engenharia de software magazine

DevMedia group Ano II - Edição 13

Edição Especial

1 ano de Engenharia de Software Magazine

Requisitos

Conheça Algumas Soluções Concretas para Problemas Práticos da Engenharia de Requisitos

Requisitos
Conheça uma abordagem para extrair os requisitos de usabilidade de uma aplicação

Projeto
Entenda os principais conceitos e benefícios do uso da Rastreabilidade

Projeto
Aprenda a elaborar diagrama de classes – Parte II

Projeto
Desenvolvimento Orientado a Componentes: o que é e um exemplo prático

Projeto
Domain Driven Design: Fique por dentro das principais definições

Desenvolvimento
Depuração: a arte de encontrar e remover defeitos

Aulas desta edição:

- Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 1
- Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 2
- Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 3

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 2 - 13ª Edição 2009 **Impresso no Brasil**

EDITORIAL

"Atualmente muitas empresas estão se movimentando no sentido de definir detalhadamente seus processos para apoiar suas atividades de desenvolvimento. O cenário da crescente adoção de modelos de maturidade como MPS e CMMI é um indicador de que as empresas nacionais estão se preocupando com a qualidade dos serviços que oferecem, conseguindo, dessa forma, uma inserção maior no mercado internacional de desenvolvimento de software. Adicionalmente, pesquisas como a encomendada pela SOFTEX apontam diversos benefícios obtidos por empresas que investiram na melhoria de seus processos.

Os modelos de maturidade exigem a implementação de diversos processos e boas práticas da engenharia de software. Uma das atividades fundamentais da engenharia de software exigidas desde os níveis iniciais dos modelos é a engenharia de requisitos.

Existem diferentes maneiras de estruturar um processo com atividades relacionadas à engenharia de requisitos em empresas desenvolvedoras de software. Acreditamos que modelos de maturidade, como o MPS, possam servir como um arcabouço para a definição deste processo. As exigências relacionadas à engenharia de requisitos no modelo MPS podem ser encontradas no Guia Geral do MPS e maneiras de implementar estas exigências podem ser encontradas no Guia de Implementação Específico. Além disso, consideramos de fundamental importância para um processo de engenharia de requisitos que ele seja capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática."

Neste contexto, a Engenharia de Software Magazine destaca uma matéria que apresenta uma iniciativa rumo ao levantamento destas dificuldades e problemas e de maneiras de estruturar um processo para lidar com estes problemas. Uma leitura muito interessante.

Além desta matéria, esta edição traz mais seis artigos:

- Rastreabilidade: Definições Iniciais;
- Desenvolvimento Orientado a Componentes;
- Domain Driven Design;
- UML – Diagrama de Classes: Encontrando classes e desenhando seu diagrama – Parte II;
- Como extrair os requisitos de usabilidade de uma aplicação.
- Depuração: a arte de encontrar e remover bugs.

Desejamos uma ótima leitura!
Equipe Editorial Engenharia de Software Magazine

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)
Doutor 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 e Coordenador do curso de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora, Professor do curso de Bacharelado em Sistemas de Informação da Faculdade Metodista Granbery, Professor e Diretor do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Fundação Educacional D. André Arcoverde, Analista de Sistemas da Prefeitura de Juiz de Fora, Colaborador da Engenharia de Software Magazine.

Eduardo Oliveira Spínola
(eduspinola@gmail.com)
É Editor das revistas Engenharia de Software Magazine, SQL Magazine, WebMobile. 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 quinta edição, temos um conjunto de 3 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 ao lado:

Tipo: Engenharia de Requisitos
Título: Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 1
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Existem diferentes maneiras de estruturar um processo com atividades relacionadas à engenharia de requisitos em empresas desenvolvedoras de software. Modelos de maturidade, como o MPS, podem servir como um arcabouço para a definição deste processo. Além disso, é fundamental para um processo de engenharia de requisitos que ele seja capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática. Uma iniciativa rumo ao levantamento destas dificuldades e problemas e de maneiras de estruturar um processo para lidar com estes problemas será apresentada nesta série de vídeo aulas. Nesta primeira aula, serão relembradas algumas definições iniciais sobre a engenharia de requisitos.

Tipo: Engenharia de Requisitos
Título: Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 2
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Existem diferentes maneiras de estruturar um processo com atividades relacionadas à engenharia de requisitos em empresas desenvolvedoras de software. Modelos de maturidade, como o MPS, podem servir como um arcabouço para a definição deste processo. Além disso, é fundamental para um processo de engenharia de requisitos que ele seja capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática. Uma iniciativa rumo ao levantamento destas dificuldades e problemas e de maneiras de estruturar um processo para lidar com estes problemas será apresentada nesta série de vídeo aulas. Nesta segunda aula, serão apresentados um processo genérico de engenharia de requisitos e alguns desafios genéricos associados à atividade de elicitação de requisitos.

Tipo: Engenharia de Requisitos
Título: Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 3
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Existem diferentes maneiras de estruturar um processo com atividades relacionadas à engenharia de requisitos em empresas desenvolvedoras de software. Modelos de maturidade, como o MPS, podem servir como um arcabouço para a definição deste processo. Além disso, é fundamental para um processo de engenharia de requisitos que ele seja capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática. Uma iniciativa rumo ao levantamento destas dificuldades e problemas e de maneiras de estruturar um processo para lidar com estes problemas será apresentada nesta série de vídeo aulas. Nesta terceira aula, serão apresentados dois problemas práticas e duas soluções concretas associadas à atividade de elicitação de requisitos.

ÍNDICE

Você não está mais sozinho.

Soluções Concretas para Problemas Práticos da Engenharia de Requisitos

Marcelo Nascimento Costa

mnc@kalisoftware.com

Diretor de Tecnologia da Kali Software. Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ. Bacharel em Ciência da Computação pela UFPA. Especialista em CMMI, tendo participado de diversas implementações e avaliações deste modelo. Professor do curso de Ciência da Computação do Centro Universitário Metodista Bennett.

Marcos Kalinowski

mk@kalisoftware.com

Diretor Executivo da Kali Software. Doutorando e mestre em Engenharia de Software da COPPE/UFRJ. Bacharel em Ciência da Computação pela UFRJ. Consultor de implementação, instrutor, avaliador e membro da equipe técnica do MPS.BR, tendo participado de diversas avaliações deste modelo. Professor do curso de Ciência da Computação do Centro Universitário Metodista Bennett. Professor da pós-graduação e-IS Expert da UFRJ.

Rodrigo Oliveira Spínola

ros@kalisoftware.com, 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.

De que se trata o artigo?

Consideramos de fundamental importância para um processo de engenharia de requisitos que ele seja capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática. Uma iniciativa rumo ao levantamento destas dificuldades e problemas e de maneiras de estruturar um processo para lidar com estes problemas encontra-se neste artigo.

Para que serve?

Existem diferentes maneiras de estruturar um processo com atividades relacionadas à engenharia de requisitos em empresas desenvolvedoras de software. Acreditamos que modelos de maturidade, como o MPS, possam servir como um arcabouço para a definição deste processo. As exigências relacionadas à engenharia de requisitos no modelo MPS podem ser encontradas em (SOFTEX, 2007a) e maneiras de implementar estas exigências podem ser encontradas em (SOFTEX, 2007b). Este artigo complementa este material através da descrição de problemas práticos associados à engenharia de requisitos e soluções concretas para os problemas apresentados.

Em que situação o tema é útil?

O intuito deste artigo é servir como instrumento para trazer conhecimento a respeito de engenharia de requisitos para a prática, discutindo como o conhecimento na área pode ser aplicado para resolver problemas práticos reais. Desta forma, o artigo pode ainda ser utilizado como complemento ao Guia de Implementação do MPS (SOFTEX, 2007b), auxiliando organizações na estruturação de um processo de engenharia de requisitos capaz de lidar com diferentes tipos de problemas.

Atualmente muitas empresas estão se movimentando no sentido de definir detalhadamente seus processos para apoiar suas atividades de desenvolvimento. O cenário da crescente adoção de modelos de maturidade como MPS (SOFTEX, 2007a) e CMMI (SEI, 2006) é um indicador de que as empresas nacionais estão se preocupando com a qualidade dos serviços que oferecem, conseguindo, dessa forma, uma inserção maior no mercado internacional de desenvolvimento de software. Adicionalmente, pesquisas como a encomendada pela SOFTEX (Travassos e Kalinowski, 2008) apontam diversos benefícios obtidos por empresas que investiram na melhoria de seus processos.

Os modelos de maturidade exigem a implementação de diversos processos e

boas práticas da engenharia de software. Uma das atividades fundamentais da engenharia de software exigidas desde os níveis iniciais dos modelos é a engenharia de requisitos (Campos et al., 2008 – artigo publicado na edição 7 da Engenharia de Software Magazine). A importância dos requisitos e alguns conceitos iniciais relacionados à engenharia de requisitos do software são destacados em (Ávila e Spínola, 2008 – artigo publicado na edição 1 da Engenharia de Software Magazine).

Existem diferentes maneiras de estruturar um processo com atividades relacionadas à engenharia de requisitos em empresas desenvolvedoras de software. Acreditamos que modelos de maturidade, como o MPS, possam servir como um arcabouço para a definição deste processo. As exigências relacionadas à engenharia de requisitos no modelo MPS podem ser encontradas em (SOFTEX, 2007a) e maneiras de implementar estas exigências podem ser encontradas em (SOFTEX, 2007b). Além disso, consideramos de fundamental importância para um processo de engenharia de requisitos que ele seja capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática. Uma iniciativa rumo ao levantamento destas dificuldades e problemas e de maneiras de estruturar um processo para lidar com estes problemas encontra-se neste artigo.

O restante deste artigo está organizado da seguinte maneira. A seção 2 define a engenharia de requisitos e descreve suas principais atividades. A seção 3 representa um levantamento de dificuldades e problemas por atividade do processo de engenharia de requisitos, além da descrição de possíveis soluções. Por fim, a seção 4 apresenta as considerações finais deste artigo.

Engenharia de Requisitos

Podemos entender requisitos como sendo o conjunto de necessidades explicitadas pelo cliente que deverão ser atendidas para solucionar um determinado problema do negócio no qual o cliente faz parte. De forma genérica, requisitos podem ser dos seguintes tipos:

• **Requisitos funcionais.** São requisitos diretamente ligados a funcionalidade do software, descrevem as funções que o software deve executar.

• **Requisitos não funcionais.** São requisitos que expressam condições que o software deve atender ou qualidades específicas que o software deve ter. Em vez de informar o que o sistema fará, os requisitos não-funcionais colocam restrições no sistema.

• **Requisitos de domínio.** São requisitos derivados do domínio da aplicação e descrevem características do sistema e qualidades que refletem o domínio. Podem ser requisitos funcionais novos, restrições sobre requisitos existentes ou computações específicas.

O refinamento funcional dos requisitos é freqüentemente realizado utilizando a notação de casos de uso, que descrevem a interação entre entidades externas e o sistema. Mais informações sobre casos de uso podem ser encontradas em (Cockburn, 2007).

A engenharia de requisitos compreende as atividades relacionadas à produção (levantamento, registro, validação e verificação) e gerência (controle de mudanças, gerência de configuração, rastreabilidade, gerência de qualidade dos requisitos) de requisitos. A **Figura 1** representa estas atividades. Maiores detalhes sobre estas atividades podem ser obtidos em (Ávila e Spínola, 2008).

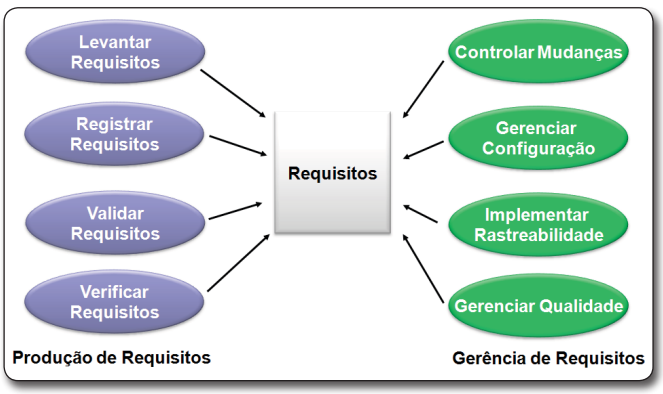


Figura 1. Atividades da Engenharia de Requisitos (Produção e Gerência)

Desta forma, os dois conceitos base (produção e gerência) devem ser considerados em conjunto ao se definir estratégias de trabalho com requisitos nas organizações (ver **Figura 2**). A produção dos requisitos fornece um conjunto inicial de requisitos para que possa ser gerenciado. A gerência de requisitos, por sua vez, assegura que mudanças nos requisitos sejam tratadas de forma adequada durante o processo de desenvolvimento. Desta maneira, ter um processo de gerência de requisitos estabelecido representa capacidade de lidar com mudanças.

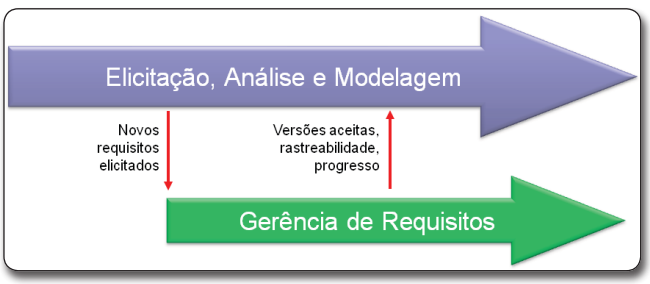


Figura 2. Produção e Gerência de Requisitos

De uma maneira geral, entre os objetivos da engenharia de requisitos destacamos (Pfleeger, 2004):

• estabelecer uma visão comum entre o cliente e a equipe de projeto em relação aos requisitos que serão atendidos pelo projeto de software;

• registrar e acompanhar requisitos ao longo de todo o processo de desenvolvimento;

• documentar e controlar os requisitos alocados para estabelecer uma baseline para uso gerencial e da engenharia de software;

• manter planos, artefatos e atividades de software consistentes com os requisitos alocados.

Para apoiar o alcance destes objetivos, é importante que se tenha um processo de engenharia de requisitos bem definido.

A **Figura 3** apresenta um modelo genérico de atividades (focado na produção de requisitos) que pode ser utilizado para descrever a maioria dos processos de engenharia de requisitos. Apesar do aparente fluxo entre as atividades, não existe uma fronteira explícita elas. Na prática existe muita sobreposição e interação entre uma atividade e outra.

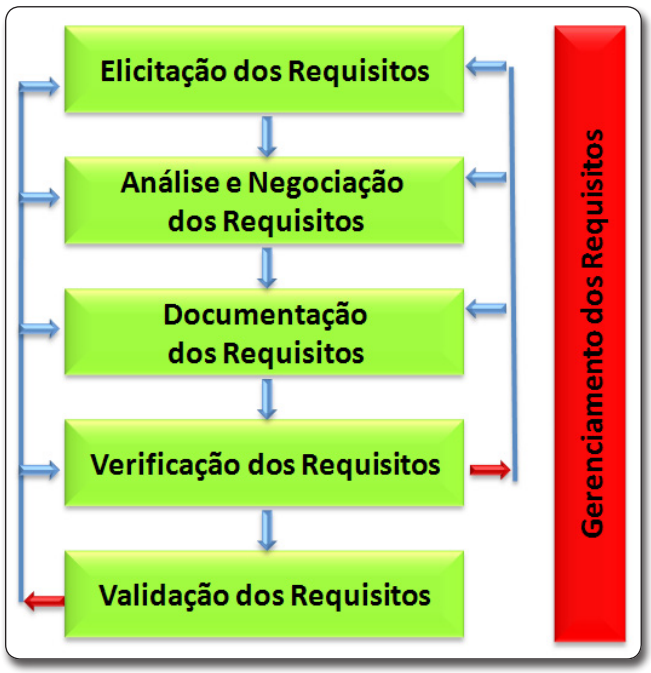


Figura 3. Processo de Engenharia de Requisitos

Sugere-se que cada organização desenvolva um processo adequado à sua realidade (Pressman, 2006). Modelos de maturidade, como o MPS, podem servir como um arcabouço para a definição deste processo. Entretanto, a prática estruturada da engenharia de requisitos, conforme definida pelos modelos de qualidade, é recente. A primeira avaliação de uma organização de software brasileira em um modelo de maturidade ocorreu há cerca de 10 anos.

Sendo uma prática recente e de fundamental importância para o desenvolvimento de software, é importante compreender seus problemas e pensar em soluções que possam ajudar na consolidação da engenharia de requisitos. Um processo de engenharia de requisitos deve ser capaz de lidar com dificuldades e problemas relacionados a requisitos que possam surgir durante o desenvolvimento de software na prática. Tendo isto em vista, a seção seguinte representa uma iniciativa rumo ao levantamento de problemas práticos e de maneiras de estruturar um processo para lidar com estes problemas.

Tratando Problemas Práticos da Engenharia de Requisitos

As subseções seguintes apresentam dificuldades e problemas agrupados por atividade da engenharia de requisitos, tendo em vista as atividades do modelo genérico de atividades descrito na **Figura 3**. Foram identificados problemas também na fase que é executada durante a execução das demais atividades da engenharia de requisitos, a gerência de requisitos. Esta lista de dificuldades e problemas foi obtida com base na experiência prática de anos de consultoria de apoio para a implantação do processo de engenharia de requisitos em empresas de

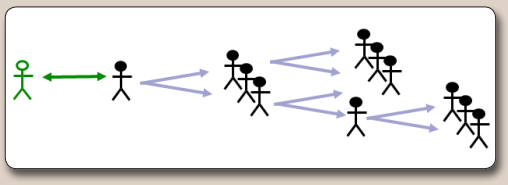
Elicitação de Requisitos	
Problema	**Discussão da Solução**
Envolver interessados inapropriados. Os envolvidos no suporte ao levantamento dos requisitos não trabalham ou não conhecem totalmente a operação do sistema. Por exemplo, a organização necessita melhorar a usabilidade (interface) do software para melhorar o tempo de atendimento dos clientes da organização. Nesta situação, o analista de requisitos precisa entrevistar os envolvidos especificamente com o operacional da aplicação e, em algumas situações, a gerência não permite que a equipe operacional se envolva na definição da solução. O problema fundamental consiste que o gerente possui uma visão apenas estratégica/tática do sistema, porém não "sofre" com os problemas específicos da usabilidade dos sistemas. Provavelmente, ele fornecerá uma solução que lhe parece mais adequada, mas que pode não resolver efetivamente o problema para qual o projeto foi definido.	Uma das maneiras de endereçar este problema é estruturar o processo de elicitação de modo a envolver uma entrevista inicial com o patrocinador (diretor, gerência) para identificar os interessados, identificando seus papéis e interesses. Adicionalmente, durante as entrevistas, pode ser interessante abordar questões como "Existem outras pessoas que poderiam responder essas perguntas?"  Uma vez identificados, os mesmos devem ser envolvidos no processo, participando de entrevistas ou de atividades como workshops de requisitos ou sessões JAD (Wiegers, 2003).
Problema	**Discussão da Solução**
Problemas políticos na organização. A organização passa por problemas políticos, incluindo a reestruturação de departamentos e papéis/responsabilidades. Nesses casos, não existe uma definição de quem vai realmente ser responsável pelo sistema. O atual responsável pela área, que utilizará o sistema, participa do processo de elicitação de requisitos, modelando o escopo do sistema de acordo com as necessidades atuais da organização. Após a aprovação desse escopo, a organização sofre a reestruturação e o novo responsável, designado para esta área, possui outra visão do negócio e solicita a redefinição completa do escopo.	Este risco pode ser mitigado com a utilização de um ciclo de vida iterativo incremental, onde nem toda a funcionalidade é definida no início do processo. Adicionalmente, para cada um dos incrementos funcionais produzidos, o processo de engenharia de requisitos deve prever a obtenção explícita do comprometimento com os requisitos definidos. A partir da obtenção deste comprometimento, os requisitos devem poder ser alterados somente através do processo de gerência de requisitos, que envolve a análise de impacto das modificações solicitadas. Dependendo do impacto, alterações contratuais podem ser necessárias.

Tabela 1. Problemas e soluções associados à elicitação de requisitos

diferentes portes e de diferentes domínios de negócios (incluindo fábricas de software), além da experiência de avaliação de empresas nos modelos de maturidade CMMI e MPS.

Os problemas apresentadas nas **Tabelas de 1 a 6** estão classificados por atividade do processo de engenharia de requisitos. Adicionalmente (**Tabelas 7 e 8**), problemas relacionados a ferramentas e recursos humanos envolvidos no processo de engenharia de requisitos foram listados. Junto à listagem de cada um dos problemas encontra-se uma discussão, com base em conhecimento acadêmico gerado a respeito de engenharia de requisitos, sobre como estruturar um processo de engenharia de requisitos para lidar com o problema.

Análise e Negociação de Requisitos	
Problema	**Discussão da Solução**
A linguagem natural e a abstração os requisitos de alto nível dificultam o mapeamento das capacidades macro em requisitos funcionais e não funcionais. Constantemente, os requisitos de usuário são descritos de uma forma muito abstrata, o que dificulta a tradução para requisitos funcionais e não-funcionais do sistema. Essa dificuldade se origina também de casos em que os usuários não têm a visão correta das necessidades do negócio que serão atendidas e qual a situação futura do sistema após a entrada em produção. Essa falta de visão pode tornar os requisitos de alto nível muito abstratos, deixando a definição de funcionalidades concretas sob responsabilidade do engenheiro de requisitos.	Para tratar esta dificuldade é necessário aumentar a interação entre o engenheiro de requisitos e o usuário. Isto pode ser atingido através de workshops de requisitos ou sessões JAD envolvendo os interessados. Outra prática que ajuda a entender bem os objetivos de negócio do cliente é iniciar as entrevistas da elicitação com perguntas abertas, do tipo: Quais são os problemas? Por que eles precisam ser resolvidos? Existem outras razões para eles serem resolvidos? Quais os benefícios esperados de uma solução bem sucedida? Como esses problemas são resolvidos hoje? Qual é a situação atual? Qual seria a situação desejada, ou seja, como você gostaria de resolver os problemas? Pode ser interessante ainda verificar a possibilidade de empregar técnicas de elicitação complementares como etnografia (observação do ambiente operacional do cliente). As definições funcionais realizadas a partir dos macro requisitos podem ser validadas junto ao usuário utilizando prototipação e revisões.
Problema	**Discussão da Solução**
Separar premissas relacionadas ao desenvolvimento do sistema dos seus requisitos. Comumente ocorre confusão entre requisitos não funcionais e premissas de desenvolvimento. Podemos definir premissas como o conjunto de condições para que o sistema possa ser desenvolvido ou possa entrar em produção. Nestes casos, é comum, por exemplo, o analista de requisitos definir as fases de desenvolvimento do sistema, o nome do usuário que deverá validar o sistema, o número de horas por cada fase, entre outros. Estes exemplos, são claramente identificados como premissas para o desenvolvimento do sistema e não podem ser classificados como requisito não-funcional de nenhum tipo.	Este problema pode ser endereçado estabelecendo templates e diretrizes que orientem a separação adequada do conteúdo do documento de requisitos e fazer uso de revisões técnicas para assegurar a qualidade dos documentos produzidos.

Tabela 2. Problemas e soluções associados à análise e negociação de requisitos

Documentação de Requisitos	
Problema	**Discussão da Solução**
Dificuldade na descrição de requisitos funcionais e não-funcionais. Os requisitos funcionais e não-funcionais (conforme definição da seção 2) devem ser claros para o entendimento do usuário e também para permitir a correta especificação dos casos de uso durante a fase de especificação de requisitos. Alguns problemas que encontramos na descrição dos requisitos funcionais: Omissão - Informação necessária não incluída; Ambigüidade – Informação passível de ter múltiplas interpretações; Inconsistência - Informações conflitantes; Fato Incorreto - Informação que não é verdadeira para as condições especificadas. Informação Estranha - Informação desnecessária.	Um instrumento que pode ser utilizado para encontrar problemas na descrição dos requisitos funcionais é a utilização de revisões técnicas formais. Entre as revisões técnicas e formais destacamos as inspeções e os walkthroughs. Um dos benefícios das inspeções é a disseminação das boas práticas na organização. Adicionalmente, treinamento pode ser conduzido focando nos problemas encontrados nas inspeções. Mais informações sobre inspeções de requisitos podem ser encontradas em (Kalinowski et al., 2007).
Problema	**Discussão da Solução**
Complexidade no detalhamento dos casos de uso para definição da solução. Constantemente, os casos de uso ficam extremamente complexos, com, por exemplo, 30 passos no fluxo principal, 5 fluxos alternativos, 5 fluxos de exceção e dezenas de regras de negócios. Nestes casos, a leitura do caso de uso se torna bastante cansativa e difícil para os usuários e, até mesmo, para os envolvidos na área de TI da organização.	Este problema pode ser de difícil solução, uma vez que a complexidade pode ser inerente à solução sendo descrita. Uma recomendação para facilitar a compreensão do caso de uso é manter os passos simplificados, contudo assegurando que todas as informações necessárias (informações recebidas, opções disponibilizadas, informações fornecidas e ações realizadas) estejam descritas. A separação do caso de uso em diversos casos de uso relacionados através de inclusões (includes) e extensões (extends) pode ser pensada, desde que os casos de uso resultantes atendam a um objetivo facilmente compreendido (Cockburn, 2007).

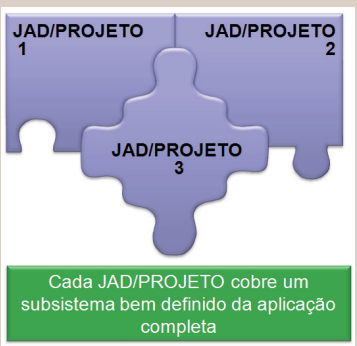
Problema	Discussão da Solução
Falta de padronização no detalhamento de casos de uso. A falta de padronização dentro de uma mesma organização leva a dificuldades no entendimento das soluções e na garantia da qualidade da especificação de requisitos produzida.	Novamente a solução envolve estabelecer um padrão dentro da organização, definindo explicitamente um template e diretrizes de preenchimento. Adicionalmente é interessante estabelecer um processo de garantia da qualidade para assegurar que os artefatos produzidos na engenharia de requisitos sigam corretamente os padrões estabelecidos.
Problema	**Discussão da Solução**
Alteração constante dos requisitos, causando retrabalho. Modificações em alguns requisitos podem causar grande impacto sobre o esforço de desenvolvimento do software. Por exemplo, a alteração de um requisito não-funcional relacionado ao desempenho do produto (tempo de resposta), pode tornar toda uma arquitetura de solução (por exemplo, em camadas) inadequada.	Em um cenário de constantes mudanças a utilização de um ciclo de vida iterativo incremental, onde nem toda a funcionalidade é definida no início do processo, pode se mostrar mais adequada. Adicionalmente, para cada um dos incrementos funcionais produzidos, o processo de engenharia de requisitos deve prever a obtenção explícita do comprometimento com os requisitos definidos. A partir da obtenção deste comprometimento, os requisitos devem poder ser alterados somente através do processo de gerência de requisitos, que envolve a análise de impacto das modificações solicitadas. A análise de impacto pode ser facilitada caso a rastreabilidade entre requisitos e os demais produtos de trabalho possa ser identificada. Dependendo do impacto, alterações contratuais podem ser necessárias.
Problema	**Discussão da Solução**
Complexidade na definição da solução funcional. Comumente documentos de requisitos possuem diversos casos de uso complexos. Alguns documentos atingem o tamanho de centenas de páginas, dificultando o detalhamento e entendimento de todos os requisitos necessários para o projeto.	Quebrar o sistema em módulos ou conjuntos de caso de uso que possam ser produzidos em iterações separadas, seguindo um ciclo de vida iterativo incremental. As especificações destes módulos devem ser tratadas em documentos separados e conseqüentemente menores.
Problema	**Discussão da Solução**
Detalhamento técnico desnecessário durante a especificação funcional do sistema. Uma parte considerável dos analistas de requisitos veio de equipes de desenvolvimento e, invariavelmente, utilizam termos técnicos (exemplos: formatos de arquivos XML, conversões de dados, comunicação pela rede, etc.) para a especificação dos requisitos. A especificação funcional deve se limitar a descrever o que o sistema realiza e não como (Wiegers, 2003).	Um instrumento que pode ser utilizado para encontrar este tipo de problema na especificação funcional é a utilização de revisões técnicas formais. Adicionalmente, treinamento pode ser fornecido para esclarecer o conteúdo apropriado da especificação funcional.
Problema	**Discussão da Solução**
Disponibilidade limitada para a realização de sessões JAD/workshops de requisitos. Em projetos muito complexos o número interessados pode ser muito grande e difícil de gerenciar. Por exemplo, para uma organização pode se tornar praticamente inviável reunir diversos interessados durante um período muito grande para definição de requisitos na sessão JAD.	Dividir o projeto em módulos (por exemplo, módulo financeiro, módulo de RH, módulo do setor de registro, módulo de materiais, etc.) que possam ser tratados em sessões JAD separadas com diferentes interessados em cada uma das sessões.  Desta forma, um número menor de interessados é necessário em cada uma das sessões.

Tabela 3. Problemas e soluções associados à documentação de requisitos

Verificação de Requisitos	
Problema	**Discussão da Solução**
Requisito não funcionais não verificáveis. Alguns requisitos não funcionais (desempenho, confiabilidade, portabilidade, usabilidade, etc.) são freqüentemente descritos de forma que não possam ser verificados durante os testes. Por exemplo, como testar que um software deve ser rápido, que uma interface deve ser amigável, que um software é confiável, ou que um software deve possuir portabilidade.	Devido à sua própria definição, requisitos não-funcionais devem ser mensuráveis. Assim, deve-se associar forma de medida/referência a cada requisito não-funcional elicitado. A organização pode disponibilizar exemplos de unidades de medida para cada tipo de requisito não funcional. Alguns destes exemplos seguem: Desempenho: tempo de resposta em segundos para uma dada configuração de ambiente, número de transações processadas por segundo, etc. Usabilidade: tempo de treinamento necessário, número de quadros de ajuda, etc. Confiabilidade: tempo médio entre falhas, taxa de ocorrência de falhas, etc. Portabilidade: sistemas operacionais destino, browsers para visualização, etc. As revisões técnicas formais devem assegurar que os requisitos não funcionais estejam descritos de forma verificável.

Tabela 4. Problemas e soluções associados à verificação de requisitos.

Validação de Requisitos	
Problema	**Discussão da Solução**
Dificuldades para validação de casos de uso por parte do usuário. O usuário comum (não envolvido com a área de tecnologia da informação) pode possuir dificuldade no entendimento da semântica da notação de herança, pois não este não consegue abstrair o comportamento comum e as diferenças entre os casos de uso pai e filho. As duas outras notações semânticas includes e extends complicam a validação pela necessidade de navegação entre os casos de uso e depois o retorno para o caso de uso que chamou o caso estendido ou incluído. Algumas situações existem mais de três níveis de chamada de casos de uso, ou seja, o caso de uso X inclui o Y, que inclui o Z e que, finalmente, chama o W. Essa estratégia dificulta a compreensão do detalhamento do caso de uso.	Esta dificuldade é inerente da descrição funcional através de casos de uso. O que pode amenizar este problema é evitar o uso demasiado de heranças entre casos de uso e treinar o usuário na leitura de documentos que envolvam descrições de casos de uso. A visualização da solução descrita nos casos de uso pode ainda ser facilitada através do uso de protótipos de telas. Outra prática interessante para a validação, quando o usuário tem dificuldade na leitura dos documentos, é a utilização de walkthroughs (Melo et al., 2007) em que uma apresentação a respeito do conteúdo do documento é realizada (muitos vezes pelo próprio autor do documento), explicando os diferentes casos de uso. Outra prática interessante neste caso seria o uso de protótipos para apoiar o entendimento dos requisitos por parte do usuário.
Problema	**Discussão da Solução**
Documentos funcionais grandes dificultam a validação. Como foi citado acima, um documento funcional muito extenso e complexo dificulta a validação da definição funcional, principalmente por parte do usuário, dificultando o entendimento da solução funcional completa. Neste cenário, muitas vezes é possível perceber que a validação por parte do usuário ocorreu de forma mais intensa nas primeiras páginas do documento.	O problema pode ser endereçado realizando mais de uma revisão de validação do documento, selecionando amostras do documento envolvendo apenas alguns dos casos de uso. Neste caso é importante identificar os casos de uso que possuem relacionamentos entre si e que devem ser validados em conjunto. Pode ser interessante ainda acrescentar protótipos no documento de especificação funcional. Protótipos tendem a fornecer uma percepção mais clara da funcionalidade a ser desenvolvida e normalmente recebem mais atenção na validação por parte do usuário do que descrições de casos de uso.

Tabela 5. Problemas e soluções associados à validação de requisitos.

Gerência de Requisitos	
Problema	**Discussão da Solução**
Dificuldades de estabelecer uma estratégia para a atualização e reutilização de casos de uso. Ter uma estratégia para reutilização de casos de uso pode ser importante, por exemplo, para a fase de manutenção em que novos casos de uso podem ser definidos reutilizando descrições anteriores. Não é comum a organização se preparar dessa maneira para a implantação do processo de gerência de requisitos, o que dificulta o armazenamento e atualização/reutilização dos casos de uso.	Para tratar este problema seria necessário criar uma biblioteca de casos de uso estruturada de acordo a divisão funcional (módulos) dos sistemas da organização e, preferencialmente, considerar cada caso de uso como um documento separado. Este documento poderia ser tratado como um item de configuração para armazenamento em um sistema de controle de versões.
Problema	**Discussão da Solução**
Como representar a atualização de casos de uso? Qual estratégia escolher para representar qual (is) passo(s) e regra(s) de negócio(s) foram alterados e qual tipo de solicitação de mudança que causou essa alteração no caso do uso. Desta forma, mantém-se um histórico da evolução do caso de uso.	Uma maneira de destacar as atualizações realizadas nos casos de uso é o preenchimento de um histórico de versões no início da descrição de cada caso de uso, informando a data, as alterações realizadas e o responsável pelas alterações. Adicionalmente, se o caso de uso é tratado como um item de configuração separado em um sistema de controle de versões, as alterações podem ser informadas no momento da atualização do repositório deste sistema.

Problema	Discussão da Solução
Dificuldade de integração das práticas de gerência de requisitos com gerência de configuração. Um problema comum é a falta de disciplina dos analistas de requisitos para trabalhar integrado com as práticas da gerência de configuração.	Para tratar este problema é necessário definir um processo de gerência de configuração. A gerência de configuração envolve o controle de versões e a gerência de solicitações de mudança. Mais informações sobre gerência de configuração podem ser encontradas em (Dantas, 2008 – artigo publicado na edição 2 da Engenharia de Software Magazine). Neste processo a evolução dos requisitos deve estar alinhada com a evolução das baselines de requisitos, ou seja, deve-se conseguir identificar quais solicitações de mudança obrigaram a alterar o documento funcional já aprovado e validado pelos stakeholders. Os requisitos acordados em uma baseline somente podem ser alterados através de solicitações de mudança.
Problema	**Discussão da Solução**
Trabalhar com o backlog de casos de uso inexistentes para sistemas em manutenção. Muitos sistemas começam a ser definidos funcionalmente através de casos de uso depois de um certo tempo de entrada em produção. Nesta situação, não existem casos de uso descrevendo o sistema, então, quando surgem demandas evolutivas (pequenas alterações no sistema), acaba sendo necessária a descrição inteira dos casos de uso para se conseguir fornecer um contexto para os stakeholders que validarão os casos de uso.	Neste caso, esforço deve ser investido na engenharia reversa dos casos de uso (documentação dos casos de uso a partir dos sistemas existentes). Para sistemas muito grandes pode ser interessante considerar uma abordagem evolutiva para a criação dos casos de uso, onde inicialmente a documentação consiste de um resumo do caso de uso (contendo, por exemplo, apenas o fluxo principal). Casos de uso específicos podem ser refinados no momento em que uma manutenção na funcionalidade que eles descrevem for realizada.
Problema	**Discussão da Solução**
Dificuldades de implantação e manutenção da rastreabilidade dos requisitos. Muitas vezes a implantação na prática de uma estratégia para estabelecer a rastreabilidade não é trivial. Muitas ferramentas são pouco flexíveis e adaptáveis e planilhas acabam sendo usadas freqüentemente. Assim, o esforço para criação da rastreabilidade pode ser alto.	Uma maneira de reduzir o esforço é estabelecer um processo de desenvolvimento que integre as atividades relacionadas ao estabelecimento da rastreabilidade com as próprias atividades de engenharia do software. Planilhas podem ser usadas, mas dificultam essa integração. Algumas ferramentas CASE permitem o estabelecimento dos links de rastreabilidade no momento da criação e atualização dos artefatos.
Problema	**Discussão da Solução**
Dificuldades de estabelecimento retroativo de rastreabilidade dos requisitos. Como criar uma matriz de rastreabilidade para sistemas pré-existentes em que a rastreabilidade não foi considerada durante o desenvolvimento.	O estabelecimento retroativo de rastreabilidade pode ser extremamente custoso e uma análise de custo/benefício deve ser considerada antes de realizar esta tarefa. Esta análise deve levar em consideração o volume de requisição de mudança no software e o impacto das modificações. Caso este volume não seja grande pode não ser interessante investir este esforço. Na medida em que manutenções forem realizadas a rastreabilidade referente a estas manutenções pode ser estabelecida.

Tabela 6. Problemas e soluções associados à gerência de requisitos

Ferramentas	
Problema	**Discussão da Solução**
Custo das ferramentas para gerência de requisitos. As ferramentas para suportar o processo de engenharia de requisitos possuem um custo considerável.	Para compensar o custo da ferramenta, o processo de engenharia de requisitos deve ser muito bem definido e compatível com a ferramenta. Adicionalmente, os recursos devem ser bem treinados e devem possuir um suporte adequado. É importante ressaltar que um processo de engenharia de requisitos pode ser estabelecido utilizando somente ferramentas gratuitas (editores, planilhas, sistemas de controle de versão, sistemas de registro de solicitações de mudança, etc). Tendo isto em vista, é necessário conhecer explicitamente as vantagens obtidas com a ferramenta a ser adquirida e realizar uma análise de custo/benefício.

Tabela 7. Problemas e soluções associados a ferramentas

Recursos Humanos	
Problema	**Discussão da Solução**
Falta de conhecimento dos analistas de requisitos em um domínio específico do problema. O domínio de aplicação pode se mostrar complexo, como por exemplo, financeiro, seguros e telecomunicações. Isto causa impacto na área de tecnologia de informação, pois idealmente o analista de requisitos deveria possuir um conhecimento de todas as áreas de domínio para poder detalhar corretamente as regras de negócio pertinentes ao domínio da aplicação. Caso não tenha o conhecimento desse domínio, o analista de requisitos poderá gerar um documento funcional com regras de negócios incorretas ou com um nível de abstração funcional muito alto.	O processo de elicitação deve prever esforço dedicado à compreensão do domínio. Este esforço deve ser explicitado tanto para o cliente quanto para a equipe do projeto. No caso de projetos em domínios muito complexos e específicos o envolvimento com os clientes deve ser intensificado e pode ser interessante considerar a contratação de suporte constante de consultores externos, especialistas no domínio.

Tabela 8. Problemas e soluções associados a recursos humanos.

Nesta seção apresentamos problemas e soluções relacionados a atividades de engenharia de requisitos. Entretanto, é importante destacar que a identificação dos problemas que precisam ser tratados pode não ser trivial. Uma maneira de encontrar problemas que permitam a melhoria do processo é a análise causal de defeitos de software (no caso da engenharia de requisitos seria a análise dos defeitos relacionados aos artefatos gerados em suas atividades). Mais informações sobre análise causal de defeitos de software podem ser encontradas em (Kalinowski e Costa, 2008 – artigo publicado na edição 3 da Engenharia de Software Magazine).

Conclusão

A engenharia de requisitos é considerada nos principais modelos de maturidade e representa uma das atividades fundamentais da engenharia de software. Estabelecer um processo de engenharia de requisitos capacita a empresa a lidar com mudanças durante o desenvolvimento de software. A importância da engenharia de requisitos é destacada em (Ávila e Spínola, 2008).

Neste artigo fornecemos uma breve descrição da engenharia de requisitos e de suas atividades, listamos problemas associados a cada uma destas atividades e discutimos soluções para estes problemas. A lista de problemas foi obtida com base na experiência prática de anos de consultoria de apoio para a implantação do processo de engenharia de requisitos em empresas de diferentes portes e de diferentes domínios de negócios, além da experiência de avaliação de empresas nos modelos de maturidade CMMI e MPS. A discussão das soluções, por sua vez, é pautada em conhecimento gerado a respeito de engenharia de requisitos através de pesquisa na área.

Assim, o intuito deste artigo é servir como instrumento para trazer conhecimento a respeito de engenharia de requisitos para a prática, discutindo como o conhecimento na área pode ser aplicado para resolver problemas práticos reais. Desta forma, o artigo pode ainda ser utilizado como complemento ao Guia de Implementação do MPS (SOFTEX, 2007b), auxiliando organizações na estruturação de um processo de engenharia de requisitos capaz de lidar com diferentes tipos de problemas. ●

Referências

Ávila, A.L., Spínola, R.O., "Introdução à Engenharia de Requisitos", Engenharia de Software Magazine, Ano:1 Edição:1, ISSN 1983127-7, 2008.

Campos, A.C.C., Munhão, A.C.B., Spínola, R.O., Kalinowski, M., "Apoiando a Implementação do Modelo de Maturidade MPS Nível G", Engenharia de Software Magazine, Ano:1 Edição:7, ISSN 1983127-7, 2008.

Cockburn, A., "Escrevendo Casos de Uso Eficazes", Editora Artmed, ISBN: 9788536304571, 2007.
Dantas, C., "Gerência de Configuração", Engenharia de Software Magazine, Ano:1 Edição:2, ISSN 1983127-7, 2008.

Kalinowski, M., Costa, M.N., "Melhorando Processos de Software através de Análise Causal de Defeitos", Engenharia de Software Magazine, Ano:1 Edição:3, ISSN 1983127-7, 2008.

Kalinowski, M., Spínola, R. O., Dias-Neto, A.C., Bott, A., Travassos, G.H., "Inspeções de Requisitos de Software em Desenvolvimento Incremental: Uma Experiência Prática", VI Simpósio Brasileiro de Qualidade de Software (SBQS), Porto de Galinhas, Brasil, 2007.

Melo, W., Travassos, G.H., Shull, F., 2001, "Software Review Guidelines", Technical Report ES – 556/01 – COPPE/UFRJ, August 2001.

Pressman, R. S., "Engenharia de Software: Uma Abordagem Prática", 6 ed., São Paulo: McGraw-Hill, ISBN: 8586804576, 2006.

Pfleeger, S.L., "Engenharia de Software: Teoria e Prática", 2a Ed., Prentice Hall, ISBN: 8587918311, 2004.

SEI, CMMI for Development (CMMI-DEV), Version 1.2, Technical report CMU/SEI-2006-TR-008. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

SOFTEX. MPS.BR – Guia Geral, versão 1.2, Disponível em: http://www.softex.br/mpsbr, junho 2007a.

SOFTEX. MPS.BR – Guia de Implementação – Parte 1 Nível G, versão 1.1, Disponível em: www.softex.br/mpsbr, junho 2007b.

Travassos, G.H., Kalinowski, M., "iMPS: Resultados de Desempenho de Organizações que Adotaram o MPS". 1. ed. Campinas: Associação para Promoção da Excelência do Software Brasileiro - SOFTEX, ISBN 978-85-99334-11-9, 2008.

Wiegers, K.E., "Software Requirements", 2nd Edition, Microsoft Press, 2003.

Como extrair os requisitos de usabilidade de uma aplicação

Artefatos de software que devem ser utilizados durante a extração dos requisitos de usabilidade de um sistema

Rodrigo S. Prudente de Aquino
rodrigo@wpage.com.br
Bacharel em Ciência da Computação pela PUC-SP e MBA em Engenharia de Software pela USP. Foi analista de sistema na Petrobras e trabalhou como Gerente de Tecnologia Web em uma das maiores agências de marketing direto do Brasil. Escritor de artigos e palestrante em universidades, Rodrigo S. Prudente de Aquino é autor do livro WPage – Padronizando o desenvolvimento de Web Sites (www.wpage.com.br) distribuído em todo Brasil. É também um dos responsáveis pela criação e modificação de várias regras heurísticas de usabilidade utilizadas pela USP para analisar Web Sites.

De que se trata o artigo?
Neste artigo veremos como extrair os requisitos de usabilidade de uma aplicação, além de apresentar alguns artefatos de forma a agilizar a extração facilitando o trabalho do engenheiro de usabilidade.

Para que serve?
Apresentar como é importante desenvolver sistemas com boa usabilidade e mostrar como esse fator é um diferencial em diversos tipos de aplicações nos dias de hoje.

Em que situação o tema é útil?
Na extração de requisitos de usabilidade para desenvolvimento de quaisquer sistemas do tipo Client/Servidor ou Web. Além disso, o tema visa conscientizar as empresas de desenvolvimento que fornecer aplicações com melhor usabilidade é um diferencial no mercado.

O conceito de usabilidade surgiu da necessidade de facilitar o manuseio de aplicações complexas, como sistemas para foguetes, aviação, trens, usinas, etc., proporcionando maior segurança para a vida humana. Com o passar do tempo o conceito foi difundido em outras áreas, como aplicações Client/Server e Web Sites, visando melhorar as características dos produtos tornando-os mais rentáveis. Mas como identificar que um software é mais usual do que outro? Neste artigo, será explicado como extrair os requisitos de usabilidade para o desenvolvimento de sistemas, obtendo informações objetivas e relevantes de como melhorar o manuseio de um produto, aumentando conseqüentemente sua rentabilidade.

Para extrair os requisitos de usabilidade de um sistema é necessário estudar quatro fatores responsáveis em direcionar o entendimento do desenvolvedor no momento da concepção do produto. Esses fatores podem ser expressos por meio de artefatos que deverão explicar sobre o perfil do usuário, a análise de tarefas, a plataforma em que o produto funcionará e os princípios gerais de design. Antes de explicar esses artefatos, será feito um overview sobre cada um deles:

• **Perfil do usuário:** artefato indicando um estudo do público alvo que utilizará o software. As informações relevantes servirão para modificar a concepção do produto;
• **Análise de tarefas:** artefato mostrando quais as ações mais críticas (importantes) o usuário fará no sistema. Em um Web Site de E-commerce, por exemplo, uma das tarefas mais críticas seria o cadastro do usuário e todo o ciclo de abertura e fechamento de pedido;
• **Plataforma utilizada:** artefato que descreve as características da plataforma em que o sistema funcionará. Conhecer bem a plataforma é um fator muito importante de forma a identificar quais recursos devem ser ou não utilizados dela;
• **Princípios gerais de design:** artefato que avalia o resultado baseado em estudos anteriores que descrevem o design do projeto, formatos, cores, etc. Através deste artefato tem-se um refinamento da solução, atacando realmente o que interessa.

Perfil do usuário

Não existe o melhor tipo de interface para todos os usuários de um sistema. Algumas interfaces podem otimizar o manuseio de uma aplicação para alguns usuários, como podem tornar um sistema extremamente complicado para outros. Por esta razão é necessário realizar um estudo sobre o perfil dos usuários que mais utilizarão a aplicação com o objetivo de que ela fique mais usual para a maioria dos "clientes". O resultado do estudo sobre o perfil dos usuários é uma lista de requisitos, podendo ser segmentada por categoria (profissão), que servirá para direcionar a construção da interface.

Para iniciar o estudo, recomenda-se que você avalie as características mais comuns entre os usuários da aplicação e em seguida faça alguns refinamentos levando em consideração;
• Características fisiológicas (atitude, motivação);
• Conhecimento e experiência (habilidades específicas);
• Características de trabalho (freqüência de uso);
• Características físicas (deficientes visuais, daltônicos, etc.).

Essas características podem ser determinadas por meio de entrevistas e/ou questionários. No caso de sistemas desenvolvidos para utilização interna da companhia, várias informações podem ser obtidas através do RH da própria empresa. No caso de sistemas que serão utilizados pelos clientes de uma companhia, entrevistas e ações de marketing podem ser realizadas a fim de que os dados sejam coletados. No final do levantamento, as informações devem ser resumidas com o objetivo de que as conclusões sejam tiradas para compor a lista de requisitos de interface. Em alguns casos é válido dividir essa lista por categoria de usuários como médicos, engenheiros, recepcionistas, técnicos, etc.

É válido destacar que se a empresa deseja lançar um sistema (produto) radicalmente inovador haverá dificuldades para extração dos requisitos de interface, sendo em alguns momentos quase impossível determinar os potenciais usuários para a aplicação. Nesta situação, o produto deve ser lançado e os ajustes na interface devem ser feitos à medida que as características dos usuários forem levantadas com a utilização da aplicação.

A seguir, mostra-se um estudo feito para um sistema fictício com o objetivo de extrair os requisitos, que mais tarde influenciarão em outros documentos e na interface do sistema.

Nome do sistema: SGCP (Sistema para Controle e Gerenciamento de Pedidos)
Área: Engenharia Mecânica
Função: Gerenciar os pedidos de uma loja de autopeças
Observação: Foram entrevistadas somente as pessoas que utilizarão o sistema

A seguir, apresenta-se a **Tabela 1** contendo os tipos de usuários que utilizarão o sistema e a respectiva descrição de cada um.

Usuário Postulado	Descrição
Operador	Usuário responsável pelo atendimento e registro de venda dos produtos: Atende Mecânicos; Atende Concessionárias; Atende Auto-Peças.
Supervisor	Usuário responsável pela supervisão da equipe de Operadores. Possui privilégios de cancelar pedidos e alterar taxa de desconto. Responsável também pelo controle de desempenho dos operadores.
Gerente	Controlar aspectos financeiros das vendas, controlar produtividade através de relatórios do sistema e possuir alçada para delegar funções.

Tabela 1. Perfil do Usuário Postulado

Após apresentar uma rápida descrição sobre perfil do usuário, veremos na **Tabela 2** o instrumento de coleta de dados de identificação do perfil do usuário. O instrumento de coleta de dados é formado por objetivos e perguntas que servirão para que um questionário seja gerado a fim de ser aplicado ao usuário final.

Objetivo	Perguntas
Determinar conhecimentos gerais do usuário.	Qual a sua idade?
	Qual o seu grau de instrução/escolaridade?
	Qual a sua opinião sobre cursos e treinamentos?
	Qual a sua facilidade em lidar com números (contas de multiplicação e divisão)?
Determinar a experiência e perfil funcional do usuário em relação à interface a ser adotada.	Qual o seu grau de utilização de computador?
	Qual o seu nível de conhecimento em Internet?
	Qual sua experiência em digitação?
Determinar a familiaridade do usuário com o sistema de pedidos.	Qual a sua experiência em sistemas de pedidos de compras?
Determinar conhecimento do usuário sobre o negócio da empresa para determinar grau de detalhamento das informações.	Como você classificaria o seu conhecimento em autopeças?
Determinar a desenvoltura do usuário em lidar com clientes e resolução de conflitos.	Qual a sua experiência em lidar com clientes?
Determinar nível de segregação das funções do sistema.	Qual a sua experiência em gerenciar pessoas/equipes?
	Qual o seu poder de decisão onde você trabalha?

Tabela 2. Instrumento de coleta de dados de identificação do Perfil do Usuário

A partir da **Tabela 2** pode-se extrair o seguinte questionário que deverá ser usado para coletar os dados que identificarão o perfil do usuário do sistema.

1. Qual a sua idade?
( ) 15-25
( ) 26-35
( ) 36-45
( ) 46-55
( ) Maior que 56

2. Qual o seu grau de instrução / escolaridade?
( ) 1º. Grau incompleto
( ) 1º. Grau completo
( ) 2º. Grau incompleto
( ) 2º. Grau completo
( ) Nível Superior incompleto
( ) Nível Superior completo
( ) Pós-Graduação incompleto
( ) Pós-Graduação completo

3. Como você classificaria seu conhecimento em autopeças?
( ) Conhecimento Alto
( ) Conhecimento médio
( ) Conheço alguma coisa
( ) Nenhum conhecimento

4. Qual o seu grau de utilização de computador?
( ) Nenhuma
( ) Pouca experiência
( ) Utiliza somente no serviço para tarefas essenciais
( ) Utiliza somente para acesso a Internet e e-mail
( ) Alta utilização

5. Qual o seu nível de conhecimento em Internet?
( ) Nunca utilizou
( ) Utiliza mas com pouca experiência
( ) Utiliza com muita frequência
( ) Sou um expert no assunto

6. Qual sua experiência em digitação?
( ) Alto
( ) Médio
( ) Baixo

7. Qual a sua experiência em sistemas de pedidos de compras?
( ) Alto
( ) Médio
( ) Baixo

8. Qual a sua experiência em lidar com clientes?
( ) Alta
( ) Média
( ) Baixa

9. Qual a sua opinião sobre cursos e treinamentos?

( ) Muito importante
( ) Importante
( ) Mais ou menos importante
( ) Sem importância

10. Qual a sua facilidade em lidar com números (contas de multiplicação e divisão)?
( ) Alto
( ) Médio
( ) Baixo

11. Qual a sua experiência em gerenciar pessoas/equipes?
( ) Alto
( ) Médio
( ) Baixo

12. Qual o seu poder de decisão onde você trabalha?

( ) Alto
( ) Médio
( ) Baixo
( ) Não trabalho

Quantidade de Pessoas Entrevistadas	1 - Qual a sua idade?					2 - Qual o seu grau de instrução / escolaridade?						3 - Qual seu conhecimento em autopeças?				4 - Qual o seu grau de utilização de computador?					5 - Qual o seu nível de utilização em Internet?				6 - Qual sua experiência em digitação?		
	15-25	26-35	36-45	46-55	> 56	2º grau incom.	2º grau com.	Superior incom.	Superior com.	Pos-grad incom.	Pos-grad com.	Alto	Médio	Pouco	Nenhum	Nenhum	Pouca experiência	somente para tarefas essenciais	Somente para internet e email	Alta utilização	Nenhum	pouca freqüência	muita freqüência	Expert no assunto	Alto	Médio	Baixo
1			X				X								X			X				X					X
2			X			X								X			X				X					X	
3	X							X							X					X			X			X	
4	X							X							X				X			X					X
5			X					X							X			X			X					X	
6	X							X					X				X						X			X	
7			X						X					X				X					X			X	
8			X						X				X					X					X				X
9		X						X				X					X					X					X
10		X							X					X			X					X					X
11		X								X					X					X				X		X	
12		X							X				X							X				X	X		

Tabela 3. Dados Obtidos

Com o objetivo de simular as entrevistas dos usuários, a **Tabela 3** apresenta os dados obtidos com as respostas fornecidas por doze participantes do processo.

A seguir, a **Tabela 4** apresenta outra visão dos dados da Tabela 3, indicando a porcentagem de respostas fornecidas para cada questão, melhorando a percepção sobre o tipo do usuário que está sendo estudado.

Requisitos obtidos em relação ao perfil básico dos usuários

Após elaborar o instrumento de coleta de dados, gerar o questionário e entrevistar os usuários, alcançando os dados da **Tabela 4**, mais especificamente a coluna (%), estima-se que pela maioria das respostas dos usuários têm-se os seguintes requisitos não funcionais de produto:

- Idade Média entre 36-45 anos;
- Usuário possui grau de escolaridade – Superior incompleto;
- Possui pouco conhecimento sobre o negócio da empresa (venda de autopeças);
- Possui alguma experiência na utilização de computador;
- Os usuários na sua maioria já utilizaram a Internet;
- Possui alguma experiência em digitação;
- Possui alguma experiência em sistema de pedidos de compras;
- Possui alguma experiência em lidar com clientes;
- Usuários acham importante ter treinamento na empresa;
- Usuários na sua maioria têm facilidade de lidar com números;
- Pelo menos metade da equipe tem facilidade em gerenciar equipes;
- Poucos possuem alto poder de decisão na empresa.

É valido lembrar que os requisitos de perfil do usuário estabelecem um impacto direto nos outros artefatos a serem gerados para que uma aplicação possua boa usabilidade. Essa extração deve ser feita pelo engenheiro de usabilidade em conjunto com o designer de interface. Mesmo depois que os requisitos de interface estiverem prontos é necessário, depois de alguns anos, reavaliar o perfil do usuário com o objetivo de saber se eles ainda são os mesmos e certificar-se que não será necessário melhorar a usabilidade. Em alguns casos,

Questões	Opções	Qtd	%
1 - Qual a sua idade?	15-25	3	25%
	26-35	4	33%
	36-45	5	42%
	46-55	0	0%
	> 56	0	0%
2 - Qual o seu grau de instrução / escolaridade?	2º grau incompleto	1	8%
	2º grau completo	1	8%
	Superior incompleto	5	42%
	Superior completo	4	33%
	Pos-grad incompleto	1	8%
	Pos-grad completo	0	0%
3 - Qual o seu conhecimento sobre peças automotivas?	Alto	1	8%
	Médio	3	25%
	Pouco	3	25%
	Nenhum	5	42%
4 - Qual o seu grau de utilização de computador?	Nenhum	0	0%
	Pouca experiência	4	33%
	Somente para tarefas essenciais	4	33%
	Somente para Internet e e-mail	1	8%
	Alta utilização	3	25%
5 - Qual o seu nível de utilização em Internet?	Nenhum	2	17%
	Pouca freqüência	4	33%
	Muita freqüência	4	33%
	Expert no assunto	2	17%
6 - Qual sua experiência em digitação?	Alto	1	8%
	Médio	6	50%
	Baixo	5	42%
7 - Qual sua experiência em Sist. de pedidos de compras?	Alto	3	25%
	Médio	5	42%
	Baixo	4	33%
8 - Qual sua experiência em lidar com cliente?	Alta	3	25%
	Média	5	42%
	Baixa	4	33%
9 - Qual a sua opinião sobre cursos e treinamentos?	Muito importante	5	42%
	Importante	3	25%
	Mais ou menos importante	4	33%
	Sem importância	0	0%
10 - Qual sua facilidade em lidar com números? (contas mult. e divisão)	Alta	5	42%
	Média	4	33%
	Baixa	3	25%
11 - Qual sua experiência em gerenciar pessoas / equipes?	Alta	2	17%
	Média	6	50%
	Baixa	4	33%
12 - Qual o seu poder de decisão onde você trabalha?	Alto	2	16%
	Médio	5	42%
	Baixo	5	42%

Tabela 4. Análise dos Resultados Obtidos

Quantidade de Pessoas Entrevistadas	7 - Qual sua experiência em Sist. de pedidos de compras?			8 - Qual sua experiência em lidar com clientes?			9 - Qual a sua opinião sobre cursos e treinamento?				10 - Qual sua facilidade em lidar com números? (contas mult. e divisão)			11 - Qual sua experiência em gerenciar pessoas / equipe?			12 - Qual o seu poder de decisão onde você trabalha?		
	Alto	Médio	Baixo	Alta	Média	Baixa	Muito importante	Importante	Mais ou menos importante	Sem importância	Alta	Média	Baixa	Alta	Média	Baixa	Alto	Médio	Baixo
1		X			X		X						X		X			X	
2	X			X					X		X					X			X
3			X			X		X			X					X			X
4			X		X			X				X				X			X
5		X			X				X				X		X			X	
6		X			X		X				X			X				X	
7			X		X			X				X				X	X		
8		X				X	X						X		X			X	
9			X			X	X				X				X			X	
10		X				X			X			X			X				X
11	X			X			X				X			X			X		
12	X			X					X			X			X				X

Cont. Tabela 3. Dados Obtidos

os usuários tornam-se melhor estudados, possuindo novas experiências e adquirindo novas práticas. O levantamento do perfil do usuário alimenta diretamente as informações utilizadas na análise de tarefas, fornecendo uma visão geral sobre o que será estudado.

Análise de tarefas

A análise de tarefas é um dos estudos mais importantes no levantamento dos requisitos de usabilidade. Ele terá grande impacto na interface e na usabilidade do sistema, possuindo relação direta com a melhoria na rentabilidade de uma aplicação. Entende-se em melhorar a rentabilidade de um sistema, por exemplo, uma interface simples de manipular, que permita ao usuário adquirir um produto com rapidez e eficiência (no caso de um site de E-commerce).

O trabalho de analisar as tarefas a serem feitas por um só usuário, pode ter início depois que a aplicação tenha sido identificada, definida e esteja com o escopo já finalizado. Com o escopo aprovado, o engenheiro de usabilidade deve identificar as características principais do sistema, levantando o que é crítico (requer mais estudo) e o que não é crítico (não sendo necessário desenvolver a análise de tarefas).

Em outras palavras, a análise de tarefas visa auxiliar na definição dos requisitos de usabilidade e deixar claro o caminho de como esses requisitos foram extraídos. Para auxiliar na identificação de uma análise de tarefas é necessário utilizar três passos básicos: levantar a informação do tipo de trabalho que está sendo automatizado, coletar e analisar dados de observações e entrevistas com usuários, e construir e validar um modelo de usuário levando em consideração as principais tarefas da organização.

O mais importante dos passos mencionados anteriormente é: Coletar e analisar dados de observações e entrevistas com usuários utilizando como contexto a aplicação a ser desenvolvida. A idéia principal desse passo é fazer com o que o desenvolvedor (neste caso o engenheiro de usabilidade) descubra qual o modelo de trabalho do usuário, aplicando essas informações nas funcionalidades e na interface do sistema a ser projetado. Normalmente, a realização de testes com possíveis usuários reais da aplicação traz muitos benefícios, fazendo com que o engenheiro de usabilidade entenda dificuldades muitas vezes não encontradas em estudos anteriores. Para o levantamento dessas informações não basta utilizar apenas os modelos de casos de uso, eles são conceitos abstratos que capturam generalidades através de poucos usuários fazendo tarefas similares. Mas como realizar o estudo acima? Inicialmente, procure o local onde o usuário utilizará a aplicação, caracterize o ambiente, conheça os jargões do usuário e observe o comportamento dele. Procure pensar como o usuário para que possa identificar ao máximo futuras melhorias de interface e entregar o produto o mais completo possível.

A seguir, mostra-se um exemplo de como realizar a análise de uma tarefa do sistema SGCP (Sistema para Controle e Gerenciamento de Pedidos – apresentado no item anterior) levantando primeiramente os dados da caracterização do ambiente, elaborando uma árvore de metas e finalmente escolhendo e descrevendo uma tarefa a ser analisada. É válido destacar que os requisitos surgem à medida que as tarefas forem analisadas, ou seja, quanto maior o número de tarefas escolhidas, maior a probabilidade de descobrir mais requisitos.

Caracterização do ambiente

O ambiente estudado para a realização do trabalho possui as seguintes características:

- Irá possuir no máximo doze pessoas, sendo onze em contato direto com o sistema;
- É uma área dividida em dez baias para operadores, supervisores e sala para gerentes;
- Todas as pessoas possuem um micro computador e telefone com ramal específico;
- O ambiente não possui muitos ruídos (apenas o barulho do toque dos telefones que não são muito altos). A razão da existência de baias é reduzir esse som dos toques dos telefonemas, além de impedir que o operador desconcentre-se olhando para a tela do computador do companheiro ao lado;
- Entre essas pessoas, tem-se um gerente, dois supervisores, oito operadores (para cada quatro operadores, tem-se um supervisor relacionado) e um administrador da rede (que não irá operar o sistema);
- O administrador da rede será o responsável pela verificação e correção dos problemas técnicos com hardware, configuração de ramais, energia elétrica, estabilizadores, backups do servidor e de elementos da rede como hubs e switches;
- O sistema ficará em uma máquina servidora localizada em uma sala separada das baias. Essa máquina possui backups de duas em duas horas e os dados são armazenados em fitas DAT (ministrado pelo administrador da rede).

Árvore de Metas

A árvore de metas é uma maneira de representar passo a passo como o usuário realiza uma operação do seu dia a dia no sistema. Com ela é possível visualizar o processo operacional como um todo, facilitando encontrar possíveis melhorias no sistema que influenciem positivamente na realização da tarefa do usuário. Utilizando o contexto da aplicação SGCP (Sistema para Controle e Gerenciamento de Pedidos), mostra-se a seguir três exemplos de árvore de metas através das **Figuras 1**, **2** e **3**, representando respectivamente as ações do usuário: realizar venda de um pedido, consultar um pedido e cancelar um pedido. Por meio das figuras, pretende-se entender as tarefas com o objetivo de incluir melhorias funcionais na aplicação de forma a aperfeiçoar o processo.

Requisitos funcionais e não funcionais obtidos da análise de tarefas

Caracterizando o ambiente em que o sistema funcionará e elaborando uma árvore de metas das principais operações do dia a dia do usuário, consegue-se extrair requisitos funcionais e não funcionais que devem ser levados em consideração no momento do desenvolvimento da aplicação.

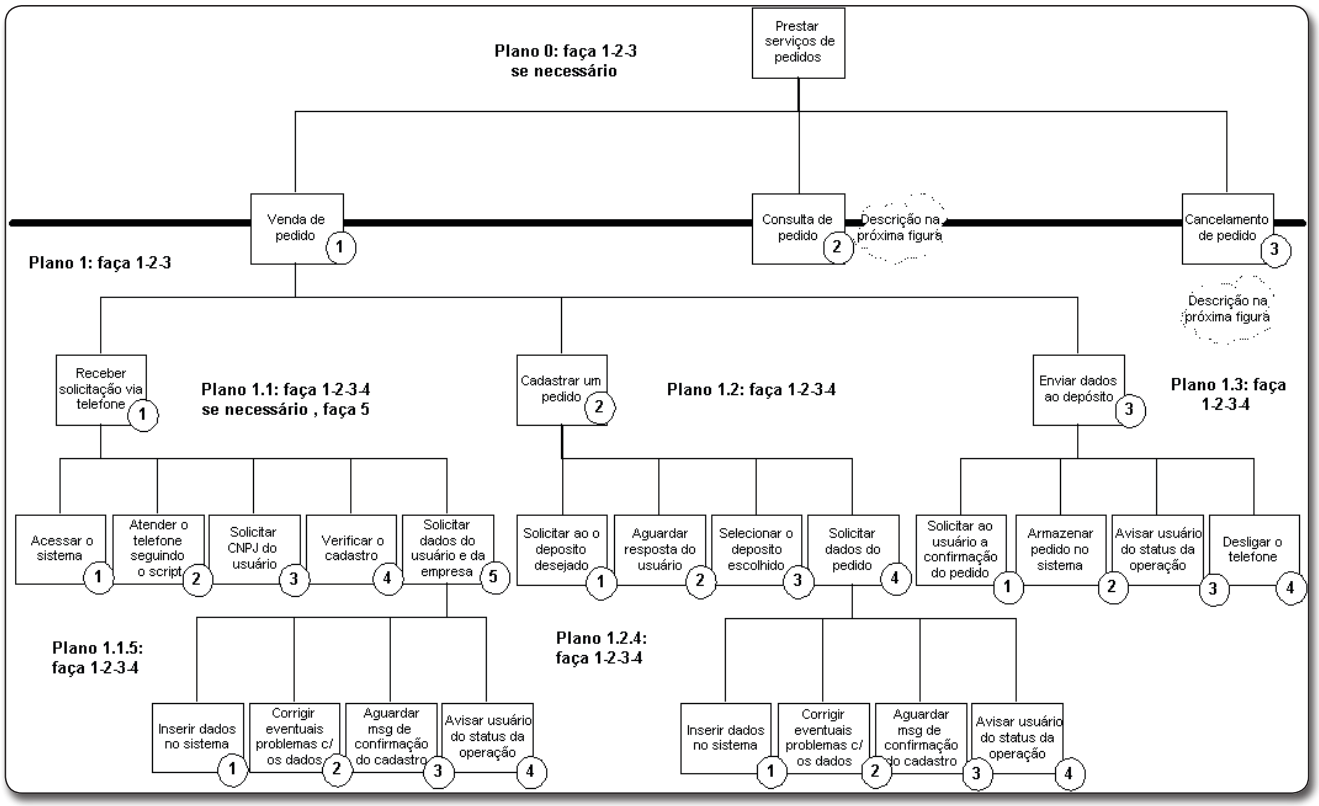


Figura 1. Árvore de Metas – Venda de Pedido

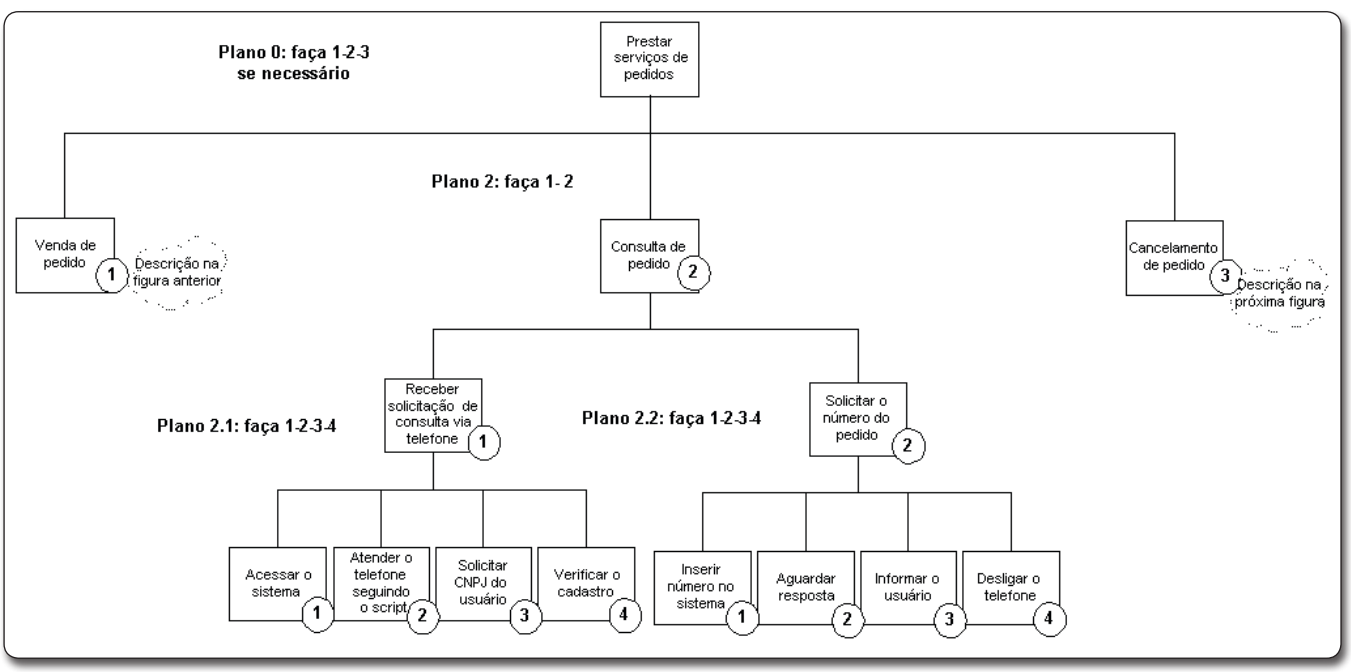


Figura 2. Árvore de Metas – Consulta de Pedido

A seguir, têm-se os requisitos extraídos de acordo com os estudos anteriores:

• A aplicação deve conter algumas teclas de atalho, de forma a otimizar o tempo de acesso do operador e do gerente a algumas áreas do software. Com isso, pode-se reduzir o tempo de espera no telefone por parte do cliente;

• Em razão do oferecimento dessas teclas de atalho, deve-se fornecer treinamento aos empregados da companhia que estarão em contado direto com o sistema;

• É importantíssima a utilização de baias separando os operadores e gerentes. Assim aumentam-se as chances de um bom atendimento ao cliente;

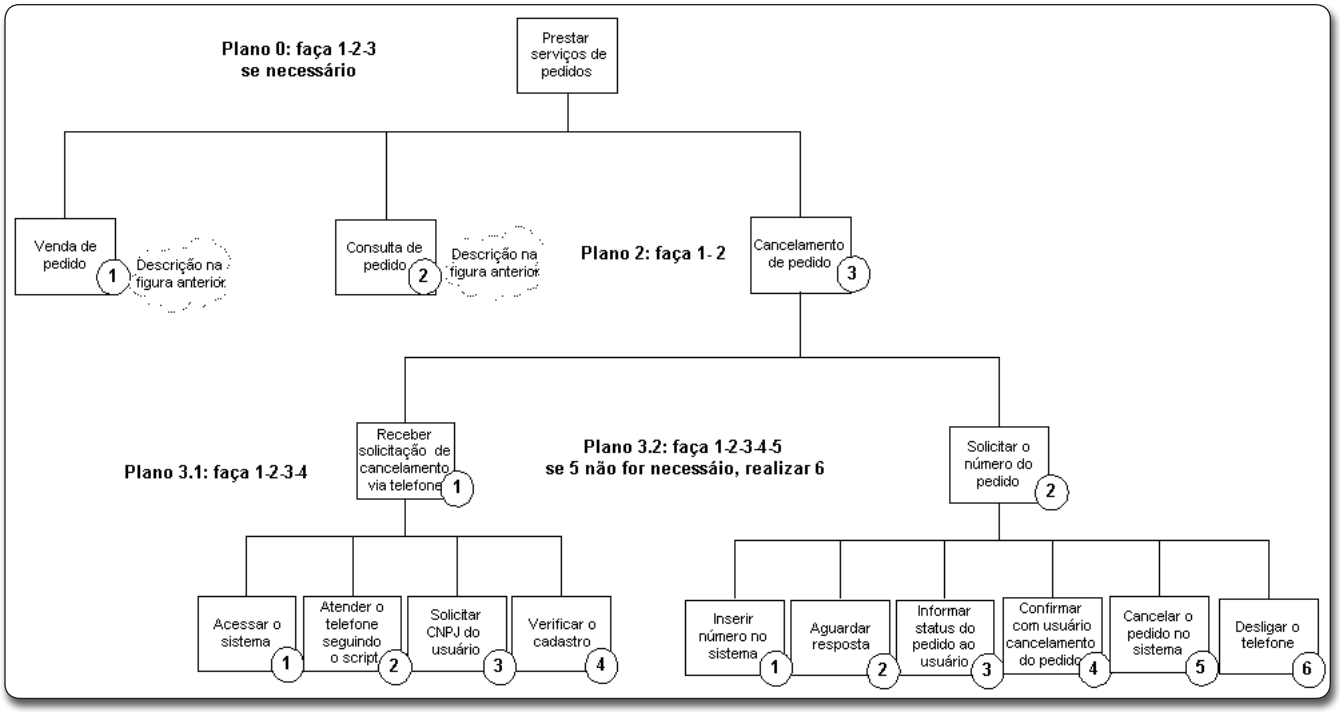


Figura 3. Árvore de Metas – Cancelamento de Pedido

• Sistema deve mostrar na tela o maior número de informações sobre o cliente, a fim de facilitar uma eventualidade confirmação dos dados;
• O sistema precisa ser rápido na busca das informações do cliente (pois o mesmo estará aguardando ao telefone);
Após ser encontrado, o sistema deve apresentar fácil acesso às informações relacionadas aos pedidos recentes do cliente;
• Sistema deve mostrar informações sobre a situação financeira do cliente para com a empresa, com o objetivo desses dados servirem para tomadas de decisões do gerente.

No desenvolvimento de software tradicional, as equipes estão preocupadas apenas em melhorar a automação, esquecendo dos benefícios que um trabalho de reengenharia pode oferecer. A reengenharia oferece uma diferente visão ao processo operacional do usuário, propiciando formas de melhorar a maneira como o usuário trabalha, por exemplo, eliminando tarefas que não causam impacto direto ao objetivo do negócio. Outra dica importante para a extração dos requisitos de usabilidade é: nunca assuma que o usuário pensa como você (você não pode assimilar simplesmente a forma de um usuário para dentro de seu projeto). Mesmo os protótipos e casos de uso ajudando na validação do projeto, eles não mostram informações sobre o comportamento e o modelo de trabalho do usuário. Não menos importante, a análise de tarefas deve ser documentada e poderá ser utilizada como um guia para os testes da aplicação.

Plataforma utilizada

O objetivo deste item é definir a relação entre a interface do usuário e as características da plataforma em que o sistema funcionará. Na maioria dos casos, os engenheiros de usabilidade trabalham com sistemas executados pelo Windows, Unix ou Linux. Podem ocorrer situações em que a empresa tenha de desenvolver o sistema operacional em que o software será executado e até mesmo o hardware.

A fim de facilitar o entendimento, o estudo será feito apenas sobre aplicações ligadas à plataforma Windows. Com essa relação é válido destacar que algumas revisões devem ser feitas no produto à medida que uma nova versão do Windows é lançada no mercado. Dessa forma, assegura-se que o produto esteja funcionando da mesma maneira que foi concebido, não sendo necessário modificar alguma de suas características em função de uma nova versão do sistema operacional.

Para realizar o estudo sobre a plataforma utilizada e levantar os requisitos a serem usados para definir o escopo do projeto deve-se extrair, por exemplo, informações baseadas no:
• Tamanho do monitor;
• Tipo de resolução (640x480, 800x600, 1024x768, etc.);
• Quantidade de cores do monitor (VGA ou Monocromático);
• Dispositivos conectados (teclado, mouse, joystick, touchscreen, etc.);
• Velocidade do processador (MHz, GHz);
• Velocidade da banda (rede ou modem);
• Sistema operacional que será executado (Windows, Linux, etc.);
• Indicação do tipo de processamento (multitarefa);
• Quantidade de efeitos especiais (3D, animações, sons, etc.);
• Tipos de objetos que serão utilizados na tela como (listbox, combobox, textbox, etc.);

Utilizando como exemplo o sistema SGCP (Sistema para Controle e Gerenciamento de Pedidos) comentado anteriormente, apresenta-se a seguir alguns exemplos de requisitos não funcionais que podem ser extraídos:

- Monitor de 13.5 polegadas;
- Resolução utilizada será 640x480;
- Quantidade de cores: Monocromático;
- Dispositivo conectado: apenas o teclado, placa de rede e modem externo;
- Velocidade do processador: 1000 MHz;
- Velocidade de banda: 10 Kbps;
- Sistema operacional: Windows;
- Processamento com multitarefa;
- Não haverá tela com animações e efeitos especiais;
- Objetos utilizados em tela: listbox, combobox, textbox, datagrid, labels, botões e imagens.

Ainda existem outros parâmetros de hardware e software que podem ser utilizados para extração dos requisitos não funcionais de uma aplicação. Eles não foram citados pois variam de acordo com o tipo de plataforma que o projetista usará. Com os requisitos definidos, várias situações de programação poderão ser modificadas, por exemplo: não adiantaria a aplicação possuir áreas sinalizadas com cores, pois os monitores utilizados pelos usuários seriam monocromáticos.

Identificar corretamente a plataforma em que o sistema funcionará é um fator muito importante, pois dela serão gerados os requisitos que por sua vez serão os responsáveis pela modelagem correta de uma aplicação sob o ponto de vista da usabilidade.

Princípios gerais de design

Mais de vinte anos de pesquisa e estudos de caso foram documentados sobre sistemas de interface do usuário. Muitas dessas pesquisas têm sido sintetizadas nos princípios gerais de design e aplicadas em larga escala sobre o estudo de perfil do usuário, análise de tarefas e tipos de plataforma utilizada.

O propósito desse estudo é identificar e revisar o que já foi atribuído ao projeto, com os princípios gerais de design. Possíveis modificações devem ser aplicadas na fase de projeto, desenvolvimento e testes com o intuito de melhorar a usabilidade do produto. Os princípios gerais de design não substituem os requisitos não funcionais extraídos de uma aplicação. Eles devem preferencialmente completar-se. Um dos exemplos da aplicabilidade dos princípios gerais de design, em um sistema que possui alta incidência de usuários do sexo masculino, seria o cuidado ao utilizar as cores verde e vermelha. Isso deve ocorrer caso o estudo sobre o perfil de usuário tenha apresentado ocorrências de daltônicos como usuários do sistema.

Os princípios gerais de design podem ser encontrados em:

- Livros sobre interface e usabilidade;
- Materiais de cursos sobre usabilidade;
- Referências encontradas em pesquisas e estudos de caso sobre interface com usuário.

Durante o desenvolvimento de um projeto, os princípios gerais de design:

- Podem ser aplicados a qualquer momento durante a análise de um requisito não funcional. As modificações devem ser feitas antes de começarem as fases de projeto, desenvolvimento e testes;
- Devem ser confrontados com os requisitos extraídos do sistema, a fim de certificar que uma boa decisão foi tomada.

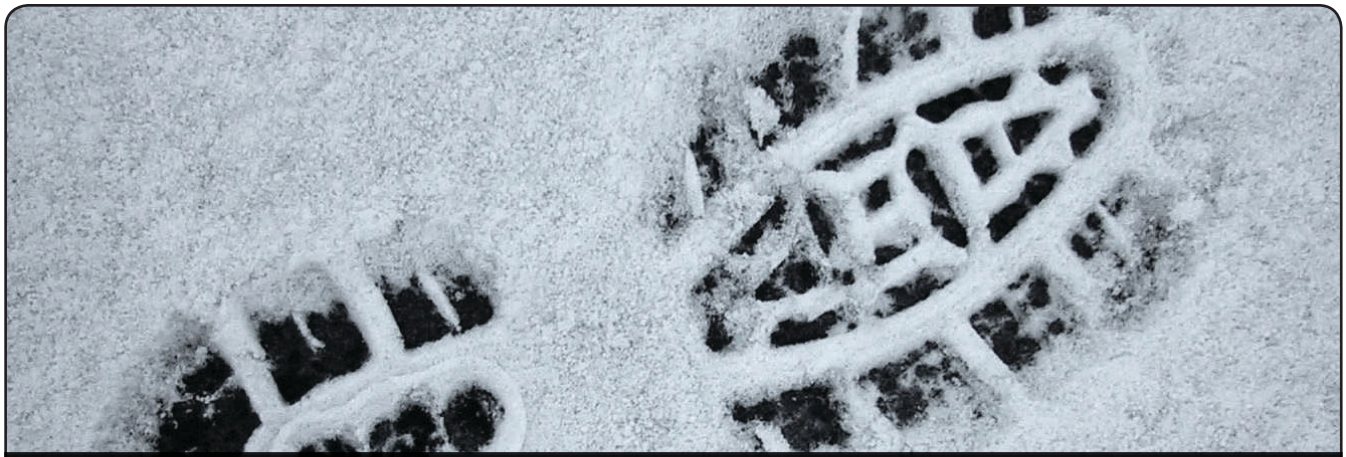
Utilizando com o exemplo o sistema SGCP (Sistema para Controle e Gerenciamento de Pedidos), os princípios gerais de design podem contribuir, por exemplo, para:

- Identificar melhor posição dos objetos na tela, por exemplo, posicionar a área de help no canto superior direito, onde é utilizado na maioria das vezes – caso não haja um requisito indicando outra posição;
- Indicar o melhor tipo de fonte para que tenha melhor legibilidade dependendo do monitor utilizado;
- Indicar os tons de cores que fiquem melhor apresentáveis em monitores monocromáticos.

Quando o engenheiro de usabilidade encontra dificuldades em extrair alguns requisitos não funcionais de um sistema, os princípios gerais de projeto entram em cena com o objetivo de indicar o que é mais utilizado de acordo com os padrões estabelecidos durante anos de pesquisa. Nessa ocasião, entra o bom senso do engenheiro em tomar a decisão, verificando se o que foi estabelecido como padrão encaixa-se realmente na aplicação em desenvolvimento.

Conclusão

Gerar todos os artefatos durante o desenvolvimento de um sistema pode nem sempre ser uma boa opção. Deve-se levar em consideração o tamanho da aplicação, o orçamento disponível e o tempo de desenvolvimento, a fim de que os melhores estudos (artefatos) sejam escolhidos. Indica-se que o engenheiro de usabilidade ou a pessoa responsável pela extração dos requisitos não funcionais de um sistema trabalhe em perfeita sintonia com a área de desenvolvimento. Dessa forma, evitam-se re-trabalhos de programação em função da descoberta de novos requisitos. Não menos importante, recomenda-se que a empresa crie o seu próprio guia de usabilidade, levando em consideração os seus projetos desenvolvidos e os estudos documentados nos princípios gerais de design. Isso fará com que a extração de requisitos de usabilidade de cada projeto torne-se mais simples no futuro, poupando o tempo da organização. ●



Rastreabilidade

Definições Iniciais

Neste artigo será apresentada uma revisão dos principais métodos existentes para lidar com o conceito de rastreabilidade: matrizes de rastreabilidade e métodos de recuperação automática de estruturas de rastreabilidade. A seguir, serão apresentadas técnicas de rastreabilidade alternativas para o rastreamento de requisitos não funcionais, baseadas em cenários, e considerações sobre a granularidade da rastreabilidade para torná-la mais gerenciável e menos custosa. Por fim, será apresentada uma definição formal da rastreabilidade, identificando as operações básicas sobre elos e artefatos.

Jerônimo Backes

jeronimobackes@gmail.com

Mestre em Engenharia de Software pela Universidade Federal do Rio Grande do Sul (2008), e graduado em Ciência da Computação pela Universidade de Santa Cruz do Sul (2006). Trabalhou até o início de 2006 com compiladores e geradores de código. Atualmente, atua como consultor em Análise de Sistemas e desenvolvimento Java pela empresa Stefanini IT Solutions em Porto Alegre, RS.

Matrizes de Rastreabilidade

Matrizes de rastreabilidade são geralmente utilizadas para exibir os relacionamentos entre a elicitação de requisitos e a representação destes requisitos em um método particular da engenharia de software. Mesmo em projetos pequenos ou de tamanho moderado, o estabelecimento de elos de rastreabilidade,

De que se trata o artigo?

Neste artigo será apresentada uma revisão dos principais métodos existentes para lidar com o conceito de rastreabilidade: matrizes de rastreabilidade e métodos de recuperação automática de estruturas de rastreabilidade. A seguir, serão apresentadas técnicas de rastreabilidade alternativas para o rastreamento de requisitos não funcionais, baseadas em cenários, e considerações sobre a granularidade da rastreabilidade para torná-la mais gerenciável e menos custosa. Por fim, será apresentada uma definição formal da rastreabilidade, identificando as operações básicas sobre elos e artefatos.

Para que serve?

Rastreabilidade é um conceito que pode ser considerado chave em projetos de desenvolvimento de software. Este artigo trata de seus principais conceitos.

Em que situação o tema é útil?

Para aqueles que pretendem ter um maior controle sobre os artefatos gerados ao longo do desenvolvimento facilitando sua manutenção e análise de impacto a partir de solicitações de alteração.

entre artefatos-chave e modelos, continua sendo uma tarefa desafiadora e cara [1]. Um dos possíveis motivos é que não há forma padronizada de armazenar ou representar elos de rastreabilidade. Tradicionalmente, eles são armazenados em uma matriz e representados como grafos.

Em termos da álgebra linear, elas exibem o mapeamento entre fonte e alvo. Tais mapeamentos são apresentados em um tipo especial de matriz, chamada de matriz de dependência, que representa a relação de dependência entre elementos da fonte e do alvo (fonte x alvo). Nas linhas, ficam os elementos fonte, e nas colunas, os elementos alvo. Nesta matriz, uma célula com o valor 1 denota que o elemento fonte (na linha) é mapeado para o elemento alvo (na coluna). Reciprocamente, isto significa que o elemento alvo depende do elemento fonte [2].

Em sua forma mais simples, a rastreabilidade se manifesta em tabelas cruzadas, nas quais os elementos de um projeto são relacionados aos requisitos que satisfazem [3]. Nesta matriz, elementos-fonte são mapeados para elementos-alvo. A **Tabela 1** [2] apresenta o mapeamento do elemento-fonte f1 para os elementos-alvo a1, a3 e a4, o que indica que a1, a3 e a4 dependem de f1. Analogamente, pode-se dizer que f1 dá origem a a1, a3 e a4.

Esta representação permite visualizar, por exemplo, que vários requisitos são implementados por uma mesma classe, permitindo que a intersecção entre classes responsáveis pela satisfação de um requisito seja não-vazia (ou seja, n classes podem ser comuns entre dois ou mais requisitos). Além disso, é possível visualizar as várias classes que podem ser necessárias para a implementação de um requisito. [4]. É possível representar a tabela graficamente, conforme a **Figura 1** [4].

		Alvos			
		a1	*a2*	*a3*	*a4*
Fontes	*f1*	1	0	1	1
	f2	0	1	0	0
	f3	0	0	1	0

Tabela 1. Exemplo de dependência entre elementos

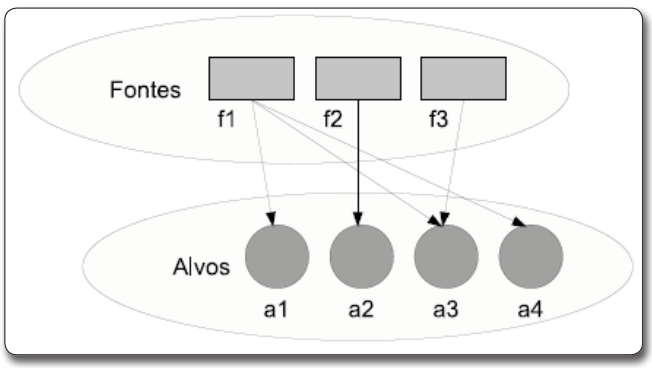


Figura 1. Representação gráfica da matriz de rastreabilidade

Elos de rastreabilidade são geralmente estabelecidos pelo relacionamento explícito entre dois artefatos, armazenando-os em tabelas, bancos de dados, ou ferramentas de gerenciamento de requisitos [5], e esta ainda é a prática atual [6]. Existem diversos modelos de rastreabilidade que diferem nos tipos de artefatos que rastreiam. Porém, a maioria dos projetos não utiliza um método sistemático de rastreamento, mas delega, para determinados indivíduos, a atividade de realizar um rastreamento ad hoc, quando for necessário [7].

Formalmente, as estruturas tradicionais de rastreabilidade são criadas seguindo a definição Elo(a,a'). Uma matriz de rastreabilidade (MR) é formada pelo conjunto {Elo(a,a')|a e a' são artefatos, em qualquer nível de abstração (eg. caso de uso, especificação de requisitos, diagrama de classe, código fonte de uma linguagem específica, etc. } [8].

Cada artefato possui um nível de abstração, que pode ser mais ou menos completo do que os outros, e apresenta diferentes pontos de vista sobre as necessidades do sistema. As relações entre estes, do menos completo para o mais completo, até a implementação, deve ser mantida para garantir que a estrutura de rastreabilidade permita:

- O mapeamento dos requisitos para um modelo posterior, até o código fonte, garantindo que a satisfação dos requisitos esteja atribuída a componentes do sistema (Forward from Requirements);
- O Mapeamento de um modelo qualquer de volta para os requisitos, evitando o gold-plating (Backward to Requirements);

Grande parte das propostas de rastreabilidade possui estas características, no entanto, algumas suportam o rastreamento somente vertical (relacionamentos entre artefatos no mesmo nível de abstração), outras somente o horizontal (relacionamentos entre artefatos em níveis diferentes de abstração). Para exemplificar a utilização de matrizes no rastreamento vertical, considere as seguintes regras de negócio (doravante denominados simplesmente como requisitos, por simplicidade):

R1 "Todo cliente tem uma conta, com um determinado limite de crédito";

R2 "O limite de crédito de qualquer conta está restrito a um determinado valor, estabelecido segundo a lei federal XYZ";

R3 "A verificação do limite de crédito é realizada de acordo com os seguintes critérios...";

R4 "Universitários têm limite de R$ 200,00, invariavelmente".

Não é necessário muito esforço de interpretação para perceber que os requisitos estão de fato relacionados, pois tratam de interesses comuns, formando a possível matriz de rastreabilidade, representada pela **Tabela 2**.

Requisitos	R1	R2	R3	R4
R1		X		X
R2			X	
R3				X
R4				

Tabela 2. Matriz de rastreabilidade entre os requisitos R1 a R4

Note que apesar do pequeno número de requisitos e relacionamentos, não é trivial interpretar a matriz da Tabela 2 e identificar o motivo por trás da existência dos elos e a inexistência de outros. No caso do exemplo, estes foram criados pelas seguintes interpretações:

R1-R2 "A lei federal XYZ estabelece o limite da conta, citado em R1";
R1-R4 "O limite da conta também pode ser definido pelo valor citado em R4";
R2-R3 "A verificação deve considerar as regras impostas pela legislação";
R3-R4 "A verificação deve considerar o caso das contas universitárias".

Porém, várias interpretações diferentes podem ser tomadas a partir dos requisitos apresentados, formando relacionamentos diferentes. Assim, para evitar este esforço de re-interpretação, é necessário anexar a cada elo as decisões, problemas, suposições e argumentos que motivaram sua criação. Analogamente, é fundamental que as mesmas informações sejam mantidas para os elos que não foram criados, evitando o esforço desnecessário de justificar novamente uma decisão já estabelecida [9].

Para exemplificar o uso de matrizes no rastreamento horizontal, considere os requisitos R1 a R4, "implementados" no código fonte da Listagem 1. Como a listagem implementa os requisitos, pode-se dizer que estes pertencem ao conjunto de relações { Elo(a, r)|a é um requisito e r é um trecho de código }.

Listagem 1. Exemplo de implementação dos requisitos

```
1  Client {
2   per sonalData ;
3   account ;
4 }
5
6  Account{
7   cash ;
8   limit ;
9   type ;
10 }
11
12 System{
13   listOfClients ;
14
15   limitVerification(Amount a, Account c){
16     . . .
17   };
18
19   withdraw(Client c, Account acc, Amount a){
20     limitVerification(a, acc) ;
21     ...
22   };
23 }
```

Requisito	Classes	Métodos	Atributos
R1	Cliente, Account		Cliente.account.limit
R2	Account		Account.limit
R3		System.limit.Verification	
R4	Account		Account {limit, type}

Tabela 3. Matriz de rastreabilidade entre requisitos e código

A **Tabela 3** representa as possíveis relações entre os requisitos e partes do código apresentado. O requisito R1 é satisfeito pela implementação das classes Client e Account, bem como pelo atributo limit em Account. R2 trata do limite da conta, então a implementação de Account.limit pode ser afetada caso R2 seja alterado (e vice-versa). Já R3 é satisfeito pela implementação de System.limitVerification, e R4 pelo tratamento de um tipo de conta (Account.type, que pode ser "conta de pessoa física", "conta de pessoa jurídica", "conta universitária", etc.). Note que a matriz de rastreabilidade depende muito da interpretação humana, sendo que outras interpretações e necessidades podem originar uma matriz completamente diferente.

Analisando mais profundamente, não é difícil identificar o potencial de crescimento de uma matriz de rastreabilidade. Os problemas de manutenção aparecem quando se tenta atualizar os elos, especialmente quando se rastreia um grande número de artefatos, com um nível de granularidade que possa ser considerado útil, como no exemplo da Tabela 3. Por exemplo, projetos com centenas de casos de uso, compostos por inúmeros passos, e relacionados a outros artefatos, são de manutenção extremamente complexa e cara. A matriz de rastreabilidade facilmente assume um tamanho ingerenciável, pois a quantidade de relações tende a crescer exponencialmente [10]. A partir deste ponto é que as técnicas de rastreabilidade baseadas em MR falham. Um dos maiores problemas reside na incapacidade de lidar eficientemente com alterações que afetam os relacionamentos estabelecidos [8].

Mesmo em projetos pequenos ou de tamanho moderado, o estabelecimento de elos de rastreabilidade, entre artefatos-chave e modelos, continua sendo uma tarefa desafiadora e cara [1]. Um dos possíveis motivos é que não há forma padronizada de armazenar ou representar elos de rastreabilidade. Tradicionalmente, eles são armazenados em uma matriz e representados como grafos. Enquanto a simplicidade é a principal vantagem dos métodos tradicionais, eles são pouco úteis na representação de todas as informações relevantes [11]. Além disso, os esforços dos desenvolvedores são retardados pela falta de suporte ferramental, e há uma percepção geral de que o custo necessário para manter uma matriz de rastreabilidade de requisitos é excessivo, em relação aos seus benefícios [6].

Métodos de Recuperação Automática

Técnicas de geração automática de elos, entre requisitos expressos textualmente e modelos posteriores (geralmente código fonte), vêm emergindo como alternativa para satisfazer o problema da cara e complexa manutenção da rastreabilidade [12, 17, 18, 4].

O problema de identificar os documentos relacionados a um dado componente do código fonte pode ser visto como uma tarefa típica e reconhecimento de padrões, ou de recuperação de informações [13]. Os elos de rastreabilidade entre artefatos também podem ser identificados utilizando simples regras, baseadas em convenções [1]. As técnicas de recuperação de informações confiam na hipótese de que o uso correto de termos do domínio entre artefatos permite rastreá-los. Nos casos em que isto não acontece, o processo de recuperação automática da rastreabilidade (RAR) torna-se ineficiente [14]. Assim, os métodos de RAR assumem a conjetura do programador são (que produz identificadores com nomes significativos, ou que sigam um padrão); e que para cada requisito, exista pelo menos um elo de rastreabilidade disponível, produzido manualmente. Estes elos já existentes auxiliam na recuperação de elos desconhecidos.

A recuperação de elos funciona na seguinte seqüência [13]:

• **Processamento de requisitos:** cada requisito é processado e um dicionário de palavras é montado, com base na frequência de cada uma. Esta fase em geral é semi-automática, pois não é possível retirar a ambiguidade da semântica contextual, nem lidar com discursos metafóricos (eg. "deve-se 'matar' o processo...");

• **Processamento do código:** para cada classe, uma lista de identificadores contendo suas ocorrências é extraída. Aqui, a intervenção humana é necessária para informar o significado de palavras curtas, com menos de três letras;

• **Recuperação do mapa de rastreabilidade:** é realizada através de um classificador.

Outras técnicas de recuperação são utilizadas por Antoniol et al. [4] para recuperar as relações de rastreabilidade de código C++ para páginas de manuais e de código Java para requisitos. Settimi et al. [15] discutem a obtenção de elos de rastreabilidade entre código e modelos UML, usando uma técnica de recuperação de informações baseada em um modelo de espaço vetorial, mas os resultados apresentaram-se limitados. Porém, a técnica reduziu o tempo necessário na manutenção da rastreabilidade. No estudo realizado por Lucia et al. [16], em um pequeno projeto, para atingir o objetivo de obter todos os elos corretos, rastreando casos de uso para classes do código fonte, foi necessário analisar 1013 elos, para identificar, entre estes, 93 elos corretos.

Considerando o extremo esforço necessário para a manutenção de elos, foram propostos métodos de RAR, que assumem a conjetura do programador são (que produz identificadores com nomes significativos, ou que sigam um padrão); e que para cada requisito, exista pelo menos um elo de rastreabilidade disponível, produzido manualmente. Estes elos já existentes auxiliam na recuperação de elos desconhecidos. As técnicas de recuperação de informações confiam na hipótese de que o uso correto de termos do domínio entre artefatos permite rastreá-los. Nos casos em que isto não acontece, o processo de recuperação da rastreabilidade torna-se ineficiente [14].

Todos os estudos interessados na aplicação de métodos para a RAR mostram que estes não são capazes de recuperar todos os elos corretos, sem também recuperar muitos falsos positivos, que devem ser descartados pelo engenheiro de software. A baixa precisão faz deste método uma tarefa tediosa, já que o engenheiro tem que gastar muito mais tempo para descartar falsos positivos do que rastreando elos corretos. Embora seja possível melhorar a performance destes métodos, eles ainda estão distantes de tornar viável a solução para o problema da recuperação de elos [16]. Mesmo assim, esta solução vem sendo cada vez mais reconhecida pela indústria como uma solução potencial para o problema da rastreabilidade [6]. Em contrapartida, Hayes e Dekhtyar [37] estudaram os resultados produzidos por analistas, e descobriram que as decisões tomadas sobre a validade ou invalidade dos elos recuperados nem sempre são corretas, e podem até piorar os resultados: elos corretos acabam sendo descartados, e elos incorretos mantidos.

Como a maioria dos métodos e ferramentas de RR existentes assumem que o desenvolvedor crie e mantenha as relações de rastreabilidade manualmente, ambientes industriais raramente estabelecem um processo de rastreabilidade, já que o estabelecimento dos links de forma manual, além de ser custoso em termos de tempo, é sujeito a erros [12]. Por este motivo, técnicas de geração automática de links entre requisitos expressos textualmente e modelos posteriores (geralmente código fonte), vêm emergindo como alternativa para satisfazer o problema da cara e complexa manutenção da rastreabilidade [12, 17, 18, 4].

Técnicas de recuperação de informações (RI) podem ser utilizadas para automatizar o processo de reconstrução de links, como por exemplo Indexação de Semântica Latente (Latent Semantic Indexing - LSI). A LSI é uma técnica de RI promissora, pois assume que há uma estrutura semântica latente em qualquer conjunto de documentos [18].

Através da RI, um usuário pesquisa por informações, utilizando palavras-chave ou consultas de texto em linguagem natural. Um sistema de procura obtém um conjunto de objetos candidatos, a partir do qual o usuário seleciona os que considera relevantes. Na maioria dos algoritmos de pesquisa, um índice de similaridade é calculado, de forma que meça o grau de similaridade entre dois documentos. Um valor de aceitabilidade é então estabelecido, de forma que qualquer valor, acima do nível de aceitabilidade, presente no índice, indica que o documento deve ser recuperado [19].

Os algoritmos de RI geralmente são avaliados utilizando as métricas recall (retorno) e precision (precisão). Tais métricas também são muito utilizadas para avaliar a utilidade de diferentes técnicas de rastreabilidade (STIREWALT; DENG; CHENG, 2005). Onde:

recall = Número de documentos relevantes obtidos/Número de documentos relevantes

precision = Número de documentos relevantes obtidos/Número total de documentos obtidos

O recall máximo pode ser obtido ao pesquisar todos os documentos de uma coleção, e a precision máxima ao pesquisar um único documento que se sabe ser correto. Por esta razão, um sistema de RI normalmente tenta maximizar recall e precision simultaneamente.

Em métodos de recuperação da rastreabilidade, uma consulta é geralmente formulada a partir de palavras encontradas em um requisito, e um algoritmo de busca a executa dentro de um espaço de busca de outros artefatos, como outros requisitos, código, casos de teste, ou modelos do projeto. Resultados de vários métodos de RI encontraram sucesso limitado com taxas de precision geralmente entre 10% e 60%, e níveis de aceitabilidade definidos para obter de 90% a 100% de recall [20, 12].

Outras técnicas de RI são utilizadas por [4] para recuperar as relações de rastreabilidade de código C++ para páginas de manuais e de código Java para requisitos. [15]

discutem a obtenção de links de rastreabilidade entre código e modelos UML, usando uma técnica de RI baseada em um modelo de espaço vetorial, cujos resultados apresentaram-se limitados. Neste, porém, a técnica de RI reduziu o tempo necessário na manutenção da rastreabilidade. Não obstante, o trabalho fornece razões pelas quais grande parte das técnicas de RI não são completamente eficientes, devido a erros de:

• **Inclusão (links falsos):** Causados pelo alto acoplamento entre artefatos, mesmo que estes artefatos representem links corretamente;

• **Omissão (links perdidos, ou não detectados):** Causados por falta de terminologia padronizada entre os responsáveis pelos requisitos, projetistas, programadores e testadores.

Outros problemas na detecção de links ocorrem na troca de perspectiva. Por exemplo, um requisito da perspectiva do usuário poderia ser descrito nos termos "o usuário deve ser capaz de ver ...", enquanto que da perspectiva do sistema seria "o sistema deve exibir ...". Assim, a técnica de RI dificilmente associa a mesma ação a partir de diferentes pontos de vista. Ela também é limitada pela diferença no nível de abstração entre artefatos, como entre código fonte e requisitos expressos em linguagem natural.

Embora os resultados sejam influenciados pelo algoritmo utilizado, e possam ser melhorados através de mecanismos de feedback e da incorporação de conhecimentos contextuais adicionais, a real limitação parece ser relacionada à qualidade do conjunto de dados. Conjuntos que usam glossários de projeto claramente definidos tendem a produzir resultados melhores do que os que usam glossários definidos sem planejamento [19].

Glossários de projeto auxiliam no fornecimento de termos importantes do domínio, além de apresentar uma breve explanação sobre o termo em seu contexto [21].

Embora métodos de rastreabilidade dinâmicos não sejam uma bala de prata, fornecem um meio muito prático na redução de esforço para a obtenção de links. A aplicação de métodos de recuperação de traços no gerenciamento de alterações durante o ciclo de vida de um software reduz o esforço potencial necessário para analisar o impacto de alterações, e provê uma solução alternativa promissora para as atividades de manutenção de links de rastreabilidade [15].

Rastreabilidade de Requisitos Não-Funcionais

Muitas organizações não rastreiam requisitos não-funcionais (RNFs). Alterações funcionais são geralmente implementadas com pouca compreensão de como qualidades do sistema – como segurança e performance – serão afetadas. Não obstante, RNFs são raramente rastreados pois tendem a afetar globalmente o sistema, o que gera a necessidade de manter um exagerado número links de rastreabilidade. Porém, a falha ao gerenciar a alteração de RNFs pode levar a resultados catastróficos, salientando a importância de se rastrear este tipo de requisito [19].

Devido à sua natureza global, RNFs são vagos e interdependentes. Por isso, geralmente só é possível descrevê-los em relação a requisitos funcionais ou abstratos. Além disso, são expressos em um nível de abstração muito alto, e devem ser refinados [22].

A grande desvantagem dos métodos de rastreabilidade é a sua necessidade de possuir um modelo conceitual completo de todos os elementos da especificação. Ao tratar RNFs, desconhece-se completamente cada tipo de RNF e os relacionamentos entre estes tipos. Outro desafio é a natureza entrelaçada dos RNFs. Os métodos de rastreabilidade atuais são baseados em um conjunto de elementos discretos interrelacionados. Por outro lado, RNFs são aspectos transversais que impactam de forma entrelaçada em diferentes requisitos [22].

Cleland-Huang [19] apresenta alguns métodos de rastreabilidade que permitem rastreabilidade de RNFs, apontando seus pontos fracos e fortes:

• **Matrizes:** Embora matrizes sejam tecnicamente capazes de suportar a rastreabilidade de RNFs, requerem que os links sejam definidos e mantidos explicitamente, além de terem problemas de escalabilidade ao representar o grande número links que podem ser gerados a partir de um único RNF;

• **Aspectos:** Interesses podem ser categorizados como funcionais e não-funcionais. Interesses não-funcionais incluem RNFs como manutenibilidade, performance, segurança, etc. e tendem ser insuficientemente concretos para ser implementados como aspectos. Por outro lado, interesses funcionais e mais concretos como logging, autenticação e rastreamento podem ser implementados como aspectos, porque seu comportamento é mais facilmente definido, podendo ser expresso em termos de regras de weaving de aspectos;

• **Métodos de recuperação de informações (RI):** Permite que links candidatos sejam gerados dinamicamente. Porém, as métricas recall e precision dos resultados não são 100% confiáveis, e o usuário deve validar os resultados;

• **Cenários:** São de propósito múltiplo e requerem menos trabalho para estabelecer a rastreabilidade, porém dificilmente provêm a cobertura completa da rastreabilidade;

• **Rastreabilidade baseada em eventos:** Proposta por Cleland-Huang, Chang e Christensen [41], utiliza uma arquitetura publish-subscribe para definir os links de rastreabilidade que suportam análise de impacto automática sobre RNFs. Os links são estabelecidos entre requisitos de performance quantitativamente definidos, restrições na especificação dos requisitos, e variáveis localizadas em modelos de simulação de performance. Quando uma alteração é proposta, o estado atual do modelo de simulação é salvo, e parâmetros especulativos são passados para o modelo. Este então é novamente executado usando os novos valores. Os resultados são extraídos e reportados, e o modelo é restaurado para o estado inicial. Este método demonstra a capacidade de automatizar a re-execução de um modelo de avaliação arquitetural, em resposta a um evento de alteração, assim que um requisito afetado seja descoberto;

• **Baseado em Design Patterns:** Fornece um método para rastrear RNFs para o projeto sem depender de uma proliferação descontrolada de links. Porém, técnicas de detecção de padrões retornam resultados imprecisos.

Em qualquer sistema não-trivial, é inviável o custo e esforço necessários para construir uma matriz de rastreabilidade, que possa capturar uma grande rede de relacionamentos entre RNFs e outros artefatos. Assim, métodos de recuperação dinâmica de links de rastreabilidade fornecem um método que requer esforço menor para obtenção de links [19].

Cenários

Cenários vêm sendo utilizados como um meio alternativo, e por vezes complementar, para expressar requisitos e o comportamento do sistema através das diversas fases de desenvolvimento. Cada cenário pode assumir representações e semânticas diferentes entre cada fase, e estas podem ser relacionadas. Assim, é possível descrever um sistema em níveis de abstração diferentes, mapear os cenários para a arquitetura do software e usá-los em casos de teste [23].

Para estabelecer relações entre cenários, é preciso compreender seu propósito em cada fase, bem como o que estes possuem ou não em comum entre as fases. Um cenário é uma sequência de eventos. Um evento pode ser descrito de várias formas: algo que ocorre, uma ação, uma interação, um passo, uma alteração instantânea de um estado para outro, etc.

Os artefatos produzidos no desenvolvimento de um software – tais como descrições de modelo, linguagens diagramáticas, especificações formais, código fonte, etc. – estão altamente relacionados, de tal forma que alterações em um devem afetar os outros. As dependências entre os elos caracterizam seu relacionamento de forma abstrata [24].

Os links entre cenários servem como artérias entre modelos como diagramas, pois dão vida para descrições de polígonos e setas, de forma a torná-las representações legítimas e úteis de sistemas. Com a ausência da rastreabilidade, ou incerteza sobre sua precisão, a utilidade dos modelos torna-se severamente limitada [25].

O significado semântico de dependências entre elos pode ser inferido através da diferença semântica entre os artefatos ligados por eles. Ou seja, elos são entidades neutras, que simplesmente descrevem dependências. Contudo, a interpretação de tais dependências e de como utilizá-las depende dos artefatos que os elos ligam (EGYED, 2003).

Naslavsky et. al.(2005) [23] identifica diversos tipos de cenários utilizados em diferentes fases do desenvolvimento:

• **Cenários de Requisitos:** são utilizados para descrever requisitos após uma fase preliminar de levantamento. É o tipo de cenário mais próximo da compreensão dos stakeholders, ao contrário dos utilizados em fases posteriores. Os cenários de requisitos descrevem sequências de eventos que ocorrem sobre o sistema proposto. Um exemplo deste tipo de cenário são Use Cases;

• **Cenários de Análise:** adicionam mais detalhes e uma perspectiva da parte interna do sistema aos cenários de requisitos. São identificadas entidades de alto nível, que farão parte do sistema. Da mesma forma, eventos, que antes eram representados por meio de linguagem natural, assumem a forma de interações entre os conceitos do sistema. Um exemplo deste tipo de cenário são diagramas de sequência da UML, ou um gráfico de no qual agentes que trocam mensagens em um nível similar às classes de um diagrama de classes de análise;

• **Cenários de Arquitetura:** acrescentam mais detalhes aos anteriores. Nesta fase, são tomadas decisões sobre quais conceitos tornar-se-ão componentes da arquitetura do sistema. Um exemplo deste tipo de cenário é um diagrama de sequência da UML, ou um gráfico para sequenciamento de mensagens, onde agentes que trocam mensagens são componentes arquiteturais e conectores;

• **Cenários de Projeto:** são cenários descritos em maiores detalhes do que os de arquitetura. Os conceitos usados neste caso são partes de objetos de componentes arquiteturais, exibindo assim, o funcionamento interno dos componentes;

• **Cenários de Código:** descrevem o fluxo de eventos a partir do nível de implementação do sistema, que são na verdade informações de rastreamento em tempo de execução, correspondendo a comandos executados ou invocações de métodos. Assim, cenários de requisitos, análise, arquitetura e de projeto, quando executados, resultam em algum cenário de código;

• **Cenários de teste:** aparecem em cada fase do ciclo de desenvolvimento, e são utilizados para testar a implementação do sistema ou outros artefatos. Os cenários definidos acima, para cada fase do ciclo de vida, podem ser utilizados como cenários de teste. Assim, cenários em cada nível de abstração são utilizados para testar se o comportamento de um cenário de nível inferior está de acordo com o comportamento esperado, descrito pelo de nível superior. Já que um cenário de nível superior pode levar a múltiplos cenários de nível inferior, é necessário testar a sua conformidade.

Existem diversas formas de definir o rastreamento entre cenários. Utilizar a mesma linguagem para descrever todos os tipos de cenários facilita o estabelecimento e manutenção do relacionamento entre eles. Pode-se utilizar cenários para relacionar conceitos e eventos identificados entre outros cenários e artefatos.

Modelar a informação de rastreabilidade de forma precisa é fundamental para manter a consistência dos vários modelos. A falta deste tipo de informação constitui um grande problema, pois reduz a utilidade dos modelos e diagramas ao ponto de os desenvolvedores questionarem a utilidade da modelagem, já que há pouco ou nenhum valor em utilizar modelos de software que não representam consistentemente o sistema real [24].

Granularidade da Rastreabilidade

O uso excessivo da rastreabilidade pode resultar em um emaranhado de links de difícil manutenção e utilização. Ou seja, não faz sentido rastrear todos os requisitos com a mesma granularidade [26].

A captura de interdependências entre os artefatos do processo de desenvolvimento é de alta complexidade, e o esforço necessário para manter o rastreamento torna o processo de RR muito custoso na prática. Além disso, o esforço para criar elos ligando requisitos ao código fonte é bastante alto (o estudo realizado

por Heindl e Biffl (2005). [27] apresenta um tempo de cerca de 45 minutos por requisito), porém este tipo de traço é o que fornece as informações mais úteis para a rastreabilidade.

Mesmo que ferramentas tentem automatizar o processo de RR, ainda não conseguem reduzir o esforço necessário para a sua manutenção, tornando a RR uma atividade custosa para implementação prática. A principal razão é que os métodos existentes não levam em consideração a diferença entre requisitos valiosos – do ponto de vista da rastreabilidade – e entre requisitos de baixíssimo valor (HEINDL; BIFFL, 2005).

Egyed et. al.(2005) [28] acreditam que considerações sobre o valor dos artefatos são necessárias, permitindo realizar o planejamento da rastreabilidade de forma sustentável. A motivação da engenharia de software baseada em valores, é que na maior parte das vezes a prática e a pesquisa consideram circunstâncias nas quais o valor dos artefatos não é levado em conta. Assim, cada requisito, caso de uso, objeto, defeito, etc., são tratados com igual importância. O desenvolvimento de software baseado em valor tem como premissa classificar os artefatos de software com alguma medida de valor, considerando sua relevância, e portanto, tratando-os de forma diferente.

No contexto da rastreabilidade de software, pode-se então concluir que nem todos os links de rastreabilidade são igualmente importantes. Por exemplo: requisitos podem ser classificados como "críticos", "importantes" e "de interesse". Mesmo que alguns requisitos "de interesse" sejam implementados, sua corretude não possui a importância de requisitos considerados "críticos".

Há pouco retorno econômico em melhorar o nível de detalhe de links de rastreabilidade além de um certo limite. Algumas aplicações podem não necessitar rastrear certos requisitos com alta granularidade. Portanto, nem todo link é necessário, e gerá-los ou mantê-los representa desperdício quando não são necessários de fato. Da mesma forma, se alguma aplicação necessitar somente de um certo nível de qualidade, gerar ou manter links mais detalhados é um desperdício se tais melhorias não se traduzirem em benefícios [29].

Heindl e Biffl (2005) [27] introduziram o processo de rastreamento de requisitos baseado no valor (Value-Based Requirements Tracing - VBRT), cujo objetivo é identificar elos baseados em requisitos priorizados e assim identificar quais são mais importantes e valiosos em detrimento de outros menos interessantes.

O processo VBRT consiste de cinco passos distintos:

• **Definição de requisitos:** o engenheiro de requisitos analisa a especificação de requisitos do software e identifica requisitos atômicos, atribuindo identificadores únicos a cada um. Após este passo, obtém-se uma lista de requisitos e seus identificadores;

• **Priorização de requisitos:** todos os stakeholders avaliam os requisitos e estimam o valor, risco e esforço de cada. O resultado deste passo é uma lista ordenada de requisitos onde estes são classificados sob três níveis de prioridade;

• **Empacotamento de requisitos:** este passo é opcional e permite identificar grupos de requisitos relacionados, com o objetivo de refiná-los para uma arquitetura intermediária;

• **Linkagem de artefatos:** são estabelecidos os links entre requisitos e artefatos. Requisitos importantes são rastreados mais detalhadamente do que os menos importantes, de acordo com os níveis de intensidade de rastreamento definidos (o VBRT cita três). O resultado deste passo é um plano de rastreabilidade;

• **Avaliação:** pode-se utilizar os elos para os mais variados propósitos, como por exemplo: estimar o impacto de alterações sobre determinados requisitos.

Os resultados no caso de estudo apresentado por Heindl e Biffl (2005) [27] mostraram que o esforço necessário para estabelecer links, considerando o valor dos requisitos, foi 35% do esforço necessário para estabelecer a rastreabilidade completa, sem afetar a qualidade das informações de rastreabilidade. Obviamente, tais resultados dependem de diversos fatores como: documentação e complexidade dos artefatos do sistema.

Dando suporte às conclusões deste artigo, o método proposto por Riebisch e Hubner (2005) [21] considera a manutenção de um número pequeno de rastros como fator-chave para o sucesso do processo de rastreabilidade.

Modelo Formal da Rastreabilidade

Cleland-Huang, Chang e Christensen (2003) [41] apresentam, formalmente, definições para:

- artefatos;
- elos;
- operações sobre artefatos;
- rastro;
- tipos de erro;
- operações sobre requisitos.

Artefatos e elos

Artefatos incluem objetos como requisitos, módulos de código, projetos, casos de teste e seus respectivos resultados, além de várias outras entidades.

Elos de rastreabilidade definem as relações que existem entre os vários artefatos do processo de engenharia de software.

Definição 1: Um artefato é uma parte da informação produzida ou modificada pelo processo de engenharia de software. Este pode tomar uma variedade de formas, incluindo modelos, documentos, código fonte, casos de teste e executáveis. Todos os artefatos, total ou parcialmente, formam os objetos rastreáveis do sistema. Seja A={a1,a2,a3, . . . ,an} o conjunto de todos os artefatos identificados no sistema. Seja R = {r1, r2, r3, . . . , rn} C A o conjunto de todos os artefatos que são requisitos.

Definição 2: Um elo Elo(a,a') representa um relacionamento explícito, definido entre dois artefatos a e a'. Se a e a' estão diretamente ligados como em a → a', onde → indica um elo, então o elo é direto. Se a e a'' estão indiretamente ligados através de um ou mais artefatos intermediários, como em a → a' → a'', então o elo é indireto. Seja a → a'', a fórmula representa um elo indireto de a até a''. Um artefato (como por exemplo a')

através do qual um elo indireto é estabelecido é chamado de intermediário. A direção do elo indica que este é estabelecido a partir do artefato à esquerda (LHS) para aquele à direita (RHS), tal que o artefato LHS apresente uma dependência sobre o artefato RHS.

Para compreender por que a manutenção da rastreabilidade é considerada tão difícil, é necessário primeiro examinar como a infraestrutura de rastreabilidade é afetada por alterações.

Definição 3: Uma alteração C pode ser introduzida sobre um artefato a em uma de duas fases. Uma alteração proposta implica que uma análise de impacto deve ser realizada para determinar como a alteração C afetará o sistema atual. Enquanto que uma alteração implementada implica que todos os artefatos afetados e seus elos relacionados devem ser atualizados para refletir a alteração. Uma alteração proposta não resulta, necessariamente, em uma alteração implementada.

Quando uma alteração C é introduzida em um artefato a, os artefatos restantes do sistema podem ser categorizados, de acordo com seu relacionamento com a, das seguintes formas:

Definição 4 (Artefato ligado): Seja Linked(a) $\subseteq$ A o conjunto de todos os artefatos ligados direta ou indiretamente ao artefato a, através de um ou mais elos de rastreabilidade explicitamente definidos.

Definição 5 (Artefatos relacionados): Seja Related(a) $\subseteq$ A o conjunto de todos os artefatos intrinsecamente relacionadas ao artefato a direta ou indiretamente, tendo ou não um elo de rastreabilidade estabelecido entre eles. Há dois motivos pelos quais Linked(a) $\neq$ Related(a). (1) Não é desejável representar cada relacionamento concebível entre artefatos com um elo, já que isto produziria um "novelo" de elos ingerenciável, muitos dos quais provavelmente nunca seriam utilizados. As decisões sobre quais artefatos relacionados serão ligados devem ser baseadas em estratégias bem definidas, em nível de projeto. Portanto, seja StrategyRelated(a) $\subseteq$ Related(a) o conjunto de artefatos que, de acordo com as estratégias definidas no projeto, devem ser explicitamente ligados a um artefato a. Idealmente, Linked(a) = StrategyRelated(a). (2) Certos elos que devem existir podem estar ausentes devido a falhas na atualização e manutenção da infraestrutura de rastreabilidade. Por esta razão, Linked(a) C StrategyRelated(a) é geralmente o caso.

Definição 6 (Artefatos afetados): Seja Impacted(C,a) $\subseteq$ Related(a) o conjunto de todos os artefatos afetados por uma alteração proposta C sobre o artefato a.

Definição 7 (Artefatos identificados): Seja Identified(C,a) $\subseteq$ A o conjunto de todos os artefatos identificados como afetados por uma alteração proposta C sobre o artefato a. A menos que o desenvolvedor identifique manualmente artefatos externos ao esquema de rastreabilidade, Identified(C,a) deve ser um subconjunto de Linked(a).

Definição 8 (Artefatos atualizados): Seja Updated(C,a, t) $\subseteq$ Identified(C,a) o conjunto de artefatos atualizados, como resultado da implementação da alteração C sobre o artefato a, em um instante de tempo t no qual todos os artefatos e elos relacionados à alteração C estão atualizados.

Além disso, certos artefatos desempenham um papel mais crítico na análise de impacto do que outros. Estes estão situados em posições centrais dentro de um grafo ou árvore de rastreabilidade, e quando inadequadamente mantidos, podem inibir a capacidade de identificar os artefatos corretos em análises de impacto futuras. Estes artefatos são definidos em termos de sua posição dentre uma série de caminhos de rastreabilidade.

Definição 9: Um caminho de rastreabilidade TP é um conjunto ordenado de elos, no qual artefatos indiretamente ligados aj e aj+n estão conectados através de uma série de artefatos intermediários diretamente ligados, tal que TP(aj,aj+n) = {Elo(aj,aj+1),Elo(aj+1,aj+2), ... ,Elo(aj+n-1,aj+n)} representa o caminho de rastreabilidade do artefato aj até aj+n. Um caminho de rastreabilidade de requisitos representa o caso especial para o qual aj é um requisito. O comprimento do menor caminho de rastreabilidade a partir de um artefato para o seu requisito mais próximo especifica o nível deste artefato. Um artefato diretamente ligado a um requisito é dito de nível 1, enquanto que um artefato com um artefato intermediário é dito do nível 2, e assim por diante. Um artefato é dito crítico se está posicionado como um artefato intermediário em múltiplos caminhos de rastreabilidade, pois qualquer falha ao mantê-lo resultará na falha da manutenção de outros artefatos em níveis menores, no mesmo caminho de rastreabilidade.

Conclusões - Problemas e desafios da rastreabilidade

Embora a rastreabilidade seja legalmente requerida na maioria das aplicações de segurança crítica, e seja reconhecida como um componente de muitas iniciativas de melhoria em processos de software, as organizações ainda lutam para implementá-la de modo que ofereça um bom custo-benefício [6]. Padrões aceitos como o CMMI nível 3 e ISO 15504 exigem que práticas básicas de rastreabilidade sejam utilizadas [6, 1], mas as atividades de rastreamento e gerenciamento de requisites podem trazer diversos custos inesperados, que em alguns casos excedem os benefícios [30]. Infelizmente, a tarefa de construir uma matriz de RR leva tempo, é árdua, cara, e sujeita a erros [6]. Por conta disto, com o desenvolvimento do software e sua manutenção, geralmente as informações de rastreabilidade não representam mais a realidade, ou nem existem mais [31].

Os métodos de rastreabilidade existentes focam-se na criação de estruturas de relacionamentos. Embora úteis, o problema é evoluir estas estruturas [32]. Entre as soluções mais aplicadas para encontrar o melhor tradeoff entre o tamanho e a utilidade das informações de rastreabilidade, estão rastrear somente requisitos críticos e reduzir a granularidade (por exemplo, a

Tabela 3 poderia relacionar somente requisitos e classes). Infelizmente, nenhuma destas é satisfatória, já que uma pequena alteração pode afetar diversas partes de um sistema, e a análise de impacto deve contar com o maior número de detalhes possível para que os efeitos colaterais possam ser detectados com precisão. Além disso, muitos esquemas de rastreabilidade fornecem pouco suporte para identificar o impacto de requisitos totalmente novos [33].

Não é necessário discutir o valor potencial que a RR pode trazer ao desenvolvimento de sistemas, afinal, a literatura relata que um dos maiores problemas na tarefa de manutenção está relacionado à compreensão de como o sistema funciona: alguns estudos apontam que de 30% a 60% dos custos nesta fase estão relacionados à descoberta do que o sistema faz [34]. Contudo, a RR é raramente utilizada, já que em muitos casos o custo envolvido não cobre os benefícios. Este alto custo vem: (1) da dificuldade em gerar automaticamente as relações de RR, com uma semântica clara e precisa; (2) da heterogeneidade e do grande número de artefatos que são criados durante o desenvolvimento; e (3) a falta de corretude e completude das relações de rastreabilidade [42].

Na prática, uma variedade de problemas de rastreabilidade geralmente ocorrem pelo uso de métodos informais de RR, falha na cooperação entre as pessoas responsáveis pela coordenação do processo, dificuldade na obtenção das informações necessárias para dar suporte ao processo, e falta de treino em práticas de RR [35]. Além disso, a pressão para reduzir o tempo de entrega e aumentar a produtividade resulta na adaptação de processos para esta realidade de busca por alto desempenho, que geralmente sacrifica atividades não-produtivas. O estabelecimento e manutenção da RR, bem como a manutenção da consistência entre os artefatos de software, são algumas destas atividades custosas e tediosas freqüentemente negligenciadas [13].

Como resultado dos problemas de manutenção, um grande número de pesquisadores vem investigando o uso da recuperação automática da rastreabilidade (RAR), utilizando métodos de recuperação de informações para gerar elos dinamicamente, tais como modelos de espaço vetorial, indexação semântica, ou modelos probabilísticos de rede. Mesmo que não seja ainda uma bala de prata, esta técnica vem sendo cada vez mais reconhecida pela indústria como uma solução potencial para a rastreabildade [6]. Porém, não existe processo de recuperação da rastreabilidade completamente automático. Todos incluem um componente humano, pois as decisões (de aceitação ou rejeição) sobre elos recuperados devem ser tomadas pelo usuário. Além disso, uma vez que os elos de rastreabilidade sejam recuperados, devem ser mantidos de forma consistente, a fim de ser utilizados em outros processos e tarefas [11]. Portanto, um problema desafiador a ser pesquisado está em assegurar uma qualidade aceitável nos elos, sem pressionar demasiadamente os desenvolvedores [1].

O problema da manutenção da rastreabilidade persiste, bem como a resistência dos desenvolvedores em aplicá-la a seus projetos. Entre os diversos fatores que geram seu alto custo, está a dificuldade em gerar elos automaticamente, com uma semântica clara e precisa [42]. Ao ignorar a semântica das informações de RR, limita-se a capacidade de processos automáticos de dedução [36].

Enquanto a simplicidade é a principal vantagem dos métodos tradicionais, eles são pouco úteis na representação de todas as informações relevantes [11]. Além disso, os esforços dos desenvolvedores são retardados pela falta de suporte ferramental, e há uma percepção geral de que o custo necessário para manter uma matriz de rastreabilidade de requisitos é excessivo, em relação aos seus benefícios [6, 30].

Todos os estudos interessados na aplicação de métodos para a RAR mostram que estes não são capazes de recuperar todos os elos corretos, sem também recuperar muitos falsos positivos, que devem ser descartados pelo engenheiro de software. A baixa precisão faz deste método uma tarefa tediosa, já que o engenheiro tem que gastar muito mais tempo para descartar falsos positivos do que rastreando elos corretos. Embora seja possível melhorar a performance destes métodos, eles ainda estão distantes de tornar viável a solução para o problema da recuperação de elos [16]. Mesmo assim, esta solução vem sendo cada vez mais reconhecida pela indústria como uma solução potencial para o problema da rastreabildade [6]. Em contrapartida, Hayes e Dekhtyar [37] estudaram os resultados produzidos por analistas, e descobriram que as decisões tomadas sobre a validade ou invalidade dos elos recuperados nem sempre são corretas, e podem até piorar os resultados: elos corretos acabam sendo descartados, e elos incorretos mantidos.

A maioria das ferramentas e métodos de rastreabilidade suportam a identificação de artefatos afetados, quando existe um conjunto preciso de elos. Contudo, grande parte destes não fornece nenhum tipo de suporte para assegurar que os artefatos e elos relacionados, afetados por alguma alteração, sejam atualizados rapidamente. A atualização é tipicamente manual, e como resultado, surgem erros. Estes podem afetar a capacidade de, futuramente, identificar os impactos de novas alterações [41].

Mesmo quando uma organização implementa um processo de rastreabilidade, sua eficiência é geralmente influenciada negativamente por dificuldades no estabelecimento, manutenção e completa utilização de elos de rastreabilidade. Entre outros problemas, os desenvolvedores frequentemente falham em manter os elos devido às dificuldades inerentes na coordenação de membros da equipe, além da impressão de que o custo de implementar a RR excede suas vantagens [10]. Não obstante, a falta de percepção do benefício por parte dos desenvolvedores faz com que as tarefas de rastreabilidade tenham baixíssima prioridade, o que resulta em dados incompletos e incorretos [38]. Assim, um dos maiores problemas da rastreabilidade é integrá-la ao processo de desenvolvimento de forma que forneça benefícios tangíveis e imediatos.

O suporte ferramental eficiente para a rastreabilidade é um fator-chave para torná-la viável. Devem ser desenvolvidos métodos nos quais a rastreabilidade seja mantida automaticamente, enquanto o usuário continua seu trabalho normal. Por exemplo, um usuário pode dizer "Estou trabalhando na satisfação deste requisito", e a partir daí, tudo o que ele tocar é rastreado para aquele requisito [42]. Além disso, para reduzir a quantidade de recursos e tempo necessários para rastrear requisitos, é necessário embutir as informações de cobertura dos requisitos no próprio software que irá atendê-los [30].

Atualmente, é muito difícil gerenciar um conjunto de requisitos de tal forma que seja possível visualizar onde estes são atendidos. Informações suficientes para isto querem que uma matriz de rastreabilidade seja mantida, estabelecendo relações entre requisitos, casos de teste, decisões de projeto, etc. [18].

Além disso, estabelecer e manter os traços traz uma grande sobrecarga sobre os engenheiros de software. Existem ferramentas que fornecem a infraestrutura para gerenciá-los, contudo estas não libertam o usuário de identificar links, ou de assegurar sua validade com o passar do tempo [29]. Consequentemente, a RR é raramente estabelecida em ambientes industriais, onde sistemas são documentados por grandes coleções de artefatos, geralmente expressos em linguagem natural [12].

A falta de percepção do benefício faz com que as tarefas de rastreabilidade tenham baixíssima prioridade, o que resulta em dados incompletos e incorretos. Porém, esta impressão revela-se falsa, pois com a apropriada gerência de um processo de RR, a equipe de desenvolvimento torna-se capaz de [28]:

- demonstrar que todos os requisitos foram testados;
- identificar quais partes genéricas de um software podem ser
- alteradas para satisfazer outros requisitos;
- justificar decisões de projeto acerca da criação de variantes de um produto;
- medir o progresso do desenvolvimento;
- entre outros.

Outro grande problema da rastreabilidade, portanto, é integrá-la ao processo de desenvolvimento de forma que forneça benefícios tangíveis e imediatos.

O trabalho de Daneva (2003) [40] apresenta a seguinte lição: assegurar que as políticas de rastreabilidade sejam obedecidas. Negligenciá-las, especialmente na manutenção de informações de rastreabilidade, trouxe problemas de gerenciamento de informação em seis de treze projetos acompanhados no trabalho citado. A atualização esporádica de requisitos fez com que a substancial quantidade de informação não pôde ser mantida, aumentando o custo e o tempo de um processo já caro.

Tantas limitações praticamente impossibilitam o uso da rastreabilidade fácil e eficientemente. ●

Referências

[1] NEUMULLER, C.; GRUNBACHER, P. Automating Software Traceability in Very Small Companies: a case study and lessons learned. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE, 21., 2006, Washington, DC, USA. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.145–156.

[2] BERG, K. van den; CONEJERO, J. M.; HERN´ANDEZ, J. Analysis of crosscutting across software development phases based on traceability. In: INTERNATIONAL WORKSHOP ON EARLY ASPECTS AT ICSE, EA, 2006. Proceedings... New York: ACM Press, 2006. p.43–50.

[3] ALMEIDA, J. P.; ECK, P. van; IACOB, M.-E. Requirements Traceability and Transformation Conformance in Model-Driven Development. In: IEEE INTERNATIONAL ENTERPRISE DISTRIBUTED OBJECT COMPUTING CONFERENCE, EDOC, 10., 2006, Hong Kong, China. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.355–366.

[4] ANTONIOL, G.; CIMITILE, A.; CASAZZA, G. Traceability Recovery by Modeling Programmer Behavior. In: WORKING CONFERENCE ON REVERSE ENGINEERING, WCRE, 7., 2000, Brisbane, Australia. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2000. p.240.

[5] FLETCHER, J.; CLELAND-HUANG, J. Softgoal Traceability Patterns. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING, ISSRE, 17., 2006, Raleigh, North Carolina. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.363–374.

[6] CLELAND-HUANG, J. Just Enough Requirements Traceability. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 30., 2006, Chicago, Illinois. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.41–42.

[7] HEINDL, M.; BIFFL, S. Risk management with enhanced tracing of requirements rationale in highly distributed projects. In: INTERNATIONAL WORKSHOP ON GLOBAL SOFTWARE DEVELOPMENT FOR THE PRACTITIONER, GSD, 2006, Shanghai, China. Proceedings... New York: ACM Press, 2006. p.20–26.

[8] MALETIC, J. I.; COLLARD, M. L.; SIMOES, B. An XML based approach to support the evolution of model-to-model traceability links. In: INTERNATIONAL WORKSHOP ON TRACEABILITY IN EMERGING FORMS OF SOFTWARE ENGINEERING, TEFSE, 3., 2005, Long Beach, CA. Proceedings... New York: ACM Press, 2005. p.67–72.

[9] RAMESH, B.; JARKE, M. Toward Reference Models for Requirements Traceability. IEEE Trans. Softw. Eng., Piscataway, NJ, USA, v.27, n.1, p.58–93, 2001.

[10] CLELAND-HUANG, J.; ZEMONT, G.; LUKASIK, W. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In: REQUIREMENTS ENGINEERING CONFERENCE, RE, 12., 2004, Kyoto, Japan. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2004. p.230–239.

[11] MARCUS, A.; XIE, X.; POSHYVANYK, D. When and how to visualize traceability links? In: INTERNATIONAL WORKSHOP ON TRACEABILITY IN EMERGING FORMS OF SOFTWARE ENGINEERING, TEFSE, 3., 2005. Proceedings... New York: ACM Press, 2005. p.56–61.

[12] SPANOUDAKIS, G. Plausible and adaptive requirement traceability structures. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, SEKE, 14., 2002, New York:. Proceedings... ACM Press, 2002. p.135–142.

[13] PENTA, M. D.; GRADARA, S.; ANTONIOL, G. Traceability Recovery in RAD Software Systems. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, IWPC, 10., 2002, Washington, DC, USA. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2002. p.207.

[14] LUCIA, A. D. et al. Improving Comprehensibility of Source Code via Traceability Information: a controlled experiment. In: IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC, 14., 2006, Athens, Greece. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.317–326.

Referências - Continuação

[15] SETTIMI, R. et al. Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, IWPSE, 7., 2004, Kyoto, Japan. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2004. p.49–54.

[16] LUCIA, A. D. et al. Can Information Retrieval Techniques Effectively Support Traceability Link Recovery? In: IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION,ICPC, 14., 2006, Washington, DC, USA. Proceedings... [S.l.: s.n.], 2006. p.307–316.

[17] EGYED, A. et al. Determining the cost-quality trade-off for automated software traceability. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE, 20., 2005. Proceedings... New York: ACM Press, 2005. p.360–363.

[18] LORMANS, M.; DEURSEN, A. V. Can LSI help Reconstructing Requirements Traceability in Design and Test? In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, CSMR, 2006, Washington, DC, USA. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.47–56.

[19] CLELAND-HUANG, J. et al. Goal-centric traceability for managing non-functional requirements. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 27., 2005. Proceedings... [S.l.: s.n.], 2005. p.362–371.

[20] CLELAND-HUANG, J.; ZEMONT, G.; LUKASIK, W. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In: REQUIREMENTS ENGINEERING CONFERENCE, RE, 12., 2004, Kyoto, Japan. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2004. p.230–239.

[21] RIEBISCH, M.; HUBNER, M. Traceability-Driven Model Refinement for Test Case Generation. In: IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS, ECBS, 12., 2005, Greenbelt, Maryland. Proceedings. . . Los Alamitos, CA: IEEE Computer Society, 2005. p.113–120.

[22] PAECH, B.; KERKOW, D. Non-Functional Requirements Engineering – Quality is essential. In: INTERNATIONAL WORKSHOP ON REQUIREMENTS ENGINEERING: FOUNDATION FOR SOFTWARE QUALITY, REFSQ, 2004, London, UK. Proceedings... Berlin: Springer-Verlag, 2004. p.27–40.

[23] NASLAVSKY, L. et al. Using scenarios to support traceability. In: INTERNATIONAL WORKSHOP ON TRACEABILITY IN EMERGING FORMS OF SOFTWARE ENGINEERING, TEFSE, 3., 2005. Proceedings... New York: ACM Press, 2005. p.25–30.

[24] EGYED, A. A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Trans. Softw. Eng., Piscataway, NJ, USA, v.29, n.2003, p.116–132, 2003.

[25] EGYED, A. A scenario-driven approach to traceability. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 23., 2001, Washington, DC, USA. Proceedings. . . Los Alamitos, CA: IEEE Computer Society, 2001. p.123–132.

[26] DOMGES, R.; POHL, K. Adapting traceability environments to project-specific needs. Communications of the ACM, New York:, v.41, n.12, p.54–62, 1998.

[27] HEINDL, M.; BIFFL, S. A case study on value-based requirements tracing. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC, 10., 2005, Lisbon, Portugal. Proceedings... New York: ACM Press, 2005. p.60–69.

[28] EGYED, A. et al. Determining the cost-quality trade-off for automated software traceability. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE, 20., 2005. Proceedings... New York: ACM Press, 2005. p.360–363.

[29] EGYED, A. et al. A value-based approach for understanding cost-benefit tradeoffs during automated software traceability, TEFSE, 3. In: INTERNATIONAL WORKSHOP ON TRACEABILITY IN EMERGING FORMS OF SOFTWARE ENGINEERING, 2005, Long Beach, CA. Proceedings... New York: ACM Press, 2005. p.2–7.

[30] DELGADO, S. Next-Generation Techniques for Tracking Design Requirements Coverage in Automatic Test Software Development. In: IEEE SYSTEMS READINESS TECHNOLOGY CONFERENCE, AUTOTESTCON, 2006. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.806–812.

[31] LUCIA, A. D.; OLIVETO, R.; SGUEGLIA, P. Incremental Approach and User Feedbacks: a silver bullet for traceability recovery. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM, 22., 2006, Philadelphia, Pennsylvania. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.299–309.

[32] MURTA, L. G. P.; HOEK, A. van der; WERNER, C. M. L. ArchTrace: policy-based support for managing evolving architecture-to-implementation traceability links. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE, 21., 2006, Washington, DC, USA. Proceedings... Los Alamitos,
CA: IEEE Computer Society, 2006. p.135–144.

[33] CLELAND-HUANG, J. et al. Automating Speculative Queries through Event-Based Requirements Traceability. In: IEEE JOINT INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, RE, 2002, Essen, Germany. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2002. p.289–298.

[34] TRYGGESETH, E.; NYTRO, O. Dynamic Traceability Links Supported by a System Architecture Description. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM, 1997, Bari, Italy. Proceedings... Los Alamitos, CA: IEEE Computer Society, 1997. p.180–187.

[35] SALEM, A. Improving Software Quality through Requirements Traceability Models. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER SYSTEMS AND APPLICATIONS, 2006, Washington, DC, USA. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.1159–1162.

[36] RAMESH, B. Factors influencing requirements traceability practice. Communications of the ACM, New York:, v.41, n.12, p.37–44, 1998.

[37] HAYES, J. H.; DEKHTYAR, A. Humans in the traceability loop: can't live with 'em, can't live without 'em. In: INTERNATIONAL WORKSHOP ON TRACEABILITY IN EMERGING FORMS OF SOFTWARE ENGINEERING, TEFSE, 3., 2005. Proceedings... New York: ACM Press, 2005. p.20–23.

[38] ARKLEY, P.; RIDDLE, S. Overcoming the Traceability Benefit Problem. In: IEEE INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, RE, 13., 2005, Washington, DC, USA. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2005. p.385–389.

[39] LORMANS, M.; DEURSEN, A. van. Reconstructing requirements coverage views from design and test using traceability recovery via LSI. In: INTERNATIONAL WORKSHOP ON TRACEABILITY IN EMERGING FORMS OF SOFTWARE ENGINEERING, TEFSE, 3., 2005, Long Beach, California. Proceedings... New York: ACM Press, 2005. p.37–42.

[40] DANEVA, M. Lessons Learnt from Five Years of Experience in ERP Requirements Engineering. In: IEEE INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, RE, 11., 2003, Monterey Bay, CA. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2003. p.45.

[41] CLELAND-HUANG, J.; CHANG, C. K.; CHRISTENSEN, M. Event-Based Traceability for Managing Evolutionary Change. IEEE Trans. Softw. Eng., Piscataway, NJ, USA, v.29, n.9, p.796–810, 2003.

[42] CLELAND-HUANG, J. Requirements Traceability - When and How does it Deliver more than it Costs? In: IEEE INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE, RE, 14., 2006, Minneapolis. Proceedings... Los Alamitos, CA: IEEE Computer Society, 2006. p.323.

UML – Diagrama de Classes

Encontrando classes e desenhando seu diagrama – Parte II

De que se trata o artigo?
Este artigo tem por objetivo apresentar as regras para se modelar um diagrama de classes, partindo da análise prática dos requisitos de um estudo de caso.

Para que serve?
Fornecer aos desenvolvedores ou estudantes da área de sistemas uma linha de entendimento com o intuito de orientá-los a modelar seus diagramas de classes.

Em que situação o tema é útil?
Para quem ainda não modelou classes, ou para quem tem experiência e quer revisar a sintaxe permitida nesse tipo de diagrama.

Ana Cristina Melo
informatica@anacristinamelo.com.br
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.

Vamos dar continuidade ao artigo anterior, no qual demonstramos a partir de um pequeno estudo de caso como é feita a modelagem de um diagrama de classes, aproveitando esse passo a passo para apresentar as regras desse diagrama.

Nessa segunda parte, veremos como refinar um diagrama de classes, apresentando algumas definições relevantes como escopo, restrições, os relacionamentos de generalização e agregação, além de algumas classes especiais como classe de enumeração, abstrata e de associação.

Refinando as classes

No último artigo, chegamos à primeira versão do nosso modelo de classes, reproduzido na **Figura 1**. Para entender melhor nosso estudo de caso, vamos apresentar também os requisitos que derem origem a esse modelo. Confira na **Tabela 1**.

Terminamos o artigo anterior chamando a atenção para alguns tipos de dados que não são os tipos básicos, como inteiro (integer), float (double), booleano (boolean), data e hora (date). Esses tipos diferentes aparecem no atributo sexo da classe Paciente e no atributo tipo da classe Telefone. São os tipos enumerados.

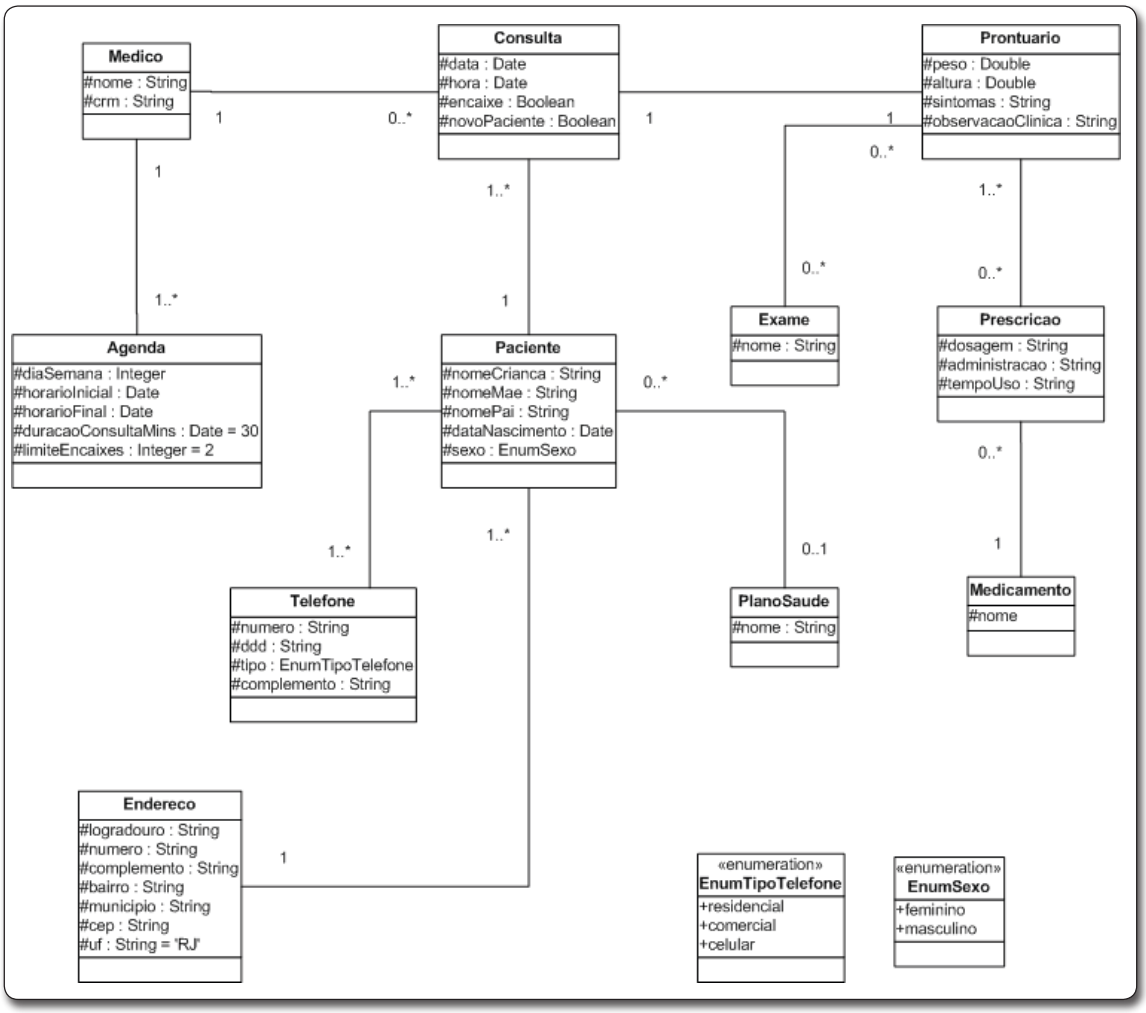


Figura 1. Primeira versão do diagrama de classes do estudo de caso do Sistema de Consultas Médicas

Sistema de Consultas Médicas
Dr. Monteiro contratou uma empresa para informatizar seu consultório.
Dr. Monteiro é pediatra e atende particular ou por plano de saúde. As consultas na agenda serão marcadas pela secretária. As consultas ocorrem de meia em meia hora, em horários distintos a cada dia da semana. Só são permitidos dois encaixes por dia. Caso o paciente seja novo, deve-se registrar apenas o nome da criança, do responsável, telefone de contato e tipo de plano.
Para cada paciente é preciso manter: nome da criança, nome dos pais, data de nascimento, endereço completo (incluindo uf), telefone residencial e celular (indicando a quem pertence), sexo, prontuário de cada consulta, histórico de peso e altura. O receituário deve ser emitido pelo sistema com as prescrições, nome e CRM do médico.
O prontuário de cada consulta deve conter a descrição dos sintomas, resultado da observação clínica, os medicamentos prescritos (com dosagem e administração) e exames pedidos. Associado a cada cliente deve estar registrado seu plano de saúde ou se é particular.

Tabela 1. Requisitos do Estudo de Caso Sistema de Consultas Médicas

O tipo enumerado é na realidade uma classe de enumeração, que é uma das classes estereotipadas pré-definidas pela UML.

Uma classe de enumeração é identificada com o estereótipo «enumeration» e é utilizada para representar constantes que unificam conteúdos não-mutáveis, que precisam ser referenciados com freqüência. Essa é a mesma base dos tipos enumerados. É o que permite, por exemplo, que ao se ter um atributo tipoTelefone, possamos testar algo assim:

```
if tipoTelefone = residencial then
```

Isso garante legibilidade no código, pois é muito mais claro se ler o trecho anterior em vez de:

```
if tipoTelefone = 0 then
```

Além da legibilidade, garantimos rapidez e segurança na manutenção. Se precisarmos alterar o valor que está por trás de uma constante, temos um único lugar onde modificar: a classe de enumeração.

Então, para criar uma classe desse tipo, coloca-se o estereótipo «enumeration» na primeira linha do compartimento do nome da classe. A lista das constantes vai para o compartimento dos atributos, apenas com o nome que os identifica. Ao se criar uma classe de negócios, basta colocar a classe de enumeração como tipo de dados, como acontece, por exemplo, na classe Telefone:

```
tipo : EnumTipoTelefone
```

Dando continuidade ao refinamento desse modelo de classes, podemos olhar mais atentamente para os relacionamentos. Veja que temos apenas o relacionamento de associação ligando as classes.

E aproveitamento o tema, devemos ressaltar que não existe relacionamento entre uma classe que use uma classe de enumeração como tipo de dados e a própria classe de enumeração. Isso se deve, pois a classe de enumeração é apenas a representação de um tipo de dados.

Utilizando adornos na associação

Repare que no relacionamento de associação aparecem várias multiplicidades (ex: 1, 0..1, 0..*). Se tentarmos entender o que expressa cada relacionamento, provavelmente conseguiremos sem grande esforço.

Por exemplo: é fácil compreender que entre Paciente e Telefone se lê "Paciente possui Telefone". Mas, entre Prontuario e Exame pode ficar a pergunta: "o que significa"?

Nesse caso podemos utilizar **adornos** de característica opcional, mas que em alguns casos melhoram a compreensão do modelo. Para o caso citado acima, a UML oferece o **nome de associação**. Assim, podemos nomear a associação, identificando-a como "solicitação". Veja como fica na **Figura 2**.

Desta forma, podemos ler que "Em um prontuário há a solicitação de nenhum ou vários exames" ou "Um exame pode ser solicitado em nenhum ou vários prontuários".

Graficamente, o nome da associação é colocado no centro do relacionamento, acompanhado de um pequeno triângulo que indica a direção na qual o nome deve ser lido. Veja um exemplo na **Figura 2**, no relacionamento entre Voo e Cidade, como o nome de associação se torna útil e indispensável quando temos dois ou mais relacionamentos entre as mesmas classes.

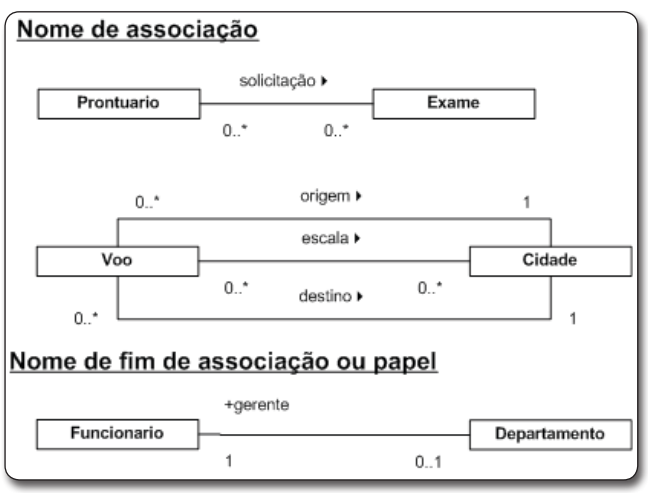


Figura 2. Outros recursos no relacionamento de associação

Além do nome da associação, temos também o nome do **papel** que a classe representa na associação. Na versão atual da UML, ele é nomeado como **nome do fim da associação**, mas ainda prefiro a nomenclatura antiga que era **nome do papel**. Revela e explica muito mais. Veja na **Figura 2** o relacionamento entre Funcionario e Departamento.

Sem identificar o papel que esse funcionário exerce nesse relacionamento ficaria difícil adivinhá-lo.

Vimos os adornos inerentes ao relacionamento de associação, porém a UML nos disponibiliza outros tipos de relacionamentos, tão importantes quanto este.

Existe na UML um caso particular do relacionamento de associação que é a **agregação**.

A agregação representa um relacionamento "todo-parte", ou seja, uma das classes representa o todo, enquanto outra representa a parte.

A associação estabelece uma ligação entre as classes, mantendo a independência de vida das mesmas. Por exemplo, ao ligar por associação a classe Prontuário à classe Exame, crio um relacionamento que representará os exames solicitados naquela consulta. Mas essas classes existem por si só. Posso acabar com o relacionamento, sem prejudicar a existência de cada classe.

No caso da agregação, uma classe é enxergada como parte de outra, indicando na fase de implementação, que para se ter acesso ao comportamento dos objetos-partes devemos fazê-lo por meio dos objetos-todo.

Na agregação desenhamos um losango (diamante) junto à classe que representa o "todo". Veja na **Figura 3**.

Há uma variação mais poderosa da agregação, indicando que a classe parte pertencerá só e somente só à classe todo, sendo esta responsável pela criação e destruição de suas partes. É o relacionamento de **composição** ou **agregação por composição**. Graficamente, basta preencher o losango (diamante) junto à classe-todo. Veja **Figura 4**.

Algumas dicas para se identificar um relacionamento como uma agregação ou composição:

- Se você puder dizer no relacionamento de associação que uma classe é "parte de", "contém" ou "pertence à" outra classe, então estamos diante de uma agregação ou composição;
- Se as partes não podem ser divididas, estamos diante de uma composição. Caso contrário, temos uma agregação.
- Se o ciclo de vida das partes estão atreladas ao ciclo de vida do todo, temos uma composição;
- Se a multiplicidade do todo é 1 ou 0..1, temos uma composição;
- Se a multiplicidade do todo puder ser maior do que 1, temos uma agregação.

Vamos observar todos os relacionamentos no nosso modelo de classes e verificar se algum deles pediria um caso mais particular.

Entre as classes Medico e Agenda não podemos enxergar a agenda sem a existência do médico. Quem melhor conhece sua agenda do que o próprio Médico? Ou seja, nesse caso, do que o objeto Medico? E considerando que cada agenda pertence a um único médico, podemos transformar esse relacionamento numa composição. Repare que aplicamos tranquilamente a frase: "Uma agenda pertence exatamente a um médico".

Figura 3. Exemplo gráfico de um relacionamento de agregação

Figura 4. Exemplo gráfico de um relacionamento de agregação por composição

Entre Paciente e Telefone, é fácil dizermos que um telefone pertence a um paciente. O mesmo ocorre entre Paciente e Endereco. Considerando que tanto telefone quanto endereço pode pertencer a mais de um paciente, aqui podemos alterar o relacionamento para uma agregação.

Já não podemos fazer isso com Medico e Consulta, pois a consulta "é feita com" um médico, e não a consulta "é parte" do médico.

Entre Consulta e Prontuario podemos dizer que a consulta contém um prontuário, ou até mesmo, o prontuário "é parte" da consulta, o que vale a alteração para uma composição.

Vejamos então o relacionamento entre Prescricao e Medicamento. Considerando que nessa solução, a partir do cadastramento de um medicamento, se cadastra uma lista de prescrições possíveis, temos uma agregação por composição entre as classes Medicamento e Prescricao.

Avançando para um outro recurso da UML, temos um outro caso envolvendo um conjunto de três classes.

Repare que a classe Consulta não existe sem as classes Medico e Paciente. Isso significa que a classe Consulta só existe porque existe o relacionamento. Desta forma, estamos diante de uma **classe de associação**, ou seja, uma classe que é fruto de um relacionamento, e que possui atributos próprios. Vejamos como fica esse relacionamento na **Figura 5**.

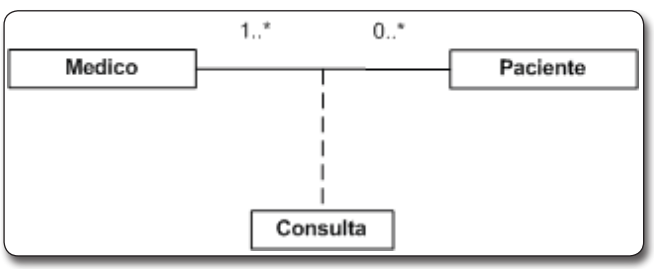


Figura 5. Exemplo gráfico de um relacionamento de agregação por composição

Lemos esse relacionamento da seguinte forma: "um médico pode consultar nenhum ou vários pacientes. Um paciente se consulta com um ou vários médicos".

Vamos observar um outro conceito usado na modelagem de classe. É o conceito de **escopo**. Podemos dizer que um elemento (atributo ou operação) tem escopo de classe ou escopo de instância, ou ainda, em outras palavras, que o elemento é estático ou não estático.

Um elemento estático ou com escopo de classe indica que o mesmo não depende que se instancie um objeto para obter seu valor. O mesmo é obtido considerando-se apenas a classe.

Por exemplo: nos requisitos desse estudo de caso é citado que as consultas ocorrem de meia em meia hora. Ao se apresentar a agenda de um determinado dia, é preciso que o sistema gere automaticamente os horários com intervalos de meia em meia hora e a partir daí verifique quais horários já estão preenchidos. Para que o sistema possa gerar essa agenda, é preciso consultar de algum lugar esse valor de meia hora, normalmente em forma de constante. Essa constante fica atrelada à classe, sendo acessada diretamente sem a necessidade de instanciar um objeto.

Supondo uma operação estática, sua execução agirá sobre toda a coleção de instâncias, sendo indiferente conhecer ou não um objeto.

Por exemplo: numa classe Cliente, a operação obterQtdClientesAtivos é classificada como estática ou de classe, pois de nada adianta instanciar um objeto, se será preciso varrer todos os objetos persistidos (gravados) para se obter essa quantidade.

A representação gráfica de um elemento estático é mostrá-lo sublinhado.

Um elemento não-estático ou com escopo de instância indica que o mesmo depende do valor de um objeto.

Por exemplo: para se saber o valor de um atributo dataNascimento de uma classe Paciente, é preciso antes identificar quem é o paciente. Da mesma forma, se desejamos saber a idade desse paciente, chamamos uma operação obterIdade, que para ser executada, dependerá do objeto e do valor específico da data de nascimento.

Usando esse conceito de escopo, alteramos para escopo de classe (estático) os atributos duracaoConsultaMes e limiteEncaixes da classe Agenda.

Aplicando todas essas alterações, vejamos como ficou nosso modelo de classes na **Figura 6**.

Outras definições do Diagrama de Classes

Vimos os relacionamentos de associação, agregação e agregação por composição. Contudo, há um relacionamento tão importante quanto os citados que é o relacionamento de generalização/especialização, baseado no conceito de herança.

O reaproveitamento é uma das principais metas da orientação a objetos. Para isso, uma das formas é refinar um modelo buscando elementos comuns, levando-os a um único lugar que centralize as regras, o tratamento e o local de manutenção. No caso da herança, esse local centralizado é a classe pai ou superclasse.

Vamos considerar o pedaço de um modelo de classes apresentado na **Figura 7**. Repare que as classes Funcionario e Cliente possuem em comum os atributos nome, cpf, sexo e dataNascimento; e a operação obterIdade. Além disso, ambas as classes se relacionam com a classe Endereco. Assim, é fácil percebermos que esses atributos são os mesmos porque são características de uma pessoa. Desta forma, podemos criar uma superclasse (ou classe pai) centralizando os atributos, operações e relacionamentos comuns. Veja como ficaria na **Figura 8**.

Para se obter esse refino, colocamos na classe pai todos os atributos e operações comuns. Tiramos os relacionamentos comuns das classes filhas e ligamos apenas à classe pai. Nas

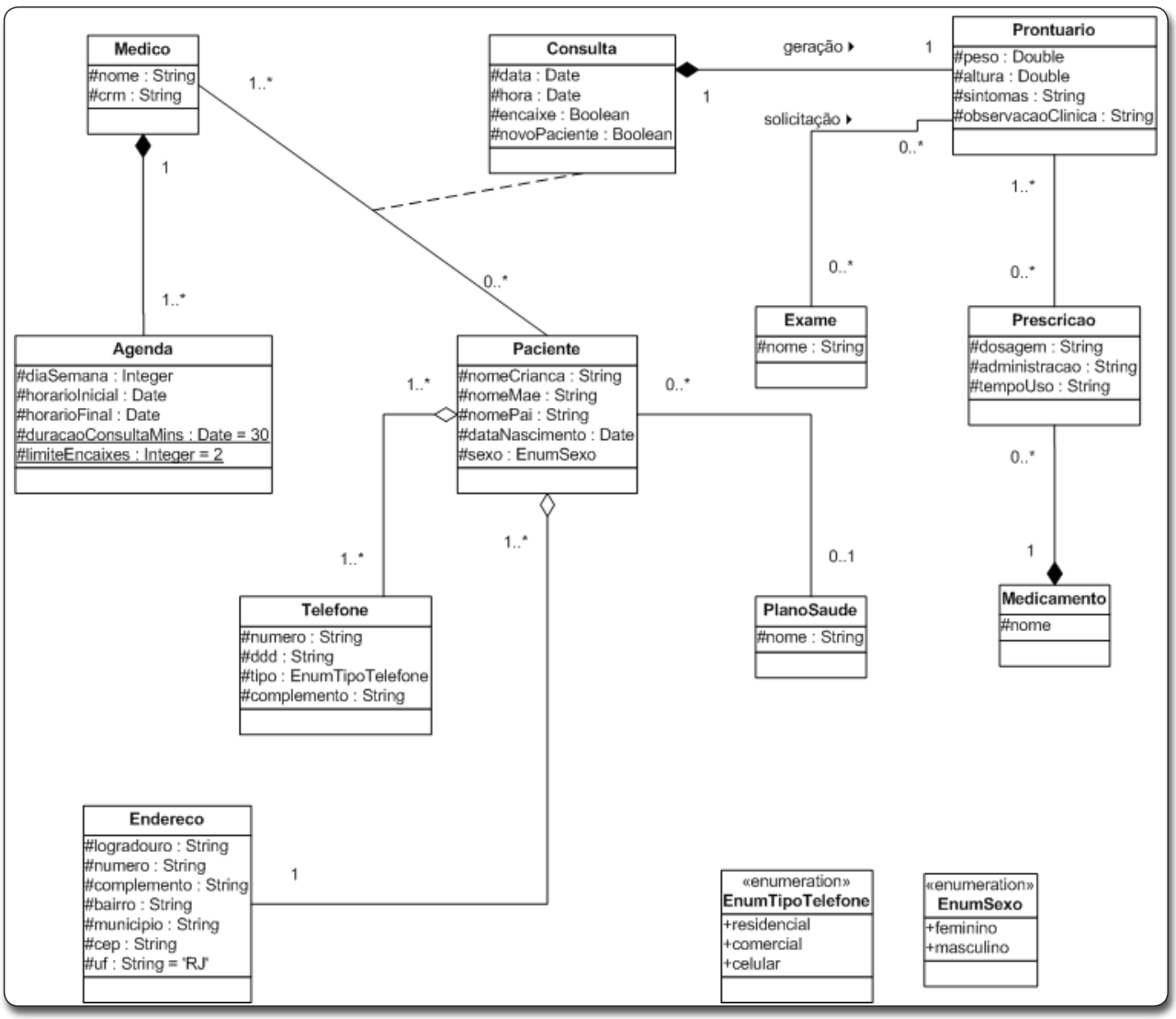


Figura 6. Modelo de classes refinado do Sistema de Consultas Médicas

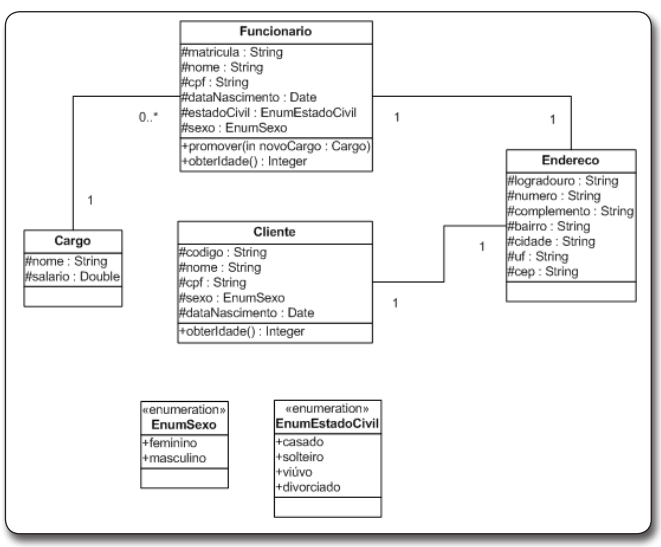


Figura 7. Trecho de um modelo de classes antes de um processo de refinamento

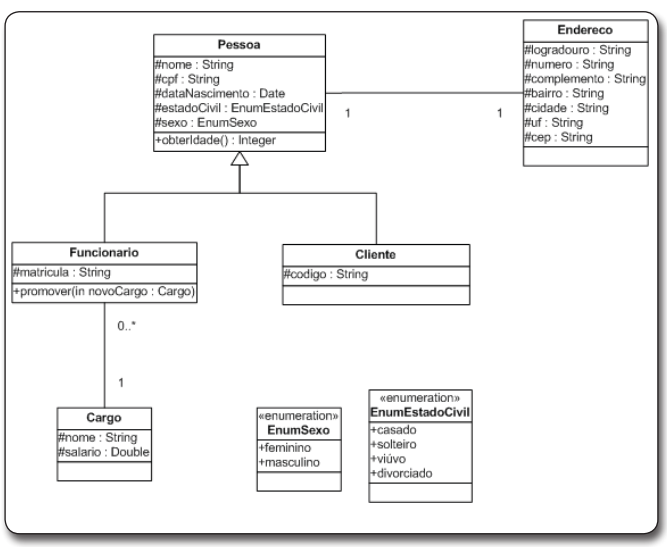


Figura 8. Trecho de um modelo de classes após refinamento e aplicação do relacionamento de generalização/especialização

classes filhas, eliminamos tudo o que "subiu", deixando apenas o que é específico a cada um. Isso garantirá que ao se acessar uma classe filha, tenhamos acesso não só aos seus próprios elementos como a todos os elementos (atributos e operações) e relacionamentos de suas classes ascendentes.

Observe que o atributo estadoCivil está associado somente à classe Funcionario, mas por ser uma característica de Pessoa também é levado para a classe pai. Não importa que a classe Cliente herde esse atributo e não use. Isso não gera nenhuma exceção.

Um outro recurso da UML é o conceito de **classe abstrata**. Vamos tomar o exemplo anterior do relacionamento de generalização/especialização. O que desejamos é centralizar as regras e os elementos. Contudo, não tem sentido imaginarmos que em algum momento fôssemos persistir um objeto Pessoa, ou seja, armazenar em arquivo os objetos Pessoa. Só esperamos as persistências de Funcionario e Cliente. Nem sempre isso é uma verdade. Supondo que a classe pai fosse a classe Funcionario e a classe filha Vendedor, então, nesse caso, seria natural esperar objetos persistentes das duas classes.

Contudo, se uma classe pai for definida como não tendo instâncias diretas, então ela deve ser definida como classe abstrata.

Graficamente, demonstra-se uma classe como abstrata, colocando o estereótipo «abstract» ou colocando-se o nome da classe em itálico.

Assim, no modelo anterior, usando uma ferramenta case, o nome da classe pessoa apareceria em itálico ou com o estereótipo logo acima.

Conclusão

Como conclusão, podemos perceber que os diagramas de classes nos oferecem diversos recursos. Para o analista, ele chegará facilmente a uma primeira solução, que não contemplará todas as possibilidades. Porém o mais importante é não encerrar aí sua modelagem, e sim sempre refiná-la, até se obter um modelo mais completo.

Referências

MELO, Ana Cristina. Desenvolvendo aplicações com UML 2.0: do conceitual à implementação. Rio de Janeiro: Brasport, 2004.

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

Desenvolvimento Orientado a Componentes

Um exemplo prático

Rodrigo Henrique Severiano
rodhensev@gmail.com.br
Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, Bacharel em Ciência da Computação pela UFJF e Técnico em Informática Industrial pelo CEFET-MG, Líder da Equipe de Arquitetura/Padrões JEE da Agência Nacional de Saúde, Analista de Sistemas da Agência Nacional de Saúde.

Regina Maria Maciel Braga
regina@acessa.com
Doutora e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, Bacharel em Informática pela UFJF, Professor Adjunto da UFJF, tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software e Banco de Dados, atuando principalmente nos seguintes temas: DBC, Orientação a Objetos, Ontologia e Integração de Dados.

Marco Antônio Pereira Araújo
maraujo@devmedia.com.br
Doutor 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?
Este artigo detalha os benefícios e vantagens de se aplicar o desenvolvimento baseado em componentes como técnica a ser utilizada para o desenvolvimento de aplicações.

Para que serve?
Auxilia no desenvolvimento de aplicações com mais agilidade, através do reuso de componentes pré-existentes e com maior grau de confiabilidade e qualidade.

Em que situação o tema é útil?
Facilitar a construção de produtos que compartilham características similares com outras aplicações.

A competição entre as empresas está se tornando cada vez mais intensa e globalizada. Freqüentemente, elas competem com pequenas diferenças em novos serviços, que são rapidamente introduzidos no mercado. Melhorar o desempenho do negócio comumente significa que as empresas devem radicalmente melhorar seu desempenho no processo de desenvolvimento de software. Assim sendo, o sucesso do negócio é medido através da produção de software e de serviços de forma mais ágil, melhor e mais barata. A agilidade significa que o software deve ser disponibilizado em tempo hábil, para alcançar uma fatia maior do mercado. Neste contexto, "melhor" significa que o software deve atender aos requisitos do negócio e sem falhas relevantes, e "mais barato" quer dizer que o software deve ser produzido e mantido a baixo custo (Jacobson et al., 1997).

O desenvolvimento de software tradicional tem apresentado diversos problemas. Poucos são os projetos de software que têm sido entregues no prazo e dentro do orçamento previsto e a qualidade nem sempre está em níveis aceitáveis de confiabilidade. Além disto, cerca de 80% dos custos do software são gastos em atividades de manutenção, consumindo tempo e reduzindo consideravelmente a competitividade das empresas (Bosch, 2000).

O Desenvolvimento baseado em Componentes (DBC) (Brown, 2000) aparece como uma técnica promissora para a resolução destes problemas. Essa técnica consiste no desenvolvimento de aplicações a partir de componentes interoperáveis, reduzindo, assim, a complexidade e o custo do desenvolvimento, melhorando a qualidade do produto de software (Kallio, 2001) (Braga, 2000) (Woodman et al., 2001).

Desenvolvimento Baseado em Componentes

D'Souza (1998) define DBC como uma técnica de desenvolvimento de software, na qual todos os artefatos – desde códigos executáveis até especificações de interface, arquiteturas e modelos de negócio, e variando desde sistemas completos até pequenas partes – podem ser construídos pela combinação, adaptação e união de componentes numa variedade de configurações.

O conceito de componente ainda não possui um consenso entre os pesquisadores. Entretanto, uma visão bem aceita é que componentes reutilizáveis são artefatos auto-contidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces em conformidade com um dado modelo de arquitetura de software, possuindo documentação apropriada e um grau de reutilização definido (Braga, 2000).

Para que possamos "montar" aplicações a partir de componentes, duas atividades se mostram importantes: composição e adaptação. A composição é o uso de componentes na construção de um componente maior ou de um software completo. A melhor forma de reutilização seria o emprego do componente sem modificações. Entretanto, nem sempre isto é possível e, às vezes, é até importante o componente ser provido de mecanismos que permitam a sua adaptação, admitindo um maior espectro de possibilidades de uso.

A busca e a seleção de componentes são também atividades importantes em um processo de desenvolvimento baseado em componentes. A facilidade de localização e de compreensão da utilidade e das características do componente confere agilidade ao processo. Portanto, uma documentação acurada e um apropriado mecanismo de classificação de componentes são fundamentais para o sucesso da localização e reutilização de componentes (Kallio, 2001) (Braga, 2000).

Processo de Desenvolvimento de Componentes

O processo de desenvolvimento de componentes envolve quatro fases: análise de requisitos, especificação dos componentes, provisão e montagem de aplicações no domínio.

Na fase de análise de requisitos existe a preocupação com a definição e especificação dos requisitos do problema, ou seja, com o entendimento dos processos do negócio da aplicação. A fase de especificação de componentes se divide em três estágios intermediários (identificação de componentes, especificação de componentes e interação de componentes), através dos quais, a partir dos modelos especificados na fase de análise de requisitos, principalmente casos de uso, são identificadas as interfaces e especificações dos componentes. A fase de provisão determina quais componentes comprar, implementar, adaptar ou integrar com base nos resultados da especificação. Assim, a busca em repositórios de componentes já existentes é muito importante nesta etapa. Por fim, a fase de montagem guia a correta integração dos componentes, integrando-os a artefatos existentes para formar uma aplicação que satisfaça as necessidades do usuário.

No sentido de apresentar essa abordagem de forma prática, é apresentado a seguir um estudo de caso no domínio de gestão acadêmica, o qual detalha o processo de desenvolvimento, dando ênfase aos artefatos reutilizáveis gerados ao longo de todo o processo.

Estudo de Caso

Como já citado anteriormente, para o êxito no desenvolvimento de componentes e na sua posterior reutilização, deve-se ter um processo de desenvolvimento baseado em componentes que vai desde a fase de estudo do domínio dos componentes a serem desenvolvidos até a sua implementação.

Serão apresentadas a seguir as etapas de análise de requisitos e especificação dos componentes, brevemente descritas na seção anterior, através dos modelos gerados no desenvolvimento de aplicações com componentes.

Assim, foram desenvolvidos componentes para o domínio de gestão acadêmica, objetivando maior controle dos dados referentes aos alunos, professores, disciplinas, cursos, salas, ou seja, um maior detalhamento das informações pertinentes a este domínio como, por exemplo, a matrícula de alunos em cursos, a matrícula de alunos em disciplinas, o trancamento de curso por parte de alunos, o trancamento de disciplinas por parte de alunos, o cadastro de disciplinas com suas respectivas cargas horárias, a associação de disciplinas com suas respectivas salas, o cadastro de professores com suas respectivas disciplinas e o lançamento de notas. Foi desenvolvido também um componente de suporte para acesso ao banco de dados, sendo um componente genérico o suficiente para ser reutilizado em qualquer domínio de aplicação.

Análise de requisitos

O primeiro passo para se entender o domínio e definir os requisitos, seria através da utilização de modelos de casos de uso. Esse tipo de modelo descreve as iterações entre o usuário e o sistema (**Figuras 1, 2 e 3**). O usuário é representado por um ator, que é aquele que interage com o sistema. Pode ser um coordenador do curso, um aluno solicitando a matrícula, uma secretária ou um professor cadastrando seus dados.

É apresentada a seguir uma breve descrição dos principais casos de uso identificados para o estudo de caso envolvendo o domínio de gestão acadêmica:

- Cadastrar novo aluno: é o ato de cadastrar um novo aluno no sistema, informando dados básicos sobre ele;
- Efetuar matrícula de alunos nas disciplinas: é o ato de matricular o aluno nas disciplinas que deseja cursar. Nada impede que um aluno se matricule numa disciplina que não seja do seu período corrente, contanto que ela não tenha nenhum pré-requisito que o aluno não tenha realizado;

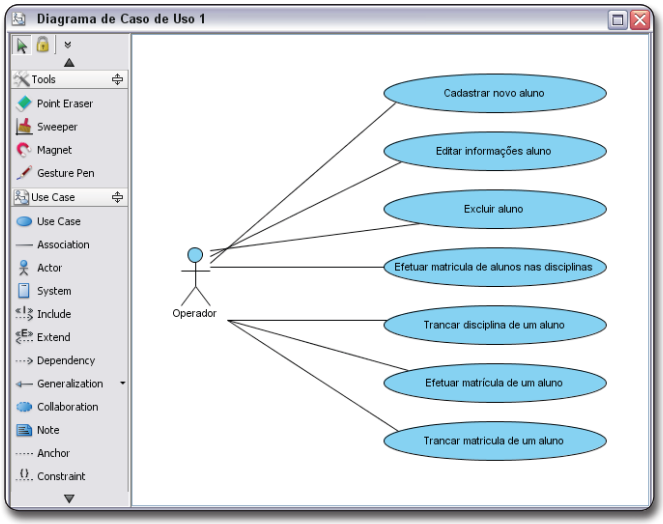


Figura 1. Casos de Uso do Domínio Gestão Acadêmica (parte 1)

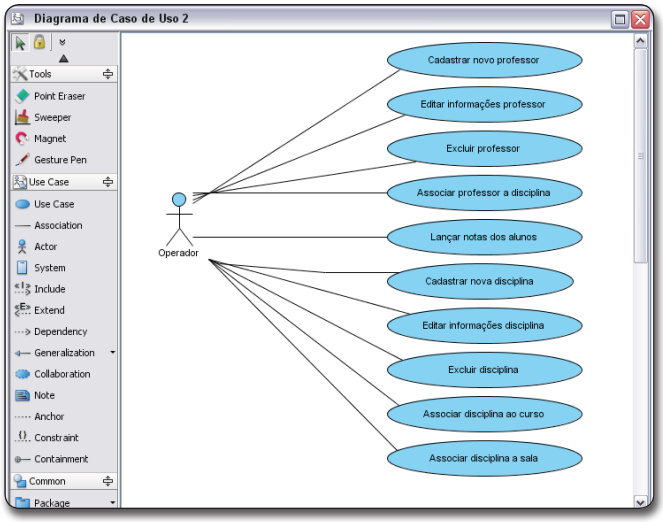


Figura 2. Casos de Uso do Domínio Gestão Acadêmica (parte 2)

• Trancar disciplina de um aluno: é o ato de interromper o curso de uma disciplina na qual o aluno estava matriculado. O aluno possui um determinado tempo para que essa ação seja solicitada;
• Efetuar matrícula de um aluno: ação de matricular o aluno no seu respectivo curso;
• Trancar matrícula de um aluno: é o ato de interromper o curso do aluno por um determinado período. Após o vencimento deste período, o aluno deve retornar ao curso se matriculando novamente, ou solicitar um novo trancamento;
• Cadastrar novo professor: é o ato de cadastrar um novo professor no sistema, informando os dados básicos sobre ele;
• Associar professor a disciplina: cadastrar no sistema quais as disciplinas que o respectivo professor irá lecionar. Um professor deve estar associado a uma ou várias disciplinas;
• Lançar notas dos alunos: ação de lançar no sistema as notas alcançadas pelos alunos nas disciplinas, ao final do período. O aluno pode ser considerado aprovado ou reprovado;
• Cadastrar nova disciplina: é a ação de cadastrar no sistema uma nova disciplina, informando os dados básicos sobre ela. Uma disciplina pode ter nenhuma ou várias disciplinas como pré-requisito;
• Associar disciplina ao curso: cadastrar no sistema quais disciplinas estão associadas a quais cursos;
• Associar disciplina a sala: cadastrar no sistema quais disciplinas são ministradas em quais salas, como forma de controle para evitar que horários coincidam na mesma sala;
• Cadastrar novo curso: é a ação de cadastrar no sistema um novo curso, informando os dados básicos sobre ele;
• Cadastrar nova sala: é o ato de cadastrar no sistema uma nova sala, informando os dados básicos sobre ela.

Em seguida tem-se a descrição textual de alguns casos de uso, os quais são detalhados em passos (**Tabelas 1** a **3**).

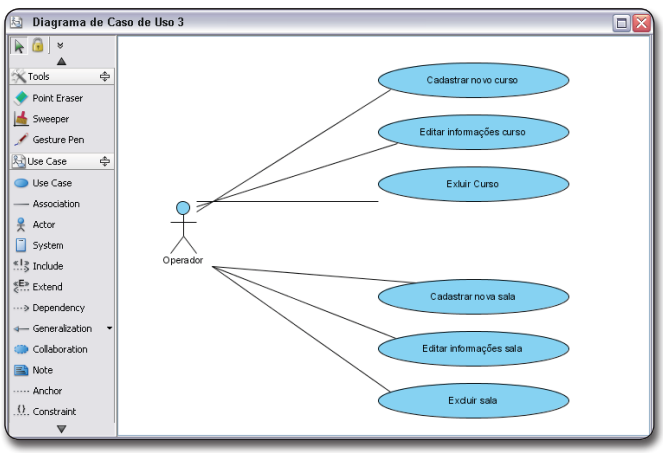


Figura 3. Casos de Uso do Domínio Gestão Acadêmica (parte 3)

Nome:	Efetuar matrícula de alunos nas disciplinas
Ator:	Operador
Objetivo:	Cadastrar no sistema o aluno com as disciplinas que o mesmo deseja cursar.
Cenário Principal	
1. Operador seleciona o código do aluno que se deseja matricular.	
2. Operador seleciona o código da(s) disciplina(s) que se deseja matricular.	
3. Sistema matricula o aluno na disciplina selecionada.	
Exceções	
a. A disciplina selecionada exige pré-requisitos que não foram cumpridos pelo aluno, devendo ser fornecida uma disciplina válida.	

Tabela 1. Descrição Textual – Efetuar matrícula de alunos nas disciplinas

O próximo passo é a construção do Modelo Conceitual de Negócios. Esse modelo tem como objetivo apresentar os conceitos e identificar os relacionamentos existentes no domínio do problema, como mostrados na **Figura 4**.

No domínio de gestão acadêmica, de acordo com o diagrama da figura anterior, pode-se concluir que um aluno pode solicitar ou estar matriculado em um conjunto de disciplinas de um curso, assim como, no caso de estar com o curso trancado

Nome:	Trancar disciplina de um aluno
Ator:	Operador
Objetivo:	Interromper o curso de uma disciplina na qual o aluno estava matriculado.
Cenário Principal	
1. Operador seleciona o código do aluno que deseja trancar a disciplina.	
2. Operador seleciona o código da(s) disciplina(s) que se deseja trancar.	
3. Sistema registra trancamento.	
Exceções	
a. O prazo de solicitação de trancamento expirou. Operação não pode ser realizada.	

Tabela 2. Descrição Textual – Trancar disciplina de um Aluno

Nome:	Lançar notas dos alunos
Iniciante:	Operador
Objetivo:	Lançar no sistema as notas alcançadas pelos alunos nas disciplinas ao final do período.
Cenário Principal	
1. Operador seleciona o código do aluno que se deseja lançar nota.	
2. Operador seleciona o código da disciplina.	
3. Operador entra com os dados referentes a nota e sua situação (A – aprovado, R – reprovado).	
4. Sistema armazena as informações	

Tabela 3. Descrição Textual – Lançar notas dos alunos

não estar associado a nenhuma disciplina. No ato da matrícula, o aluno consegue verificar se a respectiva disciplina possui pré-requisitos, ou seja, disciplinas que ele já deva ter cursado para realizar a solicitação. Estas disciplinas são ministradas por professores. Pode-se verificar também que existe uma associação entre disciplinas e salas.

Os atributos normalmente não são apresentados nesse diagrama, uma vez que se trata de um modelo de alto nível, porém, nada impede que nesse momento eles estejam especificados. A especificação dos atributos será considerada nas fases posteriores de detalhamento dos componentes e suas interfaces.

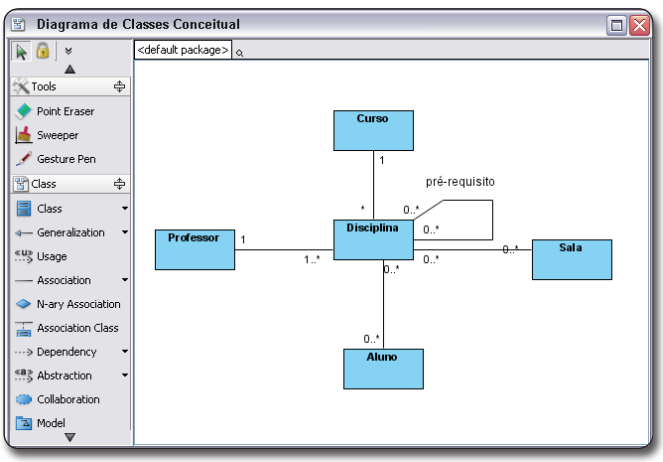


Figura 4. Modelo Conceitual de Negócios do Domínio de Gestão Acadêmica

Especificação dos componentes

O próximo passo do processo de desenvolvimento dos componentes do domínio de gestão acadêmica é a especificação dos componentes, que é dividida em três estágios intermediários: identificação de componentes, especificação de componentes e interação de componentes.

O estágio de **identificação dos componentes** tem como entrada o modelo conceitual de negócio e o modelo de casos de uso, já apresentados, e tem como principal objetivo identificar um conjunto inicial de interfaces para os componentes de negócio e especificá-los como uma arquitetura inicial de componentes do domínio.

Este estágio produz como artefato o modelo de tipos de negócios, utilizado para criação dos modelos de informações ligados às interfaces. Esse modelo representa precisamente as informações do negócio que são relevantes para o desenvolvimento de aplicativos no domínio de gestão acadêmico (**Figura 5**). Verifica-se ainda que as disciplinas possuem uma quantidade de créditos e carga horária que devem ser cumpridas pelos alunos para que consigam a sua aprovação, bem como os horários e dias da semana que são ministradas, vinculadas à sala. À medida que os alunos vão sendo avaliados, os professores lançam as notas dos mesmos nas respectivas disciplinas e a sua situação pode ser verificada a qualquer momento pelo sistema. Essas informações motivaram o aparecimento de novas classes no diagrama.

Para a identificação das interfaces, foram analisados os modelos vindos da etapa de análise do domínio, ou seja, agrupando os principais conceitos e funcionalidades do domínio e, combinado às ações identificadas nos casos de uso, é possível ter uma idéia do conjunto inicial de interfaces e especificações dos componentes.

O próximo estágio intermediário trata da **especificação dos componentes** propriamente ditos. Este estágio produz como artefatos os modelos de especificação de interfaces, de sistema e de negócios, especificação de componentes e arquitetura de componentes.

Em relação às interfaces, podem-se identificar dois tipos para os componentes: as interfaces funcionais, que são derivadas dos casos de uso já apresentados, e as interfaces de negócio que são derivadas dos tipos de negócio, também já apresentados.

Como visto, os casos de uso são detalhados em passos e esses passos são usados para auxiliar na identificação das operações necessárias no sistema. Como uma primeira abordagem, define-se uma interface para cada caso de uso e analisam-se os passos para descobrir as operações necessárias. Para componentes funcionais, a partir do detalhamento dos casos de uso, agregação destes em casos de uso similares e das operações contidas nos mesmos, origina-se métodos das interfaces. Têm-se também componentes de negócio que são derivados dos tipos, mas os funcionais são os mais importantes.

Porém, pode ser verificado que vários dos casos de uso listados anteriormente estão intimamente interligados, ou seja, fazem parte de um mesmo processo com um dependendo totalmente do outro. As operações destes casos de uso foram agrupadas em interfaces. Portanto os agrupamentos

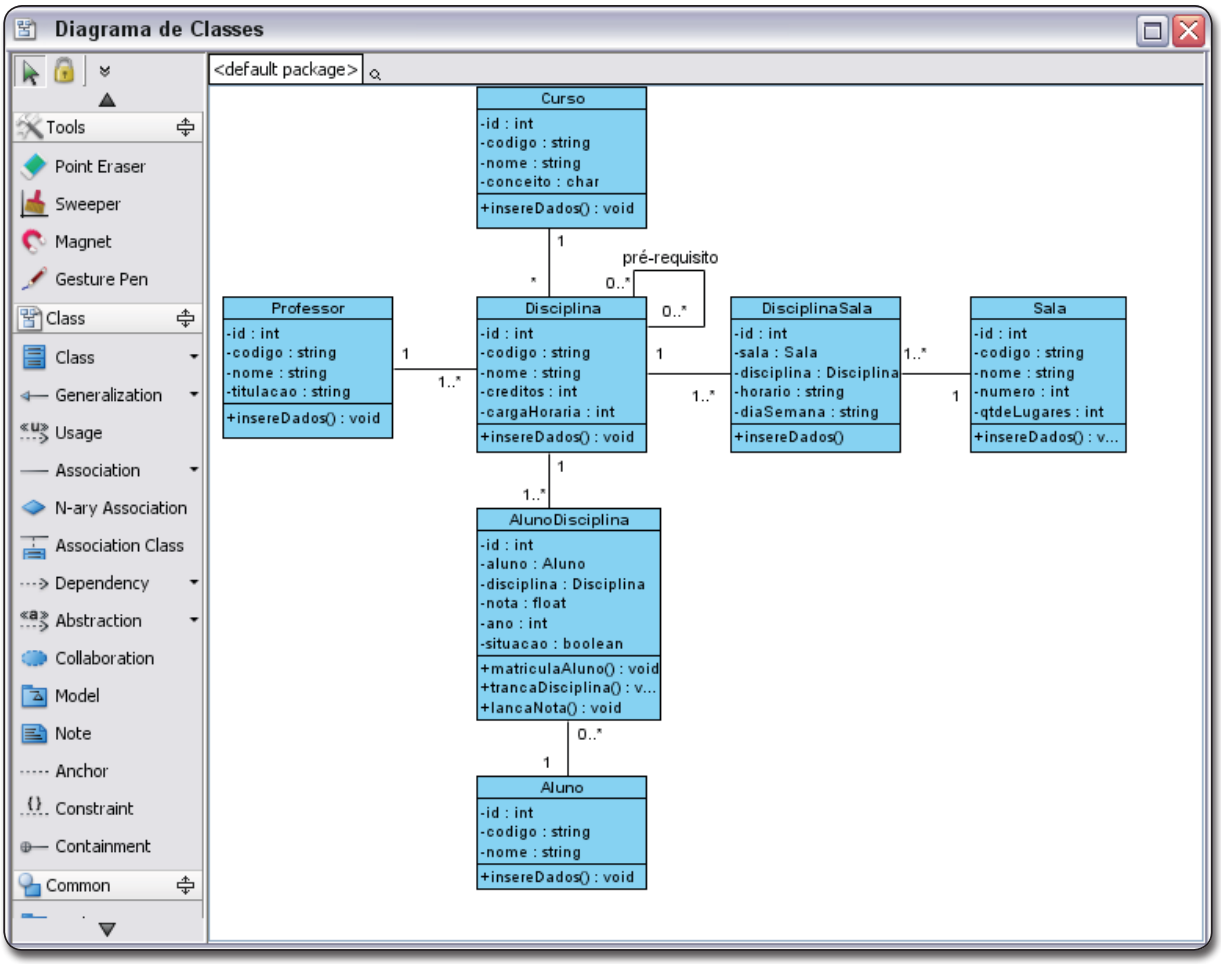


Figura 5. Modelo de Tipos de Negócios do Domínio de Gestão Acadêmica

ficaram da seguinte maneira: o componente **Gerente de Aluno** com a interface **IAluno** associada aos seguintes casos de uso **cadastrar novo aluno**, **editar informações aluno** e **excluir aluno** e a interface **IMatricula** associada aos casos de uso e**fetuar matrícula de um aluno**, **trancar matrícula de um aluno**, **efetuar matrículas de alunos nas disciplinas** e **trancar disciplina de um aluno**. Para o componente **Gerente de Professor** a interface **IProfessor** com os casos de uso cadastrar **novo professor**, **editar informações professo**r, **excluir professor**, **associar professor a disciplina** e **lançar notas dos alunos**. O componente **Gerente de Disciplina** com a interface **IDisciplina** associada aos casos de uso **cadastrar nova disciplina**, **cadastrar disciplina pré-requisitos**, **editar informações disciplina**, **excluir disciplina**, **associar disciplina ao curso** e **associar disciplina a sala**. Para o componente **Gerente de Curso** a interface **ICurso** associada aos casos de uso **cadastrar novo curso**, **editar informações curso** e **excluir curso**. E o componente **Gerente de Sala** com a interface **ISala** associada aos casos de uso **cadastrar nova sala**, **editar informações sala** e **excluir sala**.

Já para as interfaces de negócio, a partir da identificação dos tipos principais, ou seja, aluno, professor, disciplina, curso e sala, devem ser associados uma interface de negócio para cada tipo principal relacionado. Com isso, cada interface gerencia as informações do seu tipo.

Baseado nessas idéias apresentadas e juntando as várias interfaces identificadas, foram criadas as seguintes interfaces mostradas na **Figura 6**.

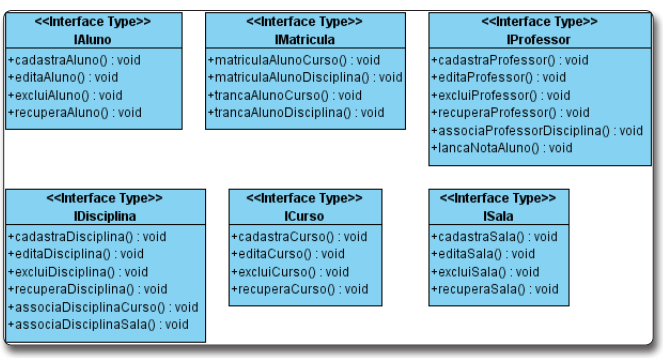


Figura 6. Interfaces do Domínio

Logo em seguida são apresentadas as especificações das interfaces de acordo com os componentes especificados. Cada interface tem seu próprio pacote. O pacote contém a interface em si e os tipos associados (**Figuras 7** e **8**).

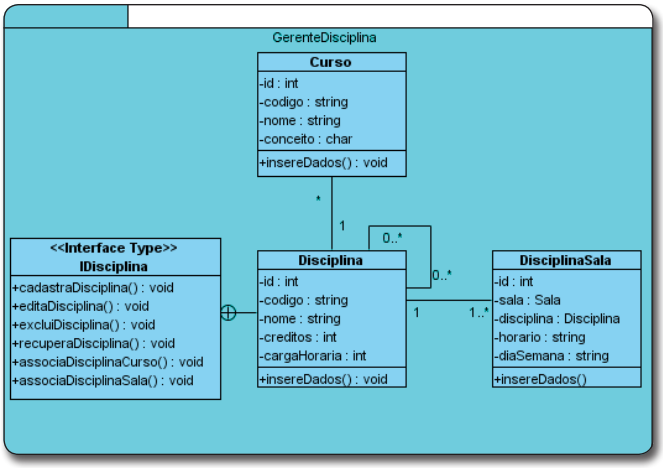


Figura 7. Pacote de Especificação de Interface – IDisciplina

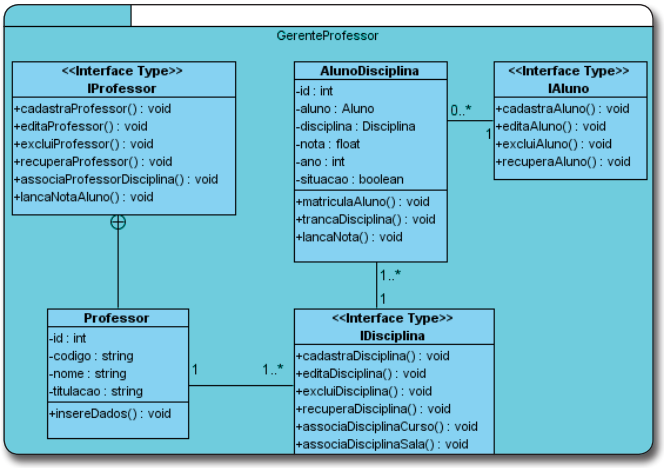


Figura 8. Pacote de Especificação de Interface – IProfessor

Em geral, para cada uma das interfaces identificadas anteriormente é criada uma especificação de componente separada, para depois ser construída a arquitetura de componentes, ou seja, como serão as interações entre os mesmos.

Em relação aos componentes propriamente ditos, ao reorganizar os agrupamentos da fase anterior e agrupando as interfaces em um mesmo componente, cada componente do domínio de gestão acadêmica ficou associado a uma ou mais interfaces da seguinte maneira: o componente Gerente de Aluno oferece as interfaces IAluno e IMatricula. O componente Gerente de Professor a interface IProfessor. O componente Gerente de Disciplina a interface IDisciplina. O componente Gerente de Curso provê a interface ICurso e o Gerente de Sala a interface ISala. Estes agrupamentos estão representados na **Figura 9**.

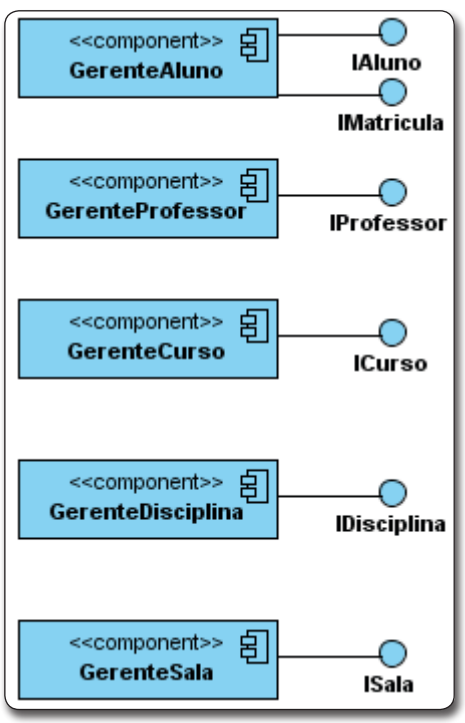


Figura 9. Diagrama de Componentes do Domínio

Para fazer acesso a base de dados (gravar, recuperar e atualizar os dados) houve a necessidade do desenvolvimento de um componente de suporte, que chamamos de Gerente de Banco de Dados (**Figura 10**).

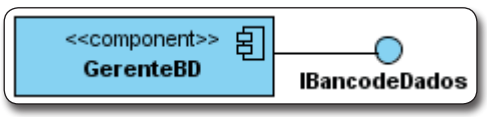


Figura 10. Gerente de Banco de Dados

A interface IBancoDeDados pode ser definida de acordo com a **Figura 11**.

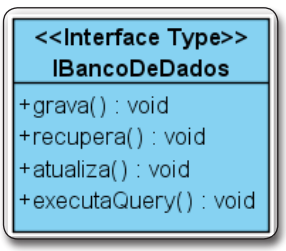


Figura 11. Especificação da Interface do Componente de Suporte para Acesso ao Banco de Dados

Este componente de suporte ficará no repositório de componentes e é genérico o suficiente para poder ser reutilizado em qualquer outro domínio.

Será analisado agora o último estágio intermediário que é a interação entre componentes. A arquitetura de componentes mostra como os componentes interagem, mostrando as interfaces providas e requeridas de cada componente. Este tipo de interação pode ser visto na **Figura 12**, onde os relacionamentos vêm do detalhamento das interfaces dos componentes, ou seja, para realizar um determinado método, podem ser necessários serviços de outros componentes.

Exemplo de Utilização dos Componentes

No intuito de exemplificar essa abordagem, foi desenvolvida uma aplicação simples para integrar estes componentes e mostrar o funcionamento dos mesmos em conjunto.

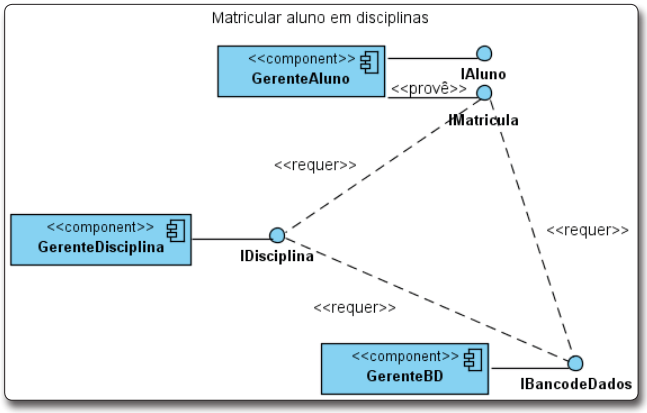


Figura 12. Diagrama da Arquitetura do Componente – Gerente de Aluno

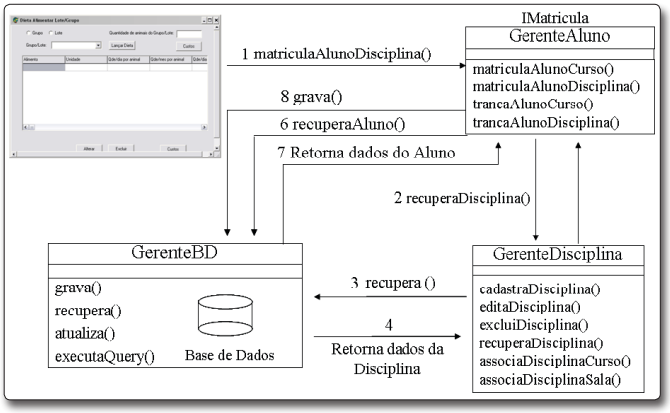


Figura 13. Arquitetura da Aplicação – Matrícula de Aluno em uma Disciplina

No desenvolvimento da aplicação, primeiramente foram criadas as interfaces (com usuário) de entrada dos dados, que estarão passando informações para os componentes através de suas interfaces (dos componentes), referentes aos parâmetros de entrada das funcionalidades dos mesmos. Por exemplo, para o componente Gerente de Aluno foram criadas janelas para realizar a matrícula do aluno no curso, para solicitação de matrícula de um aluno em uma disciplina, para trancamento do curso por um aluno e trancamento de uma disciplina. Já para o componente Gerente de Disciplina foram criadas janelas para cadastrar uma nova disciplina, para associar essa disciplina a um curso, dentre outras.

Como exemplo, é mostrado o funcionamento da aplicação (arquitetura da aplicação) utilizando o componente Gerente de Aluno, o qual pode ser visualizado na **Figura 13**. A figura apresenta a janela de entrada de dados para realizar a matrícula de um aluno em uma disciplina e todos os eventos que ocorrem desde a chamada do método matriculaAlunoDisciplina() até o momento em que o objeto é gravado na base de dados que contém os dados de matrícula.

Conclusões

Neste artigo foi relatada a descrição das etapas de desenvolvimento de componentes, apresentando ainda um estudo de caso no domínio de gestão acadêmica. Foram desenvolvidos seis componentes, sendo cinco componentes de domínio representando os controles de informações de alunos, professores, disciplinas, cursos e salas, e um componente de suporte de acesso à base de dados, o qual está sendo reutilizado dentro do domínio. ●

Referência

Bosch, J., 2000, Design and Use of Software Architectures: Adopting and evolving a product-line approach, Editora Addison-Wesley, ACM Press.

Braga, R. Busca e Recuperação de Componentes em Ambiente de Reutilização de Software. Tese de Doutorado em Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, Brasil, dezembro 2000.

Brown, A.W. Large Scale Component-Based Development, Prentice Hall, 2000.

D'Souza, D. F. e Wills, A. C., 1998, Object, Components, and Frameworks with UML: The Catalysis Approach, Editora Addison-Wesley, Massachusetts.

Jacobson, I., Griss e M., Jonsson, P., 1997, Software Reuse: Architecture, Process and Organization for Business Success, Editora Addison-Wesley.

Kallio, P. e Niemelä, E., 2001, Documented Quality of COTS and COM Components, IV ICSE Workshop on Component-Based Software Engineering

Villela, R. M. M. B., 2000, Busca e Recuperação de Componentes em Ambientes de Reutilização de Software, Tese de Doutorado, UFRJ-COPPE, Rio de Janeiro.

Woodman, M. et al., 2001, Issues of CBD Product Quality and Process Quality, IV ICSE Workshop on Component-Based Software Engineering.

Domain Driven Design

Uma Introdução

Daniel Cukier

danicuki@gmail.com

Graduado em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP), está terminando o mestrado nessa mesma instituição, cujo tema da tese é Padrões para Introduzir Novas Ideias na Indústria de Software. Trabalha há mais de 14 anos com desenvolvimento de software, atualmente é gerente de desenvolvimento na Locaweb. Autor do blog http://agileandart.blogspot.com, já participou como palestrante em eventos como Encontro Ágil e Falando em Agile.

De que se trata o artigo?

Neste artigo veremos os principais padrões de Domain Driven Design e alguns exemplos de como esses padrões podem ser aplicados.

Para que serve?

DDD induz a implantação de um cenário de melhoria contínua, podendo ser uma ferramenta extremamente útil para se desenvolver software de qualidade e que atenda bem as necessidades do cliente.

Em que situação o tema é útil?

Tanto programadores quanto arquitetos e especialistas de negócio podem se beneficiar com as técnicas de DDD, que ensinam justamente boas práticas de como modelar seu domínio, além de tornar eficiente a interação entre os vários papéis de pessoas que fazem parte do processo de desenvolvimento de software. DDD pode ser muito útil quando vários times trabalham em conjunto no desenvolvimento de um sistema complexo.

Domain Driven Design significa Projeto Orientado a Domínio. Ele veio do título do livro escrito por Eric Evans, dono da DomainLanguage, uma empresa especializada em treinamento e consultoria para desenvolvimento de software. O livro de Evans é um grande catálogo de Padrões, baseados em experiências do autor ao longo de mais de 20 anos desenvolvendo software utilizando técnicas de Orientação a Objetos. O que seria um Padrão?

Um padrão é uma regra de três partes que expressa a relação entre um contexto (1), um problema (2) e uma solução (3).

DDD pode ser visto por alguns como a volta da orientação a objetos. É verdade que o livro é um chamado às boas práticas de programação que já existem desde a época remota do SmallTalk. Quando se fala em Orientação a Objetos pensa-se logo em classes, heranças, polimorfismo, encapsulamento. Mas a essência da Orientação a Objetos também tem elementos como:

- **Alinhamento do código com o negócio:** o contato dos desenvolvedores com os especialistas do domínio é algo essencial quando se faz DDD (o pessoal de métodos ágeis já sabe disso faz tempo);
- **Favorecer reutilização:** os blocos de construção, que veremos adiante, facilitam

aproveitar um mesmo conceito de domínio ou um mesmo código em vários lugares;

• **Mínimo de acoplamento:** Com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos;

• **Independência da Tecnologia:** DDD não foca em tecnologia, mas sim em entender as regras de negócio e como elas devem estar refletidas no código e no modelo de domínio. Não que a tecnologia usada não seja importante, mas essa não é uma preocupação de DDD.

Todas essas coisas são bem exemplificadas e mostradas na forma de vários padrões em DDD. Mas o livro também mostra muitos padrões que não dizem respeito a código ou modelagem. Aparecem coisas que estão mais ligadas a processos (como Integração Contínua) ou a formas de relacionamento entre times que fazem parte do desenvolvimento de um sistema complexo. Eric Evans dividiu o livro em quatro partes, que apresentaremos a seguir.

Colocando o modelo de domínio para funcionar

Para ter um software que atenda perfeitamente a um determinado domínio, é necessário que se estabeleça, em primeiro lugar, uma Linguagem Ubíqua, ou Linguagem comum, com termos bem definidos, que fazem parte do domínio do negócio e que são usados por todas as pessoas que fazem parte do processo de desenvolvimento de software. Nessa linguagem estão termos que fazem parte das conversas diárias entre especialistas de negócio e times de desenvolvimento. Todos devem usar os mesmos termos tanto na linguagem falada quanto no código. Isso significa que, se durante uma conversa com um cliente do sistema de cobrança, por exemplo, ele disser: "Temos que emitir a fatura para o cliente antes da data limite", vamos ter no nosso código alguma coisa do tipo:

• Uma classe para a entidade Cliente;
• Uma classe para a entidade Fatura;
• Algum serviço que tenha um método emitir;
• Algum atributo com o nome de data limite.

Essa linguagem ubíqua deve ser compreendida por todos e não pode haver ambiguidades. Toda vez que alguém perceber que um determinado conceito do domínio possui várias palavras que o represente, essa pessoa deve tentar re-adequar tanto a linguagem falada e escrita, quanto o código.

Utilizando a **Linguagem Ubíqua** criamos um modelo de domínio através do **Projeto Dirigido pelo Modelo** (Model Driven Design – MDD). A idéia por trás de MDD é a de que o seu modelo abstrato deve ser uma representação perfeita do seu domínio. Tudo que existe no seu negócio deve aparecer no modelo. Só aparece no modelo aquilo que está no negócio.

Em um time que cria software temos de um lado os especialistas de negócio e de outro os desenvolvedores e arquitetos. Num processo ágil defendido pelo MDD a criação do modelo abstrato deve ser feita em grupo, com todas as pessoas juntas. Se arquitetos e analistas de negócio criarem o modelo sem a participação dos programadores, corre-se o risco de criar um modelo que não é implementável ou que usará uma tecnologia inadequada. Da mesma forma, se os programadores codificarem sem se basear num modelo consistente, provavelmente desenvolverão um software que simplesmente não serve para o domínio. Em DDD, parte das pessoas que modelam o domínio são necessariamente pessoas que colocam a mão em código (Hands-on Modelers). Se os programadores não se sentirem responsáveis pelo modelo ou não entenderem como o modelo funciona, então o modelo não terá relação alguma com o software produzido por essas pessoas.

O processo de maturação de um sistema desenvolvido usando MDD deve ser contínuo. O modelo servirá de guia para a criação do código e, ao mesmo tempo, o código ajuda a aperfeiçoar o modelo. O contato contínuo com o código trará insights aos programadores, que irão refatorar o código. Essa refatoração deverá ser feita não só no código, mas também no próprio modelo.

Blocos de construção do Model Driven Design (MDD)

Uma vez que decidimos criar um modelo usando MDD, precisamos, inicialmente, isolar o modelo de domínio das demais partes que compõem o sistema. Essa separação pode ser feita utilizando-se uma arquitetura em camadas (Figura 1), que dividirá nossa aplicação em quatro partes:

• **Interface de Usuário** – parte responsável pela exibição de informações do sistema ao usuário e também por interpretar comandos do usuário;

• **Aplicação** – essa camada não possui lógica de negócio. Ela é apenas uma camada fina, responsável por conectar a Interface de Usuário às camadas inferiores;

• **Domínio** – representa os conceitos, regras e lógicas de negócio. Todo o foco de DDD está nessa camada. Nosso trabalho, daqui para frente, será aperfeiçoar e compreender profundamente essa parte;

• **Infra-estrutura** – fornece recursos técnicos que darão suporte às camadas superiores. São normalmente as partes de um sistema responsáveis por persistência de dados, conexões com bancos de dados, envio de mensagens por redes, gravação e leitura de discos, etc.

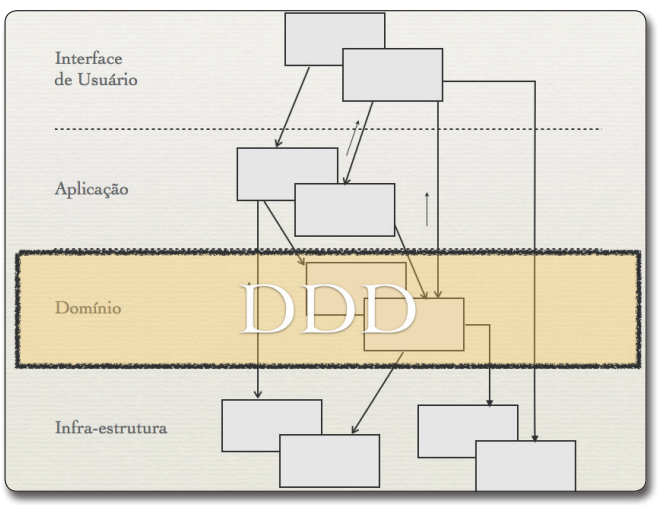


Figura 1. Arquitetura em camadas, utilizada para separar o domínio do resto da aplicação

Depois de dividirmos o sistema em camadas, nos preocuparemos apenas com a camada de domínio. Para modelar essa parte, utilizamos alguns Padrões propostos em DDD. Esses padrões são chamados de blocos de construção e serão utilizados para representar nosso modelo abstrato. Esses blocos podem ser:

• **Entidades** – classes de objetos que necessitam de uma identidade. Normalmente são elementos do domínio que possuem ciclo de vida dentro de nossa aplicação: um Cliente, por exemplo, se cadastra no sistema, faz compras, se torna inativo, é excluído, etc.;

• **Objetos de Valores** – objetos que só carregam valores, mas que não possuem distinção de identidade. Bons exemplos de objetos de valores seriam: strings, números ou cores. Por exemplo: se o lápis de cor da criança acabar e você der um novo lápis a ela, da mesma cor, só que de outra caixa, ela não vai se importar. Para a criança, o lápis vermelho de uma caixa é igual ao lápis vermelho de outra caixa. As instâncias de Objetos de Valores são imutáveis, isto é, uma vez criados, seus atributos internos não poderão mais ser modificados. Em Java, temos, por exemplo, a classe BigDecimal, muito utilizada para fazer cálculos com valores grandes. Na **Listagem 1** observamos que, para multiplicar dois valores representados pela classe BigDecimal, não alteramos os objetos com os valores dos fatores da multiplicação. Para calcular 5 milhões vezes 30 milhões construímos cada um dos fatores e então obtemos o resultado, que será armazenado numa terceira variável. Após o cálculo, cada um dos fatores continuará armazenando o valor original. A saída do código será:

```
5000000
30000000
150000000000000
```

• **Agregados** – compostos de Entidades ou Objetos de Valores que são encapsulados numa única classe. O Agregado serve para manter a integridade do modelo. Elegemos uma classe para servir de raiz do Agregado. Quando algum cliente quiser manipular dados de uma das classes que compõem o Agregado, essa manipulação só poderá ser feita através da raiz;

• **Fábricas** – classes responsáveis pelo processo de criação dos Agregados ou dos Objetos de Valores. Algumas vezes, Agregados são relativamente complexos e não queremos manter a lógica de criação desses Agregados nas classes que o compõem. Extraímos então as regras de criação para uma classe externa: a fábrica;

• **Serviços** – classes que contém lógica de negócio, mas que não pertence a nenhuma Entidade ou Objetos de Valores. É importante ressaltar que Serviços não guardam estado, ou seja, toda chamada a um mesmo serviço, dada uma mesma pré-condição, deve retornar sempre o mesmo resultado;

• **Repositórios** – classes responsáveis por administrar o ciclo de vida dos outros objetos, normalmente Entidades, Objetos de Valor e Agregados. Os repositórios são classes que centralizam operações de criação, alteração e remoção de objetos. Em linguagens como Java e .NET, repositórios são comumente implementados usando-se frameworks como Hibernate ou Nhibernate. Já em RubyOnRails, o ActiveRecord faz o papel de repositório;

• **Módulos** – abstrações que têm por objetivos agrupar classes por um determinado conceito do domínio. A maioria das linguagens de programação oferece suporte a módulos (pacotes em Java, namespaces em .NET ou módulos em Ruby). Um anti-padrão comum é a criação de módulos que agrupam as classes segundo conceitos de infra-estrutura. Um exemplo seria, ao se trabalhar com Struts, em Java, criar um pacote que conterá todas as Actions do sistema. Ao usar DDD devemos agrupar classes se esse agrupamento faz sentido do ponto de vista do domínio, ou seja, do negócio. Se tivermos, por exemplo, várias classes que compõem informações de **Paciente** num sistema médico, podemos criar um módulo chamado paciente e colocar classes como Ficha, PrescricaoMedica, RegistroDeConsulta e HistoricoDeCirurgias num mesmo pacote.

Listagem 1. BigDecimal muitas vezes é um Objeto de Valor

```
BigDecimal fiveM = BigDecimal.valueOf(5000000);
BigDecimal thirtyM = BigDecimal.valueOf(30000000);

BigDecimal result = fiveM.multiply(thirtyM);

System.out.println(fiveM);
System.out.println(thirtyM);
System.out.println(result);
```

Refatorando para compreender profundamente o modelo

Depois de elaborar um modelo de dados que reflete o seu domínio, usando os blocos de construção do MDD, o processo de aperfeiçoamento do modelo continua. O modelo deve ser refatorado e melhorado na medida em que se obtém maior compreensão de conceitos importantes do domínio. Muitas vezes alguns conceitos do domínio estão implícitos no código. O trabalho do desenvolvedor é tentar identificar esses conceitos implícitos e torná-los explícitos. Alguns padrões podem ajudar a compreender mais profundamente o modelo:

• **Interface de Intenção Revelada** – usar nomes em métodos ou classes que dizem exatamente "o que" essas classes ou métodos fazem, mas não "como" elas fazem. Dizer "como" no nome do método quebra o encapsulamento, uma vez que quem chama o código saberá detalhes de implementação;

• **Funções sem Efeitos-Colaterais** – tentar deixar o código com o maior número possível de métodos que não alterem o estado dos objetos, concentrando esse tipo de operação (alteração de estado) em Comandos;

• **Asserções** – para os Comandos que alteram estados, criar testes de unidade que rodem automaticamente, ou colocar asserções no código que validem, após a chamada dos comandos, as alterações de estado esperadas.

Por exemplo, o dono de uma loja de tintas pode querer um programa que mostre para seus clientes o resultado da mistura de cores padrões. Inicialmente temos um código como na **Listagem 2**. Não é possível, apenas olhando o nome do método

(pinta), saber o que esse método faz. O que ele pinta? Um quadro? Um desenho? Uma parede? O que ele faz com a Tinta que ele recebe como parâmetro? Temos que olhar a implementação para saber o que está acontecendo de fato neste método.

Listagem 2. Código da versão 1.0 do sistema de mistura de tintas

```
public class Tinta{
private int r;
private int g;
private int b;

public void pinta(Tinta p){
    v = v + pinta.getV(); //volume é somado
    //várias linhas de código complexo
    //para misturar tintas
    //no final, r, g e b têm seus valores alterados
   }
}
```

Um possível teste para esse código seria como na **Listagem 3**.

Listagem 3. Teste versão 1.0 do sistema de mistura de tintas

```
public void testPinta(){
//cria amarelo com volume 100
Tinta amarelo = new Tinta(100, 0, 50, 0);
//cria azul com volume 100
Tinta azul = new Tinta(100, 0, 0, 50);

//mistura azul com amarelo
amarelo.pinta(azul);

//verifica o resultado. O volume deve ser 200
assertEquals(200, amarelo.getV());
assertEquals(25, amarelo.getB());
assertEquals(25, amarelo.getY());
assertEquals(0, amarelo.getR());
}
```

Nós queremos tornar esse código mais claro, e para isso usaremos uma interface de intenção revelada. Mudaremos o código da classe Tinta e o nosso teste, como na **Listagem 4**.

Listagem 4. Teste versão 1.0 do sistema de mistura de tintas

```
public void testMisturaDeTintas(){
//cria amarelo com volume 100
Tinta amarelo = new Tinta(100, 0, 50, 0);
//cria azul com volume 100
Tinta azul = new Tinta(100, 0, 0, 50);

//mistura azul com amarelo
amarelo.misturadoCom(azul);

//verifica o resultado. O volume deve ser 200
assertEquals(200, amarelo.getVolume());
assertEquals(25, amarelo.getAzul());
assertEquals(25, amarelo.getAmarelo());
assertEquals(0, amarelo.getVermelho());
}

public class Tinta{
  private int vermelho;
  private int amarelo;
  private int azul;

  public void misturadoCom(Tinta p){
     volume = volume + pinta.getVolume(); //volume é somado
     //várias linhas de código complexo
     //para misturar tintas
  }
}
```

O código resultante já torna muito mais claro o que o método faz. O próprio nome do método (misturaCom) já diz: mistura uma tinta com outra. O antigo nome (pinta) não fazia sentido do ponto de vista do domínio, já que o cliente da loja de tintas quer saber: "que cor fica quando eu misturo essa tinta com outra?"

Projeto estratégico

As técnicas de criação e refinamento do modelo (como o uso da Linguagem Ubíqua e MDD) são importantes para começarmos a entender como desenvolver um sistema dirigido pelas regras de negócio do domínio, ou seja, como aplicar DDD. Mas a parte mais importante (e difícil) de Domain Driven Design é quando começamos a lidar com sistemas complexos. Como deve ser a interação entre sistemas que interagem entre si? Como dividir nosso trabalho, de maneira que foquemos nossos esforços naquilo que tem maior valor para o negócio? Como fazer a comunicação entre os times que desenvolvem esses sistemas?

Temos alguns padrões que nos ajudam a dividir nosso software em várias partes, que chamamos de contextos. Cada Contexto Delimitado deve estar bem claro para todos que estão envolvidos no processo de desenvolvimento. A fronteira entre contextos deve ser clara para todos, ou seja, todo mundo deve saber a qual contexto um determinado pedaço de código pertence.

Os padrões que nos ajudarão a estabelecer qual é a relação existente entre os vários times são:

• **Mapa de Contextos** – uma forma pragmática de documentar claramente os vários Contextos Delimitados que fazem parte de um sistema complexo (Figura 2). No mapa de contextos ressaltaremos os vários componentes do software e como as equipes que cuidam desse sistema interagem. Além disso, criaremos um mapa de tradução que servirá para facilitar a comunicação. Muitas vezes, existem termos usados para um mesmo conceito em vários contextos (por exemplo, no contexto do sistema de cobrança usamos o termo "Fatura", que significa a mesma coisa que o termo "Nota Fiscal" no contexto do sistema de faturamento). Toda vez que o time do sistema de cobrança for conversar com o time de faturamento, ele terá que usar adequadamente o termo "Nota Fiscal" para que o outro time entenda o que está sendo dito;

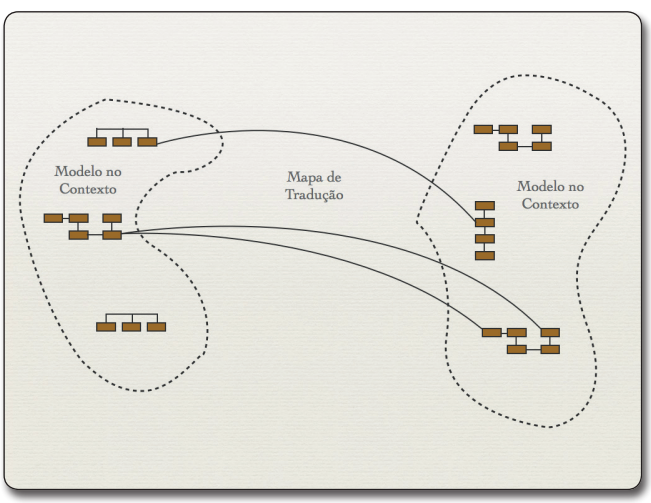


Figura 2. Mapa de Contextos

• **Produtor-Consumidor** – Quando os times possuem uma relação bem clara de Consumidor-Produtor. Produtor é o time que fornece software para o time Consumidor. Nesse tipo de relação, o Produtor assumirá compromissos de entregas para seu Consumidor, que por sua vez, tomará decisões esperando que esse compromisso seja cumprido. O time Produtor deverá rodar testes automatizados feitos pelo Consumidor (ou por ele mesmo) toda vez que fizer alguma alteração na interface que o cliente usa. Isso garantirá que nada irá quebrar do lado do Consumidor;
• **Conformista** – Algumas vezes não é possível que dois times de desenvolvimento se relacionem como Produtor-Consumidor. Mesmo que o Produtor tenha muita boa vontade, ele pode ter outras prioridades. Caso não seja possível alterar ou pedir alterações de uma parte do sistema mantida por outro time, deve-se adotar uma postura conformista e assumir que a única forma de caminhar é ter que aguardar o outro grupo e se adaptar a sua realidade. Quando a relação de conformista é estabelecida, não solicitamos mais funcionalidades ao outro time. Alteramos nossos sistemas para se adequar ao que já é oferecido pelo outro sistema;
• **Núcleo Compartilhado** – Quando dois times alteram uma mesma base de código comum (**Figura 3**). É importante que esse pedaço de código esteja bem definido e que possua muitos testes automatizados, que devem ser rodados por qualquer um dos grupos que desejar fazer alguma alteração. Todas as alterações devem ser comunicadas ao outro time e os testes precisam fazer parte de um processo de Integração Contínua. Quando se usa um núcleo compartilhado, esse pedaço de código tende a ser mais difícil de mudar, já que precisa da aceitação de várias equipes. Por outro lado, evita-se algumas duplicações, já que conceitos comuns são codificados uma única vez e utilizados por vários sistemas;

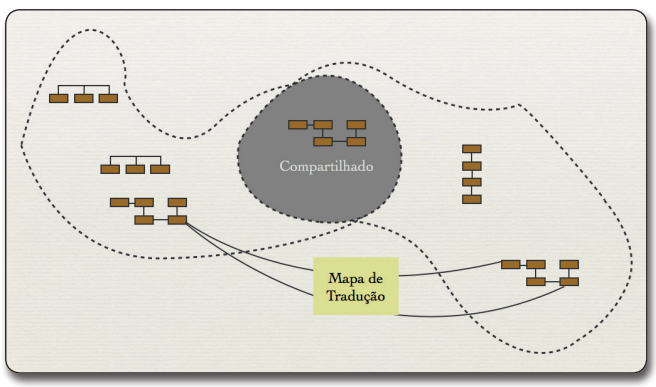


Figura 3. Núcleo Compartilhado

Camada Anti-corrupção – Quando temos um sistema legado, com código muito bagunçado e uma interface complexa, e estamos escrevendo um sistema novo, criamos uma camada entre esses dois sistemas (**Figura 4**). O novo sistema falará com essa camada, que possui uma interface bem feita. A camada anti-corrupção é responsável por traduzir e adaptar as chamadas para o sistema legado, usando uma fachada interna;

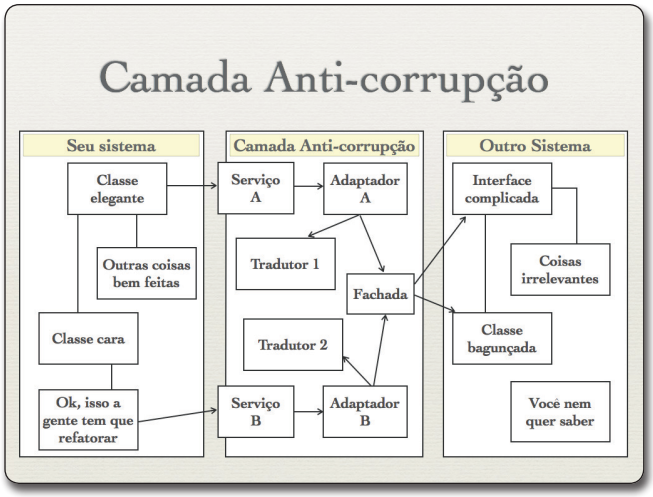


Figura 4. Camada Anti-corrupção

• **Caminhos Separados** – Quando o custo de integração é muito grande e os benefícios pequenos, times podem decidir seguir caminhos em que dois sistemas não dependam um do outro. Esse padrão é muito usado principalmente em partes pequenas e periféricas do domínio. Um exemplo seria optar por ter um cadastro de CEP local num sistema de vendas online, ao invés de integrar com o sistema dos correios;
• **Serviço Aberto de Funcionalidades e Linguagem Publicada** – Quando temos vários clientes que interagem com uma mesma parte do nosso sistema, não é conveniente que criemos adaptações para cada um desses clientes. A solução para essa situação é criar um **Serviço Aberto de Funcionalidades** e criar uma **Linguagem Publicada**. Essa linguagem é a documentação de uma API que poderá ser usada pelos vários clientes. Hoje em dia, vários serviços de internet utilizam esse tipo de abordagem: Google, Flickr, Twitter, Amazon, etc. Todos esses serviços possuem API pública, que pode ser usada por programadores que queiram criar sistemas que interajam com esses serviços.

Em busca do Núcleo

A parte mais importante e que, de fato, torna o uso de DDD interessante, é o processo chamado de Destilação do Domínio. A perfeição é atingida quando não temos mais nada para tirar daquilo que chamamos **Núcleo do Domínio** (Core Domain). Inicialmente, o nosso sistema é um bloco monolítico, que mistura código de várias camadas, classes com especialidades bem diversas. Nosso trabalho é ir separando em módulos, refatorando, extraindo métodos, classes, conceitos. É preciso organizar tudo, de forma que tenhamos bem claro quais são os conceitos centrais do nosso negócio. Esses conceitos, centrais, são nosso núcleo.

Tudo que não fizer parte do núcleo deve ser extraído e separado em sub-domínios genéricos. Apesar de serem elementos que fazem parte do nosso domínio, não são centrais. Uma vez que conceitos secundários são identificados, deve-se dar menor prioridade para eles do que para aqueles elementos que estão no **Núcleo do Domínio**. Eventualmente, vale a pena considerar adotar soluções prontas (de prateleira) para esses casos, ao

invés de manter a implementação sob responsabilidade do time de desenvolvimento. Um exemplo: imagine que estamos desenvolvendo um sistema para uma empresa de entregas internacionais. Nesse sistema temos que tratar com seriedade a questão do fuso-horário, já que esse assunto pode ser crucial para o agendamento e pontualidade nas entregas. Apesar disso, "fuso-horário" não é, provavelmente, um conceito que faz parte do núcleo do nosso negócio. Podemos desenvolver nosso próprio código de conversão de horários entre vários fusos, mas a melhor alternativa seria usar alguma biblioteca pronta. Provavelmente existe uma que irá atender às nossas necessidades.

Para tornar claro para todos o que é exatamente o Núcleo do Domínio, convém escrever um documento pequeno, de no máximo uma página, com uma Sentença da Visão do Domínio.

Um exemplo de **Sentença da Visão do Domínio seria**:

"O modelo representa compras feitas por clientes numa loja. O cliente pode passear pela loja e visualizar produtos, que estão divididos em categorias. O cliente coloca os produtos que deseja num carrinho de compras e depois passa no caixa para efetuar o pagamento, que pode ser feito em dinheiro ou cartão. Após o pagamento, os produtos são oficialmente retirados do estoque da loja."

Apesar de serem importantes, as especificações abaixo não fazem parte da visão de domínio:

"Todos os produtos são identificados por códigos de barra. Os códigos são lidos no caixa por um terminal e os preços são mostrados na tela, para o cliente, quando o código é lido. Os dados de compra são enviados através da rede sem fio e cadastrados num banco de dados, que deve ter 99,9% de disponibilidade."

Algumas vezes, essa visão resumida não é suficiente para documentar todo o Núcleo. Outra forma de ressaltar as partes que fazem parte do coração do domínio é usar o padrão **Núcleo Destacado**. Para isso, criamos um documento mais extenso, mas que não passe de sete páginas, explicando todos os elementos do núcleo e a forma como esses elementos interagem. Também podemos definir um formato especial, destacado, para representar elementos do núcleo no modelo de domínio. Assim, qualquer pessoa que estiver olhando a documentação geral do modelo, saberá claramente quais elementos são fundamentais e quais não são.

Conclusões

Extrair a essência do domínio, dentre milhares de linhas de código de um sistema complexo nem sempre é fácil. O trabalho de refinamento e busca de uma visão clara é contínuo. A refatoração é um processo incessante de busca por melhorias de projeto. Observe a sequência de imagens da **Figura 5**. Essas imagens fazem parte um estudo feito por Pablo Picasso, que durou cerca de dois meses. Durante esse tempo, o pintor refez várias vezes o mesmo desenho do touro. A cada passo ele identificava partes do desenho anterior que poderiam ser removidas, sem que a essência da idéia fosse perdida. Aplicar DDD é utilizar Padrões com o objetivo de extrair e reconhecer a essência de um sistema.

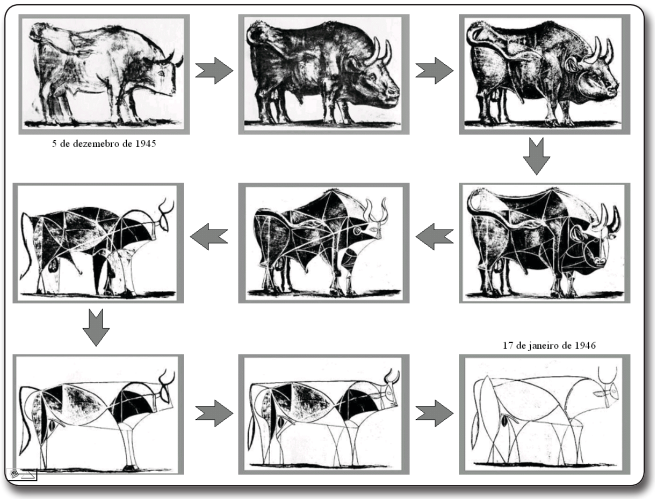


Figura 5. Refatorando para compreender a essência

Os padrões citados nesse artigo são apenas alguns dos descritos em Domain Driven Design. DDD é uma forma de desenvolver software que bastante ligado a boas práticas de Orientação a Objetos. Além disso, a própria idéia de Padrões, que promove eficácia na comunicação, é um dos valores pregados pelos agilistas. Obviamente não é possível entender DDD lendo poucas páginas. O conselho que damos é ler o livro do Eric Evans e começar a aplicar esses padrões no seu próprio negócio. Certamente são técnicas que levarão ao desenvolvimento de serviços de qualidade, sistemas seguros e fáceis de dar manutenção, levando, consequentemente, à satisfação dos seus clientes com a rapidez que o mercado de hoje exige. ●

Links

Mini-book de DDD
http://www.infoq.com/minibooks/domain-driven-design-quickly
Domain Driven Design
http://domaindrivendesign.org/
Vídeo de Introdução a DDD
http://vimeo.com/3545313
Vídeos sobre Projeto Estratégico
http://vimeo.com/3972348
http://www.infoq.com/presentations/strategic-design-evans
Vídeo Colocando o Modelo para Funcionar
http://www.infoq.com/presentations/model-to-work-evans
Eric Evans, Domain Driven Design – Tackling Complexity in the Heart of Software, 2004.

Depuração: a arte de encontrar e remover bugs

Um estudo de caso com o IDE Eclipse

Victor Vidigal Ribeiro
victorvidigal@gmail.com
Graduado do Curso de Bacharelado em Sistemas de Informação pela Faculdade Metodista Granbery e programador JSF da Granbery Consultoria Junior atuando em um projeto vinculado à COPPETEC.

Marco Antônio Pereira Araújo
maraujo@granbery.edu.br
Doutor 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 e Coordenador do curso de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora, Professor do curso de Bacharelado em Sistemas de Informação da Faculdade Metodista Granbery, Professor e Diretor do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Fundação Educacional D. André Arcoverde, Analista de Sistemas da Prefeitura de Juiz de Fora, Colaborador da Engenharia de Software Magazine.

De que se trata o artigo?
Apresentação dos principais conceitos de depuração de código fonte e demonstração de um estudo de caso utilizando o Eclipse, visando mostrar as principais funcionalidades deste IDE na depuração de código.

Para que serve?
A depuração é o processo que vai desde a localização do erro até a sua correção. Com menos defeitos, pode-se obter um produto final com maior qualidade.

Em que situação o tema é útil?
Depois de detectar erros no software a partir de seus testes, por exemplo, é preciso rastreá-los e corrigi-los. Essa é a finalidade da depuração.

Por mais experiente que seja, é natural ao programador introduzir defeitos durante a construção de um software. Além disto, esses estão se tornando cada vez mais complexos, aumentando a probabilidade de inserção de mais defeitos. Ao mesmo tempo, se exige cada vez mais do software, fazendo com que técnicas para aumentar sua qualidade sejam indispensáveis. A depuração pode ser considerada uma dessas técnicas que tem por objetivo localizar e corrigir os defeitos em um software.

Assim, da codificação de um software até suas execuções realizadas pelos usuários, é bem provável que defeitos sejam detectados e, consequentemente, precisam ser corrigidos. Quando um defeito é detectado, não se sabe exatamente onde ele aconteceu, sabe-se apenas que o mesmo existe. O processo de localização e correção destes defeitos é chamado de depuração.

A depuração deveria ser um processo ordenado. Porém, há situações em que isto não é possível, pois, muitas vezes, a manifestação externa do problema ocorrido pode não ter relação óbvia com sua causa interna. Por essa dificuldade em formalizar a depuração, alguns autores como Pressman, por exemplo, a consideram mais que uma técnica, uma arte.

Definitivamente, a depuração não é uma tarefa simples e a dificuldade em depurar é enfatizada pelo fato de não podermos contar com técnicas confiáveis.

Apensar disso, alguns cuidados devem ser tomados no momento da correção de defeitos para evitar retrabalho e aumentar a eficiência da depuração. Ao detectar um defeito, é importante analisar se há outra parte do programa com a mesma causa. Por exemplo, se foi detectado que algum problema está ocorrendo por uma inicialização errada de uma variável, todas as outras inicializações devem ser analisadas. Ou, caso não seja viável, deve-se analisar todas as inicializações de variáveis do módulo onde o defeito foi encontrado. Com isso, defeitos que possuem a mesma causa podem ser corrigidos até mesmo antes de se manifestarem.

Além disto, também é importante verificar se a alteração que está sendo realizada para corrigir o defeito encontrado não irá causar problemas em outras partes do sistema que estão funcionando. Apesar de ser importante fazer este tipo de verificação, em sistemas com baixo acoplamento, onde os módulos são mais independentes, é mais difícil desses problemas acontecem.

Outra verificação importante de se fazer é identificar o que poderia ter sido feito para que o defeito não ocorresse. Com isto, futuros problemas semelhantes aos já encontrados vão sendo evitados.

Como mencionado anteriormente, a tarefa de depuração não é simples e ainda não existem técnicas com eficiência comprovada. No entanto, ferramentas de depuração vêm sofrendo constantes evoluções e se mostrando cada vez mais indispensáveis nesta tarefa.

Estudo de Caso

Um estudo de caso será utilizado com o IDE Eclipse apresentando uma aplicação que contém uma classe chamada Aluno com seus atributos, métodos de acesso a esses atributos e um método calcularAprovacao() que retorna verdadeiro caso o aluno seja aprovado e falso caso seja reprovado. Além dessa, será desenvolvida uma classe de testes chamada CalcularAprovacaoTeste contendo cinco casos de testes que executam o método calcularAprovacao(). A aplicação calcula se um aluno foi ou não aprovado levando em consideração sua freqüência, nota 1, nota 2 e nota final. Os valores para as notas e freqüência podem variar de 0 a 100. Se um aluno não atinge um mínimo de 75 na freqüência, ele é reprovado. Caso ele possua um valor igual ou maior que 75 de freqüência suas notas serão avaliadas. Caso o aluno tenha a média das duas primeiras notas menor que 30, ele é reprovado. Caso essa média seja maior ou igual que 70, ele é aprovado. Caso contrário, a terceira nota é considerada e, caso a média entre a primeira média e a nota final seja inferior a 50, o aluno é reprovado, caso contrário, é aprovado.

O primeiro passo para iniciar o estudo de caso é criar um projeto Java com o nome CalcularAprovacao.

Depois disto, o pacote br.com.esmagazine.calcularAprovacao.model deve ser criado e também deve ser criada a classe Aluno dentro desse pacote. Essa é a classe de domínio da aplicação que contém os atributos frequencia, nota1, nota2 e notaFinal, bem como o método calcularAprovacao() que verifica se um aluno foi ou não aprovado. É possível criar um pacote clicando com o botão direito sobre a pasta src e, logo em seguida, acessando o menu New / Package.

Também deve ser criado o pacote br.com.esmagazine.calcularAprovacao.testes com a classe CalcularAprovacaoTeste, que é a classe que contém os casos de teste.

Neste ponto, a organização do projeto deve estar semelhante à **Figura 1**.

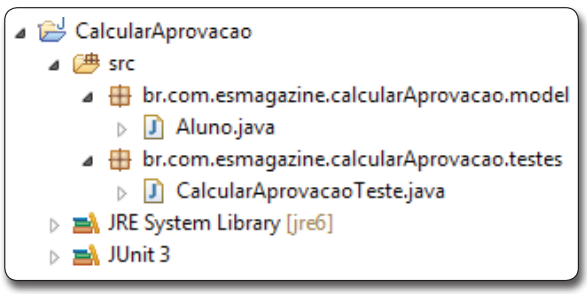


Figura 1. Organização do Projeto

Agora é preciso escrever o conteúdo das classes criadas. Como o intuito do artigo é a depuração, vamos apenas apresentar nas **Listagens 1** e **2** o conteúdo das classes Aluno e CalcularAprovacaoTeste.

Depois de construir o projeto, a classe CalcularAprovacaoTeste pode ser executada. Para isto, clique com o botão direito sobre essa classe e selecione a opção Run As / JUnit Test. Como o framework de teste JUnit já vem acoplado ao Eclipse, ele será chamado e, por sua vez, irá executar cada caso de teste criado anteriormente.

A **Figura 2** mostra o resultado dos testes executados onde é possível perceber que o teste testAlunoAprovadoNota falhou.

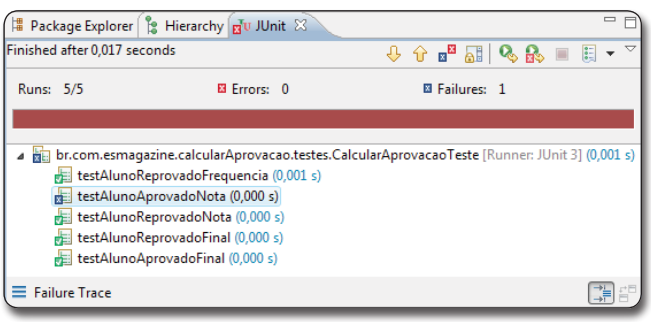


Figura 2. Resultado dos Testes

Levando em consideração que os testes estão corretos, fica claro que existe um problema no método calcularAprovacao(). Logo, um problema foi detectado, mas não se sabe exatamente em que parte do método está. A depuração tem início nesse ponto, onde se procura exatamente onde o problema está acontecendo, auxiliando na sua correção.

O primeiro procedimento que deve ser feito é definir um Breakpoint, ou ponto de parada, que são marcações no código fonte que fazem com que a aplicação pare a execução em um determinado ponto e espere um comando do programador.

Como o erro aconteceu no caso de teste testAlunoAprovadoNota

Listagem 1. Classe Aluno.java

```
package br.com.esmagazine.calcularAprovacao.model;

public class Aluno {

    private String nome;
    private int frequencia;
    private float nota1;
    private float nota2;
    private float notaFinal;

    public String getNome() {

           return nome;
    }
    public void setNome(String nome) {

           this.nome = nome;
    }
    public int getFrequencia() {

           return frequencia;
    }
    public void setFrequencia(int frequencia) {

           this.frequencia = frequencia;
    }
    public float getNota1() {

           return nota1;
    }
    public void setNota1(float nota1) {

           this.nota1 = nota1;
    }
    public float getNota2() {

           return nota2;
    }
    public void setNota2(float nota2) {

           this.nota2 = nota2;
    }
    public float getNotaFinal() {

           return notaFinal;
    }
    public void setNotaFinal(float notaFinal) {

     this.notaFinal = notaFinal;
    }

    public boolean calcularAprovacao(){

           float media;

           if (this.frequencia < 75){

               return false;

           } else {

             media = (this.nota1 + this.nota2) / 2;

             if (media < 30) {

                  return false;

             } else {

                  if (media > 70) {

                        return true;

                  } else {

                       if (((media + this.notaFinal) / 2) >= 50) {

                            return true;

                       } else {

                            return false;

                       }

                  }

             }

           }
    }
}
```

vamos colocar um Breakpoint no início deste caso de teste. Para isto, abra a classe CalcularAprovacaoTeste, clique sobre a primeira linha do caso de teste testAlunoAprovadoNota e pressione a combinação de teclas Ctrl + Shift + B. Com isto, a linha é marcada com uma bola azul. Outra forma de definir um Breakpoint é com um duplo clique sobre a barra cinza ao lado da linha desejada, ou ainda, através do menu Run / Toggle Breakpoint.

Depois de definir o ponto de parada, é preciso rodar novamente os casos de teste, mas agora clicando com o botão direito sobre a classe CalcularAprovacaoTeste e escolhendo a opção Debug As / JUnit Test.

Executando os casos de testes em modo de debug o eclipse fornece uma série de funcionalidades que auxiliam na tarefa de depuração. Como exemplo de funcionalidade pode ser citado o uso de Breakpoints já mencionados anteriormente.

Ao detectar que o fluxo de execução do sistema está passando por um Breakpoint, o Eclipse sugere que sua perspectiva visual seja alterada. Caso aceite a alteração, o eclipse muda de perspectiva fornecendo um ambiente com funcionalidades específicas para a depuração.

Com isto, o IDE ficou dividido em algumas partes. A parte esquerda superior, demonstrada na **Figura 3**, é subdividida em duas abas. Na aba Debug, também chamada de Debug View, é exibida a pilha de chamada dos métodos e cada thread executada. Nessa aba podem ser encontrados também os principais botões que controlam o fluxo da depuração. Esses botões são exibidos na **Tabela 1**, bem como seus respectivos atalhos, ícones e descrição da funcionalidade que oferecem.

Botão	Ícone	Atalho	Descrição
RemoveAllTerminated Launches		-	Limpa todas as seções de Debug terminadas
Resume		F8	Continua o fluxo de execução do programa até o próximo ponto de parada
Suspend		-	Pausa a thread atual. Pode ser utilizada em laços infinitos, por exemplo.
Terminate		Ctrl+F2	Termina a depuração
Step Into		F5	Entra no método selecionado e continua a execução dentro desse método
Step Over		F6	Apenas passa a execução para o próximo passo sem entrar no método selecionado
Step Return		F7	Volta a execução para o método que chamou o método atual
Drop to Frame		-	Retorna a execução para o início do método atual
Use Step Filters		-	Habilita/Desabilita filtros de execução. Não é possível executar as classes que estão neste filtro caso ele esteja habilitado. Pode ser utilizado para que a depuração não caia em uma classe que não se tenha o código fonte.

Tabela 1. Botões de Depuração

A aba Servers caso sua aplicação esteja rodando em um servidor como TomCat ou ClassFish, permite que se tenha controle sobre esse servidor, possibilitando pará-lo ou iniciá-lo, por exemplo.

Listagem 2. Classe CalcularAprovacaoTeste.java

```
package br.com.esmagazine.calcularAprovacao.testes;

import junit.framework.TestCase;

import br.com.esmagazine.calcularAprovacao.model.Aluno;

public class CalcularAprovacaoTeste extends TestCase{

    public void testAlunoReprovadoFrequencia() {
        Aluno aluno = new Aluno();
        aluno.setNome("João");
        aluno.setFrequencia(74);

        assertFalse(aluno.calcularAprovacao());
    }

    public void testAlunoAprovadoNota() {

        Aluno aluno = new Aluno();
        aluno.setNome("João");
        aluno.setNota1(70);
        aluno.setNota2(70);
        aluno.setFrequencia(75);

        assertTrue(aluno.calcularAprovacao());
    }

    public void testAlunoReprovadoNota() {
        Aluno aluno = new Aluno();
        aluno.setNome("João");
        aluno.setNota1(29);
        aluno.setNota2(30);
        aluno.setFrequencia(75);

        assertFalse(aluno.calcularAprovacao());
    }

    public void testAlunoReprovadoFinal() {
        Aluno aluno = new Aluno();
        aluno.setNome("João");
        aluno.setNota1(30);
        aluno.setNota2(30);
        aluno.setNotaFinal(69);
        aluno.setFrequencia(75);

        assertFalse(aluno.calcularAprovacao());
    }

    public void testAlunoAprovadoFinal() {
        Aluno aluno = new Aluno();
        aluno.setNome("João");
        aluno.setNota1(30);
        aluno.setNota2(30);
        aluno.setNotaFinal(70);
        aluno.setFrequencia(75);

        assertTrue(aluno.calcularAprovacao());
    }

}
```

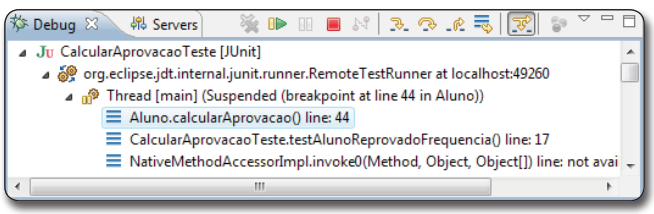


Figura 3. Abas Debug e Servers

Do lado direito superior também são encontradas duas abas vistas com mais detalhes mais adiante. Em Variables podem ser localizadas as variáveis ou objetos que já foram inicializados, incluindo os parâmetros dos métodos que estão sendo depurados.

Além de exibir os valores das variáveis ou objetos, é possível alterar esses valores manualmente clicando com o botão direito sobre a linha desejada e selecionando a opção Change Value. Depois de clicar sobre esta opção, uma janela é exibida onde é possível digitar o novo valor desejado.

A outra aba chamada Breakpoints exibe os pontos de parada que estão marcados no código. É possível habilitar ou desabilitar os pontos de parada marcando ou desmarcando o check que se localiza à sua esquerda. Também é possível remover um ou todos os pontos de parada clicando com o botão direito sobre ele e escolhendo a opção Remove ou Remove All.

O centro da tela fica dividido em duas partes, sendo que à esquerda está o código fonte. É possível adicionar ou remover pontos de parada mesmo depois que a depuração já foi iniciada. Através da **Figura 4** é possível notar que o Eclipse indica em qual ponto a execução está, colorindo a linha na cor verde e colocando uma seta azul à esquerda dessa linha.

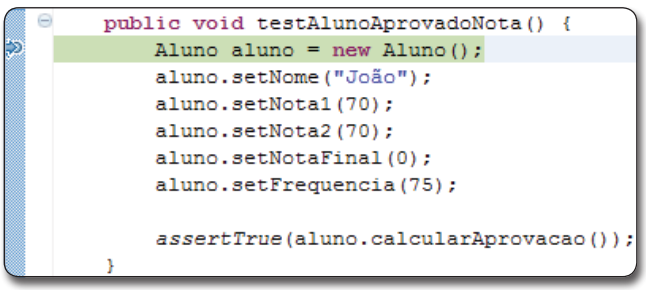


Figura 4. Ponto de execução

Ainda no centro, porém do lado direito, está localizada a aba Outline que exibe de forma hierárquica a classe atual que está sendo depurada, mostrando seus métodos e atributos, como pode ser visto na **Figura 5**.

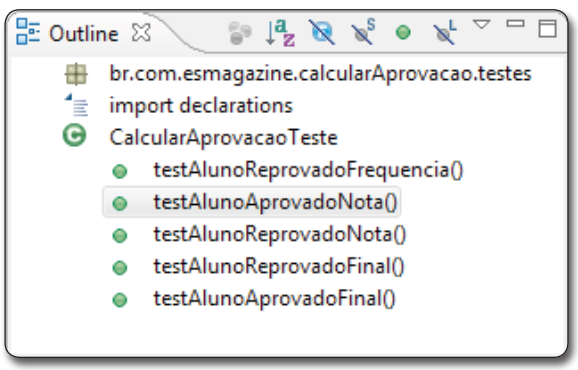


Figura 5. Aba Outline

Depois de conhecer as principais funcionalidades do ambiente de depuração do Eclipse, é preciso continuar com a depuração que foi iniciada anteriormente.

Lembrando que o sistema está parado na primeira linha do caso de teste testAlunoAprovadoNota da classe CalcularAprovacaoTeste onde um objeto do tipo Aluno é instanciado.

A opção Step Over (F6) deve ser escolhida para que o objeto aluno seja criado e o fluxo de execução passe para a próxima linha.

A próxima linha chama o método setNome para atribuir um nome ao aluno. Com a opção Step Into (F5), é possível fazer com que o fluxo de execução entre nesse método, com isto pode-se verificar se seu conteúdo está correto, ou seja, verificar

se o método setNome está realmente atribuindo o valor para o atributo nome da classe Aluno. Esta verificação deve ser feita para cada método set utilizado neste caso de teste.

A aba Variables pode ser utilizada para verificar o valor de cada atributo ou simplesmente posicione o cursor do mouse sobre o atributo desejado que o Eclipse exibirá seu valor atual.

Uma dica básica, porém muito importante, é a utilização das teclas de atalho que possibilitam a escolha dos comandos sem ter que localizar o cursor do mouse e clicar na opção desejada, pois com isso é possível ter uma maior concentração no código fonte.

A **Figura 6** mostra o estado da aba Variables no momento em que o fluxo de execução se encontra no final do método setFrequencia() que é o último método set chamado pelo caso de uso.

Em Variables, os valores que estão dentro do grupo this e que possuem um ícone vermelho à sua esquerda correspondem aos atributos da classe, e os valores que estão marcados com ícone cinza são os valores que o método set recebeu como parâmetro. Com isto, é possível verificar se os atributos da classe Aluno estão recebendo os valores esperados.

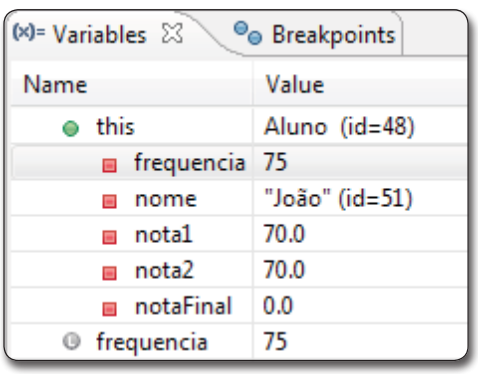


Figura 6. Aba Variables

Imagine agora que esteja no meio dessa execução e descobre-se que o atributo frequencia deveria ter o valor 80 e não 75. Teoricamente seria preciso parar a execução da aplicação, alterar o caso de teste, executar o caso de teste novamente e seguir o fluxo da aplicação até esse ponto.

Porém, o Eclipse fornece uma funcionalidade muito útil nesses casos. É possível alterar o valor dos atributos em tempo de depuração.

Para demonstrar esta funcionalidade, o valor do atributo frequencia pode ser alterado clicando com o botão direito sobre ele e em seguida selecionando a opção Change Value. Com isto, uma janela é aberta onde o valor pode ser substituído e, com este procedimento, a aplicação irá trabalhar com o novo valor nesse atributo.

Uma outra funcionalidade que pode ser útil em alguns casos é Drop to Frame . Esta funcionalidade faz com que o fluxo de execução volte para a primeira linha do método.

Para demonstrar essa funcionalidade na prática, suponha que haja a necessidade desfazer a alteração do atributo frequencia, pois foi uma alteração equivocada. Para isto, pode ser utilizada a opção Drop to Frame para retornar a execução ao início do método setFrequencia(). Como o método será executado novamente, será atribuído o valor passado como parâmetro para o atributo frequência do objeto aluno, fazendo que receba novamente o valor 75, substituindo o valor alterado manualmente através da opção Change Value.

Outra opção útil que poderia ser usada nessa situação é Step Return (F7) que faz com que o fluxo de execução volte para o método que fez a chamada do método atual. Essa opção poderia ser utilizada caso tenha sido um equívoco entrar em um dos métodos set e deseja-se continuar o fluxo de execução na próxima linha do método calcularAprovacao(). É importante comentar que esta opção apenas não mostra a execução de um método passo a passo, mas ele é executado.

Continuando o fluxo de execução com a opção Step over (F6) o sistema retorna ao caso de teste agora com o objeto aluno já criado e com seus atributos preenchidos com os valores passados pelo formulário. Como o problema ainda não foi encontrado, o fluxo de execução deve continuar.

O próximo método a ser chamado é o método calcularAprovacao() do objeto aluno que acabou de ser instanciado. Este método é chamado dentro de outro método chamado assertTrue. Este assertTrue é um recurso do JUnit que aprova o caso de teste caso o parâmetro passado retorne verdadeiro, e reprove o caso de teste caso retorne falso.

Neste ponto, é possível entrar no método calcularAprovacao() com a opção Step Into (F5) , mas para fins didáticos, vamos fazer de forma diferente.

Entre na classe Aluno e adicione um Breakpoint na primeira linha do método calcularAprovacao(). Agora, pode-se utilizar a opção Resume (F8) . Como mencionado anteriormente, essa opção faz com que o fluxo de execução continue até o próximo Breakpoint, logo, a aplicação irá parar no Breakpoint que acabou de ser inserido.

Considerando as regras de negócio explicadas no início do artigo e os valores atribuídos ao objeto aluno, o resultado do método calcularAprovacao() deveria ser verdadeiro. Executando o método passo a passo, é possível descobrir o motivo pelo qual está retornando falso, fazendo com que o caso de teste falhe.

Pressione Step over (F6) sobre a linha que faz a verificação de freqüência. O fluxo de execução passa para a linha onde a média é calculada. Até este ponto está correto, pois o valor atribuído à freqüência realmente não é menor que 75, fazendo com que o comando if direcione o fluxo de execução para a linha onde a média é calculada.

Pressione novamente a tecla Step over (F6) para obter o valor da variável media. É possível observar na aba Variables que a média calculada é 70. Para verificar se este valor foi calculado de maneira correta pode-se somar a nota1 (70) com a nota2 (70) e dividir o resultado por 2 concluindo assim que o valor calculado está correto.

Continuando a execução existe uma condição para verificar se a média é menor que 30, como não é o caso, pressione a tecla Step over (F6) para ir para o próximo passo.

Pressione a tecla Step over (F6) novamente para seguir para o próximo passo. Pode-se notar que o passo seguinte é o calculo de uma nova média que considera a notaFinal. Mas, se recorrermos às regras de negócio, mais especificamente no trecho que diz: "Caso a média entre as duas primeiras notas seja igual ou maior que 70, ele é aprovado" é possível perceber que a condição que verifica se a média é maior ou igual a 70 está errada, pois não está considerando valores iguais a 70. Portanto o erro foi encontrado e deve ser corrigido alterando o sinal de > por >= para seguir as regras de negócio.

Depois de detectar o problema e corrigi-lo, não é mais preciso executar o sistema passo a passo até o final. Pode-se optar por parar através da opção Terminate (Ctrl+F2) ou executá-lo normalmente através da opção Resume (F8).

Depois de corrigir o problema e salvar as alterações, os casos de testes podem ser executados novamente, como mostrado na **Figura 7**, para verificar se o erro foi realmente corrigido. Porém, não é preciso que a execução pare nos breakpoints inseridos anteriormente e nem é preciso retirar todos os Breakpoints, caso precise utilizá-los novamente. Para desabilitá-los, acesse a aba Breakpoints mostrada anteriormente e desmarque o check à sua esquerda, como pode ser visto na **Figura 8**.

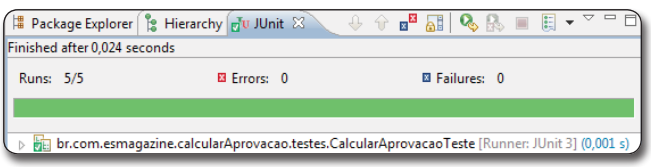


Figura 7. Execução dos casos de testes depois da depuração

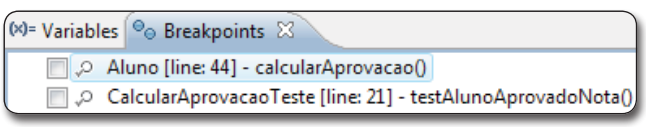


Figura 8. Breakpoint desabilitado

Suponha agora que a classe Aluno já esteja completamente testada e não se deseja mais depurar essa classe. Para isto, pode ser utilizada a opção Use Step Filters. Essa opção ativa filtros que impossibilitam depurar as classes ou pacotes que estão nesses filtros.

Um filtro pode ser criado através do menu Window / Preferences e, logo em seguida, acessando a opção Java / Debug / Step Filtering. Após isto, para adicionar a classe Aluno aos filtros, clique sobre o botão Add Class, digite Aluno na caixa de pesquisa que é exibida e selecione a classe. Neste momento, a janela de criação de filtros deve ficar semelhante à exibida na **Figura 9**.

Com este filtro criado e a opção Step Filters ativada, mesmo se o fluxo de execução estivesse parado sobre a linha demonstrada na **Figura 10** e a opção Step Into (F5) fosse selecionada, o Eclipse não iria exibir o conteúdo do método calcularAprovacao() da classe Aluno.

Outra funcionalidade importante fornecida pela ferramenta é colocar condições nos Breakpoints. Para adicionar uma condição deve-se clicar com o botão direito sobre esse Breakpoint na aba Breakpoints, ou na bola azul ao lado esquerdo da linha onde ele está e, logo em seguida clicar sobre a opção Breakpoint Properties.

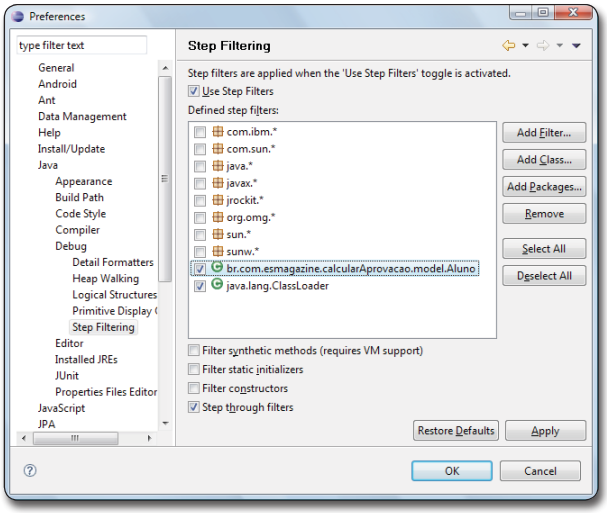


Figura 9. Criar Filtros

```
public void testAlunoAprovadoNota() {
    Aluno aluno = new Aluno();
    aluno.setNome("João");
    aluno.setNota1(70);
    aluno.setNota2(70);
    aluno.setNotaFinal(0);
    aluno.setFrequencia(75);

    assertTrue(aluno.calcularAprovacao());
}
```

Figura 10. Método CalcularAprovacao()

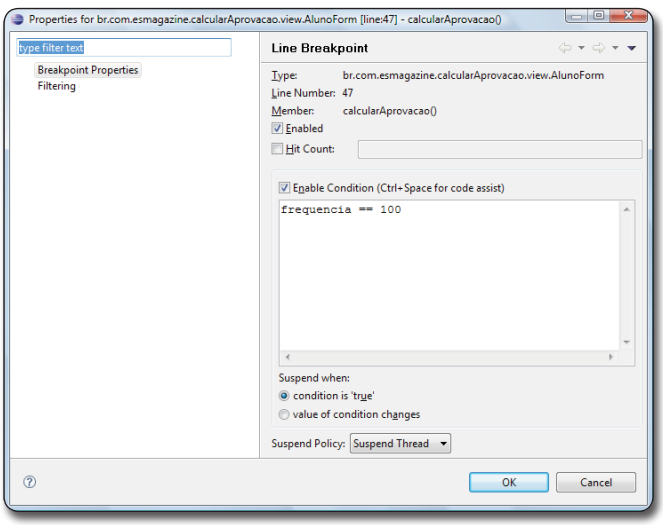


Figura 11. Adicionar Condição ao Breakpoint

Com isto, uma janela semelhante à **Figura 11** é exibida.

Nesta janela, a opção Enabled Condition deve ser marcada para que se possa digitar a condição desejada. Por exemplo, poderia ser digitado: frequencia == 100. Se essa condição tivesse sido colocada no Breakpoint do método calcularAprovacao(), o Eclipse iria parar neste Breakpoint apenas se o valor do atributo frequência for igual a 100.

Uma última função que, apesar de não estar diretamente relacionada à depuração, ajuda bastante nesta tarefa é a utilização de marcações ou Bookmarks.

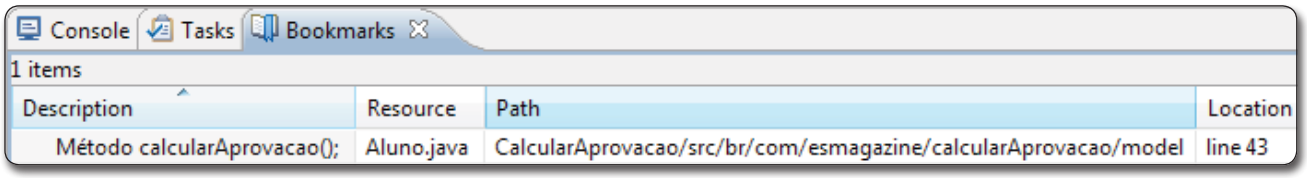


Figura 12. Marcações no código

Suponha que se esteja depurando o método calcularAprovacao() e, em algum momento, precise visualizar algum valor na classe CalcularAprovacaoTeste, por exemplo para verificar qual valor um determinado caso de teste passa para o método setNome(). Para retornar ao método calcularAprovacao(), seria preciso voltar para a classe Aluno e procurar por toda a classe até encontrar esse método novamente.

Uma maneira mais simples de voltar ao método é definindo uma marcação em uma linha desejada, por exemplo, na primeira linha do método que se deseja voltar a alterar. Para isso, clica-se com o botão direito sobre a parte cinza à esquerda da linha desejada e escolha a opção Add Bookmark. Uma janela é aberta onde deve ser digitado um nome qualquer para descrever a marcação que está sendo criada. Com isto, uma marcação é criada como pode ser visto na **Figura 12**.

Depois disto, é possível navegar por qualquer parte da classe ou até mesmo de outras classes e, quando desejar voltar para o método, basta dar um duplo clique sobre a marcação criada.

Conclusão

Este artigo procurou mostrar primeiramente que a depuração não é um processo trivial e que a falta de técnicas com eficiência comprovada em sistemas reais agrava ainda mais esta situação.

Também mostrou um estudo de caso de um fragmento de um sistema desenvolvido com o IDE Eclipse que calcula aprovação de alunos.

O que pode ser observado são as facilidades que as ferramentas de depuração oferecem como, por exemplo, execução passo a passo da aplicação e alteração dos valores das variáveis no momento da depuração. Por essas facilidades, essas ferramentas vêm se tornando cada vez mais indispensáveis ao realizar atividades de depuração.

Desde 2004 Trazendo Teoria para a Prática

Consultoria e Treinamento em Engenharia de Software

Consultoria

Arquitetura e Projeto de Software

Engenharia de Requisitos

Implantação e Melhoria de Processos

Modelagem de Processos de Negócio

Validação, Verificação e Teste de Software

Solicite nossa proposta ou se preferir agende uma reunião.

Treinamento

Engenharia de Requisitos

Gerência de Projetos de Software (com Scrum ou baseado no MPS)

Modelagem de Processos de Negócios com BPMN

Processos de Software com Ênfase em CMMI e MPS

Projeto Orientado a Objetos com UML e Padrões

Validação, Verificação e Teste de Software

Consulte nossos cursos com inscrições abertas ou
solicite uma proposta para treinamentos customizados in-company.

kalisoftware.com | info@kalisoftware.com

+55 (21) 2586-6391 - Centro Empresarial Mourisco - Praia de Botafogo 501, 1° Andar - Bloco Pão de Açúcar - Rio de Janeiro - RJ. CEP: 22250-040

Chegou o Sistema de Créditos DevMedia

Agora o **conteúdo completo** do nosso site está **ao seu alcance!**

A partir de agora todas as **2000 vídeo-aulas** do site DevMedia podem ser compradas individualmente.

Economia - Você não precisa mais assinar oito produtos diferentes para ter acesso ao conteúdo completo do site!

Todas as vídeos que você sempre quis ver a partir **R$ 0,75!**

Saiba mais sobre o Sistema de Créditos!

DevMedia GROUP

www.devmedia.com.br/creditos