmentos da indústria de software. Assim como os métodos tradicionais, os métodos ágeis têm por objetivo construir sistemas de alta qualidade que atendam às necessidades dos usuários. A principal diferença está nos princípios utilizados para atingir tal objetivo."

"Os métodos ágeis apresentam uma abordagem bastante pragmática para o desenvolvimento de software. Planos detalhados são feitos apenas para a fase atual do projeto. Para fases futuras, os planos são considerados apenas rascunhos que podem se adaptar a mudanças conforme o time aprende e passa a conhecer melhor o sistema e as tecnologias utilizadas."

Neste contexto, a Engenharia de Software Magazine destaca nesta edição uma matéria que apresenta algumas evidências que motivaram o surgimento dos métodos ágeis, explicando seus valores e princípios, com ênfase na Programação Extrema, um dos métodos ágeis que mais recebeu atenção nos últimos anos.

Além desta matéria, esta edição traz mais seis artigos:

- UML – Casos de Uso: Entendendo os casos de uso na prática – um estudo de caso – Parte I;
- Documento de Requisitos: Essencial ao Desenvolvimento de Software;
- Gerência de Reutilização de Software;
- Melhorando a Qualidade do Software e Otimizando Recursos com Teste baseado em Riscos;
- Programação Orientada a Aspectos;
- Engenharia de Software: Graduação (bacharelado) em Engenharia de Software.

Desejamos uma ótima leitura!
Equipe Editorial Engenharia de Software Magazine

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Carmelita Mulin – Atendimento ao Leitor
www.devmedia.com.br/central/default.asp
(21) 2215-0033

Kaline Dolabella
Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 2215-0033

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site SQL Magazine, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Rodrigo Oliveira Spínola - Colaborador
editor@sqlmagazine.com.br

Rodrigo Oliveira Spínola
rodrigo@sqlmagazine.com.br
Doutorando em Engenharia de Sistemas e Computação (COPPE/UFRJ). Mestre em Engenharia de Software (COPPE/UFRJ, 2004). Bacharel em Ciências da Computação (UNIFACS, 2001). Colaborador da Kali Software (www.kalisoftware.com), tendo ministrado cursos na área de Qualidade de Produtos e Processos de Software, Requisitos e Desenvolvimento Orientado a Objetos. Consultor para implementação do MPS.BR. Atua como Gerente de Projeto e Analista de Requisitos em projetos de consultoria na COPPE/UFRJ. É Colaborador da Engenharia de Software Magazine.

Marco Antônio Pereira Araújo
maraujo@devmedia.com.br
É Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ – Linha de Pesquisa em Engenharia de Software, Especialista em Métodos Estatísticos Computacionais e Bacharel em Matemática com Habilitação em Informática pela UFJF, Professor dos Cursos de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora e da Faculdade Metodista Granbery, Analista de Sistemas da Prefeitura de Juiz de Fora. É colaborador da Engenharia de Software Magazine.

Eduardo Oliveira Spínola
(eduspinola@gmail.com)
É Colaborador das revistas Engenharia de Software Magazine, Java Magazine e SQL Magazine. É bacharel em Ciências da Computação pela Universidade Salvador (UNIFACS) onde atualmente cursa o mestrado em Sistemas e Computação na linha de Engenharia de Software, sendo membro do GESA (Grupo de Engenharia de Software e Aplicações).

Caro Leitor

Para esta edição, temos um conjunto de 6 vídeo aulas. Estas vídeo aulas estão disponíveis para download no Portal da Engenharia de Software Magazine e certamente trarão uma significativa contribuição para seu aprendizado. A lista de aulas publicadas pode ser vista abaixo:

Tipo: Desenvolvimento
Título: Programação Orientada a Aspectos
Autor: Marco Antônio Pereira Araújo
Mini-Resumo: A Programação Orientada a Aspectos (POA) surge como uma proposta para a construção de sistemas complexos através da separação de interesses, como aqueles que não se aplicam à lógica de negócios como, por exemplo, segurança, desempenho, persistência, integridade de dados e tratamento de erros, dentre outros. Assim, a POA prega a divisão de um sistema em partes para uma melhor compreensão das fronteiras que definem suas funções, onde toda parte relacionada a um interesse é transportada para uma localização física separada das demais, de forma a permitir maior reusabilidade e manutenibilidade do software. Nesta vídeo aula conheceremos alguns dos principais conceitos deste paradigma de programação.

Tipo: Desenvolvimento
Título: Desempenho em Bancos de Dados como Elemento para Melhoria das Aplicações
Autor: Marco Antônio Pereira Araújo
Mini-Resumo: Ao trabalhar com consultas em bancos de dados, deve-se ter uma atenção maior com relação ao seu desempenho, uma vez que consultas mal elaboradas podem degradar consideravelmente o desempenho do sistema como um todo. Para que seja possível fazer melhorias relativas às consultas, é preciso entender como analisar seu desempenho, bem como os fatores que contribuem para sua melhoria. Este será o foco desta aula.

Tipo: Projeto
Título: Desenvolvimento Baseado em Componentes
Autor: Marco Antônio Pereira Araújo
Mini-Resumo: Desenvolvimento Baseado em Componentes (DBC) aparece como uma técnica que consiste no desenvolvimento de aplicações a partir de componentes interoperáveis, reduzindo, assim, a complexidade e o custo do desenvolvimento, e melhorando a qualidade do produto de software. Proporciona ainda o desenvolvimento de aplicações com mais agilidade, através do reuso de componentes pré-existentes e com maior grau de confiabilidade. Nesta vídeo aulas serão apresentados alguns conceitos desta abordagem.

Tipo: Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 18
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade ao estudo de caso de um sistema de gestão de festas. Neste estudo de caso, partiremos dos requisitos funcionais do software. A partir deles, será elaborado o diagrama de casos de uso do sistema. Com o diagrama de casos de uso elaborado, serão especificados três casos de uso. Por último, será elaborado o diagrama de classes com base nos requisitos funcionais e casos de uso especificados. Nesta aula, o foco é o início da elaboração do diagrama de classes. Para isto, faremos uso da heurística apresentada neste treinamento. Em particular, identificaremos as classes e atributos candidatos a partir da lista de requisitos funcionais.

Tipo: Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 19
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade ao estudo de caso de um sistema de gestão de festas. Neste estudo de caso, partiremos dos requisitos funcionais do software. A partir deles, será elaborado o diagrama de casos de uso do sistema. Com o diagrama de casos de uso elaborado, serão especificados três casos de uso. Por último, será elaborado o diagrama de classes com base nos requisitos funcionais e casos de uso especificados. Nesta aula, o foco é o início da elaboração do diagrama de classes. Para isto, faremos uso da heurística apresentada neste treinamento. Em particular, identificaremos as classes e atributos candidatos a partir dos casos de uso especificados.

Tipo: Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 20
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade ao estudo de caso de um sistema de gestão de festas. Neste estudo de caso, partiremos dos requisitos funcionais do software. A partir deles, será elaborado o diagrama de casos de uso do sistema. Com o diagrama de casos de uso elaborado, serão especificados três casos de uso. Por último, será elaborado o diagrama de classes com base nos requisitos funcionais e casos de uso especificados. Nesta aula, finalizamos a identificação das classes e atributos, indicando quais atributos pertencem a cada classe.

ÍNDICE

Introdução à Programação Extrema (XP)

Danilo Sato

danilo@dtsato.com / www.dtsato.com

Danilo Sato é Desenvolvedor, Coach e Consultor da ThoughtWorks UK. Com formação e mestrado em Ciência da Computação pelo IME/USP no tema Métodos Ágeis, é também fundador do Dojo@SP e membro da AgilCoop. Danilo já ministrou cursos sobre diversos assuntos relacionados a Métodos Ágeis e palestrou em diversas conferencias, incluindo: XP 2007 (Itália), Agile 2008 (Canadá), Agiles 2008 (Argentina) e Falando em Agile (Brasil).

Nos últimos anos, os métodos ágeis de desenvolvimento de software ganharam importância em diversos segmentos da indústria de software. Assim como os métodos tradicionais, os métodos ágeis têm por objetivo construir sistemas de alta qualidade que atendam às necessidades dos usuários. A principal diferença está nos princípios utilizados para atingir tal objetivo.

Os métodos ágeis apresentam uma abordagem bastante pragmática para o desenvolvimento de software. Planos detalhados são feitos apenas para a fase atual do projeto. Para fases futuras, os planos são considerados apenas rascunhos que podem se adaptar a mudanças conforme o time aprende e passa a conhecer melhor o sistema e as tecnologias utilizadas.

Neste artigo são apresentadas algumas evidências que motivaram o surgimento dos métodos ágeis, explicando seus valores e princípios, com ênfase na Programação Extrema, um dos métodos ágeis que mais recebeu atenção nos últimos anos.

De que se trata o artigo?

Neste artigo são apresentadas algumas evidências que motivaram o surgimento dos métodos ágeis, explicando seus valores e princípios, com ênfase na Programação Extrema, um dos métodos ágeis que mais recebeu atenção nos últimos anos.

Para que serve?

Para gerentes, tomadores de decisão e membros de uma equipe de desenvolvimento de software, este artigo apresenta uma introdução detalhada sobre métodos ágeis de desenvolvimento de software, mostrando as evidências que levaram ao seu surgimento e abordando de forma ampla alguns dos métodos ágeis mais conhecidos, focando principalmente na Programação Extrema (XP).

Em que situação o tema é útil?

Para quem já ouviu falar mas ainda não se aprofundou nos assuntos relacionados a métodos ágeis, Scrum, Lean ou XP. O artigo aborda uma ampla gama de assuntos, como: evidências, origens, metodologias, valores, princípios, práticas e formas de adoção.

Evidências

No desenvolvimento de software, é comum que os requisitos mudem enquanto a implementação ainda está acontecendo. Kajko-Mattson et al. mostram que cerca de 40% a 90% do custo durante o ciclo de vida de um projeto é gasto na fase de manutenção [1]. Muitas empresas e times de desenvolvimento acham que mudanças são indesejáveis, pois acabam com todo o esforço gasto no planejamento. No entanto, os requisitos geralmente mudam conforme o cliente vê o sistema sendo implantado e em funcionamento. É muito difícil criar um plano no início do projeto que consiga prever todas as mudanças sem gastar muito esforço, tempo e dinheiro.

Boehm chegou a afirmar que "encontrar e arrumar um defeito no software após a entrega custa cerca de 100 vezes mais do que encontrá-lo e arrumá-lo nas fases iniciais de design" [2]. Essa foi uma das principais justificativas para os métodos tradicionais gastarem mais tempo nas fases de análise de requisitos e design, apesar do próprio Boehm ter sugerido o desenvolvimento iterativo ao invés da "produção do produto completo de uma vez" [3]. Em 2001, num artigo de Boehm e Basili, houve uma redução no pessimismo ao perceber que, para sistemas pequenos, o fator era mais próximo de 5:1 ao invés de 100:1 e que, mesmo para sistemas grandes, boas práticas arquiteturais poderiam reduzir de forma significativa o custo da mudança, encapsulando as áreas de mudança em partes pequenas e fatoradas [4].

Poppendieck [5] sugere que a principal razão das mudanças no desenvolvimento de software é que o processo de negócio ao qual o software está atrelado evolui constantemente. Construir flexibilidade para acomodar mudanças arbitrárias é muito caro e pode ser um desperdício. Segundo Johnson [6], 45% das funcionalidades implementadas num sistema típico não são utilizadas nunca e 19% são raramente utilizadas. A melhor estratégia é evitar generalizações desnecessárias e fazer com que o sistema seja flexível apenas nas áreas mais propícias à mudança [7].

A maioria dos estudos de caso na indústria apontam para uma taxa relativamente alta de fracassos nos projetos de software [8]. No clássico relatório do Standish Group de 1994, o CHAOS Report [9], 37% dos fatores relacionados aos projetos em dificuldade estavam relacionados aos requisitos. O mesmo relatório de 2003 aponta que apenas 52% das funcionalidades são entregues em um projeto [10]. Outro estudo de classificação de defeitos aponta os requisitos como principal categoria de defeitos, com 41% [11]. "Nós queremos estabilizar os requisitos, mas eles não são estáveis" [12].

O Manifesto Ágil

Em fevereiro de 2001, um grupo formado por 17 desenvolvedores experientes, consultores e líderes da comunidade de software se reuniu em Utah para discutir idéias e procurar uma alternativa aos processos dirigidos a documentação e às práticas adotadas nas abordagens tradicionais da Engenharia de Software e gerência de projetos. Dessa reunião surgiu o Manifesto do Desenvolvimento Ágil de Software [13], que destaca as diferenças com relação às abordagens tradicionais e define seus valores:

- **Indivíduos e interações** são mais importantes que processos e ferramentas;
- **Software funcionando** é mais importante que documentação completa e detalhada;
- **Colaboração com o cliente** é mais importante que negociação de contratos;
- **Adaptação a mudanças** é mais importante que seguir um plano.

Apesar da importância dos itens à direita, os métodos ágeis dão mais valor para os itens destacados à esquerda. Além dos quatro valores básicos, o manifesto ágil também apresenta 12 princípios que auxiliam a difusão de suas idéias:

- A maior prioridade é a satisfação do cliente por meio da entrega rápida e contínua de software que traga valor;
- Mudanças nos requisitos são aceitas, mesmo em estágios avançados de desenvolvimento. Processos ágeis aceitam mudanças que trarão vantagem competitiva para o cliente;
- Software que funciona é entregue freqüentemente, em períodos que variam de semanas a meses, quanto menor o tempo entre uma entrega e outra melhor;
- As pessoas relacionadas ao negócio e os desenvolvedores devem trabalhar juntos no dia a dia do projeto;
- Construa projetos formados por indivíduos motivados, fornecendo o ambiente e o suporte necessário e confiando que realizarão o trabalho;
- O modo mais eficiente e eficaz de transmitir informações dentro e fora do time de desenvolvimento é a Comunicação face a face;
- A principal medida de progresso é software funcionando;
- Processos ágeis promovem o desenvolvimento em um ritmo sustentável. Os investidores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante;
- Cuidar continuamente da excelência técnica e do bom design ajuda a aprimorar a agilidade;
- Simplicidade - a arte de maximizar a quantidade de trabalho não necessário - é essencial;
- Os melhores requisitos, arquiteturas e design surgem de equipes auto-gerenciadas;
- Em intervalos regulares, o time reflete sobre como se tornar mais eficiente, refinando e ajustando seu comportamento apropriadamente.

Essas características trazem dinamismo para o desenvolvimento, motivação para o time e informações mais precisas sobre a verdadeira situação do projeto para o cliente. Enquanto as abordagens tradicionais têm um enfoque mais preditivo, os métodos ágeis são adaptativos.

O manifesto ágil apresenta uma nova filosofia para o desenvolvimento de software. Sob seus valores e princípios aparecem diversas abordagens mais específicas, com diferentes idéias, comunidades e líderes. Cada comunidade

forma um grupo distinto, porém todas seguindo os mesmos princípios. É comum inclusive a troca de conhecimento e práticas entre membros de diferentes comunidades, formando um eco-sistema em torno de diversos métodos, detalhados a seguir com maior ênfase na Programação Extrema.

Scrum

Desenvolvida nas décadas de 80 e 90 por Ken Schwaber, Jeff Sutherland, e Mike Beedle [14], o Scrum se concentra mais nos aspectos gerenciais do desenvolvimento de software, propondo iterações de duas semanas ou 30 dias (chamados Sprints) com acompanhamento diário por meio das Reuniões em Pé (ou stand-up meetings). Por dar menos ênfase aos aspectos técnicos, é geralmente combinada com práticas propostas por XP e compatível com certificações de qualidade como CMMI ou ISO 9001 [15].

Lean Software Development

Com base no Sistema de Produção da Toyota [16], o movimento Lean revolucionou a manufatura e, mais recentemente, o desenvolvimento de produtos e o gerenciamento da cadeia de suprimentos (supply chain management). Mary e Tom Poppendieck traçaram os paralelos entre os valores e práticas Lean com o desenvolvimento de software, fornecendo sete princípios [5, 7]: "Elimine Desperdícios", "Inclua a Qualidade no Processo", "Crie Conhecimento", "Adie Comprometimentos", "Entregue Rápido", "Respeite as Pessoas" e "Otimize o Todo".

Família Crystal

Alistair Cockburn propõe uma família de métodos por acreditar que diferentes abordagens são necessárias para equipes de tamanhos diferentes [17]. Apesar disso, todos os métodos dessa família compartilham propriedades como: entrega freqüente, Reflexão e Comunicação. Outra parte importante dos métodos da família Crystal é o que Cockburn chama de habitabilidade (habitability): o mínimo de processo necessário para que a equipe consiga ter sucesso.

Feature Driven Development (FDD)

Desenvolvida por Peter Coad e Jeff de Luca no final da década de 90, a FDD define duas fases compostas por 5 processos bem definidos e integrados: a fase de concepção e planejamento, composta por "desenvolver um modelo abrangente", "construir uma lista de funcionalidades" e "planejar por funcionalidade"; e a fase iterativa de construção, composta por "detalhar por funcionalidade" e "construir por funcionalidade" [18]. Outra característica interessante da FDD é a utilização de modelos UML em cores para representar classes com diferentes responsabilidades (chamados "arquétipos") [19].

Adaptive Software Development

Proposto por Jim Highsmith, esse método tenta explorar a natureza adaptativa e a incerteza no desenvolvimento de software [20]. Com base nas idéias dos Sistemas Adaptativos Complexos (relacionados à Teoria do Caos) [21], ele propõe três fases não-lineares e possivelmente sobrepostas: especulação (referindo-se ao paradoxo do planejamento), colaboração e aprendizado. Por meio de iterações curtas, a equipe cria o conhecimento cometendo pequenas Falhas, causadas por falsas premissas, e corrigindo-as aos poucos, criando uma experiência mais rica e ampla.

Dynamic System Development Method (DSDM)

Desenvolvido inicialmente por um consórcio de empresas britânicas em 1994, esse método se baseou nas idéias do desenvolvimento rápido de aplicações (Rapid Application Development ou RAD [22]) e no desenvolvimento iterativo e incremental [23]. O método começa com duas fases iniciais: um estudo de viabilidade que valida se o processo DSDM é apropriado para o projeto e um estudo de negócio, para entender as necessidades de negócio e definir uma arquitetura e os requisitos iniciais. O processo prossegue com três fases: uma iteração para o modelo funcional define protótipos e uma documentação de análise inicial, uma iteração para o design e construção do sistema, e uma última fase de implementação para entrega e implantação do produto. O DSDM ainda define princípios que incluem: participação ativa do usuário, entrega freqüente, times com poder de decisão e testes durante todo o ciclo de vida do produto.

Programação Extrema

A Programação Extrema (ou XP) foi um dos métodos ágeis que mais recebeu atenção na virada do século. Seu objetivo é a excelência no desenvolvimento de software, visando baixo custo, poucos defeitos, alta produtividade e alto retorno de investimento. Na segunda edição do livro "Extreme Programming Explained" [24], Kent Beck aprimora a definição de XP da primeira edição, enumerando suas principais características:

- XP é um método leve. O time só deve fazer o necessário para trazer valor para o cliente;
- XP é um método que enfatiza o desenvolvimento de software. Apesar de ter implicações em áreas como marketing, vendas ou operações, XP não tenta resolver os problemas diretamente ligados a elas;
- XP funciona para times de qualquer tamanho. Apesar das práticas de XP funcionarem melhor em times pequenos, seus valores e princípios podem ser aplicados em qualquer escala e XP se adapta a requisitos vagos ou em constante mudança.

Histórico

As idéias de XP originaram-se de conversas entre Kent Beck e Ward Cunningham a partir de suas experiências com desenvolvimento de software em Smalltalk. Juntos, eles escreveram o primeiro artigo sobre cartões CRC [25] e sobre a aplicação de padrões no desenvolvimento de software [26]. Ward Cunningham foi o criador do Wiki [27] e foi no seu Wiki que as primeiras discussões sobre XP aconteceram.

Muitas das características de XP, como Refatoração, Programação Pareada, adaptação à mudança, Integração Contínua, desenvolvimento iterativo e ênfase nos testes, são elementos-chave presentes na cultura da comunidade Smalltalk desde a década de 1980.

Em 1992, William Opdyke publicou sua tese de doutorado [28], contando como Kent e Ward obtinham ganhos em produtividade usando técnicas de refatoração. Mais tarde, essas técnicas seriam compiladas no livro de Martin Fowler et al. sobre refatoração [29].

Kent Beck também publicou o primeiro artigo sobre testes de unidade automatizados [30] quando desenvolveu o primeiro arcabouço para desenvolvimento de testes automatizados: o SUnit [31] para testes em Smalltalk e, mais tarde juntamente com Erich Gamma, o JUnit [32, 33] para testes em Java. Esse arcabouço foi portado para diversas linguagens, formando um conjunto de ferramentas que ficaram conhecidas como família XUnit: JUnit (Java), CppUnit (C++), CUnit (C), NUnit (.NET), pyUnit (Python), Test::Unit (Ruby), dentre diversos outros [34].

Todas essas idéias foram se fundindo na cabeça de Kent Beck quando, em 1996, ele foi chamado para ajudar no projeto C3, ou Chrysler Comprehensive Compensation System, que ficou conhecido como o berço de XP. Nele, Kent Beck utilizou pela primeira vez todas as práticas que vieram a se tornar a Programação Extrema. Sua idéia, que originou o nome Programação Extrema, era juntar as boas práticas de programação já conhecidas pela indústria e encará-las como botões de volume que seriam aumentados ao valor máximo, extremo. Se fazer revisão de código era bom, fazer Programação Pareada era revisão em tempo integral; se fazer testes automatizados cedo era bom, escrevê-los antes do código era melhor ainda.

Foi a partir dessa experiência que Kent Beck lançou, em 1999, a primeira edição do livro que difundiu XP, "Extreme Programming Explained: Embrace Change" [35]. Esse livro recebeu no mesmo ano o prêmio JOLT de produtividade da revista Software Development e, após cinco anos de experiência com a utilização e consultoria de XP, Kent Beck lançou, em parceria com a sua esposa, da área de psicologia, a segunda edição do livro que é hoje uma das principais referências sobre o assunto [24].

Abordagem

Segundo Kent Beck [24], a Programação Extrema inclui:

- Uma filosofia para o desenvolvimento de software baseada nos valores de Comunicação, Feedback, Simplicidade, Coragem e Respeito;
- Um conjunto de práticas comprovadamente úteis para melhorar o desenvolvimento de software. As práticas expressam os valores de XP;
- Um conjunto complementar de princípios, técnicas intelectuais que auxiliam a tradução dos valores em práticas, úteis quando as práticas existentes não resolvem seu problema particular;
- Uma comunidade que compartilha os mesmos valores e muitas das mesmas práticas.

Essa separação entre valores, princípios e práticas já estava presente na primeira edição de XP, porém sua importância foi reforçada na segunda edição. É possível ter uma visão mais ampla do processo quando pensamos nessas três perspectivas.

As práticas são técnicas utilizadas no dia-a-dia dos membros de uma equipe de XP. Elas são claras, objetivas e específicas. Práticas como Desenvolvimento Dirigido por Testes (Test Driven Development ou TDD) [36] ou Refatoração [29] fazem sentido somente no contexto da programação. Em outro contexto as mesmas práticas não fariam sentido. As práticas de XP são apresentadas nas Seções Práticas e Comparação com as Práticas da Primeira Versão, que compara as diferenças entre as abordagens na primeira e na segunda edição.

Valores são critérios mais amplos e universais utilizados para julgar uma determinada situação. É possível enxergar o valor do Feedback em diversos contextos, desde a programação (através da Integração Contínua, por exemplo) até a Comunicação com o cliente (através do Ciclo Semanal e do Ciclo Trimestral). XP é uma disciplina de software baseada em cinco valores, que serão discutidos na Seção Valores.

Os valores dão razão às práticas, enquanto as práticas evidenciam os valores. Para preencher o espaço vazio entre valores e práticas, Kent Beck apresenta os princípios de XP. Os princípios são técnicas intelectuais para auxiliar na tradução de valores em práticas. Eles devem ser utilizados quando as práticas propostas não se aplicam numa situação específica. Por exemplo, o princípio dos Passos Pequenos é demonstrado em diferentes práticas, como a Implantação Diária ou o ritmo imposto pelo Desenvolvimento Dirigido por Testes. Os princípios de XP são apresentados na Seção Princípios.

Valores

A filosofia de trabalho proposta por XP está baseada em 5 valores que servem de base para a aplicação das práticas e dos princípios:

Comunicação

O primeiro valor do Manifesto Ágil propõe que os indivíduos e as interações entre eles são importantes para o desenvolvimento de software [13]. A Programação Extrema expressa tal importância no valor da comunicação. XP pressupõe que a maioria dos problemas num projeto de software ocorrem por dificuldade na Comunicação [12].

A comunicação é evidenciada em muitas das práticas: uma equipe de XP funciona como um time completo, trabalhando numa área de trabalho informativa na qual todos possam sentar junto. A presença e o envolvimento real com o cliente e a programação pareada também são práticas que fortalecem e facilitam a comunicação. Além disso, as métricas

para acompanhamento do projeto também são importantes elementos de comunicação, auxiliando a equipe a melhorar o processo de desenvolvimento e o cliente a entender o andamento do projeto.

Simplicidade

O segundo valor de XP é a simplicidade. Os membros de uma equipe de XP estão freqüentemente buscando a solução mais simples para resolver seus problemas atuais. Num contexto onde a adaptação a mudanças é aceita [13] e encorajada, XP promove a preocupação com o que é mais simples para resolver os problemas de hoje, evitando desperdícios com soluções genéricas para problemas futuros.

Segundo Kent Beck, a simplicidade é o valor intelectual mais intenso de XP [24]. Encontrar a solução mais simples não é uma tarefa fácil. A Seção Práticas apresenta uma das práticas que mais ajuda os desenvolvedores numa equipe de XP a manter a ênfase constante no design simples, o desenvolvimento dirigido por testes [36].

Feedback

É importante ter uma resposta rápida sobre as ações realizadas para se adaptar às mudanças [13]. XP promove ciclos curtos e constantes de feedback, nos mais variados aspectos do desenvolvimento de software. Os valores de XP se complementam, por isso o feedback é parte importante da comunicação e da simplicidade. Diante de uma dúvida entre três diferentes soluções, tentar todas parece ser um desperdício, porém esta pode ser a melhor forma de descobrir qual solução é mais simples e mais fácil de lidar.

Segundo Ambler [37], a maior contribuição para o sucesso dos métodos ágeis é a redução dos ciclos de feedback. As práticas de XP evidenciam tal valor: a programação pareada fornece feedback em segundos enquanto o desenvolvimento dirigido por testes fornece feedback em minutos. Além disso, o acompanhamento em projetos XP é realizado diariamente pelo tracker e o uso de ferramentas para coleta e análise automatizada de métricas pode fornecer feedback ainda mais rapidamente.

Coragem

O quarto valor de XP é a coragem. Kent Beck descreve a coragem em XP como a "ação tomada diante do medo" [24]. Isso não significa que os membros da equipe devem ter coragem para fazer o que quiserem sem se preocupar com as conseqüências para o time. A coragem como valor primário, sem a influência e balanceamento dos outros valores, pode ser perigosa. No entanto, em conjunto com os outros valores, ela é muito poderosa.

Teles [38] enumera alguns pontos onde a adoção de XP exige coragem da equipe para: desenvolver software de forma incremental; manter a ênfase constante na simplicidade; permitir ao cliente priorizar funcionalidades; incentivar os desenvolvedores a trabalhar em par; investir tempo em refatoração; investir tempo em testes automatizados; estimar as histórias na presença do cliente; compartilhar o código com todos os membros da equipe; fazer a integração completa do sistema diversas vezes ao dia; adotar um ritmo sustentável; abrir mão da documentação; propor contratos de escopo variável e propor a adoção de um processo novo.

Respeito

O quinto valor, enfatizado na segunda edição de XP, serve de base para os outros quatro valores: respeito. Se os membros da equipe não se importam com os outros ou com os resultados, XP não vai funcionar [24]. É importante reconhecer que a excelência no desenvolvimento de software depende das pessoas, e elas devem se respeitar para conseguir extrair o máximo de seu potencial.

A falta de respeito pode influenciar negativamente alguns aspectos na adoção de XP: comunicação sem respeito criará conflitos internos; coragem sem respeito trará atitudes que vão contra o bem estar da equipe; programação pareada é um exercício contínuo de respeito; horas-extras excessivas irão impactar o ritmo sustentável da equipe; a colaboração entre a equipe e o cliente também exige uma comunicação aberta e respeitosa.

Princípios

Os princípios de XP funcionam como ferramentas para tradução dos valores em práticas. Tanto documentos longos quanto conversas diárias têm a intenção de comunicar. Descobrir qual é a forma mais eficiente depende parte do contexto e parte dos princípios intelectuais. Nesse caso, o princípio da humanidade sugere que a conversa satisfaz melhor as necessidades humanas de relacionamento. Os princípios que guiam a Programação Extrema são:

Humanidade

Pessoas desenvolvem software. É importante levar em conta as necessidades básicas do ser humano no desenvolvimento de software, criando oportunidades para: crescimento, contribuição, participação e relacionamento. O grande desafio é balancear as necessidades pessoais com as necessidades do time. As práticas de XP tentam atender tanto às necessidades de negócio quanto às necessidades pessoais dos membros da equipe.

Economia

Os envolvidos no desenvolvimento de software também devem se preocupar com os aspectos econômicos para evitar que o projeto seja apenas um "sucesso técnico". É importante que uma equipe de XP esteja constantemente preocupada em agregar valor de negócio ao sistema que estão desenvolvendo. Esse princípio é um dos motivos pelos quais os clientes são responsáveis pela priorização das histórias nas reuniões de planejamento. A equipe de XP deve resolver os problemas mais importantes primeiro, maximizando o valor do projeto.

Benefício Mútuo

O princípio do benefício mútuo é um dos mais importantes de XP, porém é também um dos mais difíceis de aplicar. Todas as atividades devem trazer benefício a todos os envolvidos. Escrever documentos longos é um exemplo de violação desse princípio: o programador diminui seu ritmo de produção para escrever um documento que não tem valor agora, podendo apenas trazer valor no futuro para alguém que irá dar manutenção no seu código (se a documentação continuar válida até lá). XP resolve o problema da comunicação com o futuro de uma forma mutuamente benéfica através de:

• Testes automatizados que ajudam o programador no design e na implementação das funcionalidades agora, servindo como documentação e como teste de regressão para as pessoas que irão dar manutenção no futuro;

• Refatoração constante para remover complexidade desnecessária, simplificar o design e remover defeitos, deixando o código mais limpo e fácil de entender para os futuros mantenedores;

• Nomes coerentes e baseados em metáforas que facilitam o entendimento do sistema, aumentando a velocidade atual do desenvolvimento e da integração de novos programadores na equipe.

Auto-Semelhança

O princípio da auto-semelhança sugere a aplicação da estrutura de uma solução em outros contextos, inclusive em diferentes escalas. Um exemplo sugerido por Kent Beck [24] é a aplicação de testes a priori em ambos os níveis: não só quando desenvolvendo os testes unitários com TDD, mas também para especificar os testes de aceitação com o cliente. Dessa forma, fica mais claro para os programadores que as histórias só estão prontas quando passam nos testes de aceitação, reduzindo o ciclo de feedback e simplificando o andamento da iteração.

Melhoria

XP valoriza a busca constante pela melhoria. Ao invés de buscar a perfeição, é mais importante tentar fazer o melhor trabalho possível hoje, e estar consciente de tudo que será necessário para melhorar amanhã. O princípio da melhoria valoriza as atividades que começam agora e se refinam ao longo do tempo.

Diversidade

Os times devem ser formados por uma variedade de habilidades, atitudes e perspectivas. Dessa diversidade podem surgir conflitos que devem ser vistos como oportunidades para discussão das diferentes perspectivas. Muitas vezes o melhor design surge a partir de soluções distintas. O princípio da diversidade se expressa em XP através da prática do time completo.

Reflexão

Uma boa equipe não deve apenas fazer o seu trabalho, mas sim refletir constantemente sobre as razões e as formas como estão trabalhando. Os times devem analisar seus sucessos e suas falhas, sempre em busca da melhoria contínua. A reflexão vem após a ação. O aprendizado surge para a equipe como resultado da reflexão sobre a ação. Para maximizar o feedback, as equipes de XP devem refletir e estar conscientes de seus atos.

Fluxo

Esse princípio sugere a entrega de um fluxo contínuo de software que agrega valor ao negócio, evitando pensar em fases discretas. Quanto maior o tamanho de uma atividade - iteração, release, história ou tarefa - maior o tempo gasto até descobrir se ela foi realizada com sucesso ou não, o que aumenta o risco do erro. Quanto maior o erro, mais difícil é a correção. Para aumentar o feedback e diminuir os riscos, a equipe de XP deve prover incrementos pequenos de funcionalidade, fazendo entregas pequenas e freqüentes. Práticas como integração contínua, build em 10 minutos, implantação incremental e implantação diária evidenciam o princípio do fluxo.

Oportunidade

Equipes de XP enxergam problemas como oportunidades para mudança. Na busca pela excelência, os membros do time devem demonstrar uma atitude positiva, identificando oportunidades para aprender, melhorar e desenvolver software de qualidade.

Redundância

O princípio da redundância sugere que os problemas difíceis e críticos devem ser resolvidos de várias maneiras diferentes. Muitas das práticas de XP, como programação pareada, integração contínua, envolvimento real com o cliente e desenvolvimento dirigido por testes, são redundantes na tentativa de reduzir a quantidade de defeitos e aumentar a qualidade do software produzido.

Falha

Complementando o princípio da redundância, o princípio da falha sugere que todo erro é um aprendizado. Na dúvida entre três soluções diferentes, tente implementar todas, mesmo que algumas falhem. Diante de diferentes opções de design, ao invés de perder tempo discutindo qual a melhor solução, vale mais a pena implementá-las paralelamente, aprendendo com os erros e chegando num consenso através da experiência.

Qualidade

Sacrificar a qualidade nunca é um meio eficaz de controle. Os projetos não andam mais rápido aceitando baixa qualidade. Cada incremento na qualidade tem reflexos de melhoria em diversas outras áreas do projeto, como produtividade, eficiência e motivação. Um projeto XP não considera a qualidade como uma das variáveis de controle. O custo e o tempo também são geralmente fixos, deixando o escopo como principal variável na negociação com o cliente [39].

Passos Pequenos

O princípio dos passos pequenos complementa o princípio do fluxo, fazendo com que os membros da equipe de XP se perguntem sempre "Qual o mínimo que preciso fazer para garantir que estou na direção certa?". Práticas como desenvolvimento dirigido por testes e integração contínua evidenciam o valor do ritmo imposto pelos passos pequenos. A refatoração é outro exemplo da aplicação desse princípio.

Aceitação da Responsabilidade

A responsabilidade não deve ser imposta, deve ser aceita. As práticas refletem esse princípio ao, por exemplo, sugerir que a pessoa responsável por uma História também é responsável pela sua estimativa, design, implementação e teste.

Práticas

A abordagem de apresentação das práticas de XP foi totalmente refatorada na segunda edição por Kent Beck [24]. Ao invés de exigir a utilização de todas as 12 práticas de uma vez, Kent Beck sugere que cada time deve se adaptar da maneira que achar mais apropriada. Ao invés de impor as práticas, cada mudança deve começar pelos próprios membros da equipe. Além disso, as práticas na segunda edição são 24, divididas entre práticas primárias e práticas corolárias. Práticas primárias são aquelas que podem ser aplicadas separadamente, trazendo melhoria imediata para a equipe. Práticas corolárias são mais difíceis de implementar, mostrando sua eficiência somente após domínio e experiência prévia com as práticas primárias.

Práticas Primárias

• Sentar Junto: A equipe de XP deve trabalhar num espaço amplo e aberto, onde todos possam ficar juntos, fortalecendo os elos da comunicação. É preferível trocar os tradicionais cubículos por áreas de uso comum onde os membros da equipe possam se agrupar para discutir no quadro branco, sentar juntos para trabalhar em par e espalhar gráficos e informações na parede.

• Time Completo: Equipes de XP devem ser multi-disciplinares, com todas as habilidades necessárias para o sucesso do projeto. A equipe deve ser formada não apenas por desenvolvedores, mas também por analistas de teste, analistas de negócio, clientes, especialistas em banco de dados, especialistas em interfaces gráficas, administradores de rede, etc. Todos devem trabalhar num espírito de contribuição para a equipe, visando o bom andamento do projeto.

• Área de Trabalho Informativa: Transforme o ambiente de trabalho num reflexo do projeto. Um observador interessado deve ser capaz de ter uma idéia da evolução do projeto apenas andando pela área de trabalho. Alguns exemplos de instrumentos para espalhar essas informações, chamados de radiadores de informação por Cockburn [40], são: cartões com histórias num mural, quadros brancos, notas em papel nas paredes e gráficos como, por exemplo, o burn-down chart proposto pelo Scrum [14, 41] para acompanhar a velocidade da equipe. As métricas também têm papel importante na comunicação e é na área de trabalho informativa onde elas serão atualizadas pelo tracker e apresentadas à equipe e ao cliente.

• Trabalho Energizado: O ritmo de trabalho não deve afetar a vida pessoal dos membros da equipe. Durante o planejamento, o número de horas dedicadas ao projeto deve ser definido de forma realista. Fazer horas-extra deve ser exceção e não a regra. Os membros da equipe de XP devem trabalhar apenas enquanto puderem ser produtivos e se manter energizados. O desenvolvimento de software exige criatividade que raramente aparecerá em momentos de cansaço ou indisposição [42].

• Programação Pareada: Os desenvolvedores trabalham em par para realizar suas tarefas. Isso promove o trabalho coletivo e colaborativo, une a equipe e melhora a comunicação e a qualidade do código. Os pares devem ser trocados regularmente, inclusive várias vezes por dia. Geralmente, a seleção dos pares depende da tarefa a ser realizada, da disponibilidade dos membros da equipe e da experiência de cada um. O objetivo principal é espalhar o conhecimento do sistema pela equipe inteira. Como importante efeito colateral temos também o compartilhamento de técnicas e competências entre os membros da equipe.

• Histórias: O planejamento em XP é feito com histórias escritas em pequenos cartões. Cada cartão é escrito pelo cliente e deve descrever uma unidade de funcionalidade, que geralmente representa um requisito funcional desejado. Mike Cohn propõe o seguinte formato para uma história [43]:

"Como um <usuário/papel>
Eu gostaria de <funcionalidade>
Para que <valor de negócio>"

Este formato é interessante, pois evidencia o valor de negócio associado a cada funcionalidade. Para cada história, os desenvolvedores devem dar uma estimativa sobre o tempo para implementá-la e os clientes determinam a prioridade de cada história. Essas informações são utilizadas no Jogo do Planejamento, que acontece no início dos Ciclos Semanais e dos Ciclos Trimestrais. A descrição no cartão não deve armazenar todas as informações sobre a história. Os desenvolvedores de uma equipe de XP utilizam o diálogo como principal meio de comunicação com o cliente para elucidar dúvidas sobre os detalhes da história. Os cartões devem servir apenas como um lembrete desse diálogo.

As estimativas das histórias num projeto em XP são geralmente medidas em pontos ou em "horas ideais". Equipes ágeis separam a estimativa do tamanho/complexidade de uma história do tempo que ela demora para ser implementada. Enquanto o tempo real gasto na implementação pode variar, a complexidade da história permanece a mesma. Uma "hora ideal" considera que o desenvolvedor está trabalhando focado e sem interrupção durante uma hora, porém

o tempo real gasto vai ser geralmente maior. Portanto, são uma medida do tamanho e da complexidade de uma história em relação a outras histórias do mesmo projeto. O sistema de pontos baseia-se em uma escala numérica definida pela equipe que permite a estimativa por comparação. Uma história de 4 pontos tem mais ou menos o dobro do tamanho de uma história de 2 pontos. Algumas escalas comumente utilizadas são escalas exponenciais (1, 2, 4, 8, 16, ...) ou uma seqüência de Fibonacci (1, 2, 3, 5, 8, ...). Segundo Cohn, essas escalas funcionam bem pois os valores mais altos incluem maior incerteza, refletindo a natureza preditiva das estimativas [43].

• Ciclo Semanal: O software em XP é produzido de forma iterativa e incremental. Essa prática sugere que uma equipe de XP deve planejar o trabalho de cada iteração uma semana por vez. A cada semana, os membros da equipe se reúnem para: refletir sobre o progresso realizado até o momento, planejar e priorizar as histórias da semana com o cliente e quebrar cada história em tarefas que serão implementadas pelos pares durante a semana (Jogo do Planejamento).

• Ciclo Trimestral: As releases são planejadas a cada trimestre. O plano do trimestre é de mais alto nível, geralmente representado por um tema. Temas são diferentes de histórias pois, ao invés de se preocupar com os detalhes, abrangem o todo: a forma em que o projeto se encaixa na organização. Durante o planejamento do trimestre, o time deve: identificar gargalos (principalmente externos à equipe), iniciar reparos e escolher as histórias mais alinhadas ao tema e que serão implementadas durante o trimestre.

• Folga: Inclua no plano algumas tarefas menores que possam ser removidas caso ocorra um atraso. Estimativas não devem ser consideradas um comprometimento, pois geralmente são feitas com base na experiência pessoal de cada desenvolvedor, estando sujeitas a erros. No entanto, é importante que o time se comprometa com as entregas para o cliente. Para acomodar o caráter subjetivo das estimativas, um tempo de folga deve ser incluído no plano, para que eventuais atrasos não atrapalhem a entrega da iteração ou da release, criando um vínculo de confiança e responsabilidade entre a equipe e o cliente.

• Build em 10 Minutos: O build automático do sistema inteiro e a bateria completa de testes deve rodar em até 10 minutos. Os itens em destaque são importantes: o build, assim como todas as tarefas repetitivas do projeto, deve ser automatizado, deve considerar o sistema inteiro (código-fonte e configurações de ambiente), deve rodar todos os testes e deve ser rápido o suficiente. As equipe de XP devem tentar atingir o máximo dos objetivos propostos por essa prática, pois quanto mais tempo o build demorar, menos será executado, diminuindo os ciclos de feedback e aumentando o tempo entre a introdução e a descoberta de um erro.

• Integração Contínua: O código-fonte fica armazenado num repositório compartilhado e cada par deve integrar suas alterações ao final de cada tarefa, após garantir que tudo está funcionando, realizando um build completo e rodando todos os testes. Os desenvolvedores interagem com o repositório diversas vezes por dia para trabalhar sempre numa versão atualizada do sistema. Dessa forma, o conhecimento do sistema se espalha por toda a equipe mais facilmente e a dificuldade de realizar uma integração grande se dilui em diversas integrações pequenas e freqüentes.

• Desenvolvimento Dirigido por Testes: Um dos aspectos mais importantes de XP é a ênfase nos testes automatizados. Essa prática sugere que os testes sejam escritos antes do código, trazendo benefícios como: ênfase no desenvolvimento (evitando generalizações desnecessárias), preocupação com acoplamento e coesão (geralmente o design está com problemas quando surge uma dificuldade para escrever o teste), confiança (o teste verifica o comportamento agora e no futuro) e ritmo (a próxima tarefa é sempre escrever o próximo teste ou fazer o teste passar, criando o ritmo conhecido como "vermelho, verde e refatoração" [36]).

• Design Incremental: A simplicidade é um conceito chave que permite a adaptação a mudanças. Para minimizar o custo com mudanças desnecessárias no futuro, os desenvolvedores devem sempre implementar o design mais simples - e não o mais simplista - com o mínimo da complexidade e flexibilidade necessária para atender às necessidades de negócio atuais. Porém, deve-se tomar cuidado com a interpretação dessa prática. Seu objetivo não é minimizar o investimento com design no curto prazo, mas sim manter esse investimento proporcional às necessidades do sistema conforme ele evolui. O design incremental deve ter suporte de outras práticas, como a refatoração e os testes automatizados gerados pelo desenvolvimento dirigido por testes para garantir que a equipe seja capaz de solucionar os problemas futuros com rapidez.

Práticas Corolárias

• Envolvimento Real com o Cliente: Faça com que as pessoas cujas vidas e negócios serão afetados pelo sistema façam parte da equipe. O cliente também deve fazer parte do time completo. Ele deve entender as necessidades de negócio e conhecer os verdadeiros usuários do sistema, para escrever histórias, definir prioridades e testes de aceitação e responder eventuais dúvidas sobre as funcionalidades desejadas.

• Implantação Incremental: Ao substituir um sistema legado, evite fazê-lo de uma só vez. É mais seguro substituir gradualmente partes das funcionalidades, deixando os dois sistemas funcionando ao mesmo tempo. Grandes implantações são muito arriscadas e têm custos humanos e econômicos muito altos [24].

• Continuidade da Equipe: Mantenha equipes eficientes trabalhando juntas. Existe uma tendência de tratar as pessoas como recursos substituíveis, que são trocadas de projetos diversas vezes para manter a utilização alta. No entanto, o valor no desenvolvimento de software não surge apenas do que as pessoas sabem ou fazem, mas também de seus relacionamentos e conquistas em equipe. Ignorar o valor das interações e relações para simplificar problemas de alocação é uma falsa economia [24].

• Diminuição da Equipe: Conforme a equipe melhora sua capacidade de produção, gradualmente reduza a carga sobre um dos membros, mantendo os outros trabalhando normalmente. Conforme a carga diminui sobre esse membro, ele pode ser liberado para formar novas equipes. Apesar do próprio Kent Beck não ter tido experiências com essa prática [24], incluiu-a em XP com base na sua eficácia no Sistema de Produção da Toyota [16, 5, 7]. Tentar fazer com que todos os membros pareçam ocupados pode possivelmente esconder um excesso de recursos na equipe.
• Análise de Causa Inicial: Sempre que um defeito for encontrado, conserte o problema e suas causas. O objetivo não é apenas fazer com que esse defeito específico nunca mais aconteça, mas também que o time nunca mais cometa o mesmo erro em outras situações. O processo de XP para consertar um defeito é: escrever um teste de aceitação automatizado que demonstre o problema, assim como o comportamento esperado; escrever um teste unitário com o menor escopo que também reproduz o defeito; corrigir o sistema, fazendo todos os testes passarem e, por fim, tentar descobrir a causa inicial do defeito não ter sido detectado anteriormente, realizando as mudanças necessárias para evitar que o erro aconteça novamente.
• Código Compartilhado: O repositório do código-fonte é compartilhado por toda a equipe e qualquer um pode fazer melhorias em qualquer parte do sistema. Ao invés de identificar responsáveis por cada parte do código, o time completo é responsável pelo sistema inteiro. Com isso, os membros da equipe adquirem uma ampla visão do sistema, facilitando a execução de refatorações e espalhando o conhecimento por toda a equipe.
• Código e Testes: Os únicos artefatos mantidos pela equipe são o código e os testes. Documentação deve ser evitada e, caso estritamente necessária, deve ser gerada a partir do código e dos testes. A principal forma de comunicação em uma equipe de XP é a conversa. Artefatos que se tornem obsoletos com o tempo não agregam valor ao sistema e ao negócio. Eliminar desperdícios permite melhorar as áreas que agregam valor, aquelas que definem o que o sistema faz hoje e o que a equipe pode fazer com o sistema amanhã.
• Repositório de Código Unificado: A equipe deve desenvolver em um repositório único. Ramificações podem existir, mas devem ser evitadas. Quanto maior o número de versões concorrentes do mesmo código, maior o trabalho de sincronização e mais difícil é o entendimento pela equipe. Linhas paralelas devem ser integradas rapidamente e os motivos de sua existência devem ser reconsiderados constantemente e não tidos como verdade absoluta.
• Implantação Diária: Coloque novas versões do sistema em produção toda noite. Dessa forma, o ciclo de feedback entre o que está sendo feito pelo programador e o que está sendo utilizado pelo usuário é sempre rápido e eficiente. Para que essa prática seja eficaz, muitas outras devem estar funcionando bem. É preciso garantir que o número de defeitos seja baixo e que as ferramentas de build e implantação automatizem todo o processo de entrega, possibilitando inclusive voltar uma versão, caso necessário.
• Contrato de Escopo Negociável: Contratos devem fixar tempo, custo e qualidade, deixando o escopo preciso aberto para negociação. As equipes de XP se adaptam a mudanças, permitindo que o cliente faça correções no escopo do software conforme seu aprendizado do sistema evolui. Em XP, o escopo é revisado freqüentemente para garantir que a equipe está sempre trabalhando no que é mais importante para o cliente.
• Pague-Pelo-Uso: Essa prática sugere a utilização do dinheiro como feedback final. Em sistemas pay-per-use, você cobra a cada vez que o sistema é utilizado. A conexão do fluxo de economia ao desenvolvimento de software provê informações precisas e atualizadas para direcionar melhorias no sistema. No modelo mais utilizado pela indústria, o cliente paga a cada release, porém isso coloca os interesses da equipe de desenvolvimento e do cliente em conflito. Enquanto a equipe deseja um número maior de releases com pouca funcionalidade, o cliente deseja o menor número possível de releases contendo o máximo de funcionalidade. Essa tensão gera problemas de comunicação e feedback.

Comparação com as Práticas da Primeira Versão

Devido à mudança na abordagem de apresentação das práticas de XP, Kent Beck transformou as 12 práticas originais em 24 práticas divididas em práticas primárias e práticas corolárias, conforme descrito na Seção Práticas. Algumas práticas da primeira versão ainda aparecem de forma subjetiva na descrição das novas práticas, mas para ter uma melhor base de comparação, é importante descrevê-las como foram apresentadas na primeira edição do livro [35]:
• Refatoração: A refatoração é uma técnica sistemática para reestruturar o código existente, alterando sua estrutura interna, porém mantendo seu comportamento externo [29]. Alguns exemplos de refatorações são: a remoção de código duplicado, a mudança do nome de um método ou variável e a extração de um trecho de código para um método auxiliar. O objetivo é sempre tornar o código e o design mais simples, legível, limpo e preparado para mudanças.
• Metáfora: Todos os membros da equipe, incluindo programadores e clientes, devem conversar sobre o sistema numa linguagem comum. Essa linguagem deve ser entendida tanto pelas pessoas técnicas, quanto pelas pessoas de negócio. Isso pode ser obtido através de uma metáfora comum que relaciona abstrações do sistema com objetos de um certo domínio, existentes no mundo real. Essa é uma das práticas mais difíceis de introduzir em uma equipe inexperiente, pois está diretamente ligada à comunicação e ao modo como as pessoas estão dispostas a compartilhar seus desejos e suas idéias. Essa prática estava bastante alinhada com um padrão descrito por Ward Cunningham, que ficou conhecido como Sistema de Nomes [44]. Mais recentemente, o uso dessa linguagem ubíqua para representar conceitos do domínio no código-fonte ficou popularizada com a técnica de

modelagem definida por Eric Evans, conhecida como Design Dirigido pelo Domínio (Domain Driven Design) [45].
• Padronização de Código: Antes de dar início à implementação, o time define um conjunto de padrões de codificação para escrita do código do sistema. Isso torna o código homogêneo e mais fácil de entender, melhora a comunicação, facilita a refatoração e promove a propriedade coletiva do código.

A **Tabela 1** é uma adaptação do autor com base num artigo de Michele Marchesi [46] onde é apresentada uma comparação entre a primeira e a segunda edição de XP, com destaque para a relação entre as práticas novas e as práticas originais. As práticas entre parênteses são aquelas que não aparecem explicitamente na nova edição de XP.

Adaptações das Práticas de XP

A adoção de um método ágil como XP não depende simplesmente da aplicação direta das práticas. O ciclo empírico de inspeção e adaptação dos métodos ágeis sugere que as equipes estejam em busca freqüente de melhoria e algumas adaptações são permitidas. Conforme XP começou a ser utilizada em mais projetos, uma nova prática foi sugerida: "Conserte XP quando ela falha" [47]. Uma boa fonte de inspiração para tais adaptações são as práticas sugeridas em outros métodos ágeis [48]:
• Reuniões em Pé: prática utilizada em Scrum [14, 41] que consiste numa reunião informal curta e diária, realizada no início do dia de trabalho na qual cada membro da equipe responde a três perguntas: "O que fez ontem?", "O que pretende fazer hoje?" e "Quais problemas impedem o seu progresso?". Os membros da equipe participam dessa Reunião em Pé para garantir que sua duração seja curta. Além disso, eventuais problemas que forem levantados deverão ser discutidos posteriormente apenas pelos interessados nos problemas específicos.

Tabela de Comparação entre as práticas de XP	
Primeira Edição	**Segunda Edição**
Programação Pareada	Programação Pareada
Versões Pequenas	Ciclo Semanal, Implantação Incremental e Implantação Diária
Integração Contínua	Integração Contínua
Desenvolvimento Dirigido por Testes	Desenvolvimento Dirigido por Testes
Jogo do Planejamento	Histórias, Ciclo Semanal, Ciclo Trimestral e Folga
Cliente com os Desenvolvedores	Time Completo e Envolvimento Real com o Cliente
(Refatoração)	Design Incremental
Design Simples	Design Incremental
(Padronização de Código)	Código Compartilhado
Propriedade Coletiva do Código	Código Compartilhado e Repositório de Código Unificado
(Metáfora)	Design Incremental
Ritmo Sustentável	Trabalho Energizado e Folga

Tabela 1. Comparação entre as práticas da primeira e da segunda edição de XP, adaptado de [46]

• Retrospectivas: prática originada na Família Crystal [17] e também conhecida como Reflection Workshops. Elas são reuniões realizadas ao final de cada iteração na qual o processo de desenvolvimento é avaliado, a equipe discute as lições aprendidas com a experiência e planeja as mudanças para o próximo ciclo de desenvolvimento [49]. Existem diversos formatos para as reuniões de retrospectiva, mas no mais comum a equipe discute "O que funcionou bem?", "O que pode melhorar?" e "Quais problemas nos preocupam?". Normalmente, ao final da reunião, o time terá um conjunto de ações em cada uma das categorias acima e poderá priorizá-las e escolher as mais importantes para implementar na próxima iteração. As ações escolhidas devem ser capturadas em um pôster, que será anexado à Área de Trabalho Informativa. Para garantir que as idéias sejam discutidas abertamente, é comum evitar dar nome aos culpados. Essa prática é enfatizada na diretiva principal das retrospectivas, proposta por Kerth [49]:

> "Não importa o que for descoberto, nós entendemos e acreditamos que todos fizeram o melhor trabalho possível, dado o conhecimento na época, as habilidades e os recursos disponíveis na situação em questão."

Papéis na Equipe de XP

A prática do Time Completo sugere que uma equipe de XP seja formada por uma variedade de pessoas, com todas as características e habilidades necessárias para o sucesso do projeto. A primeira edição de XP [35] estava muito mais voltada para programadores, porém, na segunda edição [24], Kent Beck descreve a importância da valorização de todos os outros papéis dentro de uma equipe. Vale ressaltar que os papéis podem ser assumidos por pessoas diferentes, em momentos distintos e que uma mesma pessoa pode desempenhar mais de um papel. A idéia é proporcionar um ambiente produtivo no qual cada membro possa contribuir da melhor forma para o projeto. Alguns dos papéis que podem fazer parte de uma equipe de XP são:
• Programadores: Responsáveis por estimar histórias e tarefas, quebrar histórias em tarefas, escrever testes e código, automatizar processos tediosos e melhorar o design do sistema. Existem dois papéis especiais para programadores. Geralmente, o mais experiente em XP atua como coach (treinador), verificando e auxiliando os membros na execução das práticas no dia-a-dia. Já o tracker está constantemente coletando e compartilhando dados sobre o andamento do projeto e do processo. O tracker é responsável por criar e espalhar cartazes e gráficos na Área de Trabalho Informativa.
• Arquitetos: Procuram e executam refatorações de larga escala no sistema, escrevem testes de carga automatizados para definir cenários de estresse e auxiliam os programadores no particionamento do sistema, mantendo a ênfase no design de alto nível.

• Analistas de Teste: Trabalham com o cliente e com os analistas de negócio para escrever testes de aceitação automatizados, definindo os cenários de sucesso e erro de cada história. Além disso, também treinam os programadores em técnicas e ferramentas de teste.
• Analistas de Negócio: Trabalham com o cliente para definir as histórias do sistema e auxiliam os programadores a interpretar o valor de negócio de cada funcionalidade.
• Projetistas de Interação: Avaliam o modo como o sistema está sendo utilizado pelos usuários finais, identificando e sugerindo novas histórias e melhorias na interface gráfica.
• Gerentes de Projeto: Facilitam a comunicação dentro do time, removendo empecilhos e coordenando a comunicação com pessoas externas à equipe do projeto (fornecedores, clientes externos ou o resto da organização).
• Gerentes de Produto: Escrevem e priorizam histórias para o ciclo semanal e definem os temas para o ciclo trimestral. Além disso, encorajam a comunicação entre a equipe e o cliente para garantir que as preocupações e necessidades mais imediatas do cliente e do usuário final sejam atendidas.
• Executivos: Trazem confiança, coragem e responsabilidade para a equipe. Além disso, avaliam os objetivos do time em relação às metas da organização, monitorando e facilitando a criação de um ambiente voltado à melhoria contínua.
• Redatores Técnicos: Por olharem o sistema do ponto de vista do usuário final, os redatores técnicos trazem feedback rápido sobre as funcionalidade do sistema e estreitam o relacionamento da equipe com o cliente, levantando dúvidas e sugerindo melhorias.
• Usuários: Por utilizar o sistema diariamente, podem ajudar a escrever e escolher histórias e tomar decisões de domínio durante o desenvolvimento. Por representar toda a comunidade de usuários, é interessante que tenham experiência com sistemas similares para tomar as decisões mais adequadas.
• Recursos Humanos: Cuidam de problemas burocráticos como contratação e avaliações. Kent Beck sugere a avaliação do time ao invés de avaliações individuais para evitar conflitos internos que atrapalhem o bom andamento do projeto [24].

Discussão das Formas de Adoção e Conclusões

Este artigo apresentou um dos métodos ágeis mais conhecidos, a Programação Extrema, descrevendo seus valores, princípios e práticas. A escolha da melhor forma de adotar XP deve levar em conta todos os fatores discutidos neste artigo e a abordagem de implantação pode variar de equipe para equipe. Enquanto algumas se sentem confortáveis com a abordagem mais rígida proposta na primeira edição, aplicando todas as 12 práticas de uma vez, outras podem preferir começar de forma mais gradual, com algumas das práticas primárias antes de partir para as práticas corolárias.

Kent Beck discute sobre os diferentes estilos de adoção fazendo uma analogia com as formas de se entrar numa piscina [50]: alguns preferem entrar de forma cuidadosa e gradual, "um pé de cada vez", evitando grandes estragos, porém gastando mais tempo; outros preferem entrar de uma vez, no estilo "bola de canhão", espalhando bastante água e passando por uma fase inicial caótica, que pode trazer maiores benefícios no curto prazo; por fim, equipes que precisam de um resultado rápido sem o risco da fase caótica, podem adotar o estilo "mergulho de cabeça" com a ajuda de um treinador externo, que vai tornar a entrada na água mais suave, pela experiência adquirida em outras situações.

Kent Beck ainda sugere alguns pontos de atenção que devem ser discutidos pela equipe para escolher a forma de adoção mais apropriada:
• Em quanto tempo o time precisa dos resultados?
• O quão dramático devem ser os resultados?
• Quanto a organização está disposta a gastar em ajuda externa?
• O quão forte são as relações entre os membros da equipe e entre a equipe e o resto da organização?

XP não deve ser utilizado em organizações cujos valores reais não se alinham com os valores de XP. Organizações que preferem dar valor a segredos, isolamento, complexidade, timidez e falta de respeito (manifestada como prepotência ou excesso de autoritarismo) não terão sucesso com a adoção de XP. Vale ressaltar ainda que uma adoção de sucesso de XP precisa abraçar os valores e princípios por trás das práticas. A adoção de algumas práticas pode trazer um pequeno benefício no curto prazo, mas as melhorias mais amplas propostas por XP só serão atingidas se houver sinergia entre os valores da equipe e de XP. Um grande choque cultural pode prejudicar a adoção de XP [51].

O discurso de XP mudou desde seu lançamento em 1999: enquanto a primeira edição enfatizava mais "como" XP funciona, a segunda edição enfatiza muito mais o "por quê". Enquanto a primeira edição era mais voltada para os programadores, a segunda edição tem um discurso mais inclusivo e flexível, trazendo benefícios para todos os envolvidos no desenvolvimento de software. ●

Referências

[1] Mira Kajko-Mattsson, Ulf Westblom, Stefan Forssander, Gunnar Andersson, Mats Medin, Sari Ebarasi, Tord Fahlgren, Sven-Erik Johansson, Stefan T¨ornquist, and Margareta Holmgren. Taxonomy of problem management activities. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, pages 1–10, 2001.

[2] Barry W. Boehm. Industrial software metrics top 10 list. IEEE Software, 4(5):84–85, Set 1987.

[3] Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. IEEE Transactions on Software Engineering, 14(10):1462–1477, Oct 1988.

[4] Barry W. Boehm and Victor R. Basili. Software defect reduction top 10 list. Computer, 34(1):135–137, Jan 2001.

[5] Mary Poppendieck and Tom Poppendieck. Lean Software Development: An Agile Toolkit for Software Development Managers. Addison-Wesley Professional, 2003.

[6] Jim Johnson. ROI, it's your job. Keynote Speech at Third International Conference on Extreme Programming (XP2002), May 2002.

[7] Mary Poppendieck and Tom Poppendieck. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley Professional, 2006.

[8] Jim Johnson, et al. CHAOS in the new millenium. Technical report, Standish Group, 2000.

[9] The CHAOS report. Technical report, Standish Group, 1994.

[10] The CHAOS report. Technical report, Standish Group, 2003.

[11] Frederick T. Sheldon, Krishna M. Kavi, Robert C. Tausworth, James T. Yu, Ralph Brettschneider, and William W. Everett. Reliability measurement: From theory to practice. IEEE Software, 9(4):13–20, Jul 1992.

[12] Craig Larman. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Professional, 2003.

[13] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. Disponível em: www.agilemanifesto.org, 2001. Acessado em: 30/10/2006.

[14] Ken Schwaber and Mike Beedle. Agile Software Development with Scrum. Prentice Hall, 2001.

[15] Christ Vriens. Certifying for CMM level 2 and ISO9001 with XP@scrum. In Agile Development Conference, pages 120–124. IEEE Computer Society, 2003.

[16] Taiichi Ohno. Toyota Production System: Beyond Large-Scale Production. Productivity Press, 1998.

[17] Alistair Cockburn. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Professional, 2004.

[18] Stephen R. Palmer and John M. Felsing. A Practical Guide to Feature Driven Development. Prentice Hall, 2002.

[19] Peter Coad, Jeff de Luca, and Eric Lefebvre. Java Modeling Color with UML: Enterprise Components and Process with Cdrom. Prentice Hall, 1999.

[20] Jim Highsmith. Messy, exciting, and anxiety-ridden: Adaptive software development. In American Programmer, volume 10, 1997.

[21] M. Mitchell Waldrop. Complexity: The Emerging Science at the Edge of Order and Chaos. Simon & Schuster, 1992.

[22] James Martin. Rapid Application Development. Macmillan Publishing Co., 1991.

[23] Jennifer Stapleton. DSDM: A framework for business centered development. Addison- Wesley Professional, 1997.

[24] Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2nd edition, 2004.

[25] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In OOPSLA, pages 1–6, October 1989.

[26] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical Report CR–87–43, Tektronix, Inc., September 1987. Presented at the OOPSLA-87 Workshop on Specification and Design for Object-Oriented Programming.

[27] Bo Leuf and Ward Cunningham. The Wiki Way: Quick Collaboration on the Web. Addison-Wesley Professional, 2001.

[28] William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois, 1992.

[29] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[30] Kent Beck. Simple smalltalk testing: With patterns. Technical report, First Class Software, Inc., Outubro 1994. Disponível em: www.xprogramming.com/testfram.htm.

[31] Kent Beck. SUnit project. Disponível em: sunit.sourceforge.net/. Acessado em: 30/10/2006.

[32] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. Java Report, 3(7), July 1998. Acessado em Jan/06.

[33] Kent Beck and Erich Gamma. JUnit project. Disponível em: www.junit.org/. Acessado em: 30/10/2006.

[34] xUnit family. Disponível em: www.xprogramming.com/software.htm. Acessado em: 30/05/2007.

[35] Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 1999.

[36] Kent Beck. Test Driven Development: By Example. Addison-Wesley Professional, 2003.

[37] Scott Ambler. Crossing the chasm. Dr. Dobb's Journal, disponível em: www.ddj.com/dept/architect/187200223, May 2006. Acessado em: 30/10/2006.

[38] Vinícius Manhães Teles. Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. Editora Novatec, 2004.

[39] Kent Beck and Martin Fowler. Planning Extreme Programming. Addison-Wesley Professional, 2000.

[40] Alistair Cockburn. Agile Software Development: The Cooperative Game. Addison Wesley Professional, 2nd edition, 2006.

[41] Ken Schwaber. Agile Project Management with Scrum. Microsoft Press, 2004.

[42] Tom Demarco and Timothy Lister. Peopleware: Productive Projects and Teams. Dorset House Publishing Company, Incorporated, 2nd edition, 1999.

[43] Mike Cohn. Agile Estimating and Planning. Prentice Hall PTR, 2005.

[44] Ward Cunningham. System of names. Disponível em: c2.com/cgi/wiki?SystemOfNames. Acessado em: 30/05/2007.

[45] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[46] Michele Marchesi. The new XP. Disponível em: www.agilexp.org/downloads/TheNewXP.pdf. Acessado em: 30/05/2007.

[47] Don Wells. Fix XP when it breaks. Disponível em: www.extremeprogramming.org/rules/fixit.html. Acessado em 31/05/2007.

[48] Danilo Sato, Dairton Bassi, and Alfredo Goldman. Extending extreme programming with practices from other methodologies. In 1st Workshop on Rapid Application Development (WDRA'07) in the Brazilian Symposium of Software Quality (SBQS'07), 2007.

[49] Norman L. Kerth. Project Retrospectives: A Handbook for Team Reviews. Dorset House Publishing Company, 2001.

[50] Kent Beck and Cynthia Andres. Getting started with XP: Toe dipping, racing dives, and cannonballs. Disponível em: www.threeriversinstitute.org/ToeDipping.pdf. Acessado em: 30/10/2006.

[51] Laurent Bossavit. The unbearable lightness of programming: a tale of two cultures. Whitepaper, disponível em: www.exoftware.com/i/white_paper/file/6/The_20unbearable_20.pdf. Acessado em: 30/10/2006.

UML – Casos de Uso

Entendendo os casos de uso na prática – um estudo de caso – Parte I

De que se trata o artigo?
Este artigo apresenta o caso de uso de uma maneira prática. Será desenvolvido um estudo de caso, passando por todas as fases da modelagem dos requisitos quanto aos casos de uso.

Para que serve?
Fornecer aos desenvolvedores ou estudantes da área de sistemas uma linha de entendimento com o intuito de orientá-los a escrever seus próprios casos de uso.

Em que situação o tema é útil?
Para quem ainda não conhece como escrever um caso de uso, ou para quem já o faz há algum tempo, mas não tem conseguido o sucesso esperado.

Ana Cristina Melo
informatica@anacristinamelo.com.br
É especialista em Análise de Sistemas e professora de graduação e pós-graduação da Universidade Estácio de Sá. Atua em análise e programação há 21 anos. Autora do livro "Desenvolvendo aplicações com UML – do conceitual à implementação", na segunda edição, e "Exercitando modelagem em UML". Palestrante em alguns eventos, entre eles, Congresso Fenasoft, OD e Sepai.

Este artigo apresentará uma das principais ferramentas oferecidas pela UML — o caso de uso, demonstrando sua utilização na prática, por meio do desenvolvimento de um estudo de caso. Nessa primeira parte detalharemos o problema, os requisitos e extrairemos os casos de uso necessários à implementação. Escreveremos os primeiros cenários, com seus respectivos protótipos, preparando o sistema para se tornar funcional.

O problema

No artigo anterior, explicamos o caso de uso como uma das principais ferramentas para capturar os requisitos do sistema. Neste usaremos os conceitos apresentados para desenvolver um estudo de caso. É preciso ressaltar que os modelos de documentos aqui apresentados são oriundos da experiência da autora como analista de sistemas, servindo apenas como exemplo para os profissionais da área. E que os requisitos descritos têm objetivo somente como exercício, não representando, na prática, as necessidades de um sistema deste tipo.

Suponha que um analista XYZ tenha aberto uma consultoria e, dentre os primeiros clientes, surja o dono de uma pequena papelaria, com uma única filial, desejando informatizar sua empresa.

O analista marca algumas reuniões e após estabelecer o escopo do sistema, dá início ao levantamento dos requisitos. De posse desses requisitos, seu trabalho é modelá-los, de tal forma que represente a expectativa desse cliente. Para essa modelagem, ele utiliza os casos de uso, pois pretende fazer uso da tecnologia de orientação a objetos. Os casos de uso e seus protótipos são apresentados para o cliente, que sugere alterações. Esse processo perdura ciclicamente até que o cliente dê sua aprovação para o analista.

Então, o projeto estará pronto para uma nova etapa, no qual as classes serão identificadas e o sistema começará a ser implementado.

Vamos acompanhar como ficou essa fase inicial da modelagem.

O levantamento de requisitos

O analista XYZ, após cada reunião elaborou uma ata que foi devidamente encaminhada para o cliente, a fim de validar com o mesmo o entendimento dos requisitos. Na **Tabela 1** vemos o exemplo de uma dessas atas, representando a primeira reunião ocorrida.

Após todas as reuniões, o analista XYZ preparou um único documento consolidando todos os requisitos. Esses requisitos estão listados abaixo:

ATA DE REUNIÃO
Data da reunião: 10/10/2008
Participantes: XYZ – analista de sistemas MNO – diretor da Papelaria ABC
Assunto: Levantamento de requisitos
O sr. MNO apresentou ao sr. XYZ a papelaria ABC, que tem cerca de 800 m2, com seções arrumadas por tipo de item: cadernos e fichários, pastas, papéis para impressão etc. A papelaria conta com vendedores que atendem ao público, recebendo comissão pelas vendas efetuadas. A comissão tem variação de acordo com a época do ano. No período de janeiro a março, a comissão é de 7% sobre as vendas. Nos outros meses é de 5%. Além disso, todo vendedor possui um salário fixo. Não há cotas a serem alcançadas. Após escolher os produtos, com a ajuda ou não de um vendedor, o cliente deve se dirigir a um deles para tirar o pedido de venda. Esse pedido conterá todos os produtos e liberará para o caixa apenas o pagamento. Só existe um caixa na papelaria, responsável apenas pelo recebimento do pagamento. Se um cliente tiver dúvidas sobre a existência de um produto ou sobre o preço do mesmo, poderá consultar um dos vendedores, que fará a pesquisa no sistema. Todos os produtos recebem um código que não está associado ao código de barras. São coladas etiquetas com esses códigos em todos os produtos. Da mesma forma, em cada prateleira há uma etiqueta com o nome dos produtos, seu código e o preço atualizado. Assim, nos produtos não há etiqueta de preço. O sr. MNO solicitou que além do sistema que registrará as vendas, seja disponibilizado um módulo para controlar o estoque. (...)
Pendências para a próxima reunião: - O sr. MNO fará uma lista de todos os relatórios que deseja, contendo todas as informações necessárias e detalhes de emissão, como freqüência.

Tabela 1. Exemplo da ata de reunião com o levantamento de requisitos

• O sistema conterá dois módulos principais: gestor de estoque/vendas e pdv (ponto de vendas).
• Não será foco dessa versão a gestão de fornecedores, pagamento dos funcionários e controle de contas a pagar.
• A papelaria conta com vendedores que atendem ao público, recebendo comissão pelas vendas efetuadas.
• Para a gestão de estoque/vendas, são desejáveis os seguintes controles:
  - cadastro de produtos
  - cadastro de fabricantes
  - cadastro de vendedores
  - atualização de estoque
  - registro de troca e devolução de produto
  - consulta de preços
  - registro de vendas
  - emissão de etiquetas de venda
  - relatório de reposição
  - consulta de faturamento diário, semanal ou mensal
  - atualizar preços e lançar promoções de produtos
• Para o PDV, são desejáveis os seguintes controles:
  - registro de pagamento das vendas
  - realizar retirada de capital
  - suprimento de capital (entrada de capital no caixa)
  - abertura do caixa
  - fechamento do caixa
• O cadastro de produtos será feito pelo departamento de estoque, e para cada produto é preciso registrar: código (gerado automaticamente pelo sistema), nome do produto (ex: caderno universitário), detalhes sobre o produto (ex: 256 folhas), unidade de venda (ex: unidade, kit, pacote, metro etc), fabricante, cor, percentual de lucro, tipo de produto (ex: cadernos e fichários, papéis para impressão etc) e a foto do produto.
• Nenhum produto pode ser excluído do estoque, apenas desativado, por meio de uma data, que determinará que a partir daquele momento não mais poderá ser vendido. Nenhum produto com itens ainda em estoque pode ser desativado.
• O estoque de cada produto será automaticamente gerado no momento do cadastramento do produto. O estoque será inicializado com zero, porém deve ser informada a quantidade mínima de estoque, que identificará a necessidade de reposição.
• As entradas no estoque são feitas a partir das notas fiscais de compras e são responsabilidade do departamento de estoque. Não haverá controle dos fornecedores, mas será preciso manter um cadastro com o nome dos fornecedores, para a futura versão do sistema. Para cada nota fiscal, armazenar número, data de emissão e fornecedor. A cada entrada de produto no estoque, deve ser registrado: quantidade e valor de custo. O sistema deverá apresentar o valor atual de venda e calcular o novo preço de venda do produto com base no valor de compra. Caberá ao usuário confirmar o novo preço ou manter o existente. Deve-se manter um histórico dos preços.
• Atualizações de preços e cadastramento de promoções devem ser feitas pela gerência. As promoções de produtos devem ser cadastradas por período (ex: preço promocional da cola 90g no período de 01/01/2009 a 15/01/2009).

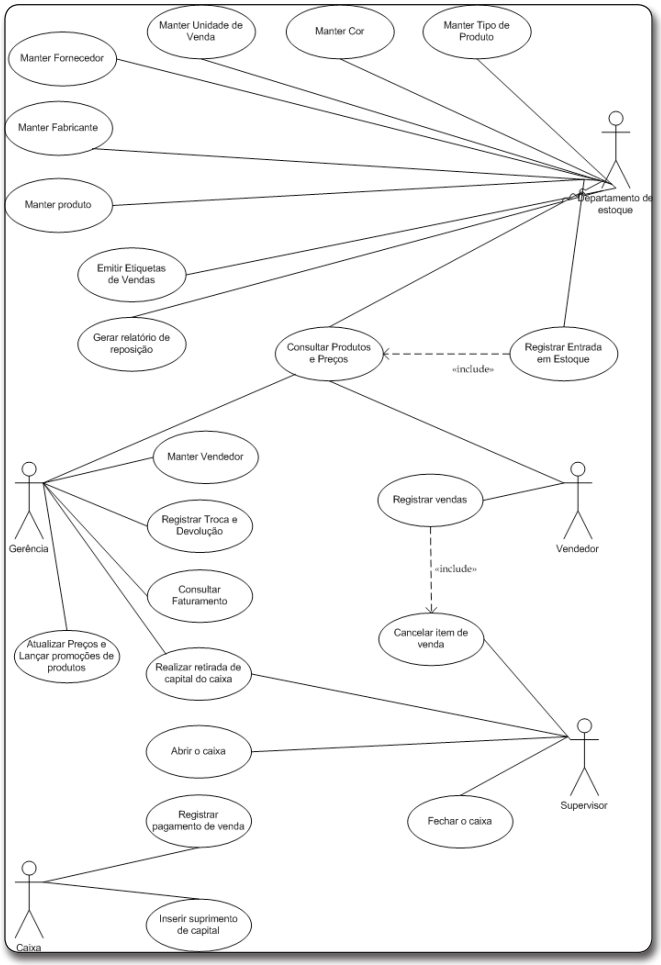


Figura 1. Diagrama de casos de uso para o sistema gestor da papelaria

• Um produto pode conter variações. Toda variação de um produto será um produto com código próprio, contudo seus dados principais (nome, fabricante e tipo do produto) são alteráveis apenas no produto principal.

• O cadastro de fabricantes será feito pelo departamento de estoque e deve conter apenas um código gerado pelo sistema e o nome.

• O cadastro de vendedores será feito pela gerência da papelaria e deve conter apenas seu código e nome. A comissão é a mesma para todos os vendedores, respeitando-se os percentuais estipulados para as épocas do ano. Sendo assim, cadastra-se o período inicial e final (dd/mm/yyyy) e a comissão que será paga sobre as vendas, independente dos vendedores.

• A troca ou devolução de produto será feita pela gerência da papelaria. Para ambos os casos, o produto deve retornar ao estoque, e o cliente receberá um vale-troca que só poderá ser utilizado no mesmo dia.

• A consulta de preços poderá ser feita pelo código e/ou pela descrição (abrangendo detalhes, cor e fabricante), possibilitando a localização de uma lista de produtos que atenda ao critério, ou até mesmo a lista de produtos que pertençam a um mesmo grupo (variação). Pode ser efetuada pelo vendedor, supervisor ou pela gerência da papelaria.

• O registro da venda será feito pelo vendedor e se dará a partir do código do produto. Se o vendedor não souber o código, poderá fazer a pesquisa pelo nome. Para cada produto é informada a quantidade comprada. Ao final de uma venda, é gerada uma nota com todos os produtos, contendo um número de venda e o total a pagar. Essa nota deve ser paga no caixa. O supervisor poderá autorizar um desconto para a venda total. Isso se dará no momento do registro da venda. São formas de pagamento aceitas: dinheiro, débito e cartão de crédito. Para débito e cartão de crédito, o sistema não se conectará à administradora de cartão ou aos sistemas bancários. O sistema apenas aguardará a confirmação do caixa quanto à autorização externa. Contudo, deve-se registrar qual a bandeira do cartão e os quatro últimos dígitos do mesmo.

• Durante a venda, qualquer item pode ser cancelado, desde que autorizado pelo Supervisor.

Modelando o diagrama de casos de uso

De posse dos requisitos, o analista pôde dimensionar o tamanho do sistema, e chegar ao seu diagrama de casos de uso.

Todos os cadastros são representados como casos de uso de manutenção (inclusão, alteração, exclusão e/ou consulta), identificados pelo verbo manter. (ex: Manter Produto indica cadastro, alteração, desativação e pesquisa de produto).

Num primeiro momento, foram identificados todos os atores responsáveis por atuar nesse sistema. Para cada ator, verificam-se suas responsabilidades. De cada uma dessas responsabilidades, nasce um ou mais casos de uso.

Ao determinar um caso de uso, devemos ter em mente seu caráter bem definido de funcionalidade. Não podemos estabelecer um caso de uso tipo "mil e uma utilidades". Um caso de uso deve se encerrar num objetivo único.

Diante do exposto, o analista chegou à seguinte **lista de casos de uso**, descrita na **Tabela 2**.

LISTA DE CASOS DE USO	
Ator: Departamento de estoque	
- Manter Produto	- Manter Fabricante
- Manter Tipo de Produto	- Manter Fornecedor
- Manter Unidade de Venda	- Manter Cor
- Registrar Entrada em Estoque	- Emitir etiquetas de vendas
- Gerar relatório de reposição	- Consultar Produtos e Preços
Ator: Vendedor	
- Consultar Produtos e Preços	- Registrar Vendas
Ator: Gerência	
- Manter Vendedor	- Registrar Troca e Devolução
- Consultar Faturamento	- Consultar Produtos e Preços
- Atualizar preços e lançar promoções de produtos	- Realizar retirada de capital do caixa
Ator: Supervisor	
- Cancelar item de venda	- Realizar retirada de capital do caixa
- Abrir o caixa	- Fechar o caixa
Ator: Caixa	
- Registrar pagamento de venda	- Inserir suprimento de capital

Tabela 2. Lista de casos de uso

Listagem 1. UC Manter Produto

```
Descrição: Este caso de uso tem por objetivo manter o cadastro
de produtos, permitindo a inclusão, alteração, desativação ou
consulta de produtos.
Atores: Departamento de Estoque
Pré-condição: existir cadastro prévio de fabricantes, tipos de
produto, unidades de venda e cores.

Cenário principal:
1.O sistema prepara uma lista de produtos cadastrados, exibindo
para cada um: código, nome do produto, detalhes, cor, fabricante e
código do produto principal (se for um produto variação).

   1.1.O usuário pode pesquisar um produto, informando os
   seguintes critérios:
      - código (com a opção de todas as variações do produto localizado) ou
      - descrição do produto (trecho ou integral) e/ou
      - fabricante (trecho ou integral)

2.O sistema habilita as seguintes opções para o usuário:

   2.1.Inclusão de produto
   2.2.Inclusão por variação de produto

      2.2.1.Para incluir uma variação, o usuário deve selecionar o
            produto principal

   2.3.Alteração de produto

      2.3.1.Para alteração, o usuário deve pré-selecionar o produto

   2.4.Desativação de produto: opção habilitada apenas para
       produtos que estejam com o saldo em estoque igual a zero.

      2.4.1.Para desativação, o usuário deve pré-selecionar o produto

3.Para a opção de "Inclusão" ou "Alteração":

   3.1.O sistema prepara uma lista de todos os fabricantes cadastrados.
   3.2.O sistema prepara uma lista de todos os tipos de produtos
       cadastrados.
   3.3.O sistema prepara uma lista de todas as unidades de venda
       cadastradas.
   3.4.O sistema prepara uma lista de todas as cores cadastradas.
   3.5.No caso de alteração, o sistema exibe o código do produto,
       desabilitado para ediçao.
   3.6.O usuário informa/edita:

      3.6.1.nome do produto
      3.6.2.detalhes do produto
      3.6.3.cor, selecionada da lista previamente preparada
      3.6.4.fabricante, selecionado da lista previamente preparada
      3.6.5.unidade de venda, selecionada da lista previamente
            preparada
      3.6.6.percentual de lucro
      3.6.7.tipo do produto, selecionado da lista previamente
            preparada
      3.6.8.no caso de inclusão: quantidade mínima em estoque
      3.6.9.foto do produto

4.Para a opção de "Inclusão por variação de produto":

   4.1.O sistema prepara uma lista de todas as unidades de venda
       cadastradas.
   4.2.O sistema prepara uma lista de todas as cores cadastradas.
   4.3.O sistema automaticamente exibe as informações do produto
       principal:

      4.3.1.nome do produto
      4.3.2.fabricante
      4.3.3.tipo do produto

   4.4.O sistema libera a inclusão de um novo produto, com base na
       variação do produto já existente.
   4.5.O usuário informa:

      4.5.1.detalhes do produto
      4.5.2.unidade de venda, selecionada da lista previamente
            preparada
      4.5.3.cor, selecionada da lista previamente preparada
      4.5.4.percentual de lucro
      4.5.5.quantidade mínima em estoque
      4.5.6.foto do produto

5.Para a opção de "Desativação":

   5.1.O usuário informa a data da desativação.

6.O usuário confirma a operação.

   6.1.O sistema atualiza o cadastro de produtos.
   6.2.Para a opção de "Inclusão" (individual ou como variação de
       produto):
     6.2.1.O sistema gera automaticamente um código de produto.
     6.2.2.O estoque inicial do sistema é criado, com as seguintes
           informações:
      - Estoque inicial = zero
      - Qtd Mínima de Estoque = quantidade informada pelo usuário
      (item 3.6.8 ou 4.5.5)

   6.3.Para opção de "Inclusão por variação de produto":

      6.3.1.O sistema automaticamente associa o produto de variação
            ao produto principal.

Cenários alternativos:
Pesquisa de produto

1.a. A pesquisa de produto deve desconsiderar caixa alta e baixa,
     acentuação e realizar a localização dos trechos em qualquer
     ordem que os mesmos apareçam no cadastro.

1.b. A pesquisa da descrição deve ser feita ao mesmo tempo nos
     campos: nome do produto, detalhes e cor, retornando os
     produtos que contenham os critérios de pesquisa em qualquer um
     desses campos.
     Permissão da opção de "Inclusão por variação de produto"

2.a. Se o usuário selecionar um produto que seja de variação,
     exibir mensagem de erro e retornar ao passo 1.
     Duplicidade de nome do produto + fabricante + detalhes

3.a. Se já houver cadastrado o mesmo nome do produto com os
     mesmos detalhes e mesmo  fabricante, que não seja uma
     variação de produto, o sistema deve exibir mensagem de erro, e
     retornar ao passo 2.
     Alteração de Variação de Produto

3.b. No caso de alteração de um produto que seja variação de
     outro, o sistema deve permitir a edição somente dos campos:
     detalhes, cor, unidade de venda e percentual de lucro,
     exibindo o código do produto principal e o código do produto
     de variação.
```

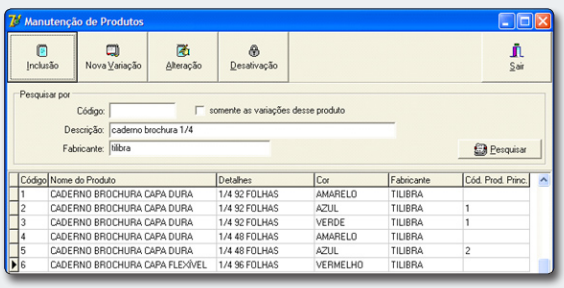


Figura 2. Protótipo (1) do UC Manter Produto – Tela principal da manutenção

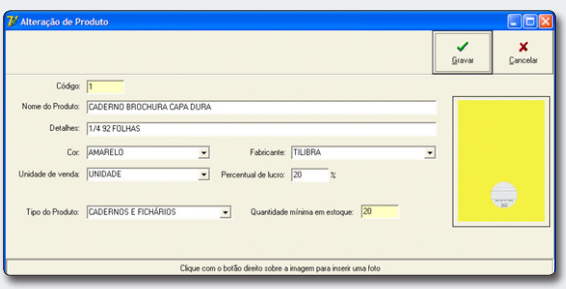


Figura 3. Protótipo (2) do UC Manter Produto – Alteração de produto

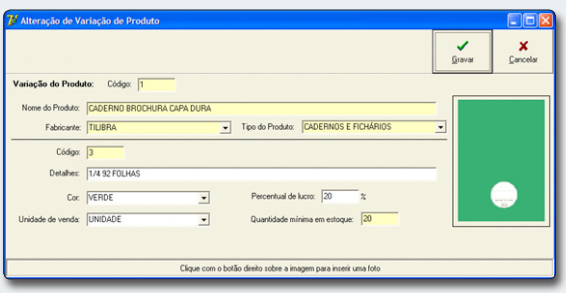


Figura 4. Protótipo (3) do UC Manter Produto – Alteração de variação de produto

Listagem 2. UC Registrar Entrada em Estoque

```
Descrição: Este caso de uso tem por objetivo registrar entrada
de produtos, disponibilizando-os para venda.
Atores: Departamento de Estoque
Pré-condição: existir cadastro prévio de fornecedores e
produtos.

Cenário principal:

1.O sistema prepara uma lista de todos os fornecedores
  cadastrados.

2.O usuário informa:

  2.1.Número da Nota Fiscal
  2.2.Data da Nota Fiscal
  2.3.Fornecedor, selecionado da lista previamente preparada

3.Para cada produto da nota fiscal:

  3.1.O usuário informa:

    3.1.1.código do produto

      3.1.1.1.O sistema pesquisa e exibe o nome do produto, os
      detalhes, a cor e o fabricante.
      3.1.1.2.O sistema exibe a quantidade atual em estoque.

    3.1.2.quantidade adquirida
    3.1.3.valor unitário da compra

  3.2.O sistema verifica o preço atual de venda do produto.

  3.3.O sistema calcula e exibe a sugestão de novo preço de
      venda, considerando o seguinte cálculo:

    3.3.1.novo preço = valor unitário da compra x percentual de
          lucro do produto

  3.4.O usuário confirma o preço vigente ou o novo preço sugerido.

  3.5.O sistema exibe todos os produtos que estão sendo
  cadastrados, contendo para cada um: código, nome do produto,
  detalhes, cor, fabricante, qtd adquirida, nova quantidade em
  estoque, valor unitário de compra, valor normal de venda vigente.

4.O sistema atualiza o cadastro de estoque.

  4.1.Em caso de alteração do preço de venda:

    4.1.1.O sistema cadastra o novo preço de venda associando
          seu início de vigência à data atual.

  4.2.Para cada produto atualizado, o sistema acrescenta na
      quantidade em estoque do produto a quantidade adquirida.

Cenários alternativos:
Nota fiscal em duplicidade
2.a. Se já houver o mesmo número de nota fiscal para o mesmo
     fornecedor, o sistema deve exibir mensagem de erro e
     retornar ao passo 2.

Pesquisa de produto
3.a. O usuário poderá localizar um produto por seu código
     ou utilizando a consulta de produtos. Include UC Consultar
     Produtos e Preços.

Exibição de preço atual de venda
3.b. Se o produto estiver em época de promoção, o sistema deve
     apresentar o preço normal fora da promoção, o preço
     promocional e seu período de vigência.

Produto inexistente
3.c. Se o código do produto não existir, o sistema deve exibir
     mensagem de erro e chamar a Pesquisa de Produto.

Produto desativado
3.d. Se o produto informado pelo usuário estiver com o status
     de desativado, o sistema deve exibir mensagem de erro e
     retornar ao passo 3.
```

Listagem 3. UC Consultar Produtos e Preços

```
Descrição: Este caso de uso tem por objetivo fornecer mecanismos
de localização de produtos, bem como conferir o preço oferecido
para um determinado produto
Atores: Departamento de Estoque, Vendedor e Gerência
Pré-condição: identificar o chamador do caso de uso

Cenário principal:

1.O usuário informa os critérios de busca:

 - código (com a opção de todas as variações do produto localizado)ou
 - descrição do produto (trecho ou integral) e/ou
 - fabricante (trecho ou integral)

2.O sistema pesquisa e exibe os produtos que satisfaçam os
  critérios informados, exibindo para cada um:

  2.1.código do produto
  2.2.nome do produto
  2.3.detalhes
  2.4.cor
  2.5.fabricante
  2.6.preço de venda
  2.7.unidade de venda
  2.8.tipo do produto
  2.9.quantidade em estoque
  2.10.foto do produto

3.Se o chamador do caso de uso assim necessitar, o usuário pode
  selecionar um produto.
  Pós-condição: retornar o produto selecionado, se for o caso.

Cenários alternativos:

Pesquisa de produto
1.a. A pesquisa de produto deve desconsiderar caixa alta e baixa,
     acentuação e realizar a localização dos trechos em qualquer
     ordem que os mesmos apareçam no cadastro.

1.b. A pesquisa da descrição deve ser feita ao mesmo tempo nos
     campos: nome do produto, detalhes e cor, retornando os
     produtos que contenham os critérios de pesquisa em qualquer um
     desses campos.
```

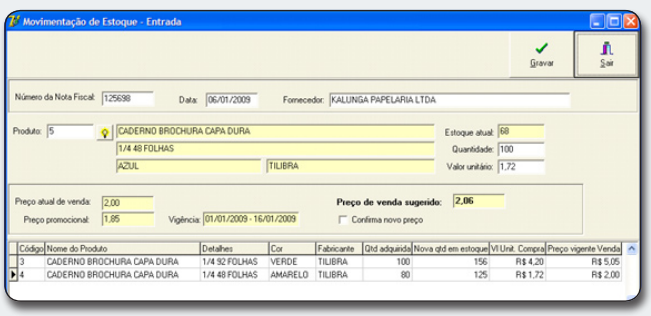


Figura 5. Protótipo do UC Registrar Entrada em Estoque

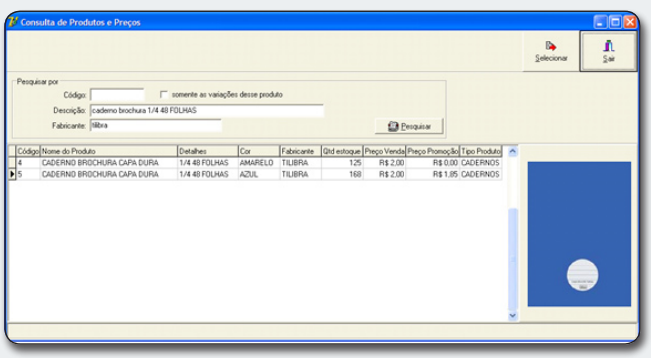


Figura 6. Protótipo do UC Consultar Produtos e Preços

A partir da lista de casos de uso, já podemos representá-la no Diagrama de Casos de Uso, conforme é demonstrado na **Figura 1.**

Escrevendo os cenários dos casos de uso

Diagrama feito, é hora de trabalhar. Até então o analista tem apenas um nome de caso de uso. Isso é ótimo para estabelecer o escopo de seu trabalho, mas não diz praticamente nada sobre o que o sistema deve fazer. O máximo de informação que se tem é "quem" executará cada tarefa.

Assim, para cada caso de uso, o analista irá escrever seus cenários. Mas baseado em que? Baseado nos requisitos levantados. Não cabe ao analista, e muito menos ao programador que receber essa documentação, inventar nenhum requisito, nenhuma regra. Ora, sendo assim, é preciso que tudo o que seja relevante esteja dito nos cenários de caso de uso. Então, significa que os requisitos também precisam estar detalhados. Se não estiverem, é hora de voltar um passo e ter mais uma reunião com o cliente.

Na **Listagem 1**, apresentamos a descrição do caso de uso *Manter produto*. Para que o analista possa chegar à escrita desse caso de uso, ele deve buscar nos requisitos tudo o que seja inerente ao cadastramento ou manutenção de um produto dentro do sistema, incluindo suas regras. A partir do caso de uso escrito, o analista desenha um protótipo para o mesmo, que ajudará não só o analista a validar o que foi escrito, como o cliente para que entenda melhor a solução apresentada. As **Figuras 2, 3 e 4** apresentam os protótipos para o caso de uso *Manter Produto*. Repare que um caso de uso não precisa estar relacionado a somente uma tela, da mesma forma que uma tela pode conter mais de um caso de uso associado a ela. Um protótipo pode ser desenhado em qualquer ferramenta, até mesmo desenhado à mão (o que, profissionalmente, não é muito recomendável).

Repare que os *itens 1 e 2* do cenário principal são refletidos no protótipo (1) – **Figura 2**. Nessa tela (ou rascunho de tela) é apresentada a consulta de produtos descrita no *item 1*, exatamente com os campos citados no caso de uso. Esse sincronismo é importante. Imagine o quanto confuso poderia ficar um programador que lesse no caso de uso a exigência de exibir nome, detalhes e fabricante no resultado da pesquisa, e ao se deparar com o protótipo, encontrasse também o campo de cor. Lembre-se que isso é um documento e precisa ser validado com o cliente. Assim, se ele aprovar esses campos, então o produto final tem que refletir essa aprovação.

Acima da grid de produtos aparece uma área para os critérios de pesquisa, de acordo com o que é descrito no *item 1.1*. As regras de pesquisa (explicitadas no cenário alternativo), nesse caso, não são visíveis no protótipo, mas são sugeridas, pois repare que o exemplo determina "caderno brochura 1/4", e o resultado localiza os produtos com a palavra "caderno brochura" no *nome do produto* e "1/4" em *detalhes*. O *item 2* do cenário principal informa que o sistema habilita algumas opções ao usuário. Essa visualização é vista na área de cima da tela do protótipo (1).

Já o protótipo (**Figura 3**) apresenta a tela que será utilizada para as opções de "Inclusão" e "Alteração". Repare que o próprio caso de uso, no *item 3*, unifica esse cadastro, simplificando a edição. Contudo, tudo o que é particular a cada opção é citado explicitamente, como o caso de exibição do código do produto que só acontece quando da opção de "Alteração". Os *itens 3.1 a 3.5* determinam a ação do sistema antes que a rotina comece a ser operada pelo usuário. Quando o usuário acessa uma tela dessas de cadastro, as combos de cores, fabricantes, unidades de venda e tipos de produto já devem vir preenchidas com o que já existe cadastrado no sistema.

O protótipo da **Figura 4** apresenta a tela que será utilizada para a opção de "Inclusão por variação de produto" e "Alteração", se essa pertencer a um produto que seja variação.

Considerando o mesmo princípio que o *Manter Produto*, só que de uma forma mais simples, são escritos os casos de uso *Manter Fabricante*, *Manter Tipo de Produto*, *Manter Fornecedor*, *Manter Unidade de Venda*, *Manter Cor* e *Manter Vendedor*, que têm por objetivo apenas o cadastramento do nome, sendo o código apenas uma chave interna que não interessará ao usuário final, que irá trabalhar apenas com o nome.

Assim, veremos na **Listagem 2** o caso de uso *Registrar Entrada em Estoque*. O objetivo desse caso de uso é dar entrada nas compras efetuadas pela Papelaria. A partir de cada compra inserida, o estoque é atualizado e os preços de venda são disponibilizados. Esse caso de uso faz uma chamada a outro caso de uso, responsável apenas pela consulta de produtos e preços. Esse caso de uso é usado em outros lugares, e por este motivo para que haja reutilização de suas funcionalidades, o relacionamento com ele é feito pelo estereótipo de inclusão. Os cenários do caso de uso *Consultar Produtos e Preços* podem ser vistos na **Listagem 3**. Repare que esse caso de uso é acessível por vários atores, e todos eles são relacionados na seção correspondente. Para que não haja sobrecarga de nomes durante a escrita do caso de uso, é sempre uma boa prática substituir o nome do ator (durante a descrição do cenário) pelo termo genérico usuário.

A **Figura 5** apresentará o protótipo do caso de uso *Registrar Entrada em Estoque*, enquanto que a **Figura 6** apresentará o protótipo do caso de uso *Consultar Produtos e Preços*.

Conclusão

No próximo artigo, veremos em detalhes casos de uso de outros formatos, como o de relatório, registro de pagamento (com as diversas formas de pagamento), consulta em tela e o de processamento e controle (como abertura e fechamento de caixa). ●

Documento de Requisitos

Essencial ao Desenvolvimento de Software

De que se trata o artigo?
Apresenta o documento de requisitos de software, destacando-o como um dos principais documentos pertinentes ao processo de desenvolvimento de software e ilustrando como ele deve ser elaborado.

Para que serve?
Informar o leitor sobre quais elementos considerar quando da elaboração de um documento de requisitos, levando em consideração o público alvo do documento que engloba cliente, desenvolvedores e gerentes, dentre outros.

Em que situação o tema é útil?
Durante o desenvolvimento de um sistema de software, no qual há a necessidade de elaborar o documento descrevendo o conjunto de requisitos do sistema de modo a informar tanto equipe de projeto quanto cliente, o que será implementado.

Antonio Mendes da Silva Filho
antoniom.silvafilho@gmail.com
Professor e consultor em área de tecnologia da informação e comunicação com mais de 20 anos de experiência profissional, é autor do livros Arquitetura de Software e Programando com XML, ambos pela Editora Campus/Elsevier, tem mais de 30 artigos publicados em eventos nacionais e internacionais, colunista para Ciência e Tecnologia pela Revista Espaço Acadêmico com mais de 80 artigos publicados, tendo feitos palestras em eventos nacionais e exterior. Foi Professor Visitante da University of Texas at Dallas e da University of Ottawa. Formado em Engenharia Elétrica pela Universidade de Pernambuco, com Mestrado em Engenharia Elétrica pela Universidade Federal da Paraíba (Campina Grande), Mestrado em Engenharia da Computação pela University of Waterloo e Doutor em Ciência da Computação pela Univesidade Federal de Pernambuco.

Um engenheiro de software é um profissional que deve ter a habilidade de antecipar e gerenciar mudanças de requisitos de um produto de software. Além disso, ele precisa saber se expressar e comunicar-se bem a fim de capturar e registrar adequadamente o documento de requisitos. Os principais problemas no desenvolvimento de um sistema de software decorrem do entendimento errado entre engenheiro de software (produtor), responsável em apresentar o documento de requisitos, e usuário (consumidor). Um documento de requisitos de software precisa ser claro, consistente e completo, porque esse documento servirá de referência aos desenvolvedores, gerente de projeto, engenheiros de software (responsáveis pelos testes e manutenção do sistema), além de servir de base para definir o escopo das funcionalidades a serem registradas num contrato. Perceba que os requisitos compreendem o cerne de qualquer produto e mudanças sobre eles

podem ocorrer ao longo do ciclo de vida de um software. Este artigo trata da importância do documento de requisitos de software e exemplifica como ele pode ser elaborado.

Requisitos de Software

Desenvolver um sistema de software requer um processo, o qual informa um conjunto de atividades a serem realizadas, quem as executam, quais artefatos de entrada são necessários e quais artefatos de saída são produzidos. Nesse sentido, detectar erros ou quaisquer outros problemas como, por exemplo, inconsistência e falta de clareza é de suma importância de modo a tornar o processo mais efetivo sob o ponto de vista de custo. Adwicionalmente, envolver o usuário no processo é determinante para o sucesso do produto e do processo. Dentro deste contexto, entender adequadamente o requisito é essencial e essa é tarefa do engenheiro de software. Um **requisito** compreende uma característica ou funcionalidade que o sistema deve possuir ou uma restrição que deve satisfazer para atender uma necessidade do usuário. Dessa forma, o engenheiro de software, desempenhando o papel de engenheiro de requisitos, deve executar duas atividades essenciais para a elaboração do documento de requisitos:

Elicitação de requisitos – atividade na qual os requisitos do sistema a ser desenvolvido são levantados;

Análise de requisitos – atividade na qual os requisitos são analisados e confirmados pelos principais interessados do projeto (isto é, os *stakeholders*) que incluem cliente, usuário final e gerente de projetos, dentre outros.

Considera-se ainda que a elicitação de requisitos objetiva definir características do sistema sob a perspectiva do cliente, enquanto que a análise de requisitos visa obter a especificação de requisitos, do ponto de vista técnico, conforme entendimento dos desenvolvedores.

Durante a realização destas atividades, o engenheiro de software está preocupado em levantar, entender, analisar e, por fim, documentar os requisitos. Para tanto, ele deve concentrar-se nas características do sistema e atributos de qualidade, e não em como obtê-los. Aqui, é preciso identificar quais requisitos fazem parte ou não do escopo do sistema a ser desenvolvido, ou em outras palavras, entender a interface do sistema considerado e o ambiente externo.

É importante ressaltar a necessidade de definir o 'limite', ou também denominado escopo do sistema, a fim de tratar os requisitos funcionais e não funcionais do sistema. Além disso, quando da elaboração do documento de requisitos, o engenheiro de software deve levar em consideração os diferentes pontos de vistas dos stakeholders de modo que o documento resultante possa comunicar adequadamente o conjunto de requisitos do sistema a ser construído.

Documento de Requisitos

O documento de requisitos delimita o escopo do conjunto de funcionalidades que um sistema deve prover, bem como descreve os atributos de qualidade que devem ser suportados. Este documento deve ser elaborado de maneira precisa, completa, consistente e, principalmente, compreensível aos stakeholders (isto é, os principais interessados no sistema). Note que o documento de requisitos será lido por várias pessoas interessadas no projeto como, por exemplo, cliente, gerente de projeto, engenheiro de testes e programadores, e, portanto, precisa comunicar com clareza os requisitos do sistema. Dessa forma, tem-se que um documento de requisitos:

- É elaborado pelo engenheiro de software e compreende o conjunto de requisitos do sistema a ser desenvolvido;
- Deve ser analisado e confirmado pelos *stakeholders*;
- Integra e relaciona um conjunto de perspectivas dos interessados do projeto;

Itens de um Documento de Requisitos	Conteúdo
1. Introdução	Contém uma descrição dos objetivos do documento, o público ao qual ele se destina e, em linhas gerais, o propósito e escopo do projeto a ser desenvolvido. Pode adicionalmente conter termos e abreviações usadas, tipos de prioridades atribuídas aos requisitos, além de informar como o documento deve evoluir.
2. Requisitos Funcionais	Esta seção descreve, de maneira sumarizada, as principais funcionalidades que o sistema de software irá realizar. Por exemplo, num sistema de biblioteca, esta seção deveria conter uma descrição das funcionalidades de autenticação de usuário e controle de acesso. Observe que o sumário das funcionalidades de um sistema se faz necessário para permitir o entendimento das funcionalidades do sistema pelos diversos stakeholders. O engenheiro de software deve organizar o conjunto de funcionalidades do sistema de modo a torná-las mais compreensíveis aos clientes e demais stakeholders. Vale ainda ressaltar que o documento de requisitos pode ser complementado por outro documento como, por exemplo, especificações de casos de uso.
3. Requisitos Não-Funcionais	Apresenta-se uma descrição geral de outros requisitos do produto que limitam opções de desenvolvimento do sistema. Isto inclui a descrição de requisitos de segurança, confiabilidade, timeout de sessão de usuário, usabilidade, dentre outros. Esta seção considera os requisitos do produto, do processo, da interface gráfica e da plataforma tecnológica empregada.
4. Escopo Não-Contemplado	Contém descrição das funcionalidades não contempladas no escopo do sistema a ser desenvolvido. Outra denominação dada a esta seção é escopo negativo. Isto visa garantir às partes interessadas no sistema (isto é, cliente e equipe de desenvolvimento) quais funcionalidades fazem parte ou não do conjunto a ser implementado.
5. Documentação Complementar	Exemplos desses documentos compreendem atas de reuniões nas quais ocorrerão levantamento e validação de requisitos, bem como o plano de projeto.
6. Apêndice	Trata-se de uma seção que pode conter, por exemplo, levantamento de perfil de usuários do sistema a ser desenvolvido e descrição do problema a ser automatizado pelo sistema de software. É importante observar que o apêndice não é parte do documento de requisitos e serve apenas como informação de apoio para os leitores do documento.

Tabela 1 – Relação de itens de um documento de requisitos.

1. Introdução

Este documento descreve um sistema que prover notícias e conteúdo online, denominado de Sistema Exemplo, a ser desenvolvido para a Empresa XYZ. Seu propósito é prover notícias sobre os mais variados conteúdos, permitindo acesso integral apenas aos usuários leitores cadastrados no sistema.

Visão geral do documento

Esta introdução fornece as informações necessárias para utilizar este documento, explicitando seus objetivos e as convenções que foram adotadas no texto. As demais seções apresentam a especificação do sistema Exemplo e estão organizadas como descrito abaixo.

Requisitos funcionais: compreende o conjunto de requisitos funcionais do sistema a ser desenvolvido, descrevendo suas prioridades.

Requisitos não funcionais: contém os requisitos não funcionais do sistema a ser desenvolvido, divididos em requisitos de produto, processo e plataforma tecnológica.

Escopo não contemplado: descreve as funcionalidades que são relacionadas com o sistema, mas que não fazem parte do escopo do mesmo e, portanto, não serão implementadas.

Documentação complementar: compreende um conjunto de documentos complementares que contêm informações relacionadas ao projeto.

Termos e convenções

Esta parte do documento explica os conceitos de termos importantes que serão citados no decorrer deste documento, conforme descrito no quadro abaixo.

Termo	Descrição
Requisitos funcionais	Requisitos de software que compõe o sistema, descrevendo ações que o sistema deverá executar quando solicitado.
Requisitos não funcionais	Requisitos de software que compõem o sistema, descrevendo atributos de qualidade que o sistema deve possuir, ou restrições que ele deve satisfazer.
Requisitos não técnicos	Requisitos não relacionados ao software como, por exemplo, material de divulgação do projeto (eventos, relatórios técnicos e outras publicações). Esses requisitos estão fora do escopo deste documento, podendo ser incluídos no Plano do Projeto.

Quadro 1. Exemplo da Seção 1 do Documento de Requisitos.

Prioridades dos requisitos

A atribuição de prioridade dos requisitos pode ser de três tipos: essencial, importante e desejável. A prioridade dos requisitos pode ser usada no gerenciamento do escopo do projeto e na definição das prioridades para o desenvolvimento do sistema.

Essencial: requisito sem o qual o sistema não entra em funcionamento. Requisitos essenciais são requisitos imprescindíveis, devendo ser disponibilizados na implantação do sistema.

Importante: requisito sem o qual o sistema entra em funcionamento, mas de forma não satisfatória. Requisitos importantes não impedem a implantação do sistema, mas devem ser implementados o mais breve possível.

Desejável: requisito que, embora não implementado, ainda permite o sistema funcionar de modo satisfatório sem comprometer as funcionalidades básicas do sistema. Requisito desejável é um requisito que pode ser entregue em qualquer momento sem prejuízo para os serviços oferecidos pelo sistema.

Quadro 1. Exemplo da Seção 1 do Documento de Requisitos.

• Serve como mecanismo de comunicação para os *stakeholders* (i.e. as partes interessadas do projeto);
• Captura e documenta os requisitos do projeto e serve de referência para testes, manutenção e evolução do sistema.

O documento de requisitos de um projeto tem o objetivo de documentar o escopo do sistema a ser desenvolvido. Nesse sentido, o documento de requisitos deve conter:
• Introdução e visão geral do documento
• Descrição de requisitos funcionais
• Descrição de requisitos não-funcionais
• Escopo não contemplado (de funcionalidades)
• Documentação de apoio

É importante perceber a importância do documento de requisitos como determinante para o sucesso de um projeto. Ele identifica quais funcionalidades fazem parte ou não do escopo do sistema. A seção seguinte apresenta um exemplo de um documento de projeto ilustrando e complementando os pontos destacados acima.

Exemplificando o Documento de Requisitos

O engenheiro de software, ao elaborar o documento de requisitos, deve buscar um compromisso de comunicar bem as funcionalidades do sistema a ser desenvolvido e da definição em detalhes com clareza e consistência para os programadores e engenheiros de testes (responsáveis pela implementação do sistema e elaboração e execução de plano de testes, respectivamente).

A Tabela 1 apresenta uma relação dos itens consideradas imprescindíveis em um documento de requisitos. A relação de itens destacados na Tabela 1 não pressupõe a intenção de ser completo, mas de apontar os itens considerados como obrigatórios num documento de requisitos. Cabe destacar que os itens sugeridos para compor um documento de requisitos, conforme apresentado na Tabela 1, leva em consideração as recomendações de

2. Requisitos Funcionais

2.1 Controle de Acesso

Esta seção apresenta a descrição das funcionalidades de controle de acesso de usuários além das funcionalidades para supervisão dos acessos ocorridos.

RF01 - Solicitar cadastro no serviço de recomendação
Este requisito permite aos usuários solicitar à Empresa XYZ o seu cadastramento no Sistema Exemplo. Essas solicitações ficam pendentes de aceitação até que sejam validadas e aprovadas por parte de um funcionário da Empresa XYZ.

Prioridade:	[X]	Essencial	[ ]	Importante	[ ]	Desejável

RF02 – Registrar cadastro de usuário no serviço de recomendação
Este requisito permite que um funcionário da Empresa XYZ valide o cadastro e libere o acesso de um usuário ao Sistema Exemplo efetuando o seu registro após o usuário confirmar leitura e aceitação (via Internet) do Termo de Responsabilidade de Acesso e Uso do Sistema Exemplo.

Prioridade:	[X]	Essencial	[ ]	Importante	[ ]	Desejável

RF03 - Alterar senha de acesso
Este requisito permite ao usuário trocar sua senha de acesso ao sistema. Para efetivar a troca de senha, os seguintes critérios de segurança serão verificados: tamanho mínimo e máximo da senha, definição de período de validade da senha e reuso de senhas anteriores. Estes critérios deverão ser definidos no banco de dados. A gerência dos critérios de segurança da senha deverá ser controlada por um sistema gerenciador de banco de dados (SGBD).

Prioridade:	[X]	Essencial	[ ]	Importante	[ ]	Desejável

RF04 – Autenticar usuário
Este requisito faz a autenticação do usuário através de seu login e senha e, em seguida, exibe um menu de opções de acordo com as funcionalidades permitidas ao usuário em conformidade com seu perfil de acesso. Toda vez que o usuário efetuar um login no sistema, deverá ser registrada a abertura de um log de acesso do usuário.

Prioridade:	[X]	Essencial	[ ]	Importante	[ ]	Desejável

RF05 – Consultar permissões de acesso
Este requisito permite que um funcionário da Empresa XYZ consulte as permissões de acesso de usuários ao sistema, obtendo informação sobre o tipo de acesso e expiração da permissão de acesso ao sistema.

Prioridade:	[ ]	Essencial	[X]	Importante	[ ]	Desejável

RF06 – Cancelar acesso de usuário
Este requisito permite que qualquer usuário autorizado (cadastrado) cancele o acesso aos conteúdos do sistema. Os dados dos usuários para cancelar o acesso ao sistema devem estar definidos no banco de dados.

Prioridade:	[ ]	Essencial	[X]	Importante	[ ]	Desejável

2.2 Outros Serviços
Esta seção descreve as funcionalidades para efetuar alterações nos cadastros de usuários, bem como alteração cadastral de dependentes.

RF06 – Alterar cadastro de usuário
Este requisito permite que um usuário possa alterar diretamente seus dados cadastrais no sistema Exemplo, bem como fazer a inclusão ou exclusão de usuários dependentes para acesso ao conteúdo do sistema.

Prioridade:	[ ]	Essencial	[ ]	Importante	[X]	Desejável

Quadro 2. Exemplo da Seção 2 do Documento de Requisitos.

documento padrão IEEE-Std 830-1998 recomendado pelo IEEE e referenciado no quadro de links deste artigo.

O conteúdo exato das seções que compõem um documento de requisitos, de um modo geral, varia de empresa para empresa. As subseções, destacadas nos **Quadros 1** a **5**, ilustram o conteúdo que compõe um documento de requisitos. O propósito do sistema Exemplo, usado aqui apenas com fins ilustrativos, é similar ao de um sistema como o de portais de jornais e revistas e outros provedores de conteúdo que permitem o acesso a conteúdo, como o portal UOL, apenas a clientes devidamente cadastrados no sistema.

Note que o **Quadro 1** apresenta uma visão geral do documento e identifica um subconjunto de termos e convenções que pode caracterizar o projeto. Todo e qualquer termo, convenção adotada ou abreviações deveriam ser apresentadas nesta seção a fim de comunicar às partes envolvidas e interessadas (i.e. os *stakeholders*) o seu significado. Isto visa prover os *stakeholders* com as denominações corretas empregadas no projeto.

A seção seguintwe apresenta a segunda parte do documento de requisitos que contém descrição dos Requisitos Funcionais (RFs), conforme ilustrado no **Quadro 2**.

O **Quadro 2** apresenta a descrição de um conjunto de

3. Requisitos Não Funcionais

3.1 Requisitos do Produto

Esta seção apresenta a descrição do conjunto de requisitos do Sistema Exemplo para prover conteúdo para usuários cadastrados.

RNF01 – Segurança

O Sistema Exemplo da empresa XYZ deve dispor de mecanismos de segurança para a autenticação de usuários e controle de acesso a conteúdos e funcionalidades do sistema, garantindo o acesso apenas para usuários cadastrados. O site deverá utilizar protocolo HTTPS, com uso de certificado digital, garantindo a autenticação do servidor, bem como proteção e confidencialidade das informações em trânsito.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF02 – Senha criptografada

O sistema Exemplo deverá prover o usuário com senha criptografada. O sistema Exemplo deve fazer uso de um algoritmo que não permita obter a senha criptografada. Este mecanismo de criptografia deverá ser implementado pelo sistema gerenciador de banco de dados (SGBD).

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF03 – Usabilidade

O sistema Exemplo deve prover o usuário com interface simples e intuitiva, de fácil navegação para facilitar o uso do mesmo por parte dos usuários.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF04 – Apresentação da interface gráfica

O sistema Exemplo deve fazer uso, exclusivamente, da língua Portuguesa para todo e qualquer texto apresentado no portal de conteúdos e adicionalmente deve ser executado no browser Internet Explorer, versão 6.0 ou superior, com resolução 800 x 600.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF05 – Ajuda online

O sistema Exemplo deve prover os usuários com Ajuda online para orientá-lo quanto ao acesso e uso das funcionalidades do sistema Exemplo.

Prioridade:	[ ] Essencial	[X] Importante	[ ] Desejável

3.2 Requisitos do Processo

Esta seção apresenta a descrição dos requisitos relativos ao processo utilizado no desenvolvimento do sistema Exemplo.

RNF06 – Arquitetura de software

A implementação do sistema Exemplo deve empregar uma arquitetura de 3 (três) camadas: apresentação, negócio e dados.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF07 – Linguagem de programação adotada

A implementação do sistema Exemplo deve utilizar a linguagem Java, adotando padrão J2EE.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

3.2 Requisitos de Tecnologia Adotada

Esta seção apresenta a descrição dos requisitos relativos às tecnologias adotadas no desenvolvimento do sistema Exemplo.

RNF08 – Disponibilidade

O portal de conteúdos do sistema Exemplo deverá estar disponível aos usuários 24 horas por dia e 7 dias por semana.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF09 – Banco de dados

A implementação do sistema Exemplo deve empregar o SQL Server 2005 Enterprise Edition como servidor de banco de dados.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

RNF10 – Componentização

Cada componente do sistema Exemplo deve ser um JAR, os quais deverão ser incluídos em um arquivo WAR (centralizador). Os arquivos JAR ficarão em projetos distintos.

Prioridade:	[X] Essencial	[ ] Importante	[ ] Desejável

Quadro 3. Exemplo da Seção 3 do Documento de Requisitos.

4. Escopo Não Contemplado

Esta seção apresenta um conjunto de funcionalidades e requisitos que não estão contemplados no escopo do Sistema Exemplo.

4.1 Controle de acesso:

• Realização de bloqueio de acesso aos usuári os que possuam mensalidades em atraso junto à empresa XYZ.
• Cadastro (inclusão, alteração, exclusão) de tipos de usuários.

4.2 Outros serviços

• Atualização de informações do portal de conteúdos.
• Mecanismo de FAQ e de busca aos conteúdos do portal.
• Atendimento a consultas através de e-mail.

4.3 Segurança

• Definição das políticas de segurança necessárias à administração do Sistema Exemplo.

Quadro 4. Exemplo da Seção 4 do Documento de Requisitos.

5. Documentação Complementar

Esta seção apresenta documentação de apoio, referenciando um conjunto de outros documentos que complementam e suportam as informações contidas no documento de requisitos.

Ata de Reunião – Levantamento de Requisitos do Módulo A do Sistema Exemplo, 12/01/2009.

Ata de Reunião – Levantamento de Requisitos do Módulo B do Sistema Exemplo, 13/01/2009.

Ata de Reunião – Levantamento de Requisitos do Módulo C do Sistema Exemplo, 14/01/2009.

Ata de Reunião – Validação de Requisitos do Sistema Exemplo, 15/01/2009.

Plano de Projeto do Sistema Exemplo.

Quadro 5. Exemplo da Seção 5 do Documento de Requisitos.

requisitos funcionais de um sistema Exemplo. Perceba que o objetivo não foi de ser completo, mas o de ilustrar como a seção que descreve os requisitos funcionais de um documento de requisitos poderia ser elaborada. Note também que as informações apresentadas neste documento têm a intenção de comunicar às partes envolvidas e interessadas o conjunto de requisitos funcionais do sistema a ser desenvolvido.

A seção seguinte apresenta a terceira parte do documento de requisitos que compreende a descrição dos Requisitos Não Funcionais (RNFs), conforme ilustrado no **Quadro 3**.

O **Quadro 3** apresenta a descrição de um conjunto de requisitos não funcionais do sistema Exemplo. Vale ressaltar que apenas alguns requisitos não funcionais são apresentados para ilustrar como essa seção do documento de requisitos poderia ser elaborada.

A seção seguinte apresenta a quarta parte do documento de requisitos que descreve o escopo não contemplado, conforme ilustrado no **Quadro 4**.

O **Quadro 4** apresenta a descrição de um conjunto de funcionalidades e requisitos não contemplados na implementação do Sistema Exemplo. Finalmente, o **Quadro 5** apresenta a quinta parte do documento de requisitos que lista um conjunto de documentos complementares.

Comentários Finais

Requisitos de software compreendem a essência de um produto, os quais definem as funcionalidades que o sistema deve prover e restrições que ele deve satisfazer. Documentar bem os requisitos de software é atividade de suma importância para um engenheiro de software que deve levar em consideração o público diverso que fará uso desse documento.

Portanto, o engenheiro de software deve ter em mente que tanto cliente quanto gerente de negócios, gerente de projeto, desenvolvedores e engenheiros de testes irão consultar as informações contidas nesse documento. Este artigo destacou a relevância desse documento de projeto e apresentou o conjunto de seções que compõem o documento de requisitos ilustrando-o para um sistema exemplo. ●

Links

Writing a Software Requirements Document
http://www2.sims.berkeley.edu/courses/is208/s02/ReqsDoc.pdf

IEEE Standard 830-1998
http://standards.ieee.org/reading/ieee/std_public/description/se/830-1998_desc.html

Military Standard – Defense System Software Development – DOD-STD-2167
www.everyspec.com/DoD/DoD-STD/download.php?spec=DOD-STD-2167.000278.pdf

Software Documentation
http://en.wikipedia.org/wiki/Software_documentation

Requirements Engineering – A Roadmap
http://www.cs.toronto.edu/~sme/papers/2000/ICSE2000.pdf

Requirements Engineering – A Good Practice Guide
http://www.comp.lancs.ac.uk/computing/resources/re-gpg/

Gerência de Reutilização de Software

Josiane Brietzke Porto

josiane_brietzke@hotmail.com

Pós-graduada em Melhoria de Processos de Software pela UFLA em 2008. Bacharel em Ciência da Computação pelo Unilasalle em 2005. Autora de artigos na área de qualidade de software (ASSE 2005, WIS 2005, CLEI Eletronic Journal, W2–MPSBR, SBQS 2007, W6-MPS.BR e Engenharia de Software Magazine). Experiência de mais de 5 anos na área de TI atuando em desenvolvimento de software e de mais de 4 anos na área de qualidade e melhoria de processos. Implementadora MR-MPS desde 2004, Certified Quality Improvement Associate (CQIA) desde 2006 e integrante desde 2008 do Comitê Setorial de Informática - Programa Qualidade RS. Atua desde 2005 na Qualità Informática em projetos de melhoria de processos baseados em MPS.BR e PGQP.

Isabel Albertuni

ialbertuni@gmail.com

Graduada em Análise de Sistemas pela FARGS em 2008. Autora de artigos na área de qualidade de software (SBQS 2007, W6-MPS.BR e Engenharia de Software Magazine). Experiência de mais de 6 anos na área de TI atuando em desenvolvimento de software e de 3 anos na área de qualidade e melhoria de processos. Integrante desde 2008 do Comitê Setorial de Informática - Programa Qualidade RS. Atua desde 2006 na Qualità Informática em projetos de melhoria de processos baseados em MPS.BR e PGQP.

De que se trata o artigo?

Neste artigo apresenta-se o processo de Gerência de Reutilização de Software, sua contribuição para as organizações intensivas em software, além de benefícios e dificuldades na implementação de suas práticas.

Para que serve?

Serve para esclarecimento e melhor entendimento do tema por profissionais da área de qualidade e pelas organizações que pretendem implementar Gerência de Reutilização de Software.

Em que situação o tema é útil?

A adoção de práticas de reutilização de software surge como uma maneira da organização melhorar a produtividade das equipes, reduzir custos e melhorar a qualidade de seus produtos.

Mais recentemente temos um cenário de demanda por uma melhor qualidade de software. Demanda porque ter uma qualidade adequada não é mais um diferencial de negócio, mas sim uma condição básica para negócios. Qualidade de software em um sentido mais amplo, incluindo uma composição de fatores como os prazos menores, custos menores, menor quantidade de defeitos, menos insatisfações de uma maneira geral, maior qualidade dos produtos desenvolvidos, maior previsibilidade, maior produtividade, maior competitividade e melhores resultados de negócio [Salviano 2006a].

Segundo Salviano (2006a), a melhoria de processo de software tem mostrado na prática ser uma abordagem viável, eficaz e eficiente para a necessária melhoria das organizações intensivas em software. A comunidade tem relatado vários casos de sucesso como, por exemplo, Herbsleb et al. [1994], DACS [1999] e Card [2002].

A melhoria de processo de software baseada em modelos (em inglês, *model based software process improvement*) pode ser definida como: uma abordagem para melhoria das organizações intensivas em software, baseada em modelos de capacidade de processo, que orienta ações para alteração dos processos utilizados para aquisição, fornecimento, desenvolvimento, manutenção e/ou suporte de sistemas de software, com o objetivo de estabelecer processos que satisfaçam de forma mais eficiente e eficaz os objetivos e necessidades de negócio da organização [Salviano 2006a].

Diante deste cenário e utilizando a abordagem de melhoria de processo de software, a adoção de práticas de reutilização de software surge como uma maneira da organização melhorar a produtividade das equipes, reduzir custos e melhorar a qualidade de seus produtos.

Neste sentido, o processo Gerência de Reutilização de Software tem como objetivo definir procedimentos tanto administrativos quanto técnicos para utilização de ativos reutilizáveis em uma organização, estabelecendo e controlando uma biblioteca para o armazenamento e recuperação destes ativos [IEEE, 2004].

Neste artigo apresenta-se o processo de Gerência de Reutilização de Software, sua contribuição para as organizações intensivas em software, além de benefícios e dificuldades na implementação de suas práticas. Para isso, o artigo está organizado da seguinte forma: a seção 2 apresenta uma visão geral sobre a Gerência de Reutilização de Software; a seção 3 descreve este processo no MPS.BR, nível de maturidade Parcialmente Definido (E); a seção 4 trata da implementação das práticas deste processo numa organização; e por fim, a seção 5 trata das considerações finais.

Gerência de Reutilização de Software

Conforme Werner (2007), a reutilização é inerente ao processo de solução de problemas utilizado pelos seres humanos. Na medida em que soluções são encontradas, estas são utilizadas em problemas similares e nossa capacidade de abstração garante a adaptação necessária ao novo contexto. O problema, portanto, não é a falta de reutilização na Engenharia de Software, mas a falta de uma sistemática ampla e formal para realizá-la.

Segundo Krueger (1992), reutilização de software é a disciplina responsável pela criação de sistemas de software a partir de software preexistente. Reutilização de software consiste no processo de reaproveitar artefatos e conhecimentos de softwares já existentes na organização para construir novos, reduzindo tempo e custo e melhorando a qualidade destes produtos [Porto, 2008]. Por isso, mecanismos para a manutenção e recuperação destes ativos devem ser implementados, assim como uma biblioteca de ativos de processo, no qual a recuperação de ativos utilizados em outros projetos possam ser adaptados (quando necessário) e reutilizados em novos projetos [SEI, 2006].

Através de reutilização de software, alguns benefícios podem ser obtidos como [Werner 2007] [CRUISE 2007]:

- melhores índices de produtividade;
- produtos de melhor qualidade, mais confiáveis, consistentes e padronizados;
- redução dos custos e esforço envolvidos no desenvolvimento e manutenção de software;
- maior flexibilidade na estrutura do software produzido, facilitando sua manutenção e evolução;
- tamanho de equipe menor, levando a uma melhor comunicação e maior produtividade, entre outros.

Por outro lado, existem algumas dificuldades relacionadas à [Werner 2007] [CRUISE 2007]:

- identificação, recuperação e modificação de artefatos reutilizáveis;
- compreensão dos artefatos recuperados;
- qualidade de artefatos reutilizáveis;
- composição de aplicações a partir de componentes;
- barreiras psicológicas, legais e econômicas;
- necessidade da criação de incentivos à reutilização;
- falta de apoio da alta direção;
- gerenciamento de projetos com reutilização de software;
- estrutura organizacional inadequada, entre outros.

Conforme relatam Werner (2007) e CRUISE (2007), atualmente existem várias iniciativas nacionais e internacionais na área de reutilização de software a fim de investigar, desenvolver e colocar em prática esta técnica.

Segundo Porto (2008), na indústria de software gaúcha e mais especificamente, no contexto do projeto Cooperativa MPS.BR - SOFTSUL, os resultados de uma pesquisa conduzida com empresas cooperadas demonstram que as práticas de reutilização de software adotadas ainda são bem iniciais e desassociadas de um programa de reutilização, políticas de incentivo ou estratégias da organização.

Conforme Becker (2008), as práticas de reuso adotadas com maior freqüência nos projetos das empresas desta pesquisa são:

- frameworks de mercado e desenvolvidos para uso próprio;
- padrões de projeto;
- adaptação de artefatos a partir de *templates* ou artefatos criados em projetos semelhantes;
- políticas de incentivos não são adotadas, porém existem iniciativas e práticas isoladas de integrantes das equipes;
- componentes prontos e outros desenvolvidos para reuso, sem adotar um repositório para compartilhamento;
- adoção de *knowledge base* com componentes reusáveis, porém pouco disseminada;
- técnicas e experiências de reuso são divulgadas em ferramenta *wiki*;
- lições aprendidas em projetos de domínios semelhantes.

Conforme Porto (2008), este cenário ainda apresenta organizações que não estão preparadas para adotar a reutilização de software de uma maneira sistemática, necessitando de disseminação de conhecimento e de pesquisas como, por exemplo, das instituições nacionais COPPE/UFRJ (2008), RISE (2008) e COMPOSE (2008).

Segundo CRUISE (2007), processos de software se referem a todas as atividades necessárias para produzir e gerenciar software, enquanto isto, processos de reutilização de software são um subconjunto de atividades necessárias para desenvolver e reutilizar ativos.

Conforme Werner (2007), o processo de Gerência de Reutilização está dividido nos seguintes sub-processos:

• Planejamento de Reutilização: propósito de definir uma estratégia de reutilização e um plano para implementação dentro da empresa;

• Criação de Artefatos: propósito de produzir software e produtos associados para a reutilização (desenvolvimento para reutilização);

• Gerência de Artefatos: propósito de coletar, avaliar, descrever e organizar artefatos reutilizáveis para garantir sua disponibilização aos processos de criação e utilização;

• Utilização de Artefatos: propósito de compor sistemas a partir de artefatos reutilizáveis (desenvolvimento com reutilização).

A seguir, na **Tabela 1**, um resumo é apresentado a fim de demonstrar o tratamento adotado pelas normas e modelos de qualidade para os processos de reutilização de software. A partir deste resumo, se observa certa similaridade e conservação no conteúdo dos objetivos principais destes processos [Porto, 2008].

ISO/IEC 15504-5: 2006 [Salviano 2006a]	**REU.1 Gerência de Ativos Reusáveis:** propósito de gerenciar a vida de ativos reusáveis da concepção até a aposentadoria;
	REU.2 Gerência de Programa de Reuso: propósito de planejar, estabelecer, gerenciar, controlar e monitorar um programa de reuso de uma organização e explorar de forma sistemática oportunidades de reuso;
	REU.3 Engenharia de Domínio: propósito de desenvolver e manter modelos de domínio, arquiteturas de domínio e artefatos para o domínio.
NBR ISO/IEC 12207 [Machado 2006]	**Gestão de Ativos:** propósito de gerenciar a vida dos ativos reutilizáveis, desde sua concepção até a sua descontinuação;
	Gestão de Programa de Reuso: propósito de planejar, estabelecer, gerenciar, controlar e monitorar um programa de reuso da organização e, sistematicamente, explorar as oportunidades de reuso;
	Engenharia de Domínio: propósito de desenvolver e manter modelos, arquiteturas e ativos de domínio.
IEEE 1517-2004 [Werner 2007]	**Integração da Reutilização nos Processos Primários do Ciclo de Vida:** propósito de tratar como a reutilização impacta nos processos de aquisição, fornecimento, desenvolvimento, operação e manutenção;
	Suporte à Reutilização: propósito de gerenciar os ativos;
	Ciclo de Vida Organizacional da Reutilização: propósito de administrar o programa de reutilização;
	Reutilização entre Projetos: propósito de tratar a engenharia de domínio.
MR-MPS [MPS.BR 2007c]	**Gerência de Reutilização (GRU):** propósito de gerenciar o ciclo de vida dos ativos reutilizáveis;
	Desenvolvimento para Reutilização (DRU): propósito de identificar oportunidades de reutilização sistemática na organização e, se possível, desenvolver um programa de reutilização para desenvolver ativos a partir de engenharia de domínio de aplicação.

Tabela 1. Reutilização de Software nas Normas e Modelos [Porto, 2008]

Na próxima seção são apresentados com maior detalhamento o programa MPS.BR e seus componentes (entre eles, o MR-MPS) e, na seqüência, uma visão geral deste processo no nível de maturidade E.

Gerência de Reutilização no MPS.BR Nível E

O programa MPS.BR foi criado em 2003 e seu objetivo é a melhoria de processo do software brasileiro. Este programa está sob a coordenação da Associação para Promoção da Excelência do Software Brasileiro (SOFTEX), com apoio do Ministério da Ciência e Tecnologia (MCT), da Financiadora de Estudos e Projetos (FINEP) e do Banco Interamericano de Desenvolvimento (BID).

O MPS.BR baseia-se em conceitos de maturidade e capacidade de processo para a avaliação e melhoria da qualidade e produtividade de produtos de software e serviços correlatos. Este programa está estruturado nos seguintes componentes: Modelo de Referência (MR-MPS), Método de Avaliação (MA-MPS) e Modelo e Negócio (MN-MPS) [MPS.BR 2007c].

O MR-MPS possui sete níveis de maturidade seqüenciais e acumulativos, que são: A (Em Otimização), B (Gerenciado Quantitativamente), C (Definido), D (Largamente Definido), E (Parcialmente Definido), F (Gerenciado) e G (Parcialmente Gerenciado).

O nível de maturidade E tem como foco principal a padronização dos processos da organização, por meio da definição de processos padrão. Estes devem ser definidos a partir dos processos e melhores práticas já existentes na organização, o que constitui o primeiro passo de uma contínua avaliação e melhoria dos processos. A definição de processos padrão inclui, além dos processos do nível E, todos os processos que pertencem aos níveis G e F do MR-MPS [MPS.BR 2007d].

Neste nível de maturidade também são exigidos o estabelecimento de uma biblioteca de ativos e de um repositório de medidas. E, ainda, a organização deve também neste nível implementar uma estratégia de gerenciamento de ativos reutilizáveis para aumentar a eficiência e a eficácia dos processos de software da organização por meio da reutilização de produtos de trabalho projetados para utilização em múltiplos contextos [MPS.BR 2007d].

Este nível é formado pelos processos de Gerência de Projetos (GPR – Evolução), Avaliação e Melhoria do Processo Organizacional (AMP), Definição do Processo Organizacional (DFP), Gerência de Recursos Humanos (GRH) e Gerência de Reutilização (GRU), além dos outros seis processos dos níveis G e F. No que se refere a atributos de processo, é formado pelos atributos AP 3.1 (O processo é definido) e AP 3.2 (O processo está implementado), além dos outros três atributos de processo dos níveis G e F, conforme pode ser visto na **Tabela 2**.

O processo GRU é abordado com maior nível de detalhamento nesta seção em função de ser objeto deste artigo. Para maiores detalhes sobre o nível E e demais processos, consulte MPS.BR (2007d).

O processo GRU tem como objetivo definir procedimentos tanto administrativos quanto técnicos para utilização de ativos reutilizáveis em uma organização, estabelecendo e controlando uma biblioteca para o armazenamento e recuperação destes ativos. Entende-se como ativo reutilizável qualquer artefato relacionado a software que esteja preparado, isto é, empacotado de maneira própria a ser reutilizado pelos processos da organização [MPS.BR 2007d].

De acordo com MPS.BR (2007d), para que estes ativos possam ser usados de maneira efetiva, é necessário estabelecer um procedimento sistemático de armazenamento, recuperação e divulgação. Assim, o processo se apresenta como instrumento a ser aplicado neste contexto, promovendo mecanismos para estabelecimento e manutenção de uma infra-estrutura que torne viável a reutilização de ativos em uma organização.

Conforme definido na seção anterior, o propósito de GRU no MR-MPS é gerenciar o ciclo de vida dos ativos reutilizáveis. Isto é, o modo como os ativos devem ser usados na organização não cabe à GRU. Seu objetivo corresponde em práticas para viabilizar a seleção e a recuperação de ativos.

A seguir, são listados os resultados esperados de GRU que precisam ser observados e implementados para se obter sucesso no objetivo do processo [MPS.BR 2007d]:

• **GRU1**: Uma estratégia de gerenciamento de ativos é documentada, contemplando a definição de ativo reutilizável, além dos critérios para aceitação, certificação, classificação, descontinuidade e avaliação de ativos reutilizáveis;
• **GRU2**: Um mecanismo de armazenamento e recuperação de ativos reutilizáveis é implantado;
• **GRU3**: (Nos níveis E e D) Os dados de utilização dos ativos reutilizáveis são registrados;
• **GRU4**: Os ativos reutilizáveis são periodicamente mantidos, segundo os critérios definidos, e suas modificações são controladas ao longo do seu ciclo de vida.
• **GRU5**: Os usuários de ativos reutilizáveis são notificados sobre problemas detectados, modificações realizadas, novas versões disponibilizadas e descontinuidade de ativos.

Na **Tabela 2** são apresentados os atributos de processo, ou seja, as características mensuráveis da capacidade do processo que também precisam ser observadas e implementadas para GRU no nível de maturidade E.

A seção seguinte propõe uma forma de GRU ser implementada em organizações intensivas em software. Esta forma pretende servir apenas como referência, pois este processo deve ser implementado de acordo com as características e necessidades de negócio de cada organização.

Atributos de Processo (AP) / Resultados Esperados (RAP)
RAP1.1 O processo é executado.
RAP 1. O processo atinge seus resultados definidos.
AP2.1 O processo é gerenciado.
RAP 2. Existe uma política organizacional estabelecida e mantida para o processo.
RAP 3. A execução do processo é planejada.
RAP 4 (A partir do Nível F). Medidas são planejadas e coletadas para monitoração da execução do processo.
RAP 5. Os recursos necessários para a execução do processo são identificados e disponibilizados.
RAP 6. As pessoas que executam o processo são competentes em termos de formação, treinamento e experiência.
RAP 7. A comunicação entre as partes interessadas no processo é gerenciada de forma a garantir o seu envolvimento no projeto.
RAP 8. Métodos adequados para monitorar a eficácia e adequação do processo são determinados.
RAP 9 (A partir do Nível F) A aderência dos processos executados às descrições de processo, padrões e procedimentos é avaliada objetivamente e são tratadas as não conformidades.
AP 2.2 Os produtos de trabalho do processo são gerenciados.
RAP 10. Requisitos para documentação e controle dos produtos de trabalho são estabelecidos.
RAP 11. Os produtos de trabalho são documentados e colocados em níveis apropriados de controle.
RAP 12. Os produtos de trabalho são avaliados objetivamente com relação aos padrões, procedimentos e requisitos aplicáveis e são tratadas as não conformidades.
AP 3.1 O processo é definido
RAP 13. Um processo padrão é definido, incluindo diretrizes para sua adaptação para o processo definido.
RAP 14. A seqüência e interação do processo padrão com outros processos são determinadas.
AP3.2 O processo está implementado
RAP 15. Dados apropriados são coletados e analisados, constituindo uma base para o entendimento do comportamento do processo, para demonstrar a adequação e a eficácia do processo, e avaliar onde pode ser feita a melhoria contínua do processo.

Tabela 2. Atributos de Processo Aplicáveis à GRU [MPS.BR 2007d]

Implementando o processo de Gerência de Reutilização de Software

Uma forma de implementar processos na organização é por meio do modelo IDEAL, desenvolvido pelo *Software Enginnering Institute* (SEI) afim de apoiar na melhoria de processo de *software* [SEI, 2008].

Suas fases correspondem a:

• *Initiating* (Início): identificação da necessidade com estímulo à melhoria, definição de escopo, patrocínio, intra-estrutura;
• *Diagnosing* (Diagnóstico): consiste em avaliar o *status* atual e identificar as práticas correntes, identificação de pontos fracos, pontos fortes e oportunidades de melhorias;
• *Establishing* (Estabelecimento): planejamento das ações a serem realizadas, definição das prioridades, alocação de equipes envolvidas;
• *Acting* (Ação): por em prática o planejamento e ações necessárias para o desenvolvimento e implantação do processo e melhorias;
• *Learning* (Aprendizado): documentação e análise das ações.

Apresentamos na **Tabela 3** uma forma de implementação do processo de GRU,

Este modelo pode ser realizado de forma cíclica, ou seja, quando identificadas melhorias, pode-se retornar a atividade de diagnóstico ou então direto para etapa de ação,

Implantação de GRU com base no Modelo IDEAL	
Início	Identificar a necessidade de aplicação do processo de GRU. Uma forma de realizar este levantamento é identificar a relevância da aplicação do processo no contexto atual, verificando o grau (Baixo, Médio ou Alto) da Importância da aplicação e o Risco com relação a manter as práticas atuais da empresa. Nesta etapa também será definido o escopo em que será desenvolvido, neste caso, a aplicação do processo de GRU.
Diagnóstico	Identificar a situação atual com base nas práticas existentes na organização, que atendam ao processo de GRU. Uma forma prática para este levantamento é identificar os pontos fracos, pontos fortes e melhorias de acordo com os resultados esperados de GRU (GRU1 a GRU5).
Estabelecimento	Definir metas, cronograma, plano de ação para atender as necessidades de melhoria, de acordo com os pontos levantados no diagnóstico para atendimento completo dos resultados esperados. Esta etapa consiste no planejamento da implantação do processo, assim como ocorre no planejamento de um projeto de software, identificar os recursos necessários (infra-estrutura, recursos humanos, ferramentas, financeiro, etc), treinamentos (se necessário), aplicação em projetos piloto e envolvidos.
Ação	Consiste na realização do planejado, executar as atividades necessárias para a implementação do processo de GRU. Por exemplo: reuniões para definição do processo, documentação do processo, prover treinamentos para as equipes pilotos, aplicação do piloto.
Aprendizado	Nesta etapa, são verificadas e analisadas as ações realizadas para identificar melhorias e aplicá-las. Para isto, indicadores podem ser definidos para melhor controle, bem como aplicação de auditorias, a partir delas é possível identificar melhorias tanto no processo definido quanto na aplicação do mesmo, sendo possível melhor capacitação dos envolvidos na realização do processo.

Tabela 3. Exemplo de Ciclo de Implementação e Melhoria do Processo de GRU

até a adequação do processo de forma que atenda todos os resultados esperados do processo de GRU.

A definição do processo de GRU pode ser feita com base na notação ETVX (*Entry criteria*, *Task*, *Verification*, *and eXit criteria*), na qual são identificados o propósito, os critérios de entradas, entradas, atividades, verificação e critérios de saída, bem como responsáveis pela realização de cada atividade, verificação e responsáveis por artefatos (Radice, 1988).

A **Figura 1** representa uma forma de definir o processo de GRU com esta notação adaptada. A representação está na primeira atividade "Identificar Ativos Reutilizáveis" para exemplo de aplicação. Tendo como Critérios de Entrada os Critérios para Seleção de Ativos Reutilizáveis (um guia que orienta e apresenta a classificação para ativos reutilizáveis); Entrada representada por artefatos e componentes de software; Atividade Identificar Ativos Reutilizáveis; Verificação através da revisão dos artefatos/componentes propostos para ativos reutilizáveis e por fim, a saída desta atividade resulta na Lista de Artefatos revisados e elencados a Ativos reutilizáveis.

Outro ponto a ser considerado na implementação de GRU deve ser a forma como o processo será medido e monitorado na organização. De acordo com Filho (2008), esta definição foi uma das principais dificuldades na implantação deste processo. Nesta experiência, foram levados em consideração aspectos como verificação de alcance dos objetivos do processo, utilidade na identificação de

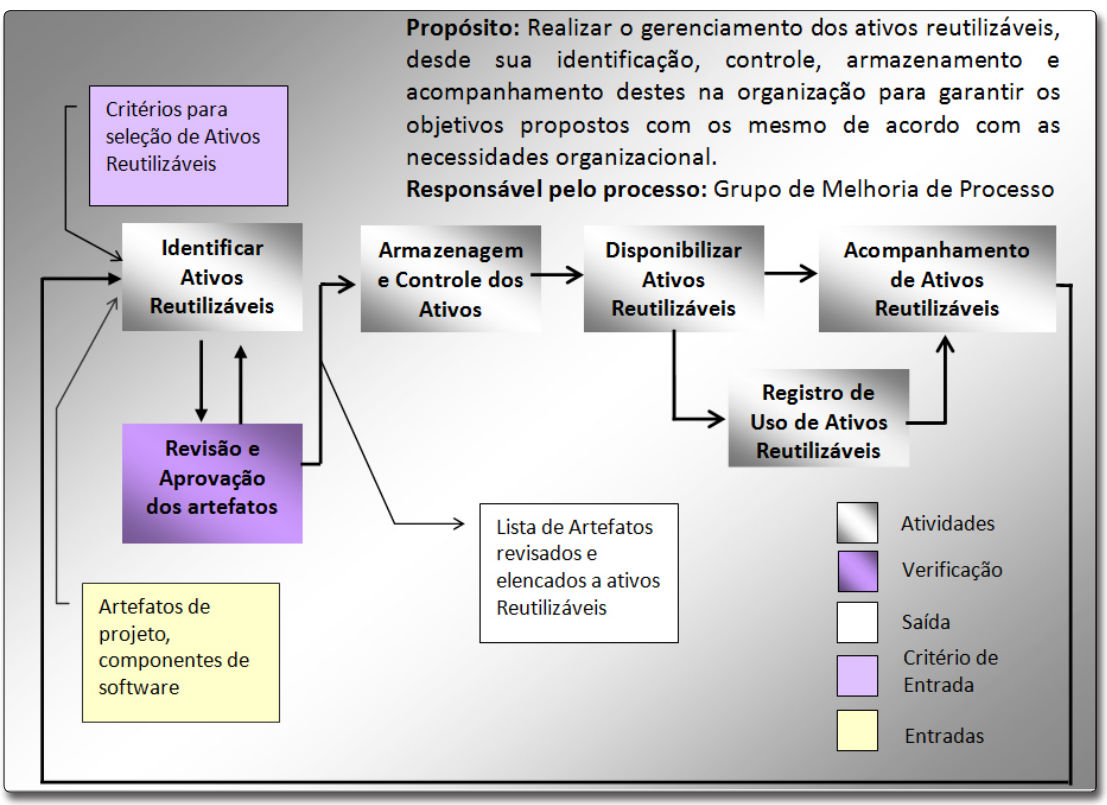


Figura 1. Exemplo de Implementação do processo de Gerência de Reutilização com a notação ETVX

oportunidades de melhoria, representatividade da métrica e custo de medição associado. Sendo que foram identificados dois indicadores: taxa de reutilização dos ativos e evolução da base de ativos reutilizáveis.

Conclusão

A Gerência de Reutilização deve apoiar a organização a reaproveitar artefatos e partes de software, melhorando a qualidade do produto, garantida pela utilização de ativos já testados e reutilizados em outros projetos, assim como o ganho de produtividade, atalhando etapas de desenvolvimento.

Vimos neste artigo que Gerência de Reutilização proporciona outros benefícios à organização como redução do custo e esforços envolvidos no processo de desenvolvimento, bem como otimização da flexibilidade da estrutura do software, que facilitará sua manutenção posteriormente.

Todavia, cuidados devem ser tomados para não onerar a aplicação deste processo, no qual seu objetivo é gerenciar o ciclo de vida dos ativos reutilizáveis. O mecanismo de como será gerenciado, desde critérios para captação de ativos reutilizáveis, controle de uso a manutenção destes, deve ser feito de acordo com o perfil da empresa. De acordo com Filho (2008), o uso de controles manuais poderá tornar o processo exaustivo e propenso a erros, podendo ser minimizado com processos automatizados.

Embora seja um processo recente no MR-MPS (desde junho 2007), quando bem implementado, seus resultados estão voltados a objetivos estratégicos de qualquer organização: melhorar a produtividade e a redução de custo de desenvolvimento. ●

Links

MPS.BR
www.softex.br/mpsbr/_home/default.asp

COMPOSE
www.compose.ufpb.br

COPPE/UFRJ - Reutilização de Software
http://reuse.cos.ufrj.br

C.R.U.I.S.E – Component Reuse in Software Engineering
http://cruise.cesar.org.br/index.html.

RISE - Reuse in Software Engineering
http://www.rise.com.br

Referências

Becker, Carlos (2008) "Reuso de Software no Contexto das Empresas da Cooperativa MPS.BR - SOFTSUL", In: II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software, Porto Alegre.

COMPOSE (2008) "Component Oriented Service Engineering", *http://www.compose.ufpb.br.*

COPPE/UFRJ (2008) "Laboratório de Engenharia de Software – Equipe de Reutilização de Software", *http://reuse.cos.ufrj.br.*

CRUISE (2007) "C.R.U.I.S.E – wComponent Reuse in Software Engineering", *http://cruise.cesar.org.br/index.html.*

Filho, Reinaldo C. Silva; Katsurayama, Anne Elise; Santos, Gleison; Murta, Leonardo; Rocha, Ana Regina (2008) "A Experiência na Implantação do Processo de Gerência de Reutilização no Laboratório de Engenharia de Software da COPPE/UFRJ", In: IV Workshop de Implementadores (W2 – MPS.BR) e II Workshop de Empresas (W6 – MPS.BR), Belo Horizonte.

IEEE (2004) "Std 1517 - IEEE Standard for Information Technology - Software Life Cycle Processes - Reuse Processes", Institute of Electrical and Electronics Engineers.

Krueger, C.W. (1992) "Software Reuse", In: ACM Computing Surveys, Vol. 24, Nº 02.

Machado, Cristina Ângela Filipak (2006) "Definindo Processos do Ciclo de Vida de Software usando a Norma NBR ISO/IEC 12207 e suas Ementas 1 e 2", Curso de Pós-Graduação "Latu Sensu" a Distância: Melhoria de Processo de Software, UFLA, Lavras.

MPS.BR (2007a) "SOFTEX – MPS.BR – Implementações em Grupos de Empresas –G1", *http://www.softex.br/mpsbr/_implementacoes/MPS.BR_SOFTSUL.pdf.*

MPS.BR (2007b) "SOFTEX – MPS.BR – Avaliações MA-MPS – Qualità Nível F do MPS.BR", *http://www.softex.br/mpsbr/_avaliacoes/avaliacao.asp?id=1434.*

MPS.BR (2007c) "SOFTEX - MPS.BR – Guia Geral", *http://www.softex.br/mpsbr/_guias/MPS.BR_Guia_Geral_V1.2.pdf.*

MPS.BR (2007d) "SOFTEX - MPS.BR – Guia de Implementação – Parte 3: Nível E", *http://www.softex.br/mpsBr/_guias/MPS.BR_Guia_de_Implementacao_Parte_3_v1.1.pdf.*

Porto, Josiane Brietzke (2008) "Gerência de Reutilização Alinhada ao MPS.BR Nível E", Monografia de Pós-graduação em Melhoria de Processo de Software, Universidade Federal de Lavras (UFLA), Lavras.

Radice, Ronald A.; Phillips, Richard W (1988) "Software Engineering, An Industrial Approach", Englewood Cliffs, New Jersey: Prentice Hall.

RISE (2008) "Reuse in Software Engineering", *http://www.rise.com.br.*

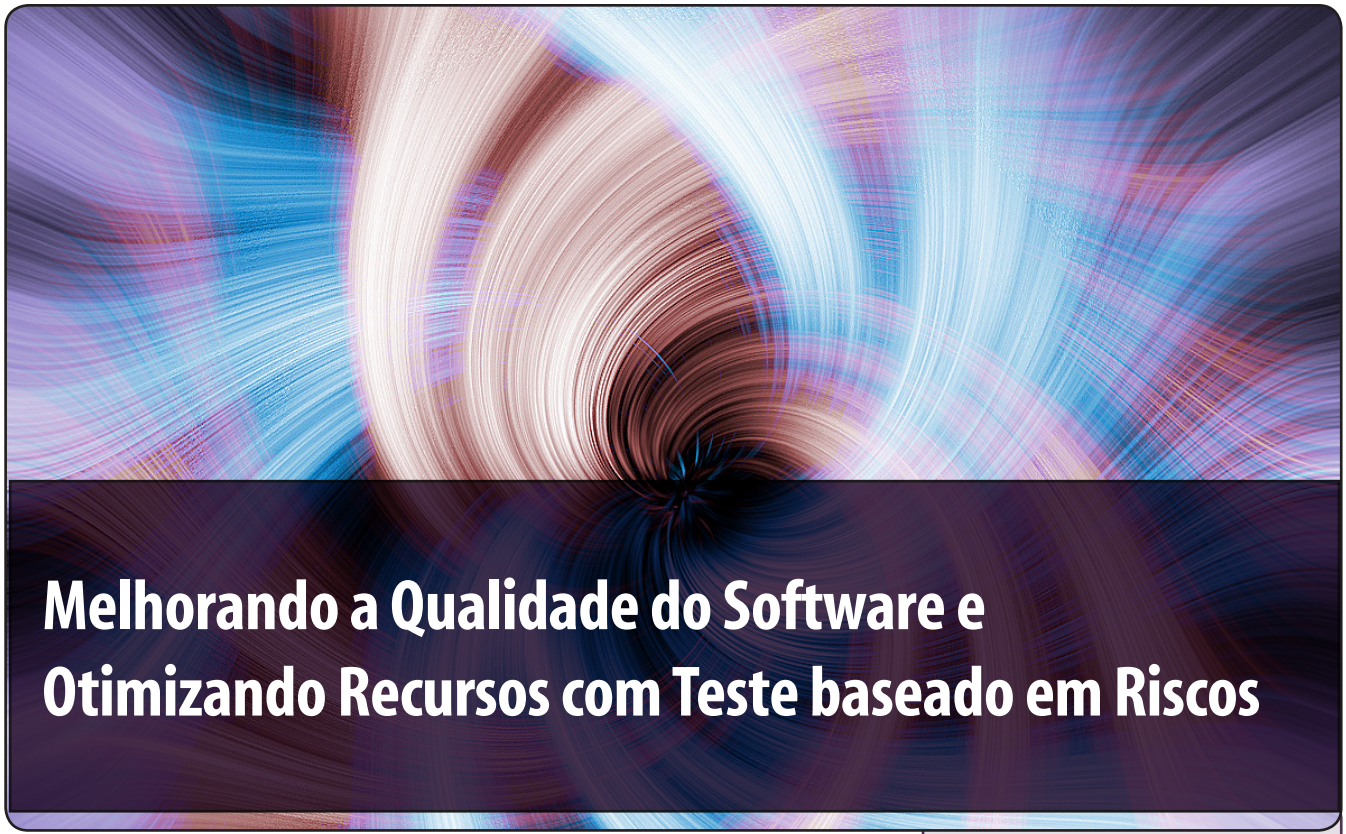
Salviano, Clênio Figueiredo (2006a) "Melhoria e Avaliação de Processo de Software com o Modelo ISO/IEC 15504-5:2006", Curso de Pós-Graduação "Latu Sensu" a Distância: Melhoria de Processo de Software, UFLA, Lavras.

Salviano, Clênio Figueiredo (2006b) "Uma Proposta Orientada a Perfis de Capacidade de Processo para Evolução da Melhoria de Processo de Software", Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas (FEEC-Unicamp). SEI. SOFTWARE ENGINEERING INSTITUTE.CMMI for Development (CMMI-DEV), Version 1.2, Technical report CMU/SEI-2006-TR-008. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

SEI. SOFTWARE ENGINEERING INSTITUTE (2008) "The IDEAL Model", *http://www.sei.cmu.edu/ideal/.*

Werner, Cláudia Maria Lima (2007) "Gerência de Reutilização de Software", Minicurso de Gerência de Reuso – MPS.BR, SOFTEX, Belo Horizonte.

Werner, Cláudia Maria Lima (2008) "GLOBO Ciência – Reutilização de Software", *http://reuse.cos.ufrj.br/files/videos/20080624_GloboCiencia.html.*



Melhorando a Qualidade do Software e Otimizando Recursos com Teste baseado em Riscos

Ellen Souza

eprs@dsc.upe.br

Professora Assistente do Curso de Bacharelado em Sistemas de Informação da Unidade Acadêmica de Serra Talhada (UAST) da Universidade Federal Rural de Pernambuco (UFRPE). Mestre em Engenharia da Computação pela Universidade de Pernambuco (UPE). Residente do Curso Seqüencial de Formação Complementar em Teste de Software pelo Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), em parceria com a Motorola. Graduada em Ciência da Computação pela Universidade Católica de Pernambuco (UNICAP).

Cristine Gusmão

cristine@dsc.upe.br

Professora Assistente do Departamento de Sistemas e Computação da Escola Politécnica da Universidade de Pernambuco (POLI UPE), onde leciona várias disciplinas na graduação e pós-graduação (especialização e mestrado) e das Faculdades Integradas Barros Melo. Doutora e Mestre em Ciência da Computação pela Universidade Federal de Pernambuco. Graduada em Engenharia Elétrica Eletrotécnica pela Universidade Federal de Pernambuco.

As empresas, de forma geral, têm despertado para a importância da atividade de teste de software como forma de melhorar a qualidade dos seus produtos e manterem-se competitivas no mercado. Além disso, a complexidade das tecnologias utilizadas e dos softwares produzidos tem crescido, tornando-se necessária a utilização de processos, técnicas e ferramentas que permitam a realização de teste de software de maneira sistematizada, com o objetivo de aumentar a qualidade do software com o menor custo possível.

Da mesma forma, a gerência de riscos tem sido fortemente debatida, estudada e utilizada em ambientes de

De que se trata o artigo?

Apresentação de uma visão geral da abordagem de teste baseado em riscos e técnicas disponíveis para o planejamento, projeto e execução de testes de software.

Para que serve?

Fornecer uma motivação para o uso da abordagem de teste baseado em riscos, destacando seus benefícios, limitações e aplicações.

Em que situação o tema é útil?

O processo de teste de software exerce um importante papel dentro da garantia de qualidade de software assegurando que os requisitos satisfazem as necessidades das partes envolvidas. Este processo, entretanto, requer bastante esforço, dentro de restrições de custo e prazo. A abordagem de teste baseado em riscos permite identificar funcionalidades com grande probabilidade de apresentar falhas, de forma a alocar melhor os recursos disponíveis, diminuindo o tempo e custo necessários para se testar. Este artigo apresenta a abordagem de teste baseado em riscos, suas aplicações, limitações, bem como as técnicas disponíveis para o planejamento, projeto e execução de testes funcionais e estruturais.

desenvolvimento de software com o propósito de administrar oportunidades, aumentando a probabilidade de entregar o software com o menor custo, no menor tempo possível e com maior qualidade.

A atividade de teste de software, no entanto, é complexa e demanda tempo considerável para ser realizada, chegando a custar até metade do valor inicial de desenvolvimento de um software.

O teste baseado em riscos (RBT – Risk-based Testing), por sua vez, permite priorizar esforços e alocar recursos para os componentes de software que necessitam ser testados mais cuidadosamente a partir da identificação, análise e controle dos riscos técnicos associados aos requisitos do software.

O consultor James Bach é considerado o pai da abordagem de Teste baseado em Riscos. Ele descreveu a idéia em seu artigo intitulado: *The Challenge of Good Enough Software*, em outubro de 1995, na revista *American Programmer*. Somente em 1999, [Bach 1999] apresenta uma técnica baseada em heurísticas para RBT e desde então, diversas técnicas têm sido propostas e utilizadas para testar produtos de software com diferentes domínios, em empresas como IBM® [Chen 2002] e outras. Estas técnicas são apresentadas neste artigo, destacando seus pontos fortes, aplicações e limitações.

Por que Realizar Teste baseado em Riscos?

É bastante freqüente organizações depararem-se com restrições de tempo e recursos para o desenvolvimento de software, em especial, para realização de teste de software. A abordagem RBT permite justamente fazer melhor uso dos recursos disponíveis, priorizando os requisitos do software que necessitam ser testados prioritariamente. Assim, a quantidade de teste planejada e realizada para o software será na proporção do risco envolvido. Quanto maior o risco, mais teste é necessário. Da mesma forma que requisitos ou funcionalidades com baixa exposição ao risco não necessitam ser testados exaustivamente.

Além disso, de acordo com estudos realizados, os defeitos com maior severidade podem ser descobertos mais cedo, tendo em vista que os requisitos mais problemáticos serão testados prioritariamente, e conseqüentemente serem corrigidos mais cedo.

RBT também é uma valiosa ferramenta para o gerente de teste que pode planejar as atividades e alocar/solicitar recursos considerando, não somente a informação das funcionalidades do software que estão em desenvolvimento ou manutenção, mas também quais riscos estão associados a cada funcionalidade e quais funcionalidades são mais críticas com relação ao teste de software.

Por fim, como conseqüência de uma atividade de teste bem planejada, a qualidade do software é melhorada.

Teste de Software não é baseado em Riscos?

É consenso que a realização de qualquer atividade de teste diminui o risco de um software apresentar falhas em produção. Teste é primariamente uma forma de controlar os riscos associados aos requisitos ou funcionalidades de um software. A partir dessas definições podem surgir as seguintes perguntas: Porque então o termo Teste baseado em Riscos? Esse termo não parece redundante? De certa forma, o termo é sim redundante, entretanto ele é utilizado para destacar um conjunto de atividades da gerência de riscos que pode ser utilizado para estabelecer melhorias nos processos de teste de software, tais como:

1. Identificar riscos associados aos requisitos ou funcionalidades do software.
2. Analisar e Priorizar os riscos identificados, através dos valores de probabilidade de ocorrência e impacto. Como os riscos estão associados aos requisitos, estes últimos são priorizados indiretamente.
3. Projetar casos de teste com base nas estratégias para tratamento dos riscos identificados. Um ou mais casos de teste podem ser projetados para mitigar um risco.
4. Controlar os riscos priorizados através da execução dos casos de teste. Quando executamos um caso de teste baseado em riscos e este falha, o software necessita ser corrigido para que o risco seja mitigado.

Os processos de teste disponíveis na literatura tratam os riscos associados aos requisitos de software de forma *ad hoc* [Bach 1999]. Casos de teste são planejados, projetados, executados e controlados com o propósito de diminuir riscos. Mas que riscos são estes que estão sendo tratados? São os riscos mais importantes? As funcionalidades que apresentam maior risco foram testadas adequadamente? Essas e outras perguntas podem ser respondidas com a adoção de processo ou técnicas de teste baseado em riscos.

Quais Riscos são Tratados pela Abordagem RBT?

O Instituto de Engenharia de Software (SEI) [Carr *et al.* 1993] define risco como a possibilidade de sofrer perdas nos objetivos do projeto, tais como: impactar na qualidade do produto final, atrasar cronograma, aumentar custos ou mesmo falhar o projeto.

Os riscos de software podem ser classificados de diversas formas. Neste artigo, é apresentada a classificação definida pelo SEI, que fornece uma taxonomia para os riscos de acordo com a sua origem (**Figura 1**):

1. Engenharia de Produto: problemas no produto que estão relacionados às ações técnicas; também conhecidos como riscos técnicos.

2. Ambiente de Desenvolvimento: problemas no processo (desenvolvimento) produtivo do software.

3. Restrições dos Programas: problemas que acontecem no projeto e no processo devido a ações da gerência.

A **Figura 1** apresenta um subconjunto da classificação de riscos de software definida pelo SEI [Carr *et al* 1993]. Os riscos relacionados à engenharia de produto são os que podem ser tratados pelo teste de software, mais especificamente os que estão relacionados aos requisitos do software, como estabilidade, completude, claridade, viabilidade, validade, precedente, escala e outros. O risco está associado à possibilidade de uma

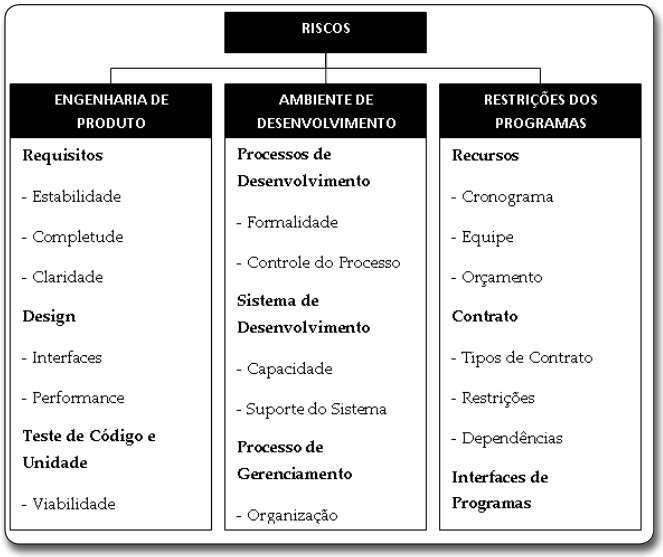


Figura 1. Classificação dos riscos de software.

funcionalidade do software funcionar de forma inadequada, ou até mesmo não funcionar.

Para exemplificar, suponha que a funcionalidade Cadastrar Usuário foi definida, documentada e será implementada. Ao revisar a funcionalidade, a equipe de testes identificou que um dos fluxos não está muito claro, é muito complexo ou gerou margem para diversas interpretações. Prontamente, o analista de teste projeta um caso de teste para verificar se esse fluxo foi implementado da forma correta pelos desenvolvedores e mitigar o risco identificado.

Após a identificação dos riscos, estes necessitam ser priorizados de acordo com a sua probabilidade de ocorrência e impacto. Alguns estudos [Amland 1999] indicam que funcionalidades extensas, complexas, desenvolvidas sob pressão, com pouco tempo e por desenvolvedores inexperientes ou sem conhecimento da tecnologia utilizada, possuem uma grande probabilidade de apresentar falhas. Similarmente acontece com funcionalidades que estão em constante manutenção ou que têm apresentado muitas falhas. Esses indícios nos ajudam a identificar a probabilidade de ocorrência de falhas.

Com relação ao impacto, uma forma bastante simples de priorizar os riscos é verificando o impacto de ocorrência de uma falha em uma funcionalidade para o cliente. Se a atividade fim de um cliente é vender livros e existe um risco associado à funcionalidade de venda, o seu impacto é bastante alto, uma vez que o cliente poderá ter sérios prejuízos caso as vendas não possam ser realizadas.

Técnicas e Processo de Teste baseado em Riscos

Nesta seção, são apresentadas algumas das técnicas e processos disponíveis para a abordagem de teste baseado em riscos, que é utilizada, mais fortemente, no planejamento e na estratégia de execução dos casos de testes [Bach 1999]. Seu uso, entretanto é recomendado também já na de fase construção ou projeto dos casos de teste, como forma de otimizar o processo de teste de software, projetando casos de teste apenas para os requisitos prioritários.

A **Figura 2** apresenta o modelo de atividades do processo de gerência de riscos, alterado por [Amland 1999] para incluir elementos relevantes ao teste baseado em risco. As caixas ovais representam as alterações realizadas por Amland e são artefatos produzidos ou atividades do teste de software realizadas com o apoio de atividades (retângulos) do processo de gerência de riscos.

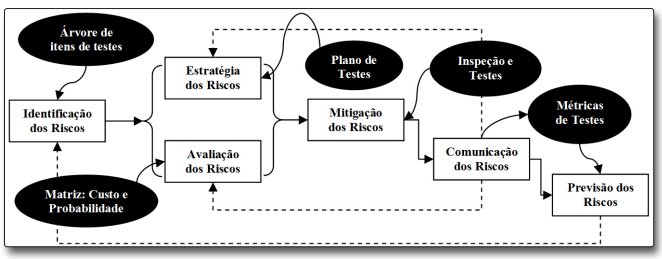


Figura 2. Atividades relevantes para o teste baseado em riscos [Amland 1999].

Para cada atividade do ciclo de vida do processo de teste de software, há uma correspondente na gerência de riscos, com o intuito de fornecer o tratamento e acompanhamento adequado para os riscos técnicos identificados.

A atividade de identificação de riscos produz como resultado um lista de riscos associados aos requisitos a serem testados. A partir da avaliação qualitativa e/ou quantitativa, os riscos são priorizados. Também, estratégias para mitigação dos riscos são elaboradas de acordo com sua prioridade e estas informações servem como base para criação do plano de testes.

Quando os testes ou inspeções são realizados, os riscos são mitigados, ou pelo menos, são minimizados para que o impacto dos fatores de riscos seja menor. A comunicação dos riscos fornece os dados para definição e acompanhamento dos riscos através de um conjunto de métricas.

Técnica Baseada em Heurística

[Bach 1999] define uma técnica baseada em heurística utilizando duas abordagens distintas para a atividade de identificação dos riscos: na abordagem "Inside-Out", o software é estudado e é questionado repetidamente, o que pode dar errado neste produto; na abordagem "Outside-In", uma lista de potenciais riscos é utilizada, juntamente com detalhes do software. Para cada risco, é identificado se este se aplica ou não a uma determinada funcionalidade.

Nessa técnica, três tipos de listas são utilizadas para auxiliar a identificação dos riscos:

1 - Lista de categorias de critérios de qualidade: são categorias desenvolvidas para atenderem a diferentes tipos de requisitos. Um exemplo deste tipo de lista é apresentado na **Tabela 1**, que foi desenvolvida a partir dos critérios do padrão ISO 9126 e HP FURPS (*Functionality, Usability, Reliability, Performance, Supportability*).

Capacidade	O requisito realiza a função requerida?
Confiabilidade	A funcionalidade resiste a falhas em todas as situações?
Usabilidade	Quão fácil é a utilização da funcionalidade pelo usuário?
Instalabilidade	Quão fácil é a instalação do software?
Manutenibilidade	Quão econômicas são as manutenções corretivas e evolutivas?
Portabilidade	Quão econômica é a portabilidade para outra plataforma?

Tabela 1. Lista de categorias de critérios de qualidade [Bach 1999].

2 - Listas genéricas de riscos: são aquelas que possuem riscos universais a qualquer sistema. Um exemplo é mostrado na **Tabela 2**.

Complexo	Qualquer coisa desproporcionalmente grande.
Novo	Qualquer coisa que não possua histórico no produto.
Modificado	Qualquer coisa que tenha sido modificada ou melhorada.
Crítico	Qualquer coisa cuja falha poderia causar um dano substancial.
Preciso	Qualquer coisa que deva atender a requisitos exatos.
Popular	Que será muito usado.
Terceirizado	Qualquer coisa usada no seu produto, mas que foi desenvolvida fora do projeto.
Distribuído	Qualquer coisa que esteja espalhada em relação a tempo ou espaço, e que seus elementos devam trabalhar juntos.
Falhou recentemente	Qualquer coisa com uma história recente de falha.

Tabela 2. Lista genérica de riscos [Bach 1999].

3 - Os catálogos de risco: São listas de riscos que pertencem a um domínio em particular. A **Tabela 3** apresenta uma versão resumida do que seria um catálogo de riscos para o instalador de um software.

Instalação dos arquivos errados.
Arquivos corrompidos.
Hardware não foi configurado de forma apropriada.
Outras aplicações corrompidas.
O protetor de tela interfere no instalador.
Não há detecção de aplicações incompatíveis.
O instalador silenciosamente substitui ou modifica arquivos críticos ou parâmetros.
O processo de instalação é muito lento.
O processo requer o constante monitoramento do usuário.
O processo de instalação é confuso.

Tabela 3. Catálogo de riscos para um instalador [Bach 1999].

Após a identificação dos riscos usando umas das abordagens explicadas anteriormente, valores são atribuídos a cada um dos riscos em uma escala de interesse. Os requisitos associados aos riscos com maior valor são testados primeiramente.

A principal dificuldade dessa abordagem está no conhecimento prévio do negócio a fim de realizar a análise dos riscos da maneira mais fiel possível. Existe a possibilidade da análise ter sido realizada de forma errada e, conseqüentemente, os "verdadeiros" riscos não terem sido atacados da forma correta.

Técnica Baseada em Métricas

Esta técnica foi desenvolvida por [Amland 1999] e consiste em um conjunto de métricas para a análise de riscos com o objetivo de subsidiar um processo de teste. Essas métricas foram aplicadas em estudo de caso de uma aplicação de instituição financeira. Existem três fontes de análise de riscos representadas pela **Equação 1**:

$$\mathrm{Re}(f) = P(f) * \frac{C(c) + C(v)}{2}$$

Equação 1. Cálculo para exposição do risco [Amland 1999].

1 - Qualidade da função (área) a ser testada: projetos de má qualidade, programador inexperiente, funcionalidade complexa, e outras variáveis resultam em funções com maior exposição a falhas. Essa função corresponde à probabilidade **P(f)**.

2 - As conseqüências de uma falha em uma função do ponto de vista de um cliente em uma situação de produção podem ser representadas pela probabilidade de ameaça legal, perda de posicionamento no mercado e pelo não cumprimento de regulamentações governamentais por causa de falhas, entre outras. Estas conseqüências representam o custo para o consumidor **C(c)**.

3 - As conseqüências de uma falha em uma função do ponto de vista do vendedor do serviço estão relacionadas à probabilidade de publicidade negativa, altos custos de manutenção de software, e outros, devido a uma função com falhas. Estas conseqüências representam o custo para o vendedor **C(v)**.

Tendo em vista o grau de exposição ao risco **Re(f)**, as áreas com risco mais alto têm a maior prioridade nos testes. Durante os testes, à medida que falhas vão ocorrendo, os graus de exposição ao risco vão sendo atualizados e a prioridade das funcionalidades pode ser modificada.

Um exemplo do grau de exposição ao risco para a função Fechar Contas é mostrado na **Tabela 4**. O Custo é calculado através da média aritmética de **C(v)** e **C(c)**. Os números entre parênteses são os pesos definidos para cada métrica. A probabilidade é calculada como a média ponderada das métricas, dividida pela maior média ponderada de todas as métricas, resultando em uma probabilidade na faixa de [0,1].

A técnica definida por Amland foi utilizada para testar dois módulos de uma aplicação financeira contendo,

	CUSTO			PROBABILIDADE						
Função	C(v)	C(c)	Média	Nova Função (5)	Qualidade (5)	Tamanho (1)	Complexidade (3)	Média Ponderada	Probabilidade	Exposição ao Risco
Fechar Conta	1	3	2	2	2	2	3	7,75	0,74	1,48

Tabela 4. Exposição ao risco para a função Fechar Contas [Amland 1999].

respectivamente, doze e trezentas funcionalidades. Sendo que, para o segundo, por falta de tempo, foram avaliadas as vinte funcionalidades mais importantes definidas pelo cliente. Um dos maiores problemas relatados por Amland foi a falta de conhecimento de algumas funcionalidades, que dificultou a análise e produziu resultados não confiáveis.

Um nível mínimo de teste foi definido para as funcionalidades com baixa exposição ao risco e testes extras foram definidos para funcionalidades com maior exposição ao risco. O cliente avaliou o produto entregue como de excelente qualidade. O número de defeitos encontrados foi similar ao das versões anteriores. No entanto, o tempo gasto para concluir os testes foi menor e o número de recursos utilizados também foi menor.

O grande desafio para definição da técnica, segundo Amland, foi a identificação dos indicadores de custo de falha e de qualidade. Foi utilizada uma abordagem simples, mas satisfatória de acordo com os estudos apresentados. Assim, como na maioria das técnicas, é mantido foco somente na análise dos riscos.

Técnica para Código-fonte Orientado a Objetos

A idéia desta técnica é que, através da aplicação de métricas de complexidade de software orientado a objetos, pode-se chegar a classes que possuem maior probabilidade de apresentar falhas. Seis métricas de medição de projetos orientadas a objeto são utilizadas, identificadas e aplicadas pelo SATC (Software Assurance Technology Center) da NASA (Goddard Space Flight Center). Segundo os autores, foi comprovado [Rosenberg *et al* 1999] que o código mais complexo tem uma maior incidência de erros ou problemas. As métricas são:

1 - Número de métodos (Number of Methods – NOM): é a contagem dos diferentes métodos existentes em uma classe.

2 - Número ponderado de métodos por classe (The Weighted Methods per Class –WMC): é a soma ponderada dos métodos em uma classe.

3 - Acoplamento entre objetos (Coupling Between Objects – CBO): é a contagem do número de outras classes nas quais uma classe está acoplada.

4 - A resposta a uma classe (The Response for a Class – RFC): é a cardinalidade do conjunto de todos os métodos que podem ser invocados em resposta a uma mensagem para um objeto da classe.

5 - Profundidade na árvore (Depth in Tree – DIT): é o número de saltos saindo de uma classe até a raiz da hierarquia de classes. Quando existe herança múltipla, a maior DIT é utilizada.

6 - Número de filhos (Number of Children – NOC): é o número de subclasses que herdam diretamente da classe na hierarquia.

Por mais de três anos, o SATC tem coletado e analisado código orientado a objetos escritos tanto em C++ quanto em Java. Mais de 20.000 classes de mais de 15 programas foram analisadas. Os seguintes valores limites para as métricas individuais, apresentados na **Tabela 5**, foram derivados do estudo da distribuição das métricas coletadas.

Uma métrica nunca dever ser utilizada sozinha para avaliar os riscos do código. São necessárias, pelo menos, duas ou três métricas para dar uma indicação de problema em potencial. Portanto, para cada projeto, o SATC cria uma tabela de classes que possuem alto risco. Classes que têm ao menos duas métricas que excedem os limites recomendados possuem alto risco.

MÉTRICA	OBSERVAÇÃO
NOM	Preferencialmente menor que 20 e aceitável até 40.
WMC	Preferencialmente menor que 25 e aceitável até 40.
CBO	Aceitável até 5.
RFC	Aceitável até 50.
DIT	Aceitável até 5.
NOC	Não há consenso. Quanto maior, maior a probabilidade de erro.

Tabela 5. Valores limites para métricas individuais.

A **Tabela 6** mostra um exemplo de métricas para classes de um projeto Java. As células em vermelho representam métricas fora do limite aceitável.

CLASSE	NO. MÉTODOS	CBO	RFC	RFC/NOM	WMC	DIT	NOC
1	54	8	536	9.9	175	1	0
2	7	6	168	24	71	4	0
3	33	4	240	7.2	105	2	0
7	54	8	361	6.7	117	2	2
8	62	6	378	6.1	163	2	0
10	63	7	235	3.7	156	2	0
11	81	10	285	3.5	161	2	0
12	42	5	127	3	69	3	0
13	20	17	324	16.2	139	4	4
14	46	5	186	4	238	1	3

Tabela 6. Métricas coletadas de um projeto Java.

A análise das classes é fácil de ser aplicada, inclusive pode ser realizada de forma automática. No entanto, os autores não propõem estratégias de testes para o código, tampouco formas de rastreamento das funcionalidades impactadas por classes com alto risco de falhas.

Técnica para Teste de Regressão baseado em Riscos

Essa técnica define uma estratégia para execução de testes de regressão baseada em especificação [Chen 2002]. A justificativa principal é que teste de regressão baseado em código é bom para teste de unidade, mas tem problemas de escalabilidade. À medida que o tamanho do sistema cresce, torna-se difícil gerenciar a informações dos testes e criar matrizes de rastreabilidade. Nesta técnica, dois conjuntos de testes de regressão são definidos:

1 - Testes de regressão para os componentes que foram alterados: pelo menos um teste é executado para cada componente inserido ou alterado.

2 - Testes de regressão baseados em riscos para os componentes que "aparentemente" não sofreram modificações: para estes, uma análise de riscos é realizada a partir de questionários submetidos aos participantes do projeto.

Essa técnica utiliza as métricas de análise de risco desenvolvidas por Amland e apresentadas anteriormente.

A fase de seleção de casos de teste envolve os seguintes passos:

1 - Estimar o custo de cada caso de teste. Ou seja, o custo que uma falha possa causar. O custo é calculado da mesma forma proposta por [Amland 1999];

2 - Derivar a severidade para cada caso de teste. Esse valor é computado a partir do número de defeitos e da severidade destes defeitos;

3 - Calcular o grau de exposição de risco para cada caso de teste. A exposição ao risco é calculada a partir do custo e da severidade, e;

4 - Selecionar os casos de teste que têm os maiores valores de exposição ao risco.

A seleção de cenários baseados em riscos deve obedecer duas regras:

1 - Selecionar os cenários para cobrir os casos de teste mais críticos, e;

2 - Garantir que os cenários cubram tantos casos de teste quanto possível.

A seleção de cenários baseado em riscos tem os seguintes passos:

1 - Calcular a exposição de riscos para cada cenário. Calculado a partir da soma da exposição ao risco dos casos de testes associado a este cenário;

2 - Selecionar os cenários com maior exposição ao risco;

3 - Atualizar a matriz de rastreabilidade, removendo os cenários selecionados e os testes já cobertos e recalculando a exposição ao risco, e;

4 - Repetir os passos 1, 2 e 3, o quanto necessário.

Esta técnica se mostrou bastante eficiente de acordo com os resultados apresentados. Além disso, a análise realizada através de questionários submetidos aos desenvolvedores, com questões que os mesmos dominam, não constituiu uma tarefa complexa. Com a ajuda de ferramentas, todo o processo é realizado de forma rápida.

Como a abordagem toma como base a documentação do sistema, essa deve encontrar-se sempre atualizada. Riscos não são identificados nesta técnica e assume-se que os casos de testes já estão prontos. Dependendo da quantidade de casos de testes, essa técnica pode ser inviável, uma vez que a análise de riscos é feita a partir dos casos de teste.

Técnica RBT baseada em Uso

Esta técnica define uma estratégia baseada em uso, onde qualquer atividade de teste implica em uma diminuição dos riscos [Besson 2004]. De acordo com o autor, independente da quantidade de testes executados, sempre haverá um risco. A idéia é investir o mínimo de esforço em teste possível para maximizar a redução dos riscos.

Os esforços de teste são controlados tomando como base a utilização do sistema, seguindo a teoria de Pareto, que afirma que vinte por cento das funcionalidades permitem ao usuário realizar oitenta por cento do seu trabalho.

A **Figura 3** apresenta o esforço necessário para eliminar todos os riscos identificados (reta em vermelho). Seguindo a idéia de Pareto, a relação entre risco e esforço ficaria mais parecida com o gráfico representado pela linha em azul. Logo, diminuir o risco em cinqüenta por cento pode ser alcançado em um curto espaço de tempo se uma priorização dos testes é feita a partir

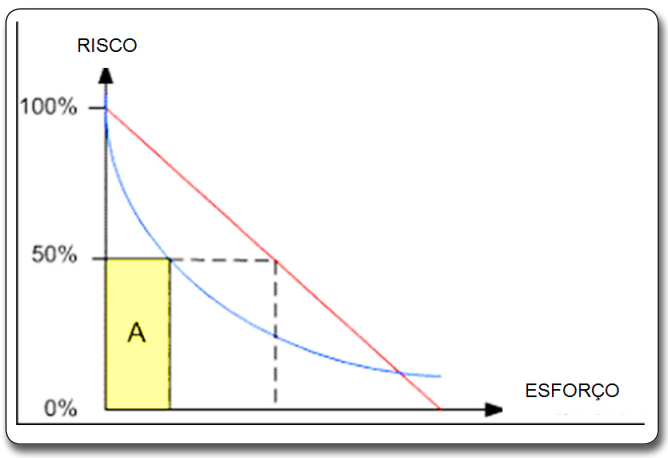


Figura 3. Esforço necessário para reduzir em 50% os riscos.

A técnica é dividida em cinco passos:

1 - Identificar funcionalidades vitais que podem previnir o usuário de utilizar o sistema se um defeito for encontrado, ou seja, um defeito com alta severidade. Uma forma eficiente de listar essas funcionalidades é a partir de pesquisas com os usuários finais do sistema, especialistas no domínio do sistema ou através de dados estátisticos de uso de versões anteriores. Uma vez que o risco aumenta com a frequencia de uso, as funcionalidades mais usadas terão maior risco.

2 - Projetar casos de teste para cada funcionalidade listada no Passo 1.

3 - Estimar tamanho (em horas ou minutos) do esforço necessário para executar os casos de teste identificados.

4 - Ordenar os casos de teste em ordem ascendente, de acordo com o esforço. Dessa forma, os casos de teste com menor esforço são executados primeiro.

5 - Iniciar a execução dos casos de teste na ordem estabelecida no Passo 4 até que o tempo termine.

A análise dos riscos é, de certa forma, fácil de ser realizada, uma vez que o usuário especifica as funções que são mais utilizadas no sistema, ou seja, as funções que permitem ao usuário realizar oitenta por cento das suas atividades.

Como a execução dos testes é baseada em esforço, os testes que gastam menos tempo são executados primeiro até que o tempo acabe. Essa técnica pode ser útil em culturas organizacionais avessas ao teste.

No entanto, esta técnica não garante que as funcionalidades

que estão sendo atacadas sejam as mais propensas a apresentar falhas.

Modelo de Processo de Teste de Software baseado em Riscos

O *RBTProcess* é um modelo de processo de teste de software baseado em riscos proposto por [Souza e Gusmão 2008], construído a partir de atividades presentes no gerenciamento de riscos e nos processos de teste de software disponíveis na literatura. O *RBTProcess* possui quatro fases distintas juntamente com seus marcos e atividades, como mostrado na **Figura 4**.

1 - Planejamento: esta fase tem como foco o planejamento inicial dos testes com base na análise dos riscos. O marco desta fase é a priorização dos requisitos através da identificação e análise dos riscos.

2 - Projeto: nesta fase, os casos de teste planejados são projetados com base na análise dos riscos. O marco desta fase é a criação de casos de teste que verificam a existência ou não dos riscos identificados.

3 - Execução: os casos de teste planejados e projetados são executados. O marco desta fase é a mitigação dos riscos identificados através da execução de casos de teste que verificam a existência ou não dos riscos identificados.

4 - Controle: os resultados da execução dos testes são coletados e controlados. Na disciplina Avaliar Testes, é verificado o conjunto de números, estratégias, tempo de execução e solicitações de mudança dos testes. Na disciplina Controlar Riscos, é realizado o acompanhamento dos riscos. Os riscos mitigados são eliminados da lista de riscos identificados e serve de entrada para o planejamento da próxima iteração. O marco desta fase é o controle dos riscos mitigados.

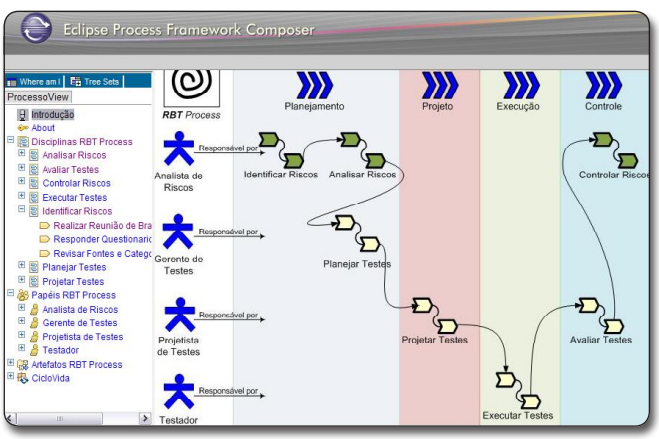


Figura 4. Página do RBTProcess [Souza e Gusmão 2008].

A seguir, uma breve descrição das disciplinas que compõem o modelo de processo:

1 - Identificar Riscos: nesta disciplina, os riscos são identificados através de questionário baseado em taxonomia de riscos (Taxonomy Based Questionnaire – TBQ). O objetivo do TBQ é diminuir o nível de abstração da atividade de identificação de riscos, uma vez que os engenheiros de teste têm conhecimento básico sobre identificação de riscos. A equipe do projeto responde a perguntas relacionadas aos requisitos, sem saber que estão identificando riscos. A **Tabela 7** apresenta um exemplo das questões contidas no TBQ para o elemento Requisito da classe Engenharia de Produto. Neste exemplo, caso o entrevistado responda "sim", indica a existência de riscos na funcionalidade ou requisito analisado.

Classe: Engenharia de Produto	
Elemento: Requisito	
Atributo:	[01] Os requisitos da funcionalidade estão mudando enquanto ela está sendo desenvolvida?
Estabilidade	Se SIM, qual parte da funcionalidade está mudando?
Atributo:	[02] Ainda existe algo para ser especificado nesta funcionalidade?
Completude	Se SIM, qual parte da funcionalidade não está especificada?

Tabela 7. Questões de estabilidade e completude para o elemento Requisito.

Após a identificação dos riscos através do TBQ, é realizada uma reunião *brainstorm* com toda a equipe de desenvolvimento a fim de validar os riscos levantados na atividade anterior. São utilizadas as listas de riscos propostas por [Bach 1999] para guiar a reunião.

O artefato de saída dessa disciplina é uma lista de riscos identificados para cada requisito.

2 - Analisar Riscos: tem como objetivo a priorização dos requisitos a serem testados com base na análise dos riscos técnicos identificados. A fórmula utilizada para priorização dos riscos foi proposta por [Amland 1999] com algumas adaptações. Para a probabilidade **P(f)**, é calculada a média dos valores atribuídos às métricas Nova Funcionalidade, Projeto/Qualidade, Tamanho, Complexidade e Dependência. Sendo que a última foi proposta neste processo e corresponde ao nível de dependência com as demais funcionalidades do software.

Os participantes desta atividade são os mesmos que realizaram a identificação dos riscos. Um guia é fornecido para atribuição dos indicadores das métricas de acordo com a atividade realizada pelo participante. Para a métrica de qualidade, por exemplo, são sugeridos os valores apresentados na **Tabela 8**. Se a pessoa que está respondendo o questionário é um desenvolvedor e não tem conhecimento da linguagem e ambiente de desenvolvimento, ela deve informar o valor "uma" para a métrica de Projeto/Qualidade.

Indicador	Desenvolvedor	Testador	Analista de requisitos
1	Não tem conhecimento das tecnologias/ferramentas utilizadas no desenvolvimento do software	Funcionalidade não apresentou defeitos nas últimas versões	Requisito está estável
2	Tem conhecimento razoável das tecnologias/ferramentas utilizadas no desenvolvimento do software	Funcionalidade apresentou alguns defeitos nas últimas versões	Requisito ou funcionalidade tem sofrido poucas alterações
3	Domina a tecnologias/ferramentas utilizadas no desenvolvimento do software	Funcionalidade apresentou muitos defeitos nas últimas versões	Requisito ou funcionalidade tem sofrido muitas alterações

Tabela 8. Exemplo de indicadores para a métrica de Projeto/Qualidade

3 - .**Planejar Testes**: tem como objetivo direcionar e controlar as atividades de teste com base na avaliação de riscos. Consiste na definição dos requisitos que serão testados ou não, tomando, como base, a análise dos riscos. A priorização dos requisitos é realizada a partir dos valores de exposição ao risco. O *RBTProcess* sugere que, a cada iteração, sejam testados, por exemplo, um conjunto de requisitos de acordo com a sua prioridade. Os requisitos classificados como de baixa prioridade somente são testados se houver tempo disponível.

Diversas estratégias podem ser utilizadas com base na análise dos riscos. Uma delas é, por exemplo, a automação dos testes para os requisitos classificados como de alta prioridade.

4 - Projetar Testes: tem como objetivo identificar e descrever os casos de teste para os requisitos a serem testados com base na análise de riscos. Consiste na identificação dos cenários, além da definição da abordagem e tipo de testes que serão utilizados. O processo sugere diferentes coberturas para os requisitos de acordo com sua importância: **i. requisitos com baixa exposição ao riscos**: testar somente os cenários relacionados aos riscos caso haja tempo disponível; **ii. requisitos com média exposição ao riscos**: testar os cenários relacionados aos riscos, fluxo principal e fluxos alterados/incluídos e **iii. requisitos com alta exposição ao riscos**: testar os cenários relacionados aos riscos e todos os fluxos do requisito (principal, exceção e alternativo).

Para cada risco identificado, testes são criados com o propósito de mitigar os fatores de riscos. O estilo de criação dos casos de testes para execução recomendado pelo processo é independente, de forma que os casos de testes possam ser executados em qualquer ordem.

5 - Executar Testes: esta é a atividade em que, de fato, os riscos são mitigados através da execução de casos de teste baseado em riscos. Tem como objetivo executar testes e verificar a corretude do software, avaliando os resultados e registrando os problemas encontrados. O testador é responsável por esta atividade e executa os testes na ordem de priorização definida pela análise dos riscos.

6 - Avaliar Testes: tem como objetivo medir quantitativamente e qualitativamente o progresso dos testes e gerar um relatório de avaliação. O projetista de testes avalia o resultado da execução dos casos de teste sem se preocupar com os riscos que foram mitigados. Esta atividade não sofre alterações para a abordagem de teste baseado em riscos, uma vez que todo o controle a acompanhamento dos riscos está definido na disciplina Controlar Riscos, sob a responsabilidade do analista de riscos.

7 - Controlar Riscos: essa disciplina é responsável pelo acompanhamento dos riscos identificados. O analista de riscos é responsável pela identificação e documentação do percentual de riscos mitigados por funcionalidade. Além da criação do relatório de avaliação dos riscos, o analista de riscos é responsável pela atualização do documento de identificação e análise dos riscos, principal entrada para a atividade de planejamento dos testes.

Em um primeiro momento, o *RBTProcess* aparece como um processo pesado mas, de acordo com os estudos realizados, as atividades da gerência de riscos de software incluídas no processo não exigem muitos esforços. Os resultados obtidos no estudo de caso forneceram uma forte indicação de que a abordagem RBT permite: concentrar os esforços de teste nos requisitos de software que possuem maior probabilidade de apresentar falhas e mostrar que os defeitos, com maior severidade, podem ser descobertos mais cedo, tendo em vista que os requisitos mais problemáticos são testados prioritariamente

Análise Comparativa

Com exceção da técnica para código-fonte orientado a objetos, todas as outras utilizam a abordagem funcional ou caixa-preta para construção dos casos de testes no nível de sistema. Apenas a técnica baseada em heurística e o *RBTProcess* propõem diferentes formas de identificação de riscos.

A técnica baseada em métricas objetiva redução do custo da fase de teste do projeto e a redução de futuros custo de manutenção do software em potencial através da otimização do processo de teste. As métricas propostas levam em consideração que funcionalidades complexas, novas, construídas com pouca qualidade e grandes (tamanho) possuem mais probabilidade de apresentar falhas. A técnica não fornece nenhuma orientação sobre a criação dos casos de teste.

A técnica baseada em riscos para teste de regressão objetiva reduzir a quantidade de casos de teste a serem executados, além de possibilitar que as áreas mais críticas sejam testadas prioritariamente. Nesta técnica, assume-se que os casos de teste já estão prontos.

A técnica baseada em uso utiliza a informação de percentual de uso da funcionalidade para priorização. Isto a torna fácil de ser utilizada. No entanto, não garante que as funcionalidades que estão sendo atacadas sejam as mais propensas a apresentarem falhas e precisem ser mais bem testadas.

A finalidade do *RBTProcess* é fornecer conhecimento e recursos necessários para que os engenheiros de teste possam utilizar a abordagem RBT em todas as atividades do teste de software, através da definição de disciplinas e papéis, e fornecimento de artefatos, guias, *templates* e métricas.

A **Tabela 9** apresenta uma análise comparativa das principais técnicas e processo apresentados neste artigo. Para a comparação, foram elencadas atividades mínimas de gerência de riscos e teste de software.

Considerações Finais

Apesar da abordagem RBT se mostrar simples, engenheiros de teste ainda encontram dificuldades em aplicá-la na prática [Goldsmith 2006], principalmente, porque a análise de riscos é algo complexo e necessita de conhecimento para sua aplicação. Não existem ainda ferramentas comerciais específicas para RBT o que, provavelmente, dificulta sua aplicação e disseminação.

TÉCNICA/ PROCESSO	IDENTIFICAR RISCOS	ANALISAR RISCOS	PLANEJAR TESTES	PROJETAR TESTES	EXECUTAR TESTES	AVALIAR TESTES	CONTROLAR RISCOS
Baseada em Heurística	Listas de critérios de qualidade, genéricas e catálogo de riscos	Equação de Barry Boehm	Não faz menção a esta atividade	Não fornece detalhes para esta atividade	Matriz de rastreabilidade	Não faz menção a esta atividade	Não fornece detalhes para esta atividade
Baseada em Métricas	Não faz menção a esta atividade	Métricas específicas para Teste de Software	Não faz menção a esta atividade	Não faz menção a esta atividade	Ordem de exposição ao risco	Não faz menção a esta atividade	Fornece um conjunto de métricas para controle de progresso dos testes
Baseada em Uso	Não faz menção a esta atividade	Utiliza a informação de percentual de uso	Não faz menção a esta atividade	Não faz menção a esta atividade	Percentual de uso da funcionalidade e tempo de execução do caso de teste	Não faz menção a esta atividade	Não fornece detalhes para esta atividade
Testes de Regressão	Não faz menção a esta atividade	Métrica proposta por Amland	Não fornece detalhes para esta atividade	Leva em consideração que os casos de teste estão prontos	Ordem de exposição ao risco	Não faz menção a esta atividade	Não fornece detalhes para esta atividade
Código Fonte Orientado a Objetos	Não faz menção a esta atividade	Métrica para código fonte OO	Não faz menção a esta atividade	Não faz menção a esta atividade	Não faz menção a esta atividade	Não faz menção a esta atividade	Não faz menção a esta atividade
RBTProcess	Questionário, listas e reunião de brainstorm	Métrica proposta por Amland com adaptações	Planeja as iterações de acordo com a exposição ao risco	Indica tipos de testes de acordo com a exposição ao risco	Ordem de exposição ao risco	Inclui atividade de avaliação de processo de teste	Conjunto de métricas para controle de progresso dos testes

Tabela 9. Análise comparativa das principais abordagens RBT.

É consenso entre os autores das técnicas apresentadas neste artigo, que a abordagem RBT permite reduzir esforços de teste e identificar funcionalidades críticas, entretanto, nenhum autor informa o percentual de esforço reduzido e a confiabilidade alcançada com a adoção da abordagem.

A constante busca pelo desenvolvimento de software com melhor qualidade e com baixo custo tem motivado inúmeras pesquisas na área engenharia de software, em especial na área de teste baseado em riscos.

Ferramentas estão em desenvolvimento [Souza e Gusmão 2008] visando auxiliar na utilização e disseminação da abordagem RBT que tem muito a contribuir na qualidade de software. ●

Dê seu feedback sobre esta edição!

A Engenharia de Software Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/esmag/feedback

Referências

[Amland 1999] Amland, S. (1999) "Risk Based Testing and Metrics: Risk analysis fundamentals and metrics for software testing including a financial application case study", 5o International Conference EuroSTAR'99.

[Bach 1999] Bach, J. (2009) "James Bach on Risk-Based Testing: How to conduct heuristic risk analysis@, Software Testing & Quality Engineering Magazine, p.23-28.

[Besson 2004] Besson, S. (2004) "A Strategy for Risk-Based Testing", disponível em <StickyMinds.com>.

[Carr et al 1993] Carr, M. J.; Konda, S.L.; Monarch, I.; Ulrich, F. C.; Walker, C. (1993) "Taxonomy Based Risk Identification", Technical Report CMU/SEI-93-TR-6. Software Engineering Institute, Carnegie Mellon University/USA.

[Chen 2002] Chen, Y. (2002) "Specification-based Regression Testing Measurement with Risk Analysis", dissertação de Mestrado, University of Ottawa/Canada.

[Goldsmith 2006] Goldsmith, R. (2006) "Early and Effective: The Perks of Risk-based Testing", Software Test and Performance Magazine, v.3, p.24-30.

[Rosenberg et al 1999] Rosenberg, L. H.; Stapko, R.; Gallo, A. (1999) "Risk-based Object Oriented Testing" 24o Annual Software Engineering Workshop, NASA SEW24.

[Souza e Gusmão 2008] Souza, E.; Gusmão C. (2008) "RBTProcess - Modelo de Processo de Teste de Software baseado em Riscos", 13º WTES - Workshop de Teses e Dissertações em Engenharia de Software, SBES'08.

Você não está mais sozinho.

Programação Orientada a Aspectos

Um exemplo prático utilizando AspectJ

Thamine Chaves Leite de Abreu
thamine.abreu@gmail.com
Atualmente cursa especialização em Desenvolvimento de Aplicações para Web no Centro de Ensino Superior de Juiz de Fora (CES/JF), Bacharel em Sistemas de Informação pela Universidade Severino Sombra (USS), Desenvolvedor de Sistemas Web na Granbery Consultoria Júnior em projeto para a Fundação COPPETEC, possui experiência de 3 anos em desenvolvimento de sistemas Java (web/desktop).

Leonardo da Silva Mota
leonardo.smota@hotmail.com
Atualmente cursa especialização em Desenvolvimento de Aplicações para Web no Centro de Ensino Superior de Juiz de Fora (CES/JF), Bacharel em Sistemas de Informação pela Universidade Severino Sombra (USS), Desenvolvedor de Sistemas Web na Granbery Consultoria Júnior em projeto para a Fundação COPPETEC, programador certificado Java (SCJP), atuou como professor assistente no curso de Sistemas de Informação da USS e dos cursos de informática da Fundação de Apoio a Escola Técnica (FAETEC), possui experiência de 3 anos em desenvolvimento de sistemas Java (web/desktop).

Marco Antônio Pereira Araújo
maraujo@devmedia.com.br
É Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, Especialista em Métodos Estatísticos Computacionais e Bacharel em Matemática com Habilitação em Informática pela UFJF, Professor dos Cursos de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora e da Faculdade Metodista Granbery, Analista de Sistemas da Prefeitura de Juiz de Fora, Editor da Engenharia de Software Magazine.

De que se trata o artigo?
Apresenta fundamentos da programação orientada a aspectos, através do AspectJ, uma extensão da linguagem Java, exemplificando seu uso e seus principais componentes através de um estudo de caso.

Para que serve?
Demonstrar a utilização da orientação a aspectos em conjunto com conceitos de Orientação a Objetos (OO), permitindo a compreensão das estruturas básicas em AspectJ e sua relação com a linguagem Java.

Em que situação o tema é útil?
Na construção de sistemas complexos, criando alternativas através da separação de interesses, que permitam maior reusabilidade e manutenibilidade do software.

A Engenharia de Software busca o aperfeiçoamento dos produtos de software, através de técnicas e métodos para facilitar sua reutilização, manutenção e evolução. Dentre as diversas práticas de desenvolvimento sugeridas, uma das mais consolidadas e eficientes, é a divisão de um sistema em partes para compreensão das fronteiras que definem suas funções. Essa divisão pode ser conhecida também como *separação de interesses (separation of concerns)*. O ideal nessa abordagem seria que toda parte relacionada a um interesse fosse

transportada para uma localização física separada das demais, que se relacionam a outros interesses, facilitando seu estudo e compreensão.

O paradigma de orientação a objetos emprega a separação de interesses através dos objetos, seus estados e operações, visando a redução da complexidade no desenvolvimento de software e aumentando sua produtividade. Embora a Orientação a Objetos (OO) dê suporte a essa divisão, ela ainda não é totalmente contemplada, devido aos interesses sistêmicos não serem tratados da forma adequada e muitas vezes se encontrarem espalhados por todo o sistema. Interesses sistêmicos são requisitos que não se aplicam à lógica de negócios como, por exemplo, segurança, desempenho, persistência, integridade de dados e tratamento de erros, dentre outros. Em decorrência disso, os sistemas OO mais complexos tendem a ter forte acoplamento, fraca coesão e grande redundância, o que dificulta sua compreensão, manutenção, evolução e reutilização. Quando um interesse se espalha por todo o sistema e se mistura entre outras funcionalidades, é denominado de interesse transversal, ou *crosscutting concern*. Como exemplo de interesses transversais pode-se citar logging, segurança, tratamento de erros, persistência de dados ou autenticação, entre outros.

A Programação Orientada a Aspectos surge com uma boa proposta para solucionar esse problema, através do encapsulamento dos *crosscutting concern* em módulos separados do restante do código. Esses módulos são denominados *aspectos*. Dessa forma, ao invés dos interesses estarem espalhados pelo código do sistema, eles são combinados nos locais desejados (ver **Figura 1**). A idéia é permitir que os objetos tratem apenas dos interesses de negócio a que são destinados sem se importar com a forma com que os interesses sistêmicos serão tratados pelos aspectos. Por isso, o ideal é que um objeto não saiba da existência de um aspecto, sendo obrigação deste acessar e tratar os interesses sistêmicos de acordo com a necessidade de um objeto.

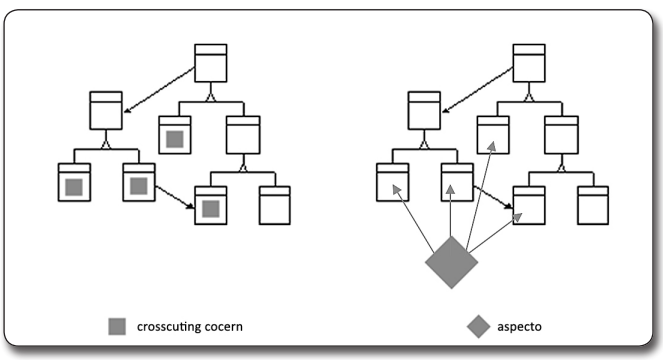


Figura 1. Diferença entre interesses transversais espalhados no sistema e a abordagem de modularização através dos aspectos.

Dentro desse contexto, este artigo tem por objetivo apresentar o desenvolvimento de sistemas utilizando AspectJ, uma extensão da linguagem Java, através da ferramenta AJDT – AspectJ Development Tool (ver seção Links) auxiliando na criação de aspectos que serão desenvolvidos através de um estudo de caso em Java, que permitirá ao leitor maior entendimento dos conceitos abordados. A plataforma utilizada para a realização do estudo de caso foi a IDE Eclipse. Para o entendimento da linguagem, faz-se necessário a compreensão dos conceitos básicos sobre o desenvolvimento orientado a aspectos, que serão explicados na seção seguinte.

Fundamentos básicos da Programação Orientada a Aspectos (POA)

Programação de Orientada a Aspectos (POA) foi criada para complementar a programação orientada a objetos através de novos conceitos e técnicas de modularização de código. Para isso, a programação orientada a aspectos envolve basicamente três etapas de desenvolvimento (ver **Figura 2**):

1 - Decomposição aspectural (*aspectual decomposition*): nesta etapa, os interesses transversais são identificados e separados dos interesses de negócio;

2 - Implementação de interesses (*concern implementation*): cada interesse identificado é programado separadamente, dando origem aos aspectos;

3 - Recomposição aspectural (*aspectual recomposition*): após a implementação dos aspectos, é realizada a integração dos aspectos com os componentes desenvolvidos, através do combinador aspectural, ou *aspect weaver*.

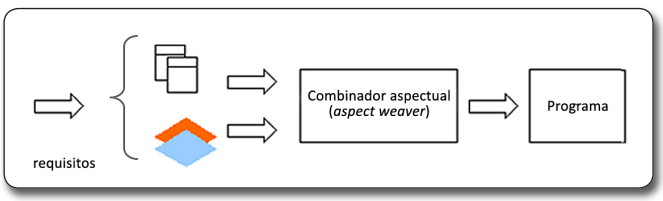


Figura 2. Etapas do processo de desenvolvimento orientado a aspectos

Um programa desenvolvido utilizando-se a POA é composto pelos seguintes elementos:

1 - Linguagem de componentes: responsável pela implementação dos interesses de negócio, sendo a linguagem base para a construção do sistema, não prevendo nada a respeito do que deve ser implementado na linguagem de aspectos.

2 - Linguagem de aspectos: responsável pela implementação de estruturas que irão definir o comportamento de um aspecto e a forma como este será executado.

3 - Combinador de aspectos (*aspect weaver*): semelhante a um compilador, porém não gera um programa compilado e sim, um novo código que combina os componentes escritos em linguagem de componentes com os escritos em linguagem de aspectos.

Introdução ao AspectJ

Para a implementação dos conceitos de POA, foi criada pela equipe da Xerox Parc, em 1997, o AspectJ (ver seção Links), que atua basicamente como uma extensão da linguagem Java. Além dos elementos oferecidos pela POO (Programação Orientada a Objetos) como classes, métodos e atributos, dentre outros, são acrescentados novos conceitos e construções ao AspectJ, apresentados a seguir:

Pontos de Junção ou *Join Points*: são pontos na execução do programa Java onde o aspecto será aplicado como, por exemplo, em chamadas de métodos, chamadas de construtores, execução de tratamento de exceções, pré-inicialização de objetos, dentre outros.

Pontos de Corte ou *Pointcut*: são construções que permitem indicar em que *join points* o aspecto irá interceptar a execução da aplicação. Ou seja, ele indica em que situações o aspecto entrará em ação. Possui a seguinte sintaxe: *pointcut <nome> (argumentos): corpo;*

Para que um *pointcut* seja definido, é necessário que haja pelo menos um designador, que seria a situação de interceptação pelo aspecto. O designador se localiza no corpo da declaração do *pointcut*. A **Tabela 1** mostra os principais designadores e suas funções.

***Advice*:** os *pointcuts* selecionam em que pontos de junção serão executadas as ações de determinado aspecto, porém quem indica esse comportamento são os *advices*. Existem três tipos de *advices*:

- **before:** executado antes da ação do *join point* que foi definido no *pointcut*. Sua sintaxe básica é: *before(): <nome do pointcut>{ <corpo> }*
- **after:** executado depois da ação do *join point* que foi definido no *pointcut*. Sua sintaxe básica é: *after(): <nome do pointcut>{ <corpo> }*

Existem ainda três variações desse advice:

- ***after():*** será sempre executado;
- ***after() returning***: só será executado caso o *join point* (normalmente um método) retorne algo, caso ele não chegue a essa linha, esse *advice* nunca será executado;
- ***after() thowing***: será executado caso o *join point* (normalmente um método) lance uma exceção.

- **around:** será executado no lugar do *join point* que foi definido no *pointcut*. Dentro desse *advice* ainda pode-se chamar a execução do *join point* original. Sua sintaxe básica é: *around():<nome do pointcut>{ <corpo> }*

Declaração entre tipos ou *Inter-Type*: são declarações de membros e classes que interferem na hierarquia e estrutura de classes do programa Java. A sintaxe para a declaração de um atributo é a seguinte:

<modificador de acesso> <tipo> <nome da classe>.<nome do atributo> = <atribuição>;

Caso os atributos ou métodos sejam declarados com modificador de acesso private, ficarão restritos ao escopo do aspecto, caso sejam declarados como public, podem interferir no programa Java, causando conflitos caso haja outros membros com o mesmo nome.

Aspectos ou *Aspects*: são unidades que modularizam os interesses transversais (*crosscutting concern*), através do encapsulamento dos pontos de junção, pontos de corte, *advices* e declarações entre tipos. Sua declaração básica é: *aspect <nome do aspecto> {…}*

Designador	Função	Exemplo
call(Signature)	Os métodos/construtores declarados em call serão interceptados sempre que forem chamados. Sua identificação é feita de acordo com a assinatura definida.	call (String Cliente.salvarDados()); interceptará toda chamada ao método salvarDados() existente na classe Cliente, que tenha como tipo de retorno uma String.
execution(Signature)	Os métodos/construtores declarados em execution serão interceptados durante sua execução. Sua identificação é feita de acordo com a assinatura definida.	execution (boolean Cliente.defineCategoria(Categoria c)); interceptará a execução do método defineCategoria(Categoria c) existente na classe Cliente, que tenha como tipo de retorno um booleano.
get(Signature)	Os atributos definidos em get serão interceptados toda vez em que forem acessados. Sua identificação é feita de acordo com a assinatura definida.	get(String Cliente.nome); interceptará os acessos à variável nome, da classe Cliente.
set(Signature)	Os atributos definidos em set serão interceptados toda vez em que forem atribuídos. Sua identificação é feita de acordo com a assinatura definida.	set(String Cliente.nome); interceptará as atribuições feitas à variável nome, da classe Cliente.
this(Type pattern)	O tipo definido em this, indica que qualquer ponto de junção em que o objeto corrente é uma instância de Type (uma classe qualquer) será interceptado. Desta forma, pode-se utilizar a referência à instância Type para disponibilizá-la a outros componentes do AspectJ.	this(cliente); interceptará os pontos de junção que definam o tipo da instância cliente.
target(Type pattern)	O tipo definido em target indica que qualquer ponto de junção em que o objeto alvo é uma instância de Type (uma classe qualquer) será interceptado. Desta forma, pode-se utilizar a referência à instância Type para disponibilizá-la a outros componentes do AspectJ.	target(cliente); interceptará os pontos de junção que definam o tipo da instância cliente.
args(Type pattern)	Qualquer declaração de argumentos que respeitem a ordem e os tipos definidos em args será interceptada.	Args(int,int); capturará no pointcut declarações que contenham dois inteiros como argumento, disponibilizando-os aos advices.
within(Type pattern)	Qualquer join point que ocorra em uma instância de Type será interceptado.	within(Cliente) interceptará qualquer join point que ocorra na classe cliente.
handler(ExceptionType)	Qualquer join point que declarar o tipo da exceção em handler será interceptado.	handler(IOException); interceptará todo join point que declare uma exceção do tipo IOException.

Tabela 1. Designadores de *pointcuts*.

Ambiente de desenvolvimento

Esta seção tem como objetivo explicar a instalação da ferramenta AJDT para o melhor entendimento da utilização de aspectos, de modo que não serão abordadas todas as funcionalidades, somente as de maior relevância. A ferramenta já possui a linguagem AspectJ integrada, portanto, basta instalá-la para iniciar um projeto utilizando aspectos.

Para instalar a AJDT, é necessário clicar na opção Help / Software Updates / Find and Install. Na janela seguinte deve-se selecionar a opção New Remote Site e, logo em seguida, será necessário nomear o plug-in que será instalado. Para o exemplo, nomeamos de AJDT e, no campo URL, deve ser colocada a URL de instalação do plug-in, conforme a **Figura 3.** De acordo com a versão do Eclipse, a URL de download pode ser diferente e, no estudo de caso, utilizou-se a versão 3.2. Para saber mais detalhes sobre outras URLs de instalação deve-se visitar o site do AJDT (seção Links).

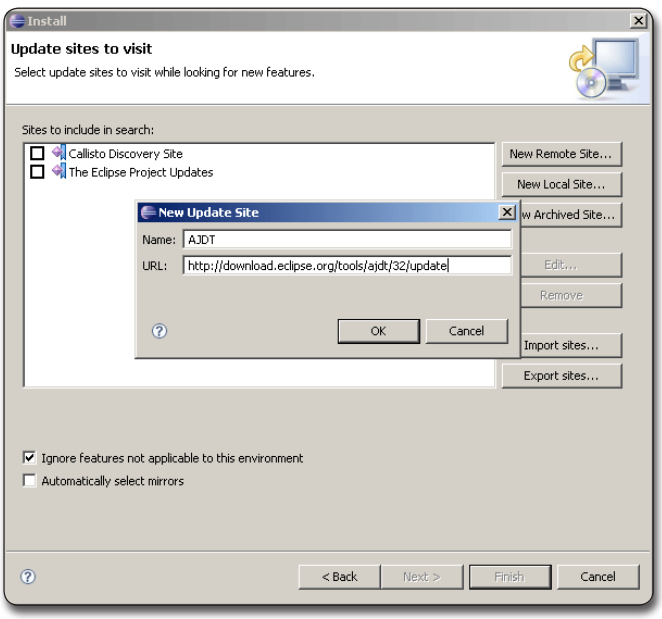


Figura 3. Instalação do plugin da AJDT

Clicando em OK, será exibida uma janela mostrando a ferramenta que será instalada. Selecione-a e clique em Next. Na janela seguinte, basta aceitar o termo de compromisso e clicar novamente em Next. Na próxima janela, deve-se clicar em Finish. A partir desse momento será efetuado o download da ferramenta. Após isso, será apresentada uma janela para a instalação do plug-in, que exibirá as informações sobre o que será instalado (ver **Figura 4**). Clique em Install All. Após completar a instalação, será exibido um aviso recomendando restartar o Eclipse e, para que a instalação tenha efeito, clique em Yes.

Estudo de caso

Utilizaremos um fragmento de código de um sistema de cadastro de clientes e trataremos de um interesse sistêmico básico, o *logging*, que será utilizado para registrar todas as possíveis exceções geradas pelo sistema.

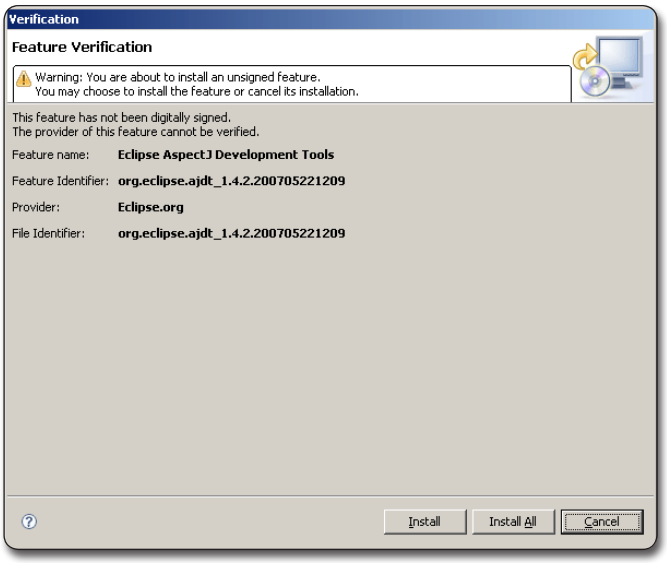


Figura 4. Informações sobre a instalação da AJDT

Para a criação de um novo projeto que utilize o AspectJ, clique com o botão direito do mouse em qualquer área branca do Package Explorer e selecione New / Project / AspectJ Project / Next. Na janela que será exibida, deve-se nomear o projeto. No estudo de caso, o nomeamos como CadastroClientes (ver **Figura 5**).

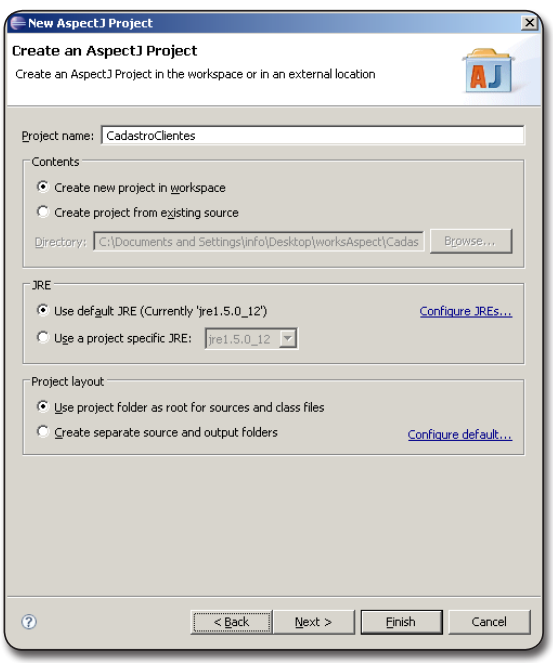


Figura 5. Janela de criação de um projeto AspectJ

Clique em Finish para finalizar a criação do Projeto. Após criado, podemos inserir pacotes e classes como se estivéssemos em um projeto Java. O diferencial é que podemos criar aspectos também. Para isso, clique com o botão direito sobre o nome do projeto ou pacote desejado e escolha a opção New / Other, localize a pasta AspectJ e, após expandi-la, escolha a opção Aspect (ver **Figura 6**).

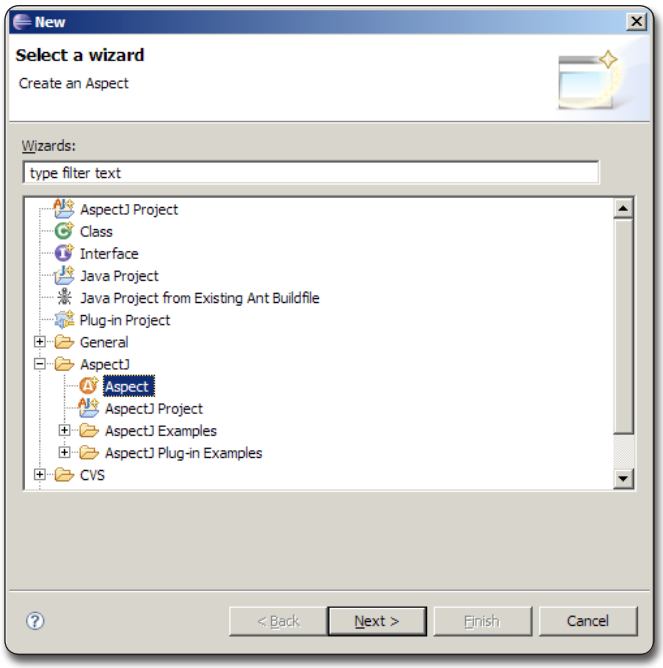


Figura 6. Criação de um aspecto.

Após selecionar esta opção, clique em Next. Na tela seguinte, será necessário nomear o aspecto. No estudo de caso o nomeamos como TratarExcecoes, clicando em Finish para finalizar sua criação. O arquivo criado tem extensão ".aj". A **Listagem 1** mostra o código fonte do aspecto, que será explicado em seguida.

O propósito desse aspecto é tratar todas as exceções geradas pelo sistema e inseri-las em um log, indicando a classe e o problema que ocorreu. Para isso, utilizamos também o log4j, uma API que permite escrever em arquivos e/ou exibir mensagens previamente configuradas no console da IDE, como auxílio à depuração de código. O log4j não é o foco do artigo, para mais informações acesse o site cujo endereço está na seção Links.

A primeira linha após as importações das classes faz a declaração do aspecto, que foi nomeado como "TratarExcecoes". É importante verificar a ausência da palavra-chave *class* responsável pela declaração de uma classe. Em seu lugar, será encontrada a declaração de um aspecto através da palavra chave *aspect*. Logo após a declaração do aspecto, declaramos também o *pointcut* no qual queremos que o aspecto incida, através das seguintes linhas extraídas da **Listagem 1**:

```
pointcut tratamentoExcecao(Exception e): handler(Exception+)
&& !within(this) && args(e);
```

Assim, apresenta a declaração do *pointcut* cujo nome é "tratamentoExcecao" e que tem como argumento um objeto do tipo Exception. Após a declaração, começamos a definir em que *join points* o nosso *pointcut* irá atuar. O designador *handler* indica que interceptará qualquer ponto do programa que declarar uma exceção do tipo Exception ou, como o indicado no caractere "+", qualquer uma de suas subclasses.

Além de *handler*, utilizamos também o designador *within* que, nesse caso, não considera as exceções declaradas dentro do próprio aspecto, indicado pelo operador de negação "!". Por fim, o *args* captura os argumentos passados como parâmetros no *pointcut*. É seu dever expor o contexto (informação capturada do *join point*, no caso a referência ao objeto Exception) desejado para que o *advice* possa usá-lo. Resumindo, a função desse *pointcut* é tratar todas as exceções que ocorram no programa, desconsiderando as exceções que possam ocorrer dentro do próprio aspecto e capturando a exceção lançada.

No estudo de caso, utilizamos alguns operadores e caracteres especiais. Caso algum deles fossem trocados, a forma como o pointcut interceptaria os *join points* seria alterada. A **Tabela 2** lista os caracteres especiais que podem ser utilizados no AspectJ.

Caractere	Função
+	Qualquer subclasse da classe informada antes do caractere
*	Qualquer seqüência de caractere, não contendo pontos
..	Qualquer seqüência de caractere, podendo conter pontos

Tabela 2. Caracteres especiais do AspectJ

Listagem 1. Aspecto TratarExcecoes

```
package aspectos;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public aspect TratarExcecoes {

        pointcut tratamentoExcecao(Exception e): handler
        (Exception+) && !within(this) && args(e);

before(Exception e): tratamentoExcecao(e){
                Object o = thisJoinPointStaticPart.
                getSourceLocation();
       getLogger(o, e);
   }

private static Logger logger = null;

public Logger getLogger(Object o, Exception excecao){
   if (logger == null) { //logger ainda nao configurado

                Properties propriedades = new Properties();
                InputStream fis = getClass().getClassLoader().
                      getResourceAsStream
                                ("log4j.properties");
        try {
           propriedades.load(fis);
        } catch (IOException e) {
         BasicConfigurator.configure()
                        ((Logger)Logger.getLogger("")).error(e);
        }
        PropertyConfigurator.configure(propriedades);
                ((Logger)Logger.getLogger("geral")).error
                ("("+o.toString()+") - " + excecao);
   }
  return logger;
       }
}
```

Após a declaração do *pointcut*, declaramos um *advice*, que irá executar uma ação especificada nele. No estudo de caso, sua função é escrever a mensagem de exceção capturada no *pointcut* "tratamentoExcecao" em um log, conforme as linhas abaixo, extraídas da **Listagem 1**:

```
before(Exception e): tratamentoExcecao(e){
Object o = thisJoinPointStaticPart.getSourceLocation();
getLogger(o, e);
}
```

O *advice* será aplicado antes da execução dos *join points*, interceptados no *pointcut*. Isso fica explícito ao utilizarmos o tipo *before*. Como o objetivo é escrever a exceção em um log e trabalhar com o *pointcut* "tratamentoExcecao", devemos declarar no *advice* o argumento declarado no *pointcut* (Exception e). Após a declaração do *advice*, deve-se mostrar a qual *pointcut* este será vinculado. Essa situação está representada no trecho "tratamentoExcecao(e)" no qual também captura a referência à exceção declarada, através da variável de referência "e".

A ação do *advice* só é inicializada na linha seguinte, onde se obtém o objeto *thisJoinPointStaticPart* para retornar a localização (através do método *getSourceLocation()*) de onde ocorreu o erro no programa de componentes e o atribui à variável de referência "o", do tipo Object. Logo abaixo o método *getLogger* é chamado, passando como parâmetro as referências aos objetos obtidos anteriormente.

No exemplo, foi utilizado um objeto reflexivo para recuperar a localização de onde a exceção ocorreu no programa Java. Reflexão é a capacidade de um programa reconhecer detalhes internos em tempo de execução que não estavam disponíveis no momento da compilação do programa. Existem três objetos que utilizam a reflexão para retornar informações sobre onde o aspecto está sendo executado. Eles podem ser chamados em qualquer lugar de um *advice*:

- ***thisJoinPoint***: contém informações dinâmicas (objetos, variáveis, etc.) sobre o *join point*.
- ***thisJoinPointStaticPart***: contém informações estáticas (nome, assinatura, etc.) sobre o *join point*.
- ***thisJoinPointEnclosingStaticPart***: contém informações estáticas (nome, assinatura, etc.) sobre o contexto que contém o *join point*. Caso o ponto de junção seja uma chamada de método, o contexto é a execução do método onde é feita a chamada.

Dentro do próprio *advice* declaramos uma variável estática "logger", do tipo Logger, que será utilizada para realizar o *logging* das exceções. Logo após essa instrução, declaramos o método "getLogger" que é o responsável pela inserção das mensagens de exceção. Este método poderia estar dentro de uma classe, porém, como o objetivo é deixar o código do programa de componentes o mais conciso possível, com apenas métodos que tratam de interesses do negócio da aplicação, ele foi inserido no próprio aspecto. Basicamente, no escopo do método um objeto do tipo InputStream é configurado de acordo com o arquivo de propriedades "log4j.properties", localizado na raiz do projeto, e a exibição da mensagem de erro é acionada. O ponto principal desse método, de acordo com a abordagem POA, é apresentada na linha abaixo, extraídas do método getLogger da **Listagem 1**:

```
((Logger)Logger.getLogger("geral")).error("("+o.toString()+")-"+
excecao);
```

As referências "o" e "excecao" foram criadas através do objeto reflexivo e da captura da mensagem de erro, respectivamente. Isso só foi possível com o uso do *advice*, que permitiu a manipulação dos objetos e a passagem de suas referências ao método getLogger(), o que tornou o código mais simples e de fácil leitura.

A seguir, na **Listagem 2**, será exibido um trecho de código de uma classe Java que pode lançar exceções. O código é bem simples e foi criado apenas a título de demonstração de como o código na linguagem de componentes se comportará. Os locais onde podem ser lançadas exceções foram destacados para facilitar a visualização do código.

Listagem 2. Método main da classe Principal.java

```
public static void main(String[] args) {
try{
   BufferedReader buf = new BufferedReader(new
            InputStreamReader(System.in));
        Integer opcao = 0;
         do{
          System.out.println("**Menu**");
          System.out.println("Escolha uma
                              opção");
          System.out.println("1 - Cadastrar
                               Cliente");
          System.out.println("2 - Sair");
          opcao = Integer.parseInt
                              (buf.readLine());
          String nome = null;
          Integer telefone= null;
          String endereco= null;

          switch(opcao){
              case 1:{
                System.out.println("Digite
                                    o nome do cliente:");
                nome = buf.readLine();
                                    System.out.println("Digite
                                     o telefone do cliente:");
                                    telefone = Integer.
                                     parseInt(buf.readLine());
                                    System.out.println("Digite o
                                     endereço do cliente:");
                                    endereco = buf.readLine();

                                    Cliente cliente = new
                                     Cliente(null, nome,
                                      endereco, telefone);
                                    ClienteDAO clienteDAO =
                                     (ClienteDAO)GenericDAO.
                                      getDAO(cliente);
                                    clienteDAO.save(cliente);
                                                          }
              }
          }while(opcao == 1);

        }catch(Exception e){

        }
   }
```

Pode-se notar que as possíveis exceções foram "tratadas" através dos blocos *try* e *catch* mas, na verdade, não há nenhum tipo de tratamento dentro do *catch*. É o aspecto TratarExcecoes que incidirá sobre essa declaração e realizará toda a ação. Nesse contexto, poderão surgir diversas dúvidas, uma delas é

o porquê devem-se declarar os blocos *try/catch* se iremos tratar todas as exceções no aspecto. Devemos declará-los, pois em vários pontos do código, chamamos métodos que são possíveis candidatos a lançar exceções e o compilador Java nos força a tratá-los ou lançá-los (através da cláusula *throws*), justamente pelo fato do programa Java não saber absolutamente nada sobre o aspecto. Ao se deparar com um método que declara lançar uma exceção (*throws*), o AspectJ não consegue capturá-la e, por isso, no exemplo foram utilizados os blocos para o "tratamento" da mesma.

Para rodar a aplicação, basta clicar com o botão direito do mouse sobre o nome do projeto, selecionar Run / Aspect/Java Application. Caso seja necessário, uma janela será exibida para a escolha da classe principal, ou seja, que contém o método *main*. Após escolhê-la, clique em Ok.

Após rodar o sistema, uma exceção pode ser forçada ao não preencher o telefone do cliente. Como o telefone é necessário para a conversão a ser realizada, o programa lança uma exceção e pára sua execução, além de adicionar a seguinte linha no log de erros: "16/01/09 10:51 ERROR (Principal.java:42) - java.lang.NumberFormatException: For input string: """". O aspecto registra essa linha através do método getLogger(), que basicamente configura e carrega as especificações do arquivo de propriedades do log4j e, através das referências "o" (que é a localização de onde foi lançada a exceção) e "exception" (que é a exceção), as informa no log, registrando-as.

Utilitários da ferramenta AJDT

A ferramenta AJDT possui alguns mecanismos para a visualização de como um determinado aspecto está incidindo sobre as classes do sistema. Para exibir o visualizador, no menu Window, escolha a opção Show View / Other. Na janela que será exibida, expanda o diretório Visualiser, e escolha as opções Visualiser e Visualiser Menu, clicando em OK (ver **Figura 7**). Na parte inferior da IDE, serão exibidos os dois visualizadores escolhidos.

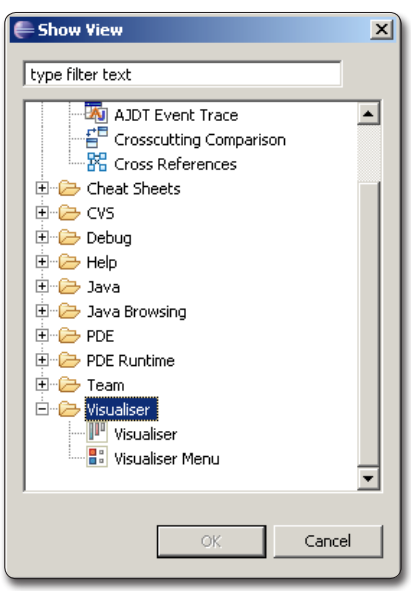


Figura 7. Visualizadores da ferramenta AJDT

Clique sobre o nome do projeto (CadastroClientes), localizado no Package Explorer, e o visualizador exibirá as classes do sistema. Para identificar as classes que podem ser interceptadas pelo aspecto, basta observar que as mesmas são exibidas em branco com uma marcação em azul, que indica a posição dos pontos de junção que foram especificados no aspecto. As classes que não serão interceptadas, ou seja, que não possuem os pontos de junção definidos, são representadas em preto (ver **Figura 8**). A linha azul indica a incidência do aspecto "TratarExcecoes".

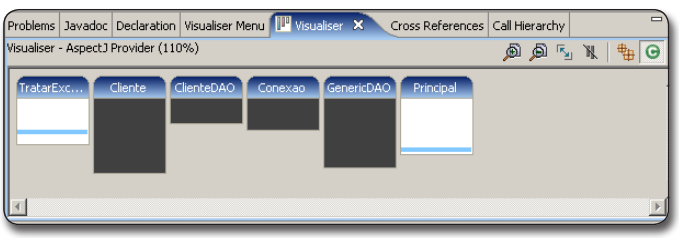


Figura 8. Visualização dos possíveis join points que tratam exceções

Como é possível observar na figura, existem apenas dois lugares que tratam exceções, e que poderiam ser capturados pelo aspecto. Um deles é na classe Principal.java e o outro no próprio aspecto, mais precisamente no método getLogger. A indicação de que há um ponto de junção declarado em determinado local não garante que o mesmo será realmente interceptado, pois isso depende das regras definidas no *pointcut*. Neste exemplo, estamos capturando todos os lugares que tratam exceções, exceto dentro do próprio aspecto, embora este esteja sendo classificado, através da figura, como sendo um local de possível interceptação.

Ainda neste visualizador, existem opções exibidas no canto superior direito:

- Zoom in : aumenta o zoom de visualização das classes afetadas.
- Zoom out : diminui o zoom de visualização das classes afetadas.
- Fit to view : expande as classes de forma que sua visualização se ajusta ao tamanho da tela.
- Limit visualization to affected bars : limita a exibição somente às classes "afetadas" pelo aspecto (ver **Figura 9**).
- Change to group view : exibe os possíveis *join points* onde o aspecto poderá interceptar, organizados por pacotes (ver **Figura 10**). Pode-se notar que as classes de cada pacote continuam sendo exibidas, só que agrupadas.
- Change to member view : exibe os possíveis *join points* onde o aspecto poderá interceptar, organizados por classes, que é a opção padrão.

O Visualiser Menu (ver **Figura 11**) exibe o nome do

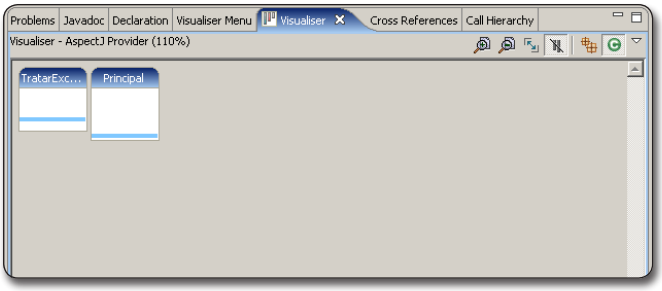


Figura 9. Visualização limit visualization to affected bars

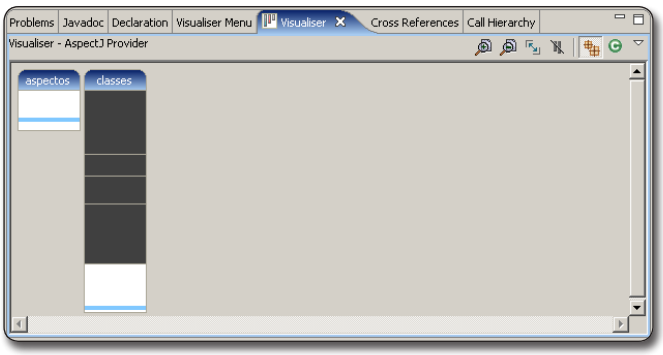


Figura 10. Visualização change to group view.

aspecto, ao lado de uma caixa de seleção de cores e um componente de seleção. Esta caixa indica a cor de identificação do aspecto quando este for exibido no Visualiser. Neste caso, utilizamos a cor padrão, que é azul. Caso o projeto tenha mais de um aspecto, cada um poderá ter uma cor que o representa para que possa ser identificado facilmente através das classes ou pacotes do projeto. Essa cor pode ser trocada, bastando-se clicar sobre a caixa de seleção de cores. A outra opção é a habilitação (ou não) da exibição do aspecto no Visualizer.

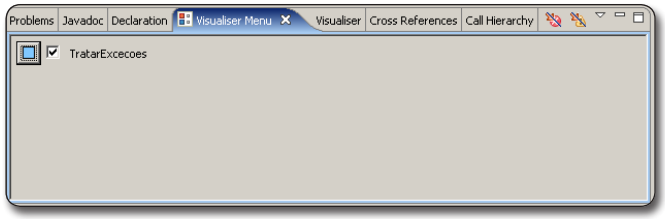


Figura 11. Visualiser Menu

Conclusão

Este artigo mostrou alguns dos principais fundamentos e características do AspectJ, uma extensão da linguagem Java que implementa a programação orientada a aspectos, visando a manutenibilidade e reusabilidade de código, através da modularização de interesses sistêmicos. O AspectJ se mostra muito eficiente, principalmente pelo fato de integrar-se aos conceitos da OO. Uma das suas principais vantagens é o fato do programa principal não saber nada a respeito dos aspectos, o que o torna independente e coeso. ●

Links

AJDT

http://www.eclipse.org/ajdt/downloads

log4j

http://logging.apache.org/log4j

Dê seu feedback sobre esta edição!

A Engenharia de Software Magazine tem que ser feita ao seu gosto.
Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/esmag/feedback

Engenharia de Software

Graduação (bacharelado) em Engenharia de Software

Fábio Nogueira de Lucena
fabio@inf.ufg.br
Professor associado do Instituto de Informática (UFG), Doutor em computação (Unicamp), PMP, Implementador do MPS.BR(r), SCJA, SCJP, SCWCD e SCMAD.

Auri Marcelo Rizzo Vincenzi
auri@inf.ufg.br
Professor adjunto do Instituto de Informática (UFG), Doutor em computação (ICMC/USP e University of Texas at Dallas) e particularmente interessado em testes de software.

Juliano Lopes de Oliveira
juliano@inf.ufg.br
Professor associado do Instituto de Informática (UFG), Doutor em computação (Unicamp), Diretor de pesquisa da Estratégia Tecnologia da Informação Ltda (empresa Avaliadora do MPS.BR) e atualmente possui intensa atuação em implementação, avaliação e melhoria de processos de software.

Plínio de Sá Leitão Júnior
plinio@inf.ufg.br
Doutor em computação.

A educação em Engenharia de Software é, senão o fator de maior influência, um dos mais relevantes na capacidade de produção de software de um país. Neste artigo é apresentada uma proposta de formação de um perfil altamente qualificado para o desenvolvimento de software tanto para o mercado local quanto global. O egresso deste curso deve adquirir uma base sólida e sustentável para uma promissora carreira na indústria de software. O curso é inovador em vários aspectos ao adotar práticas mais efetivas para a formação do egresso e surge no contexto de uma comunidade formada por quase mil empresas de TI.

De que se trata o artigo?

Proposta pedagógica de curso de graduação (bacharelado) em Engenharia de Software. Noutras palavras, organização do curso (escritório de projetos e fábrica de software), disciplinas, instrumentos didáticos e outros.

Para que serve?

Permitir que profissionais em atuação discutam e forneçam orientações acerca de como se aprender Engenharia de Software. Tais profissionais sabem o que é relevante, como dominar esta área e assim por diante. Tradicionalmente, atuais profissionais são oriundos de cursos como Ciência da Computação, Engenharia de Computação, Sistemas de Informação e outros como Matemática e engenharias, por exemplo. Em tais cursos, em geral, Engenharia de Software é tema de uma disciplina e, portanto, a área não é tratada adequadamente.

Em que situação o tema é útil?

O ensino de Engenharia de Software pode ser considerado um fator importante e de grande impacto na capacidade de produção de software de um país. Portanto, este é um tema de interesse dos atuais profissionais, agremiações de empresas produtoras de software, além de instituições de ensino interessadas em iniciativas similares.

Contexto

Software é parte de praticamente todo e qualquer sistema moderno. Em conseqüência, a formação do profissional que produz tal objeto torna-se interesse de muitos, seja por questões éticas ou profissionais. Adicionalmente, a demanda existente por mão-de-obra qualificada tem crescido mais do que a oferta, o que comumente é apresentado como motivo de preocupação para o crescimento do setor em muitos países. Em Goiás há iniciativas em curso por meio do Arranjo Produtivo Local de Software, capitaneado pelo SEBRAE-GO e pela comunidade de empresas do setor, que engloba cerca de mil empresas. O objetivo é expandir a atuação das empresas deste grupo no valioso mercado de software.

A definição e abertura do curso de Engenharia de Software contribui com estas iniciativas, e é insumo de outra que visa a elaboração de uma versão do curso a ser explorada completamente por meio da TV Digital.

O curso superior em Engenharia de Software forma bacharéis, ou seja, o egresso não é engenheiro. Embora não seja comum no Brasil, existe uma quantidade grande de cursos com este foco ao redor do mundo, onde o primeiro deles encontra-se em atividade desde 1985, no Imperial College (Londres) [1]. Houve significativo esforço para acomodar o que de melhor orientações da ACM [2] e IEEE Computer Society [3] oferecem, juntamente com normas internacionais, a análise de outros cursos similares pelo mundo, e a experiência do Instituto de Informática (responsável pelo curso) acumulada em mais de duas décadas de ensino superior e mais de uma dezena de especializações na área, além do curso de mestrado em computação. Tal instituto é uma unidade da Universidade Federal de Goiás (UFG).

No restante do texto é fornecida uma visão panorâmica do projeto pedagógico, compilada dos aspectos considerados mais relevantes e ditada pelo espaço disponível. O projeto pedagógico na íntegra pode ser obtido em [4].

Fundamentação do curso

O escopo do conhecimento abordado pelo curso baseou-se no *Software Engineering Body of Knowledge* (SWEBOK) [5]. Neste sentido, o curso não "inventa" ou exerce um perigoso papel de definir o que faz parte ou não da Engenharia de Software. Ao contrário, limita-se ao que o documento sugere. Convém ressaltar, contudo, que a forma de trabalhar este conhecimento confere personalidade distinta a cada curso.

Em [1] são apresentadas várias orientações acerca da organização de cursos de graduação em Engenharia de Software. À semelhança deste e de outros documentos empregados, não se teve a pretensão de segui-los rigorosamente, nem tampouco de negligenciá-los (aqui é suficiente esclarecer que algumas decisões se espelharam em contribuições aqui reutilizadas). Em geral, escolhas foram estabelecidas em prol do contexto onde o curso está inserido.

Da perspectiva de processo de software foi eleito o MPS.BR(r) [6], enquanto gerência de projetos adota o PMBOK [7]. Naturalmente que as fontes não são excludentes e, neste caso, a fonte mais específica foi empregada. Ou seja, embora o MPS.BR contemple gerência de projeto, este assunto foi abordado da perspectiva do PMBOK.

Os danos que podem ser causados por software e a dependência da sociedade moderna por tais objetos tornam aspectos éticos relevantes. Neste sentido, o curso adota o código de ética [8] como base para a postura esperada do egresso do curso.

Quando se pensa em mercado global é imprescindível alinhar a linguagem empregada pelo fornecedor com uma fonte respeitada internacionalmente e acessível aos clientes. Neste caso, o padrão IEEE Std 12207-2008 [9] prescreveu a língua adotada. Embora relevante, esta norma não é suficiente para atender padrões de qualidade capazes de conferir distinção entre fornecedores de software. Da mesma forma, esta norma é útil à profissionalização da engenharia de software ao mesmo tempo em que também não é suficiente para tal. Para tratar estas questões, em todas as disciplinas do curso (assunto de outra seção deste artigo), onde aplicável e disponível, normas internacionais foram adotadas. Por exemplo, a disciplina Gerência de Projetos destaca o padrão IEEE Std 1058-1998 [10] para a documentação do plano de um projeto. De forma similar, para a documentação a ser oferecida aos usuários a referência é o padrão IEEE Std 1063-2001 [11]. Estas e outras normas não devem ser equivocadamente interpretadas como restrições a serem atendidas. Ao contrário, sabe-se que qualquer aplicação inteligente da engenharia de software exige a personalização do processo de software para o projeto, para a equipe que irá executá-lo e condizente com a cultura da organização. Isto nos remete a outro padrão, IEEE Std 1074-2006 [12].

Outro componente típico da definição de cursos superiores é a perspectiva tecnológica. Neste caso, foi adotada Java e tecnologias correlatas como "eixo principal" para unir disciplinas. O objetivo é potencializar o que se aprendeu em um semestre naqueles vindouros. Por exemplo, caso se adota Java como linguagem de programação ao aprender a programar, então esta mesma linguagem será ainda mais "dominada" ao término da disciplina de Estrutura de Dados. Desta forma, ao ilustrar programação distribuída usando Java, faz-se uso de base já conhecida e oferece ao estudante a possibilidade de lidar com as dificuldades da programação distribuída sem se distrair com idiossincrasias sintáticas de outras linguagens. Dito isto, "Java e tecnologias correlatas" pode ser substituída por outra. Acreditamos, contudo, que a adoção clara de uma "trilha tecnológica" é aconselhável, e com resultados diferentes de se adotar mais de uma, o que consideramos ser o mesmo resultado quando não se adota nenhuma específica. Por fim, a escolha também deve ser compatível com o perfil dos docentes.

Estas foram as principais bases de fundamentação do curso. Estar fundamentado em fontes respeitadas, contudo, não é suficiente. A dinâmica do curso é igualmente relevante.

Curso orientado por projetos

Internalizar os conceitos da Engenharia de Software exige preparação. Para tal optou-se pela execução de projetos. Tudo no curso ocorre no âmbito de um ou mais projetos. Ou seja, o curso é tratado como uma organização baseada em projetos (não é funcional nem matricial). Isto é válido para a gestão propriamente dita do curso (administração) e para o principal serviço que oferece: ensino. Neste último caso os projetos podem ser classificados em acadêmicos (há um projeto para cada estudante do curso) e naqueles executados no âmbito da Fábrica de Software (principal laboratório do curso).

Sabemos que uma Fábrica de Software não é bem-vista por todos. Independente do significado atribuído, para o curso se resume no ambiente onde a Engenharia de Software é instanciada.

O primeiro tipo de projeto tem como objetivo a formação do estudante no tempo, no custo e com a qualidade esperados. Destes itens talvez o custo mereça pequena observação: custo de um estudante é dado pela razão da quantidade de horas que o estudante levou em disciplinas pela quantidade de horas do curso (perto de 3000 horas). Naturalmente, quanto maior a razão maior é o custo, onde o custo esperado é 1. Este custo pode ser inferior a 1.0, caso ocorram aproveitamentos.

O segundo tipo de projeto visa complementar a formação do estudante, seja por criar oportunidade de contato com assunto além do conteúdo das disciplinas ou por fomentar o exercício prático do conhecimento distribuído entre elas. Por exemplo, o desenvolvimento de uma aplicação para o iPhone pode envolver o emprego de Objective-C (apenas para ilustrar que o mundo não se resume à "trilha tecnológica" adotada nas disciplinas e discutida anteriormente).

Para gerenciar todos estes projetos é necessária uma organização distinta do que encontramos normalmente.

Organização do curso

Projetos da Fábrica de Software são definidos pelo *escritório de projetos*, que também é responsável pela gestão destes e daqueles acadêmicos (um para cada estudante). Por exemplo, é atribuição do escritório de projetos selecionar e detalhar projetos relevantes para execução no âmbito da Fábrica, bem como o acompanhamento dos projetos por meio de índices (conforme sugerido acima). Também é responsabilidade desta unidade organizacional, entre outras, a alocação de professores para ministrar disciplinas, por exemplo. Ou seja, o trabalho comumente atribuído à coordenação de um curso superior passa para o Escritório de Projeto. Coordenador do curso, docentes e funcionários são distribuídos entre estas unidades.

Estrutura curricular

Algumas disciplinas do curso ocorrem em mais de uma edição. Por exemplo, *Pacote de Trabalho* perfaz um total de 800 horas, distribuídas em 50 edições de 16 horas cada. Ou seja, o estudante terá que cumprir satisfatoriamente cinqüenta edições desta disciplina. Esta é a disciplina que permite a prática de Engenharia de Software por meio da participação em projetos, daí o nome pacote de trabalho (elemento de mais baixo nível em uma Estrutura Analítica de Projeto ou da conhecida sigla em inglês WBS). Duas outras disciplinas envolvem mais de uma edição: *Leitura de Software* e *Integração*. Cada uma destas ocorre em oito edições, ou seja, o estudante estará matriculado nestas duas disciplinas em todos os oito semestres previstos para a conclusão do curso.

As disciplinas do curso são apresentadas classificadas conforme as várias áreas que contribuem com o curso. Outras interpretações são admissíveis. Tentou-se seguir a classificação apresentada pelo SWEBOK [5], onde a Engenharia de Software possui corpo de conhecimento próprio e apoiado sobre outras áreas como Ciência da Computação e Matemática. A classificação apresentada abaixo distribui as disciplinas do curso em quatro grupos: (a) Gerência de Projeto; (b) Matemática, (c) Ciência da Computação e (d) Engenharia de Software. A exceção é a disciplina *Pacote de Trabalho*, que pode ser atribuída a qualquer uma destas quatro áreas, uma outra área ou combinação destas. Detalhes sobre esta disciplina são fornecidos adiante. Desta forma, excetuando-se a disciplina *Pacote de Trabalho*, a distribuição de percentuais por estas quatro áreas é, respectivamente: 15%, 1%, 40% e 43%, conforme ilustrado na **Figura 1**. Adicionalmente, se considerarmos uma possível distribuição das 800 horas atribuídas a *Pacote de Trabalho*, o resultado também é ilustrado na **Figura 1**, onde os percentuais relativos podem passar por mudança considerável.

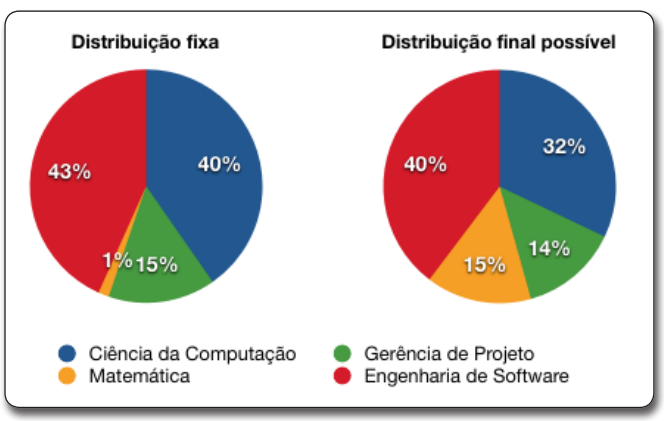


Figura 1. Percentuais de carga horária do curso por área de contribuição sem contemplar as edições da disciplina Pacote de Trabalho (gráfico da esquerda), e contemplando-as (gráfico da direita)

Atente-se para o fato de não estarem incluídas as disciplinas optativas nos gráficos acima. Algumas destas incluem *Pesquisa Operacional* e *Engenharia Econômica*. A primeira é útil no tratamento de alguns problemas. A segunda, de grande relevância para empreendimentos que visam produzir software, em particular porque tais iniciativas envolvem valores não desprezíveis.

Gerência de Projeto contribui com *Gerência de Projeto de Software* e *Integração*. Matemática contribui com *Matemática Discreta*. A Ciência da computação (área de base da Engenharia de Software) contribui com:

- Introdução à Computação;
- Introdução à Programação;
- Estrutura de Dados;
- Arquitetura de Computadores;
- Banco de Dados;
- Redes de Computadores;
- Sistema Operacional;
- Linguagens de Programação;
- Interação Homem-Computador;
- Ética, normas e postura profissional;
- Integração de Aplicações;
- Desenvolvimento de Software Concorrente;
- Desenvolvimento de Software para Persistência;
- Desenvolvimento de Software para a Web;
- Construção de Software;
- Técnicas Avançadas de Construção de Software.

As disciplinas de Engenharia de Software incluem:

- Introdução à Engenharia de Software;
- Método de Desenvolvimento de Software;
- Engenharia de Software;
- Requisitos de Software;
- Processo de Software;
- Segurança;
- Projeto Detalhado de Software;
- Arquitetura de Software;
- Gerência de Configuração de Software;
- Qualidade de Software;
- Verificação e Validação;
- Manutenção de Software;
- Métodos e Ferramentas da Engenharia de Software;
- Mercado interno e externo de software;
- Leitura de Software.

A ementa de cada uma destas disciplinas está detalhada no projeto pedagógico do curso [4]. A descrição da ementa de cada disciplina contém, além do que convencionalmente é oferecido: (a) o processo pertinente à disciplina segundo o MPS.BR [6]; (b) o mesmo para o padrão IEEE Std 12207-2008 [9] e (c) as tecnologias pertinentes à disciplina (segundo a trilha Java adotada), dentre outras informações. Três disciplinas, contudo, merecem destaque, *Integração*, *Leitura de Software* e *Pacote de Trabalho*, por não serem encontradas em cursos similares.

Integração tem como principal objetivo "manter em sintonia todo o curso", ou seja, manter docentes, discentes e as unidades organizacionais mencionadas acima "sintonizadas". Por exemplo, esta disciplina será empregada como instrumento de comunicação para distribuir informações e gerenciar expectativas. Ações como esta poderiam ser "informalmente" executadas pela coordenação do curso, o que é incompatível com a ênfase na gestão de projetos adotada no curso e carente na região, mesmo porque exige esforço não só da administração, mas também de membros de um projeto (os estudantes). Tem efeito similar à área de integração conforme o PMBOK [7]. Esta disciplina é compulsória, todo estudante estará matriculado nesta disciplina em todos os semestres do curso, assim como na disciplina Leitura de Software.

Leitura de Software, como sugere o nome, é uma disciplina onde se "lê" software. Da mesma forma que estudantes de literatura lêem clássicos, engenheiros de software também devem estar ambientados com "bons" exemplares do que eles irão produzir. Isto é comum em várias outras áreas. Por exemplo, estudantes de artes dedicam-se à observação de ensaios fotográficos. Engenheiros civis analisam plantas disponíveis de obras. Estudantes de medicina estudam laudos emitidos por especialistas. O mesmo fazem estudantes de direito sobre processos julgados. Acreditamos que benefício similar obtido por estas outras profissões também se aplica à engenharia de software.

A familiaridade com o código aberto de um produto de sucesso, por exemplo, permite que o estudante crie um referencial útil aos empreendimentos com os quais irá se envolver. Desde a formatação do código, arquitetura de software, soluções e tecnologias adotadas, até questões como a gerência de configuração e gestão do projeto, dentre outras, podem ser exploradas e contribuir com a formação do egresso. Por fim, em mais uma tentativa de esclarecer e elucidar o objetivo e justificativa desta disciplina, convém ressaltar que a produção de software nos dias atuais faz uso extensivo de padrões (padrões de projeto, padrões de análise, padrões arquiteturais e outros). Um padrão permite reutilizar conhecimento que se mostrou útil em determinado domínio, ou seja, não são exatamente produzidos, mas aproveitados de empreendimentos anteriores. Um bom engenheiro de software deve ser capaz de ler um software e perceber aspectos positivos e negativos.

Pacote de Trabalho. Esta é a disciplina com maior carga horária total: 800 horas. Em cada edição, a ementa correspondente é definida por um pacote de trabalho (pedaço do esforço de um projeto). Projeto este que pode estar em execução na Fábrica de Software ou ser o projeto acadêmico de cada estudante. No primeiro caso a ênfase está na prática da Engenharia de Software, no segundo, na complementação da formação do estudante. Estes objetivos podem estar "combinados" em um único projeto. Por exemplo, um projeto em execução na Fábrica de Software pode conter pacotes de trabalho para desenvolver, implementar e testar algoritmo paralelo para usufruir do grande número de núcleos presente nas atuais placas gráficas. Naturalmente, em tal projeto também devem constar pacotes de trabalhos dedicados a treinamento no ambiente de desenvolvimento em questão, inclusive na linguagem empregada e fundamentos

sobre o assunto. Neste sentido, a disciplina Pacote de Trabalho também é um instrumento poderoso de adaptação do curso. O exemplo também não é por acaso, mas para ilustrar que pesquisa, estudantes do mestrado e do curso podem coexistir em um mesmo projeto. Em tempo, o mesmo é válido para a extensão, pois um projeto pode ter como patrocinador uma empresa, pode ser gerenciado por uma empresa, pode ter como membros funcionários experientes de uma empresa (além de estudantes do curso). Nesta outra perspectiva, a disciplina Pacote de Trabalho também oferece mecanismo de integração entre ensino, pesquisa e extensão.

Por que tais disciplinas em ênfase?

A seção anterior exibe a concentração de esforços nas disciplinas de Engenharia de Software. Em um curso de Engenharia de Software parece natural. Convém, contudo, apresentar uma fundamentação, baseada na percepção da maturidade do contexto local (empresas de TI). As informações abaixo omitiram propositadamente valores e percentuais, mas fornecem uma noção do desenvolvimento das empresas da região conforme percebido pela academia.

Da perspectiva de pessoal, neste contexto, observa-se clara presença de dois tipos de profissionais: aqueles hábeis em aspectos de construção (baixo nível) e hábeis administradores (além do escopo deste texto). Dito de outra forma, em sua maioria, localmente encontram-se profissionais com desenvoltura em ferramentas de controle de versão, mas que praticamente desconhecem gerência de configuração. Há pessoas a frente de projetos, mas que desconhecem e não aplicam técnicas de medida de desempenho como valor agregado. Há pessoas responsáveis por garantia da qualidade, mas que a reduz a testes e assim por diante. Um indicador é a formação dos profissionais em atividade. Em pesquisa recente, um percentual pequeno era formado (qualquer que seja o curso superior).

Da perspectiva das empresas, neste contexto, observam-se iniciativas para melhoria do cenário corrente, mas ainda de resultados tímidos para a economia do estado. São poucas empresas com iniciativas de melhoria de processos de software, por exemplo. São poucas as empresas entre as maiores, conforme estudo anual da revista INFO (em geral destacam-se apenas três empresas).

Trata-se de um cenário que, respeitando exceções, tem muito a crescer. Não é preciso muito esforço para detectar que dentre quase mil empresas que produzem software, justamente Engenharia de Software é uma área pouco dominada. Todo o curso foi elaborado com base nesta conclusão. Respeitando diferenças de maturidade, a carência de profissionais em Engenharia de Software não é um problema localizado, mas atual e que ocorre em outras partes [13].

Considerações finais

Software é conhecido pelo mercado vultoso. Participar deste mercado de forma efetiva não se faz apenas com vontade, nem mesmo é suficiente uma equipe empreendedora com boas idéias. Se não houver competência, no sentido de conhecimento, habilidade e atitude, então provavelmente não haverá o sucesso. O curso de Engenharia de Software apresentado tem como visão ser um *vetor reconhecido na produção de software e no progresso regional* por meio da atuação de seus egressos. Para atingir isto o curso se concentra na competência do profissional que pretende formar. Acreditamos que esta é uma ação imprescindível para o progresso deste setor (qualquer que seja a região) e capaz de trazer frutos para a sociedade. ●

Links e referências

[1] Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, ACM e IEEE Computer Society, 2004.

[2] Association for Computing Machinery. URL: *http://www.acm.org/*.

[3] IEEE Computer Society. URL: *http://www.computer.org/*.

[4] Projeto Pedagógico do Curso Superior em Engenharia de Software. URL: *http://engenhariadesoftware.inf.br/*.

[5] Software Engineering Body of Knowledge (SWEBOK). *URL: http://www.swebok.org/*.

[6] Melhoria de Processo do Software Brasileiro (MPS.BR). URL: http://www.softex.br/mpsbr/.

[7] A Guide to the Project Management Body of Knowledge (PMBOK), 4th edition, PMI, 2008.

[8] Software Engineering Code of Ethics and Professional Practice. URL: http://www.acm.org/about/se-code.

[9] IEEE Std 12207-2008, Systems and software engineering - Software life cycle processes.

[10] IEEE Std 1058-1998, IEEE Standard for Software Project Management Plans.

[11] IEEE Std 1063-2001, IEEE Standard for Software User Documentation.

[12] IEEE Std 1074-2006, IEEE Standard for Developing a Software Project Life Cycle Process.

[13] Software Engineering Education in India: Issues and Challenges, Kirti Garg e Vasudeva Varma, 21st CSEET, 2008.