Desenvolvimento: Linhas de produtos de software: introdução, conceitos e desafios para sua adoção

engenharia de software magazine

Ano 2 :: Edição 15

Metodologias Ágeis
Entenda como realizar a priorização de funcionalidades

Engenharia de Software
Introdução à governança de tecnologia de informação

Projeto
Aprenda a elaborar diagrama de sequência

Projeto
Incorporando restrições à UML com OCL

Teste de Software

- Principais etapas do processo de teste de software
- Desafios para os testes numa fábrica de software
- Plano de teste - um 'mapa' essencial para o teste de software

Brinde! Compre esta edição e ganhe 5 vídeo-aulas

- Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 7
- Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 8
- Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 9
- Introdução à Matriz de Rastreabilidade – Parte 1
- Introdução à Matriz de Rastreabilidade – Parte 2

Conhecimento faz diferença!

engenharia de software magazine

Requisitos
Desenvolvimento de software dirigido por casos de uso – Parte 2

Planejamento
Conheça abordagens e modelos que apóiam a gerência de riscos

DevMedia Ano I - Edição 03

Melhoria de Processos de Software com o uso de Análise Causal de Defeitos

Planejamento
Plano de Projeto: Um 'Mapa' Essencial à Gestão de Projetos de Software

Requisitos
Entenda o que são requisitos não funcionais e como eles podem impactar a arquitetura de seu sistema

Projeto
Saiba como identificar e especificar componentes de negócio usando como base casos de uso e diagramas UML

Metodologias Ágeis
A importância dos testes automatizados

Verificação, Validação & Teste
Ferramentas Open Source e melhores práticas na gestão de testes.

Aulas desta edição:

- Introdução ao MS Project - Parte 01
- Introdução ao MS Project - Parte 02
- Introdução à Engenharia de Requisitos - Parte 07
- Introdução à Engenharia de Requisitos - Parte 08
- Introdução à Engenharia de Requisitos - Parte 09
- Coleta e análise de métricas com Metrics for Eclipse
- Teste Unitário com JUnit
- Teste de Cobertura com Eclemma
- Teste Funcional com Selenium-IDE

Ano: 01 – Edição 02

Gerência de Configuração
Desenvolva software de forma eficien[te] e disciplinada

Planejamento
Conheça os principais conceitos envolvidos na gerência de riscos

Processo
MPS.BR – Mitos e Verdades de um Modelo de Maturidade

Análise [de] Pontos

Projeto
Entenda os principais conceitos de SOA – Service Oriented Architecture

Requisitos
Desenvolvimento de software dirigido por casos de uso

Projeto
Aprenda a construir diagramas da UML com base em bons princípios de modelagem OO

Requisitos
Conheça algumas das principais técnicas para apoiar a identificação de requisitos

Edição Especial

Requisitos
Conheça os principais conceitos envolvidos na Engenharia de Requisitos

Projeto
Entenda o conceito de Arquitetura de Software e como trabalhar com os Estilos Arquiteturais

Especial | Processos

Melhore seus processos através da análise de risco e conformidade | Veja como integrar conceitos de Modelos Tradicionais e Ágeis | Veja como integrar o Processo Unificado ao desenvolvimento Web

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 - 15ª Edição 2009 Impresso no Brasil

Corpo Editorial

Colaboradores
Rodrigo Oliveira Spínola
rodrigo@sqlmagazine.com.br

Marco Antônio Pereira Araújo
Eduardo Oliveira Spínola

Diagramação
Gabriela de Freitas - gabrieladefreitas@gmail.com

Capa
Romulo Araujo - romulo@devmedia.com.br

Na Web
www.devmedia.com.br/esmag

Apoio

Parceiros:

Atendimento ao Leitor

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

Cristiany Queiroz – Atendimento ao Leitor
http://www.devmedia.com.br/mancad/
(21) 2220-5375

Kaline Dolabella
Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 2220-5375

Publicidade

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

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

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

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

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

EDITORIAL

Um importante fator de sucesso de um software é sua qualidade. Existem diversas maneiras de se avaliar a qualidade de um produto, uma dessas maneiras é a realização de atividades de testes de software por uma equipe especializada no assunto. Neste contexto, a Engenharia de Software Magazine destaca nesta edição três matérias muito interessantes que abordam o tema.

- **Testes de Software, um processo criativo:** Este artigo aborda o tema processo de testes de software mostrando uma maneira de agrupar as atividades de testes em etapas, com objetivos bem definidos. Ele mostra que a organização das atividades pode ter um impacto positivo significativo na qualidade do produto gerado, e que muito mais do que burocracia é preciso criatividade para um bom desempenho de tais atividades.
- **Manutenção:** Desafios para os Testes numa Fábrica de Software: Neste artigo serão apresentadas dicas e sugestões para superar os principais desafios e entraves associados às atividades de testes na manutenção de softwares em um ambiente de Fábrica de Software.
- **Plano de Teste - Um 'Mapa' Essencial para Teste de Software:** Este artigo apresenta o plano de teste de software, destacando sua importância no processo de desenvolvimento de software, mostrando como elaborá-lo e exemplificando os itens que devem compor o referido documento.

Além destas matérias, esta edição traz mais cinco artigos:

- Priorização Voltada para Resultados;
- Linhas de Produtos de Software;
- UML – Diagrama de Seqüências;
- Incorporando Restrições à UML;
- Governança de Tecnologia de Informação.

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 edição, temos um conjunto de 5 vídeo aulas. Estas vídeo aulas estão disponíveis para download no Portal da Engenharia de Software Magazine e certamente trarão uma significativa contribuição para seu aprendizado. A lista de aulas publicadas pode ser vista abaixo:

Tipo: Engenharia de Requisitos
Título: Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 7
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 sétima aula, serão apresentados problemas e sugestões de solução associadas a atividades de verificação e validação de requisitos.

Tipo: Engenharia de Requisitos
Título: Soluções Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 8
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 oitava aula, serão apresentados problemas e sugestões de solução associadas a atividades de gerenciamento de requisitos.

Tipo: Engenharia de Requisitos
Título: Concretas para Problemas Práticos da Engenharia de Requisitos – Parte 9
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 nona e última aula desta série, serão apresentados problemas e sugestões de solução associadas a atividades de gerenciamento de requisitos, ferramentas e recursos humanos associados às atividades da engenharia de requisitos.

Tipo: Projeto
Título: Introdução à Matriz de Rastreabilidade – Parte 1
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: pode ser considerado um conceito chave ao longo do desenvolvimento de projetos de software. Este conceito define que é possível mapear a relação entre os diferentes elementos elaborados durante o desenvolvimento de software. Esta relação é identificada a partir das transformações que existem em informações ao longo do processo de desenvolvimento e podem ser representadas através de "rastros" em uma matriz de rastreabilidade. Nesta vídeo aula apresentaremos as definições iniciais associadas à rastreabilidade.

Tipo: Projeto
Título: Introdução à Matriz de Rastreabilidade – Parte 2
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Rastreabilidade pode ser considerado um conceito chave ao longo do desenvolvimento de projetos de software. Este conceito define que é possível mapear a relação entre os diferentes elementos elaborados durante o desenvolvimento de software. Esta relação é identificada a partir das transformações que existem em informações ao longo do processo de desenvolvimento e podem ser representadas através de "rastros" em uma matriz de rastreabilidade. Nesta segunda parte, apresentaremos a importância do conceito de rastreabilidade e como ele pode ser aplicado através das matrizes de rastreabilidade.

ÍNDICE

08 - Priorização Voltada para Resultados
Dairton Bassi

14 - Incorporando Restrições à UML
Thiago Carvalho de Sousa

22 - UML – Diagrama de Sequências
Ana Cristina Melo

30 - Testes de Software, um processo criativo
Melissa Barbosa Pontes

36- Manutenção: Desafios para os Testes numa Fábrica de Software
Daniel Scaldaferri Lages

42 - Plano de Teste
Antonio Mendes da Silva Filho

48 - Linhas de Produtos de Software
Pasqueline Scaico, Alexandre Scaico e Fillipe Lourenço da Cunha Lima

55 - Governança de Tecnologia de Informação
Monalessa Perini Barcellos e Alex Sandro Barreto Rodrigues

Você não está mais **sozinho.**

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online 21 3382-5025

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Priorização Voltada para Resultados

De que se trata o artigo?

Neste artigo entenderemos os problemas causados pela falta de priorização nas funcionalidades durante a implementação de projetos de software e apresentaremos técnicas usadas em metodologias ágeis para determinar as prioridades de desenvolvimento.

Para que serve?

Com bons critérios de priorização, a equipe de desenvolvimento terá foco e poderá trabalhar unida para atingir seu objetivo. Além disso, o produto em desenvolvimento terá resultados palpáveis mais rapidamente. Isso permitirá que ele seja avaliado mais cedo e aprimorado.

Em que situação o tema é útil?

A eleição de prioridades adequadas evita inúmeros desperdícios de recursos e de tempo, tornando o projeto mais propenso ao fracasso. A escolha adequada das funcionalidades que serão implementadas pode ser determinante para o cumprimento do prazo e dos custos do projeto e também pode antecipar a obtenção de versões preliminares do software.

Priorizar significa determinar uma ordem, que pode ser, por exemplo, de importância ou de precedência. Neste processo, naturalmente alguns elementos são favorecidos em detrimento de outros.

Na produção de software, decisões erradas na priorização das funcionalidades ou de módulos de um projeto podem causar estouro de orçamento, perda de prazos, retardamento na geração de receita, adiamento do uso do software, falta de foco no desenvolvimento e, em última instância, cancelamento do projeto.

Historicamente, a indústria de software não tem sido hábil em fazer priorizações e tem sofrido dos males que essa deficiência traz: grandes desperdícios de tempo e de recursos que comprometem o projeto e, em muitos casos, causam o seu fracasso.

Uma amostra representativa da indústria de software de países desenvolvidos foi estudada por Jim Jonhson em 2002. Ele descobriu que 64% das

Dairton Bassi

dbassi@gmail.com

Mestre em Engenharia de Software com ênfase em Métodos Ágeis pelo IME-USP. Bacharel em Ciência da Computação pelo IME-USP. Co-fundador da AgilCoop e criador do Encontro Ágil (www.encontroagil.com.br). Especialista em implantação de metodologias ágeis. Ministra cursos e palestras sobre métodos ágeis. Atuou como programador, líder de desenvolvimento e consultor em diversas instituições do setor público e privado.

funcionalidades implementadas são raramente ou nunca utilizadas. Este número revela que somente cerca de 1/3 do que foi desenvolvido, de fato, precisava ser feito e, portanto, dois terços dos recursos e do tempo foram mal empregados. Ratificando os resultados de Jonhson, em 2007, o PMI Brasil apontou resultados semelhantes na indústria brasileira. 66% das empresas do setor de tecnologia têm problemas para cumprir o custo dos projetos e em 82% para cumprir os prazos.

Estes números sinalizam sérios problemas relacionados à priorização. Se as prioridades fossem melhor elegidas, os 64% de funcionalidades raramente usadas provavelmente não precisariam ser feitas. Com isso, haveria uma grande economia de recursos, os projetos seriam muito mais simples e, seria possível concluir os 34% que têm valor muito mais rápido. Se isso fosse feito, os problemas com custos e prazos certamente não seriam gritantes.

Há diversas maneiras de fazer a priorização do desenvolvimento de um software. A escolha de uma delas geralmente é guiada pela filosofia da empresa ou pelas características do projeto. Neste artigo apresentamos algumas técnicas para determinar as prioridades de desenvolvimento com foco no valor de negócios trazido por cada funcionalidade. Essas técnicas envolvem a equipe de desenvolvimento, mas contam principalmente com a participação de clientes e também de potenciais usuários.

Quem é o responsável?

É muito fácil pensar que os responsáveis pela priorização, ou pela falta dela, são os gerentes do projeto, que significa que eles seriam exclusivamente culpados pelos resultados que a indústria tem apresentado. Esta seria uma análise parcial e incorreta da situação. Uma boa priorização só é possível quanto o processo de desenvolvimento reserva espaço para que ela aconteça. Dado isso, é fundamental que os envolvidos na priorização conheçam o mercado no qual o software será inserido e, principalmente, o ponto de vista dos usuários.

Modelos de desenvolvimento com escopo e prazo grandes e fixos são um dos principais responsáveis por problemas de priorização. Como nestes modelos todas as funcionalidades do escopo devem estar prontas no fim do prazo, os envolvidos não sentem tanta necessidade de priorizar, afinal, a ordem de implementação não é relevante para atender o acordo com o cliente. Isso faz com que a ordem de desenvolvimento seja forte e exclusivamente influenciada pelos aspectos técnicos do desenvolvimento. Por exemplo, as funcionalidades podem ser ordenadas de acordo com as tecnologias que elas requerem em sua implementação, ou conforme o grau de dificuldade de implementação. Em outros casos, a ordenação das funcionalidades é influenciada pela indisponibilidade de alguns recursos da equipe, como DBAs, designers, etc.

Além dessas restrições desfavorecerem o uso de priorização alinhadas com os interesses comerciais, o formato de contrato que fixa prazo e escopo é o preferido por ambas as partes envolvidas na produção do software: clientes e desenvolvedores. Isso acontece porque tanto clientes como desenvolvedores querem o contrato pelo mesmo motivo: aumentar a sua segurança. O comprador quer listar as funcionalidades que ele deseja para ter uma garantia de que irá recebê-las. O desenvolvedor também quer listar as funcionalidades para estabelecer um limite sobre o tamanho do seu trabalho e de suas responsabilidades. O grande problema deste modelo é que o comprador irá querer incluir o máximo de funcionalidades, pois ele não sabe precisamente quais ele irá querer ou de fato usar daqui a alguns meses ou anos. Na dúvida, se uma funcionalidade será ou não útil, é melhor incluí-la. "E se alguém precisar..."

Com as funcionalidades e o prazo definidos, todas elas precisam ficar prontas, por isso são tratadas igualmente ou planejadas sob pontos de vista que não refletem as prioridades de negócio do projeto, criando uma seqüência de implementação motivada puramente por questões técnicas. Priorizações que consideram somente o ponto de vista técnico podem implicar, depois de alguns meses, em uma grande quantidade de código comercialmente inútil. Isso acontece quando funcionalidades sem valor para os usuários são implementadas no início do projeto, enquanto funcionalidades importantes ficam para o fim, correndo o risco de não serem implementadas caso o projeto atrase ou seja cancelado justamente por falta de resultados.

Quando a ordem importa

Priorizar, basicamente, significa definir uma ordem para que as funcionalidades sejam implementadas. O estabelecimento de critérios para ordená-las beneficia o desenvolvimento de diversas maneiras, independente do tipo de processo que se use. A eleição destes critérios é importante para determinar uma direção para o desenvolvimento e facilitar a definição de metas intermediárias para a equipe.

Para compor os critérios de priorização, podem ser usados vários tipos de informação. Alguns exemplos são: a opinião dos usuários, o retorno financeiro ou a dificuldade de implementação. Bons critérios de priorização consideram mais de uma informação, como por exemplo, o retorno financeiro e a dificuldade de implementação. Assim, fatores cruciais para o sucesso comercial do software (retorno financeiro) são ponderados com variáveis que indicam a viabilidade de produção (dificuldade de implementação).

Em projetos onde é preciso conseguir receita para garantir a sua continuidade, ou para justificar mais investimentos, a ordem de implementação deve estar alinhada com os interesses comerciais. Neste caso, priorizar pelo retorno financeiro ou valor agregado são boas opções. Com este critério de implementação, as funcionalidades mais importantes estarão prontas rapidamente e, mesmo sem que o produto inteiro esteja pronto, será possível testar e fazer demonstrações que exibem o potencial do software final.

A apresentação de funcionalidades que tornam o software capaz de gerar receita muda a percepção que o cliente tem a respeito do projeto. Seja ele um cliente externo, um investidor ou níveis superiores da própria empresa que faz o desenvolvimento, o projeto que era visto como uma fonte de gastos, ou como um investimento de alto risco, passa a ser tratado como

um ativo valioso. Como conseqüência, a equipe de desenvolvimento também passa a ser mais valorizada.

Se a urgência pelo produto for grande, uma priorização voltada para as principais funcionalidades pode criar rapidamente uma versão simplificada, ou beta, do produto. Dessa forma, as próximas funcionalidades podem ser criadas enquanto uma versão já gera receita. Isso coloca o projeto em uma condição auto-sustentável e a equipe de desenvolvimento em uma situação mais confortável com relação à cobrança por resultados.

Desenvolvimento iterativo, priorização iterativa

Modelos de desenvolvimento iterativos têm se mostrado adequados ao dinamismo com que as regras e as necessidades do mercado se comportam. Por meio de iterações e desenvolvimento incremental é possível segmentar a produção de um grande software e refinar as suas características desde o início do projeto, evitando pagar o alto preço de grandes correções tardias.

Empresas que usam métodos ágeis, como XP ou Scrum, tendem a fazer iterações curtas com duração de algumas semanas cada uma. Com isso, é possível rever e ajustar o planejamento, além de validar com freqüência o que já foi produzido. Pequenas iterações também ajudam na manutenção das prioridades, pois com revisões periódicas, novas demandas podem ser consideradas e o desenvolvimento pode se manter sensível às oscilações do mercado.

Periodicidade da Priorização

O tamanho das iterações está relacionado com características do projeto. Por exemplo, projetos sujeitos a muitas mudanças devem usar iterações curtas para evitar que as necessidades mudem durante a iteração. Por outro lado, se houver dificuldades para conseguir feedback ou uma validação sobre o que é desenvolvido, iterações maiores são mais adequadas, pois permitem que haja tempo hábil para a entrega.

Também é importante que a periodicidade com que as prioridades são revistas mantenha uma relação com o tamanho das iterações. O ideal é que a periodicidade das prioridades seja um múltiplo do tamanho da iteração, podendo ocorrer, por exemplo, junto com o planejamento de cada iteração ou a cada duas iterações. Se a periodicidade não estiver sincronizada com as iterações, o modelo de desenvolvimento forçará mudanças de prioridade enquanto a equipe está no meio de uma etapa do desenvolvimento. Isso pode criar um cenário onde os programadores trabalham para produzir funcionalidades que deixaram de ser importantes, o que tornará a equipe pouco confiante e sem motivação para concluir a iteração.

O Scrum, por exemplo, prevê a criação de um backlog com todas as funcionalidades desejadas, porém, a cada iteração, uma nova priorização acontece para a escolha do que será implementado no próximo sprint (iteração). Em XP, durante o planejamento da release costuma-se fazer uma divisão inicial das funcionalidades em iterações. Isto facilita a obtenção das primeiras estimativas. Nas próximas priorizações, é possível fazer permutas entre as funcionalidades à medida que novas prioridades são identificadas. Essas mudanças acontecem no planejamento da iteração com o consentimento de clientes e desenvolvedores.

Para os cenários onde não é possível ter todos os recursos (DBAs, designers, etc.) 100% do tempo, independente da metodologia, o ideal é minimizar a distorção que as indisponibilidades causam na priorização. Para fazer isso, determine a ordem ideal de implementação ignorando as indisponibilidades e depois tente ajustar o uso dos recursos para que a priorização real seja a mais próxima possível da priorização ideal.

Técnicas de Priorização

As técnicas que apresentamos a seguir são fortemente baseadas em interesses de negócios, portanto levam em consideração o ponto de vista dos clientes e usuários. Os programadores, apesar de estarem completamente envolvidos com a criação do software, não devem ter a responsabilidade de determinar as prioridades de negócio. Estas decisões devem ser tomadas por aqueles que estão envolvidos com questões financeiras e estratégicas do projeto, como por exemplo, as táticas comerciais e as ações de marketing.

Opinião do Cliente

Os clientes devem estar altamente envolvidos com o desenvolvimento, pois os programadores, e mesmo o gerente, na maioria das vezes não são especialistas no domínio de negócios do software, portanto eles não têm condições de determinar sozinhos as prioridades do projeto.

Envolver os clientes e colocar as primeiras versões do software em contato com potenciais usuários é uma forma de coletar opiniões de pessoas que poderão avaliá-lo pensando de que maneiras aquele programa ainda em construção pode se tornar rapidamente mais útil e poderoso.

Para coletar as opiniões de potenciais usuários, uma reunião pouco formal, no estilo de demonstração é o ideal. Após a apresentação das funcionalidades, algumas perguntas ajudarão a entender os principais interesses dos usuários. Se o grupo de usuários for grande, cada um pode responder em uma folha de papel perguntas simples como: “Quais são as próximas três funcionalidades que você gostaria que esse sistema possuísse?” acompanhada de uma discussão com o grupo sobre as respostas. Se os usuários já usam um software e o novo vem para substituí-lo ou para disputar mercado, a pergunta pode ser: “Quais funcionalidades você usa com frequência que este programa ainda não tem?”, e em um momento mais avançado do desenvolvimento a pergunta seria: “Que funcionalidades você gostaria que o sistema tivesse para tornar o seu trabalho mais fácil?”. Se os potenciais usuários não possuem qualquer solução em software, a pergunta pode ser: “O que mais este software precisa fazer para ser útil para você?” ou “O que faria você usar este software?”.

Dependendo do processo de desenvolvimento usado, perguntas parecidas com estas podem ter sido feitas em uma fase inicial do projeto, porém as suas respostas provavelmente foram usadas para entender as necessidades, mas não para determinar prioridades. Além disso, as necessidades podem ter mudado, esta é uma forma de verificar se elas continuam as mesmas.

Estas reuniões de demonstração podem acontecer periodicamente durante o desenvolvimento do produto. Mensalmente ou sempre que as maiores prioridades apontadas na reunião anterior tiverem sido implementadas. Se as demonstrações acontecerem desde a primeira iteração, o software poderá ser utilizado e avaliado desde o início, reduzindo a possibilidade da equipe de desenvolvimento despender tempo em funcionalidades que não agregam valor ao software.

Em geral, após algumas semanas já é possível fazer a primeira demonstração para clientes ou potenciais usuários, mesmo que o software não tenha todas as funcionalidades ou mesmo uma interface gráfica profissional. Naturalmente, é importante alertá-los de antemão de que esta ainda é uma versão precoce que precisa da colaboração e opinião de todos para crescer e se tornar o produto que desejam.

Priorização por Valor e Risco

Outra maneira de priorizar considera o valor de negócios que a funcionalidade traz para o software, que vamos chamar simplesmente de valor, e a dificuldade de implementação, que inclui o risco de atraso e as incertezas a respeito da implementação, chamamos essas dificuldades de risco. Esta técnica é simples e produz um diagrama que pode ser lido facilmente e oferece uma visão panorâmica do projeto sob o ponto de vista das duas variáveis consideradas: valor e risco.

O resultado deste exercício de priorização será visto em um plano cujos eixos são o valor e o risco. Ele pode ser desenhado em uma lousa, em um quadro branco ou em uma cartolina sem a necessidade de definir uma escala, basta apenas determinar a direção em que o valor e o risco crescem em seus respectivos eixos.

Para mensurar o valor e o risco de cada funcionalidade, a participação de clientes e desenvolvedores é essencial. O cliente determina a quantidade de valor que cada funcionalidade agrega ao produto final e a equipe de desenvolvimento dimensiona o risco de implementação. Desta forma, cada funcionalidade pode ser representada por um ponto no plano, conforme a **Figura 1**.

Neste exercício a meta não é obter estimativas para criar cronogramas. O objetivo é dimensionar comparativamente a quantidade aproximada de valor e risco de cada funcionalidade, por isso não é preciso associar números para as medidas. O importante é perceber quais funcionalidades são, de fato, valiosas e de alto risco através de comparações diretas com as demais. Ao dispor os pontos (funcionalidades) no gráfico, as posições começam a ser definidas pelas opiniões dos participantes e depois são ajustadas por comparação com os outros pontos.

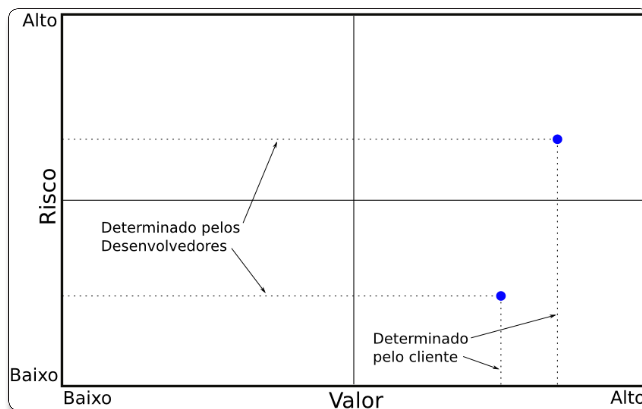


Figura 1. Plano com os eixos de Valor e Risco com funcionalidades representadas por pontos.

Depois de dispor as funcionalidades no plano, este é dividido em quadrantes. Desta forma, as funcionalidades ficarão agrupadas em quatro categorias, de acordo com suas quantidades de *valor* e *risco*. As categorias são: *alto risco e alto valor, baixo risco e alto valor, baixo risco e baixo valor, e alto risco e baixo valor.*

A seguir, uma estratégia de priorização dos quadrantes é escolhida conforme a metodologia, o projeto e a experiência da equipe. Para projetos que precisam comprovar sua viabilidade ou equipes experientes com métodos ágeis, a prioridade mais alta pode ir para as funcionalidades do quadrante com valor e risco mais altos. Fazê-las primeiro é interessante porque quando estas estiverem implementadas, haverá uma grande quantidade de valor agregado ao software e ao mesmo tempo, uma grande quantidade de risco terá sido eliminada do projeto. Em seguida vem o quadrante de *alto valor* e *baixo risco* e depois o de *baixo valor* e *baixo risco*, conforme a **Figura 2**. Com esta estratégia, caso não seja possível produzir as primeiras funcionalidades, que são de alta dificuldade, o projeto pode ser reavaliado sem ter tido custos elevados, pois os impedimentos foram identificados cedo. Também pode ser percebido que a implementação será muito mais demorada do que o previsto. Isso permite que as estimativas sejam ajustadas ainda no início do projeto.

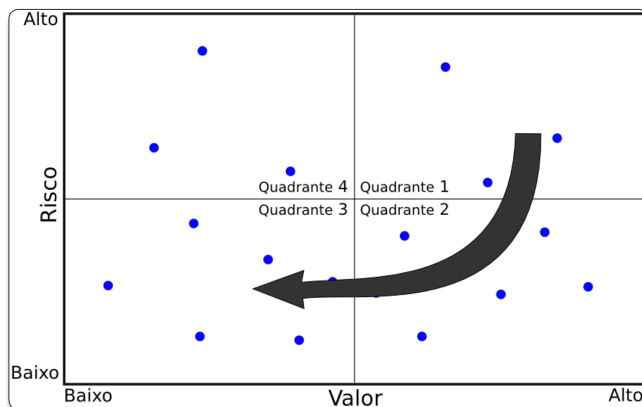


Figura 2. Sequência de implementação para equipes experientes ou projetos com muita incerteza.

Equipes com pouca experiência em desenvolvimento ágil ou projetos sem grandes desafios técnicos podem usar outra sequência de priorização. Começando pelo quadrante de *alto valor* e *baixo risco*, para produzir funcionalidades importantes e rapidamente chegar a uma versão do software em produção e, em seguida, implementar as funcionalidades dos quadrantes *de alto valor e alto risco* e *baixo valor e baixo risco* e, se necessário, as do quadrante de *baixo valor e alto risco*. A **Figura 3** exibe essa seqüência de implementação.

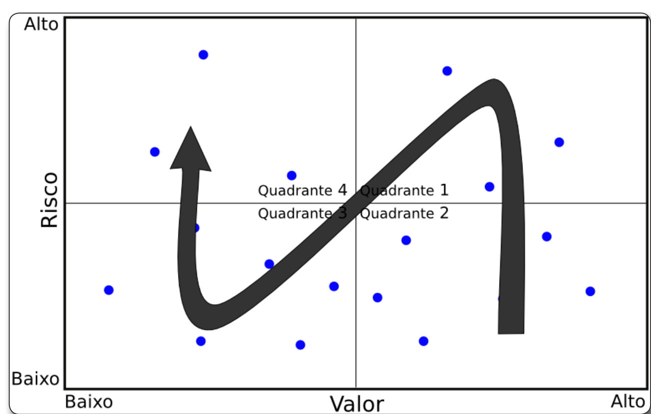


Figura 3. Sequência de implementação para equipes pouco experientes ou projetos sem desafios técnicos

Muitas vezes, independente de implementação escolhida, as funcionalidades do último quadrante não chegam a ser implementadas porque o software já atende às necessidades dos usuários e o investimento para produzir funcionalidades de alto risco não é recompensado.

Conclusões

Uma priorização adequada das funcionalidades faz parte da etapa de planejamento do desenvolvimento. Ela ajuda o projeto a aproveitar melhor seus recursos e, quando revista periodicamente, mantém o foco nas funcionalidades de maior interesse, o que aumenta significativamente as chances de sucesso do projeto.

Os critérios de priorização podem ser variados, contudo, para projetos que buscam retorno financeiro, uma boa tática é usar este fator como um dos critérios de priorização. Dessa forma, o desenvolvimento é orientado pela mesma variável que também guia a cobrança dos resultados da equipe.

A obtenção das prioridades pode ser feita a partir de técnicas de fácil execução. As que vimos neste artigo visam à criação rápida de uma versão funcional do software. Por isso consideram fortemente as opiniões de usuários, desenvolvedores e especialistas no domínio de negócios da aplicação. Contudo, as mesmas técnicas podem ser usadas considerando outros critérios de priorização.

Referências

Jim Johnson. ROI, it's your job. In Keynote Speech at 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP'2002), May 2002.

PMI Chapters Brasileiros. Estudo de benchmarking em gerenciamento de projetos brasil. Technical report, www.pmi.org.br, 2007.

Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.

Ken Schwaber. Agile Software Development with Scrum. Microsoft Press, 2004.

Mike Cohn. Agile Estimating and Planning. Prentice Hall, 2006.

Dairton Bassi. Experiências com desenvolvimento Ágil. Master's thesis, Instituto de Matemática e Estatística da Universidade de São Paulo - IME/USP, 2008.

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

Modéstia à parte, sua melhor opção para se destacar no mercado!

A Escola Superior da Tecnologia da Informação oferece as melhores opções em cursos, formações, graduações e pós-graduações para profissionais de desenvolvimento e programação.

São programas voltados para a formação de profissionais de elite, com **aulas 100% práticas, corpo docente atuante no mercado**, acesso à mais atualizada biblioteca de TI do Rio, laboratórios equipados com tecnologia de ponta, salas de estudo e exames.

PÓS-GRADUAÇÃO

- **Engenharia de Software: Desenvolvimento Java**

GRADUAÇÃO

- **Análise e Desenvolvimento de Sistemas**

FORMAÇÕES

- **Desenvolvedor Java**
- **Desenvolvedor Java: Sistemas Distribuídos**
- **Gestor de TI**
- **Desenvolvedor Web .NET 2008**
- **MCITP Server Administrator**
- **SQL Server 2008**

Acesse nosso site e conheça todos os nossos programas: **www.infnet.edu.br/esti**

TURMAS NO RIO DE JANEIRO

ESCOLA SUPERIOR DA TECNOLOGIA DA INFORMAÇÃO

www.infnet.edu.br - cursos@infnet.edu.br - Central de Atendimento: (21) 2122-8800

EDUCAÇÃO SUPERIOR ORIENTADA AO MERCADO

Incorporando Restrições à UML

Os conceitos principais por trás da linguagem OCL

De que se trata o artigo?

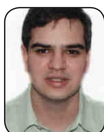
Neste artigo será apresentada uma introdução à OCL, a linguagem de restrição oficial da UML.

Para que serve?

Impor restrições a alguns objetos é de fundamental importância para esclarecer, verificar e validar modelos UML. Este artigo trata sobre como é possível fazer isso através da OCL.

Em que situação o tema é útil?

Em que situação o tema é útil: Para aqueles que não conseguem apenas com os diagramas UML definir todos os aspectos relevantes da especificação do sistema.



Thiago Carvalho de Sousa

thiagocsousa@gmail.com

Doutorando em Engenharia de Computação e Sistemas Digitais (EP-USP). Mestre e Bacharel em Ciência da Computação (IME-USP). Trabalha há mais de 8 anos com desenvolvimento de software e já atuou como gerente de projetos, analista de negócios e engenheiro de requisitos na Oracle, Johnson & Johnson, Goodyear, NET/Embratel e Grupo Claudino. Atualmente é consultor da Alstom Transport e professor licenciado das disciplinas de Engenharia de Software e Métodos Formais do CEUT e da FAETE.

A UML (Unified Modeling Language) é uma linguagem para especificação, documentação, visualização e desenvolvimento de sistemas orientados a objetos. Sintetiza os principais métodos existentes, sendo considerada uma das linguagens mais expressivas para modelagem de sistemas e por isso de grande aceitação na indústria. Por meio de seus diagramas é possível representar sistemas de softwares sob diversas perspectivas de visualização. Facilita a comunicação de todas as pessoas envolvidas no processo de desenvolvimento de um sistema - gerentes, coordenadores, analistas, desenvolvedores - por apresentar um vocabulário de fácil entendimento.

A primeira versão da UML foi apresentada em junho de 1996 e submetida à OMG (Object Management Group), consórcio de empresas que define padrões de modelos e linguagens. A revisão desse modelo mostrou uma grande deficiência na clareza e consistência das definições da UML. Em particular, uma dificuldade encontrada foi que a semântica da UML poderia ser interpretada de formas ambíguas. O problema foi minimizado com a elaboração de uma nova versão da linguagem, a qual foi publicada em 1997. O mais importante incremento nesta versão foi a criação da OCL (Object Constraint Language), que acompanha oficialmente a evolução da UML até os dias atuais.

A linguagem OCL

A OCL (ou linguagem para especificação formal de restrições, em português) é uma linguagem declarativa para descrever as regras que se aplicam aos modelos UML. É uma linguagem de texto precisa que fornece afirmações em um modelo orientado a objeto que não possam ser expressadas pela notação diagramática. A OCL complementa os modelos UML fornecendo expressões que não têm nem as ambiguidades da língua natural (eg. "o sistema deve identificar uma pessoa com um telescópio"), nem a dificuldade inerente de se usar matemática complexa.

Com a OCL é possível testar a construção dos modelos, retirar métricas ao nível de projeto e ainda especificar vários tipos de restrições, que poderão refletir regras de negócio, através de pré/pós condições e invariantes.

Como a OCL é uma linguagem formal, semelhante a uma linguagem de programação, torna-se, também, possível a criação de geradores de código tendo como entrada os modelos e as especificações/restrições nessa linguagem e como saída programas fontes em linguagens de programação ou "triggers" para bancos de dados. Inclusive, atualmente a OCL vem se tornando um componente fundamental no novo padrão MDA-QVT (Query-View-Tranformation) para transformação de modelos da OMG.

A OCL pode ser utilizada para:

- Especificar invariantes em classes e tipos do modelos de classes.
- Especificar tipos invariantes para estereótipos.
- Descrever pré e pós-condições em operações.
- Como uma linguagem de navegação entre associações.
- Como uma linguagem de consulta.
- Para especificar o alvo das mensagens e ações.
- Para especificar regras de derivações para atributos.
- Especificar restrições sobre operações.

A estrutura da OCL está intimamente ligada a outros modelos da UML, sendo bastante usada com o Diagrama de Classes através de uso de notas nos diagramas. É composta por expressões escritas em forma de afirmações. É uma linguagem que possui tipos de dados pré-definidos, assim como algumas palavras reservadas. Nas próximas seções falaremos sobre a sintaxe de cada uma dessas características principais.

Expressões OCL

Toda expressão OCL é declarativa no sentido de que expressa o quê a restrição representa no sistema e não como essa restrição é implementada. A avaliação de uma expressão quase sempre resulta em um valor booleano e nunca muda o estado do sistema no modelo.

As expressões OCL são utilizadas para definir condições invariantes nas classes representadas em um modelo e também são utilizadas para especificar as pré e pós-condições em operações aplicadas a classes deste modelo.

Expressões OCL também podem ser utilizadas para fazer consultas a um modelo de classes da UML. Essas consultas podem ser úteis para validar modelos de classes na fase de projeto. Nesse caso, a avaliação dessa expressão não devolve um valor booleano, e sim valores de um tipo específico da OCL.

Tipos de Expressões

As expressões OCL podem ser de três tipos:

- Expressões que representam condições invariantes em classes de objetos;
- Expressões que representam pré-condições de operações aplicáveis a uma classe de objetos;
- Expressões que indicam as pós-condições de operações aplicáveis a uma classe de objetos;

Contexto de uma Expressão

As expressões OCL requerem que as restrições estejam ligadas a um contexto de um modelo. O contexto de uma expressão pode ser uma classe de objetos ou pode ser uma operação aplicável a um objeto.

Para representar um contexto em OCL utilizamos a seguinte palavra reservada:

```
context  <contexto>
```

Invariantes

Invariantes são condições que os objetos modelados devem respeitar durante toda sua existência no sistema. Em OCL, para indicar que uma expressão é uma invariante, utilizamos a palavra reservada inv: após a declaração do contexto. Uma expressão típica em OCL e que representa uma condição invariante tem o seguinte formato:

```
context <contexto>
     inv: <expressão>
```

Pré-condições

Pré-condições são declarações que refletem o estado no qual deve se encontrar o sistema para que as operações sobre os objetos possam ser executadas. Em OCL, quando queremos expressar uma condição na forma de pré-condição, utilizamos a palavra reservada pre: após a declaração do contexto. O contexto deve possuir explicitamente a indicação sobre qual operação a pré-condição ocorre. A expressão típica em OCL usada para representar uma pré-condição tem a seguinte estrutura:

```
context <contexto>
      pre: <expressão>
```

Pós-condições

Pós-condições são declarações que apresentam o estado do sistema após a execução de uma determinada operação de um objeto. Em OCL, para declararmos uma expressão na forma de pós-condição utilizamos a palavra reservada post: após a declaração do contexto. O contexto deve possuir explicitamente a indicação sobre qual operação a pós-condição ocorre. Para as pós-condições temos a seguinte expressão típica:

```
context <contexto>
      post: <expressao>
```

Palavras Reservadas

Algumas palavras que desempenham funções especiais no contexto da expressão foram criadas para estruturar melhor certas restrições em OCL.

Self

Numa expressão em OCL, a palavra reservada self é usada para referenciar explicitamente uma instância do contexto.

@Pre

Numa expressão em OCL, a palavra reservada @pre acompanha um atributo ou associação, indicando o seu valor antes do inicio da operação.

Result

Numa expressão em OCL, a palavra reservada result indica o resultado gerado por uma operação.

Let ... in

Algumas vezes uma pequena parte da expressão é utilizada mais de uma vez. As palavras reservadas let...in permitem a definição de uma variável (que contém a pequena parte da expressão) a qual pode ser usada na restrição a ser expressada.

If… Then…. Else….. End If

Para uma restrição que envolve condições, foram criadas as palavras reservadas **if...then...else...end if**.

Tipos de Dados

Uma das características da OCL é ser uma linguagem tipada. Toda informação manipulada, nas expressões construídas em OCL, pertence a um dos seguintes grupos de variáveis:

• Tipos Básicos: Real, Integer, String e Boolean;
• Coleções: Set (elementos únicos e não ordenados), Bag (elementos duplicados e não ordenados), Sequence (uma bag ordenada), e OrderedSet (um set ordenado);
• Tipos dos modelos UML: todas as classes, atributos, interfaces, associações, consultas e enumerações introduzidas por um diagrama de classes.

Operações sobre Tipos Básicos

• **Real:** +, -, *, /, >=, <=, >, <, abs(), floor(), round (), max(r:Real), min(r: Real)
• **Integer:** +, -, *, /, >=, <=, >, <, abs(), div(i: Integer), mod (i:Integer), max(r:Integer), min(r: Integer)
• **String:** size(), concat(s: String), subString(lower: Integer, upper: Integer), toInteger(), toReal()
• **Boolean:** or(b:Boolean), xor(b:Boolean), and(b:Boolean), implies(b: Boolean)

Operações sobre Coleções

Para realizar uma operação sobre uma coleção utilizamos a seguinte sintaxe básica:

```
[tipo de dado] -> [operação]
```

As principais operações são:

• **size(): Integer** – encontra o tamanho da coleção;
• **isEmpty(), notEmpty():** Boolean - verifica se a coleção é vazia ou não;
• **first(): Object** – encontra o primeiro elemento da coleção;
• **last(): Object** – encontra o ultimo elemento da coleção;
• **sum (): Integer/Real** - soma os elementos da coleção;
• **count(object): Integer** – numero de ocorrências de um objeto;
• **includes (object): Boolean** – verifica se um objeto está numa coleção;
• **excludes (object): Boolean** – verifica se um objeto não é de uma coleção;
• **includesAll (collection): Boolean** – verifica a inclusão de uma coleção em uma outra;
• **excludesAll (collection): Boolean** – verifica a exclusão de uma coleção em uma outra;
• **including (object): Collection** – adiciona um elemento a coleção;
• **excluding (object): Collection** – exclui um elemento da coleção;
• **union (collection): Collection** – une duas coleções;
• **asSet(): Collection** - transforma uma bag em um set, eliminando os duplicados;
• **select (condition):Collection** – restringe a coleção os objetos sob uma certa condição;
• **reject (condition): Collection** - exclui da coleção os objetos sob uma certa condição;
• **forall (condition): Boolean** – verifica se todos os objetos satisfazem uma certa condição;
• **exists (condition): Boolean** – verifica se pelo menos um objeto satisfaz uma certa condição
• **collect (condition): Collection** - cria uma coleção derivada de outra, mas que também contém outros elementos.

Exemplo de Uso 1

A OCL se encaixa perfeitamente na metodologia de desenho por contrato (design by contract). Essa metodologia foi desenvolvida em meados da década de 80 por Bertrand Meyer e tem crescido muito nos últimos anos, tendo entre seus principais adeptos Craig Larman. O desenho por contrato (DbC) é um método de implementação que visa a construção de sistemas orientados a objetos mais confiáveis, na medida em que provê mecanismos para detecção de violações da sua especificação, em especial violações de invariantes. A principal idéia do DbC é que entre as classes e seus clientes seja estabelecido explicitamente um contrato. Nele, o cliente deve garantir certas condições antes de invocar os métodos da classe (pré-condições), que por sua vez deve garantir algumas propriedades após ter sido executado (pós-condições). Veremos a seguir como é a utilização da OCL como ferramenta para a especificação de restrições do sistema, usando dois exemplos para elucidar a teoria apresentada.

O primeiro exemplo é um clássico proposto por Jos Warmer baseado na empresa fictícia Royal e Loyal (R&L). A R&L gerencia programas de fidelidade para empresas parceiras

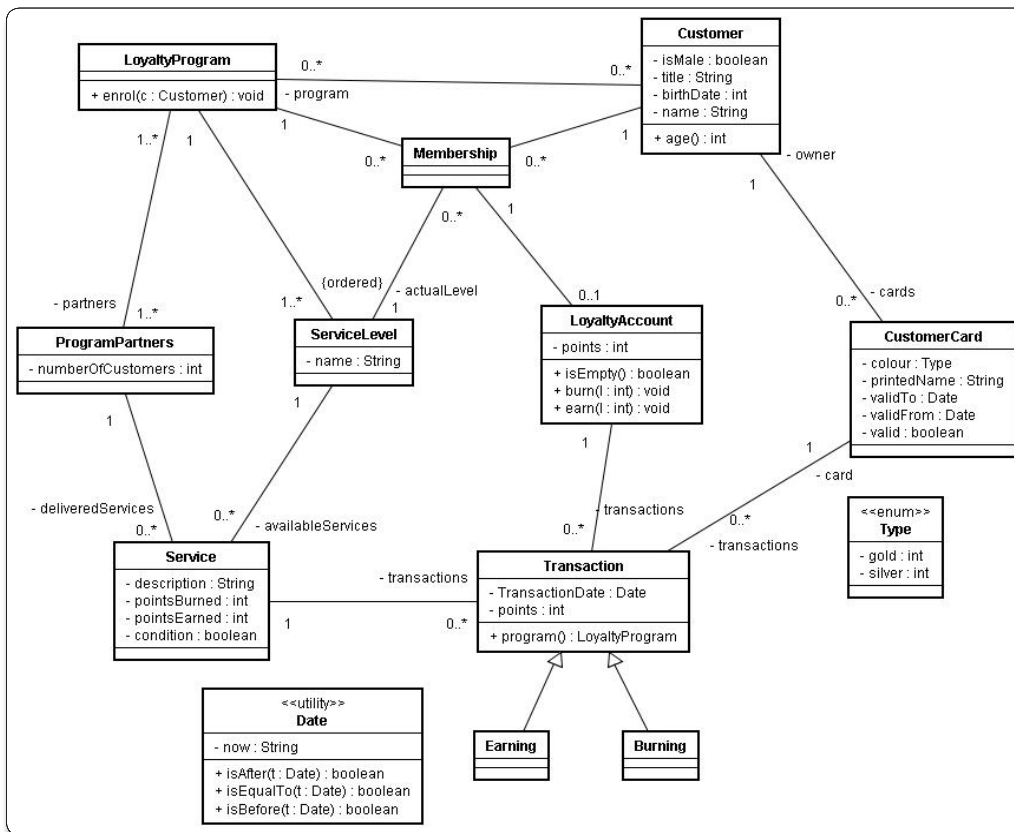


Figura 1. Diagrama de Classes.

que oferecem vários tipos de bônus (eg. milhas aéreas). A classe principal é a LoyaltyProgram. Um sistema que tem apenas um programa de fidelidade possuirá um único objeto dessa classe. A empresa que oferece a seus clientes um programa de fidelidade é chamada ProgramPartner. Mais de uma empresa pode participar de um mesmo programa. Nesse caso, os clientes que entram no programa fidelidade podem lucrar com serviços prestados por qualquer das empresas participantes.

Todos os clientes entram no programa através do preenchimento de um formulário e obtém um cartão de fidelidade. Os objetos da classe Customer representam as pessoas que entraram no programa. O cartão de fidelidade é representado pela classe CustomerCard e pode ser do tipo ouro ou prata, dependendo dos gastos do cliente. A maioria dos programas permite que os clientes acumulem pontos de bônus. Cada parceiro decide quando e quantos pontos de bônus são atribuídos a uma determinada aquisição (Service). Os pontos podem ser usados para "comprar" serviços específicos de um dos parceiros do programa. Para contabilizar os pontos de bônus que são salvos pelo cliente, cada membro é associado a uma LoyaltyAccount. Há dois tipos de transações: uma para obter pontos (Earning) e outra para resgatá-los (Burning). Para administrar os diferentes níveis de serviços, a classe ServiceLevel foi introduzida ao modelo. Um nível de serviço é definido pelo programa de fidelidade e usado por cada membro (Membership). Vejamos a seguir o diagrama de classes (**Figura 1**) desse sistema e mais detalhes sobre a especificação de requisitos (**Quadro 1**).

....

R11: Todo cliente deve ter no mínimo 21 anos

R12: A data "valid from" do cartão de fidelidade deve ser anterior a data "valid to"

R13: Um cartão fidelidade só pode ser emitido para um membro

R14: A quantidade de níveis de serviços é exatamente dois

R15: O nome impresso no cartão de fidelidade deve ser precedido pelo título ao qual o cliente deseja ser chamado

R16: O numero de cartões válidos para cada cliente deve ser igual ao numero de programas que ele participa

R17: Quando um programa não possui acréscimo e nem resgate de pontos, não existe necessidade dos clientes terem uma conta

R18: O numero de clientes de um parceiro é a soma dos participantes de um ou mais programas daquele parceiro

R19: O primeiro nível de serviço é o "prata"

R20: O máximo de pontos que pode ser resgatado em cada parceiro é de 10000

R21: O título do cliente é "Mr." se for homem, caso contrario é "Ms."

R22: O nível de serviço atual deve ser um serviço oferecido pelo programa de fidelidade

R23: O conjunto de transações em uma conta deve ter pelo menos uma transação com mais de 500 pontos

R24: A existência de pontos em conta implica que pelo menos uma transação foi realizada.

R25. O numero de pontos resgatados em uma aquisição nunca deve exceder o numero de pontos ganhos por esse serviço.

....

Quadro 1. Especificação de Requisitos

A partir desses requisitos e visualizando o diagrama de classes, podemos criar as seguintes expressões OCL:

```
context Customer                              (R11)
      inv: age( ) >= 21


      - - age( ) é um método da classe  Customer e pode ser
      usada em uma
      - - expressão OCL como consulta.

context CustomerCard                          (R12)
      inv: validFrom.isBefore(ValidTo)

      - -  isBefore ( ) é um método da classe Date
      e validFrom e validTo  são
      - - atributos de  CustomerCard do tipo Date.

context Membership                            (R13)
      inv: card.customer = customer

context LoyaltyProgram                        (R14)
      inv: serviceLevel->size() = 2

      - - a pré-definida operação size da OCL
      devolve o tamanho da coleção de
      - - níveis de serviços.

context CustomerCard                          (R15)
      inv: printedName = customer.title.
      concat(customer.name)

      - - title é do tipo String e pode ser
      concatenado (concat) com o nome do
      - - cliente, que também é do tipo String,
      para formar o nome a ser
      - - impresso no cartão.

context Customer                              (R16)
      inv: program->size() = cards->select
      (valid = true) ->size()

context LoyaltyProgram                        (R17)
      inv: partners.deliveredServices-
      >forAll(pointsEarned = 0 and pointsBurned = 0)
      implies membership.loyaltyAccount->isEmpty()

      - - se um parceiro oferece um programa onde
      todos (forAll) os serviços
      - - não oferecem acréscimo ( pointsEarned = 0)
      e nem resgate de pontos
      - - (pointsBurned = 0), então a coleção de contas
      (loyaltyAccount) de um   - - cliente deve ser
      vazia (isEmpty).

context ProgramPartner                        (R18)
      inv: numberofCustomers = loyaltyProgram.
      customer->asSet->size()

      - -  a pré-definida operação asSet da OCL
      transforma  uma bag em um
      - -  set, eliminando os clientes repetidos.

context LoyaltyProgram                        (R19)
      inv serviceLevel.name ->first() = 'Silver'
      - - o modelo define que a associação entre
      LoyaltyProgram e ServiceLeve
      - - é ordenada, sendo silver o primeiro elemento.

context LoyaltyProgram                        (R20)
      inv: partners.deliveredServices.transaction->
      select(Burning)-> collect (points)->sum( )<= 10000
      - - as transações de resgate (Burning) devem
      somar (sum) no máximo
      - - 10000 pontos

context Customer                              (R21)
      inv: title = (if isMale = true then 'Mr.'
      else 'Ms.' endif)

context Membership                            (R22)
      inv: program.serviceLevel-> includes (actualLevel)

context LoyaltyAccount                        (R23)
      inv: transaction->collect(points) -> exists
      (p:Integer | p > 500)

context LoyaltyAccount                        (R24)
      inv: points> 0 implies transaction->exists (points> 0)

context ProgramPartner                        (R25)
      inv: self.services.transaction->select(Burning)
      -> collect(points) ->sum() <= self.services.
      transaction->select(Earning)->collect(points) ->sum()
```

Note que o requisito "R18: O numero de clientes de um parceiro é a soma dos participantes de um ou mais programas daquele parceiro" permite uma interpretação ambígua, pois pode ser tanto a quantidade total de participações nos programas ou a quantidade de pessoas distintas. Com a OCL é possível esclarecer requisitos ambíguos (no caso, o analista preferiu optar pela segunda opção) e colocá-los em uma notação matemática precisa, sem margem para varias interpretações, mas de fácil entendimento. Nesse exemplo usamos a OCL apenas para relatar os invariantes do sistema, mas poderíamos usá-la também para delimitar as pré-condições e as pós-condições. Faremos isso a seguir.

Exemplo de Uso 2

O segundo exemplo é baseado em um fictício sistema bancário. Os clientes (*Customer*) possuem uma conta (*Account*), que pode ser do tipo corrente (*current*) ou poupança (*deposit*). Toda conta do tipo corrente possui um limite de cheque especial (*odLimit*). Os clientes podem sacar, depositar (em cheque (*check*) ou dinheiro (*cash*)) e solicitar empréstimos (*Credit*) usando bens (*Security*) como garantia. Vejamos a seguir o diagrama de classes (**Figura 2**) desse sistema e mais detalhes sobre a especificação de requisitos (**Quadro 2**).

A partir desses requisitos e visualizando o diagrama de classes, podemos criar as seguintes expressões OCL:

```
context Customer
      inv: age >= 18         (R1)

context Account
      inv: self.balance >= -self.odLimit    (R2)

      - - o saldo sempre deve ser maior que o
      cheque especial negativamente.

context Account
      inv: self.holder.getAge() < 25 implies  self.
      odLimit = 0   (R3)

context Account
      inv: self.accountType = #deposit implies
      self.odLimit = 0    (R4)
```

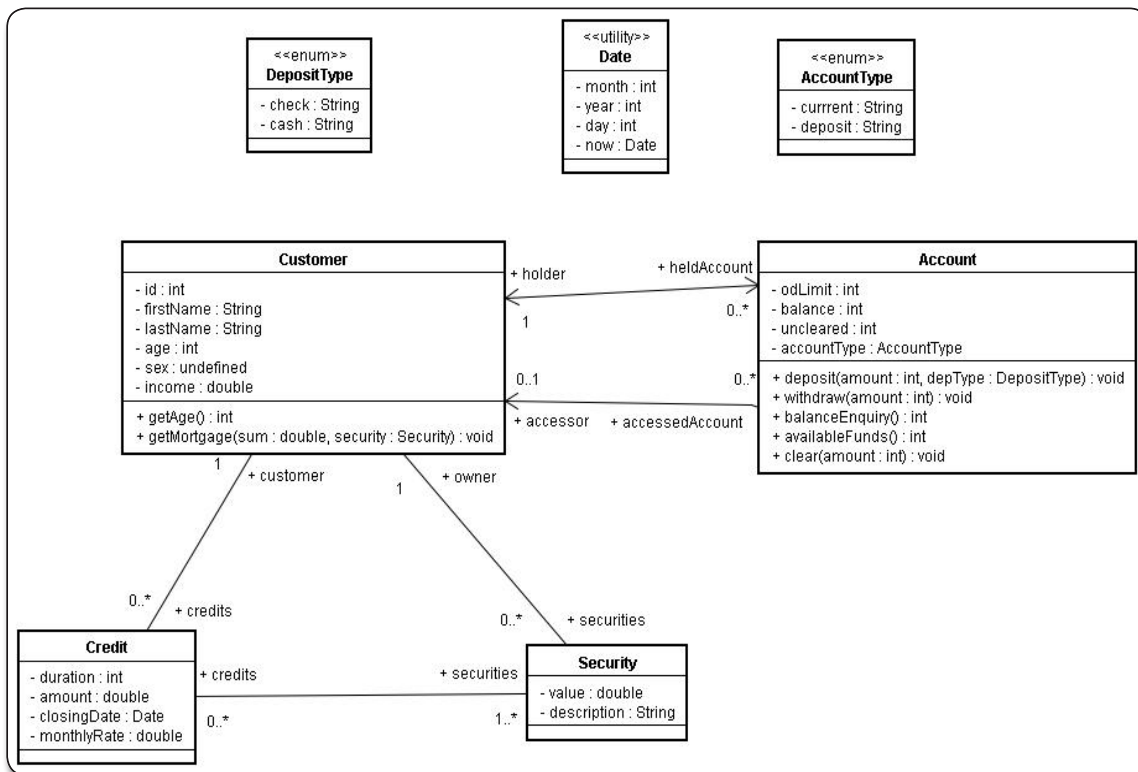


Figura 2. Diagrama de Classes do cenário 2

```
      - - se a conta é do tipo poupança (deposit),
      então não existe cheque - - especial.

context Account::deposit (amount: Integer,
depositType: DepositType)
      pre: amount > 0 and accessor = holder   (R26)
      post:
      (depositType = #cash implies balance =
       balance@pre + amount) or  (R27)
      (depositType = #check implies uncleared =
       uncleared@pre + amount)

      - - se o depósito for em dinheiro (cash),
      então o saldo é
      - -  automaticamente incrementado. Se o
      depósito for em cheque (check), - - então o
      saldo dos depósitos bloqueados é incrementado.

context Account::clear (amount: Integer)
      pre: self.uncleared >= amount  (Requisito Novo)
      post:
      (uncleared = unclered@pre - amount) and (R28)
      (balance = balance@pre + amount)

      - - quando um  depósito é liberado, o saldo
      dos  depósitos bloqueados é
      - - decrementado e o valor é acrescentado ao
      saldo real (balance).

context Account::withdraw (amount: Integer)
      pre: amount > 0 and accessor = holder  and
      balance-amout>= -odLimit                (R32)
      post:  balance = balance@pre - amount   (R33)

context Account::balanceEnquiry(): Integer
      pre: accessor = holder                  (R57)
      post: result = balance
```

….
R1: Todo cliente deve ser maior de idade
R2: O limite de cheque especial nunca pode ser excedido
R3: Clientes menores de 25 anos não possuem cheque especial
R4: Contas do tipo Poupança não possuem cheque especial
….
R26: Depósitos nulos e não realizados pelo titular são inválidos
R27: Se o depósito for em dinheiro, o saldo deve ser incrementado. Se o depósito for em cheque, o valor ficará bloqueado em uma conta de saldos bloqueados até o cheque ser compensado.
R28: Quando um cheque é compensado, o seu valor é incorporado ao saldo e a conta de saldos bloqueados é atualizada.
….
R32: Não são permitidos saques nulos, que excedam o limite de cheque especial e não realizados pelo titular da conta.
R33: O saldo deve ser atualizado após um saque.
….
R57: O saldo e o saldo total somente podem ser vistos pelo titular da conta.
R58: O saldo total é calculado pela soma do saldo com o limite do cheque especial
….
R62: Um cliente só pode pedir um empréstimo se possuir conta do tipo corrente e com cheque especial de pelo menos 10000
R63: Um cliente só pode pedir um empréstimo se a soma de todos os seus empréstimos não exceder em 50% do seu salário
R64: Um cliente só pode pedir um empréstimo se a soma dos valores de todos os bens for pelo menos 20% maior que o valor dos empréstimos
R65: Depois de pagar a mensalidade de todos os empréstimos, o saldo da conta deve ser positivo
R66: O empréstimo mínimo é de 10000 e o máximo de 1000000
R67: Os bens dados como garantia devem ser obrigatoriamente do cliente
….
R74: Será possível consultar quais os clientes com saldo total maior que 100000.
….

Quadro 2. Especificação de Requisitos

```
context Account::availableFunds(): Integer
      pre: accessor = holder                 (R57)
      post: result = balance + odLimit       (R58)

      - - o saldo total é a soma do saldo real com
      o limite do cheque especial.

context Customer::getMortgage(sum: double, security: Security)
     pre:
     self.accountType = #current and odLimit >= 10000 and
     (sum + self.credits.monthlyRatio -> sum() <=
      self.income * 0.5) and
     (security.value + self.securities.value->sum()
      >= 1.2 * self.credits.amount -> sum() + sum)
      (R62) (R63) (R64)

      - - o cliente possuir conta do tipo corrente
      (current),  seu  cheque
      - - especial ser maior ou igual a 1000
      (odLimit >= 10000), a soma dos
      - - pagamentos mensais (monthlyRatio) ser
      menor que a metade de seu
      - - salario (income) e o valor do bens
      penhorados (incluindo o atual,
      - - security.value) ser maior que 20%  do
      valor do empréstimos
      - - (incluindo o atual, sum) é pré-condição
      para realizar um novo
      - - empréstimo.

context Customer                            (R65)
      inv: heldAccount.balance -  credits.
      monthlyRate->sum() >= 0

context Credit                              (R66)
      inv: amount >= 10000 and amount <=1000000

context Security
      inv: securities.owner->forAll(owner | owner =
      customer)                             (R67)

context Customer
      inv: self.heldAccount->select(balanceEnquiry()
      > 100000)                             (R74)
```

Mais uma vez podemos ver que a OCL transforma os requisitos (descritos em linguagem natural ambígua, como o requisito R32) em notação matemática precisa e de fácil entendimento. Além disso, durante essa transformação, o analista pode lembrar de pontos importantes "esquecidos" no documento de especificação, como verificar se a conta de saldos bloqueados possui um valor maior ou igual ao que vai ser desbloqueado. A vantagem da OCL se completa quando se usa ferramentas automatizadas (eg. Together) para gerar códigos fontes (eg. uma consulta SQL a partir do requisito R74) e testar se todas as possíveis entradas/saídas das operações atendem as pré e pós-condições.

Conclusão

A evolução da engenharia de software pode ser comparada à da engenharia civil. Com o passar do tempo, esta precisou se basear em cálculos estruturais formais a fim de garantir a qualidade dos imóveis construídos. O que se percebe é que essa evolução natural para um formalismo mais apurado está também ocorrendo na engenharia de software para alguns problemas específicos, uma vez que as exigências de confiabilidade e segurança de um software têm aumentado nos últimos anos.

Nesse contexto, apresentamos no artigo as noções básicas da OCL, a linguagem oficial de restrições da UML, que vem se destacando a cada dia, sendo a grande aposta da OMG como peça fundamental da arquitetura MDA-QVT.

Links

OMG OCL Specification Site
http://www.omg.org/docs/formal/06-05-01.pdf
Artigo "Introduction to OCL", de Jos Warmer e Anneke Kleppe
http://www.klasse.nl/ocl/ocl-introduction.html
"Applying Design by Contract", de Bertrand Meyer
http://www.cs.ucl.ac.uk/students/A.Masalskis/files/contract.pdf
Artigo "Introduction to OCL in Together", de Dan Massey
http://conferences.codegear.com/article/32200

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

TechWeek

Dias 2 e 3 de outubro
São Paulo

O maior evento de
web e wireless
do País!

Inscrições limitadas

http://www.devmedia.com.br/wm2009

Maiores informações através do email
evento@devmedia.com.br
ou pelo telefone (21) 3382.5038

patrocínio

TEKANN
MOBILE SOLUTIONS

apoio

realização

DevMedia
group

UML – Diagrama de Sequências

Descobrindo como modelar um diagrama de sequências

Ana Cristina Melo

informatica@anacristinamelo.com.br

É especialista em Análise de Sistemas e professora de graduação e pós-graduação da Universidade Estácio de Sá. Atua em análise e programação há 21 anos, sendo os últimos 11 anos no serviço público. 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.

A UML, na sua versão atual, nos oferece treze diagramas que tratam de aspectos diferentes de todas as fases de desenvolvimento de um sistema. Todos têm suas utilidades, mas, na prática, sabemos que é possível desenvolver a maioria dos sistemas orientados a objetos utilizando apenas três deles. Desses, já apresentei aqui os Casos de Uso e o Diagrama de Classes. E hoje, trabalhando em como modelar um Diagrama de Sequências, podemos dizer que vocês terão o conhecimento básico para desenvolver cerca de 70% dos sistemas orientados a objeto.

Veremos o que significa esse diagrama e como o mesmo se integra com os outros modelos.

O Diagrama de Sequências é um Diagrama de Interação

O Diagrama de Sequências é o principal dos quatro diagramas de interação. Os outros são: Diagrama de Comunicação, Diagrama de Visão Geral e Diagrama Temporal.

Um diagrama de interação tem por responsabilidade mostrar a interação entre os objetos de um sistema por meio de uma visão dinâmica. Essa interação entre objetos é representada por meio de mensagens. Ao se identificar as mensagens, estamos identificando os serviços oferecidos pelas classes. E, por sua vez, identificar os serviços significa que estamos descobrindo quais os métodos necessários a cada classe. Por isso, normalmente só chegamos a uma versão final do modelo de classes depois que passamos pelo diagrama de sequências,

De que se trata o artigo?

Este artigo tem por objetivo apresentar as regras para se modelar um diagrama de sequência, partindo da integração com os casos de uso e modelo de classes.

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 sequência.

Em que situação o tema é útil?

Para quem ainda não modelou diagramas de interação, ou para quem tem experiência e quer revisar a sintaxe permitida nesse tipo de diagrama.

pois só com ele conseguimos enxergar claramente todos os métodos que serão necessários para atender aos casos de uso.

Apesar de todos os diagramas terem um objetivo final comum, cada um atende a uma característica particular.

O **Diagrama de Sequências** enfatiza a troca de mensagens dentro de uma linha de tempo sequencial.

O **Diagrama de Comunicação** enfatiza o relacionamento estrutural entre os objetos, sem se preocupar com o tempo determinado para cada interação.

O **Diagrama de Visão Geral** é uma variação do diagrama de atividades que mostra de uma forma geral o fluxo de controle dentro de um sistema ou processo de negócios. Cada nó ou atividade dentro do diagrama pode representar outro diagrama de interação.

O **Diagrama Temporal** mostra a mudança de estado de um objeto numa passagem de tempo, em resposta a eventos externos. Este último é mais utilizado quando o objetivo principal do diagrama é a determinação do tempo exato, o que é indicado para sistemas de tempo real.

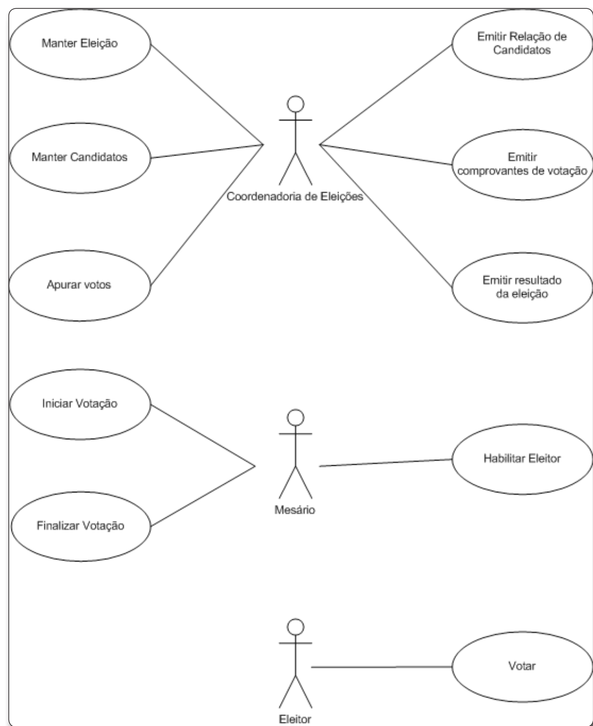


Figura 1. Diagrama de casos de uso para o sistema gestor de votação interna de uma empresa

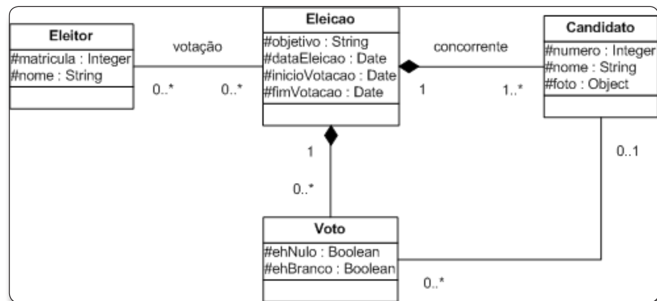


Figura 2. Diagrama de classes (versão inicial) para o sistema gestor de votação interna de uma empresa

Integrando os modelos

O Diagrama de Sequências não surge do nada e sim de uma modelagem feita a partir dos casos de uso, com o auxílio das classes já identificadas num modelo de classes. No caso de uso, estabelecemos a ordem das funcionalidades, sem nos preocuparmos com a implementação. Ao modelarmos o diagrama de sequências, representaremos por mensagens cada item descrito nos cenários principal e alternativos. Essas mensagens podem ser expressas do ator para o sistema, ou da interface para os objetos.

Para exemplificar essa modelagem, tomaremos por base um pequeno estudo de caso que se refere a um sistema de votação interna de uma empresa. Para isso, apresentaremos nas **Figuras 1** e **2** os diagramas de casos de uso e a versão inicial do diagrama de classes, respectivamente. Nas **Listagens 1, 2, 3, 4** e **5** apresentaremos os cenários dos principais casos de uso, para que vocês possam compreender como chegamos ao modelo de classes e como o pequeno sistema funciona.

Listagem 1. UC Manter Eleição

```
Descrição: Este caso de uso tem por objetivo manter o cadastro
de eleições, permitindo a inclusão, alteração, exclusão ou
consulta de eleições.
Atores: Coordenadoria de Eleições
Cenário principal:
1. O sistema prepara uma lista de eleições cadastradas, exibindo
   para cada uma: objetivo, data da eleição, status da apuração.
   1.1. O usuário pode pesquisar uma eleição, informando os
        seguintes critérios:
   - objetivo
   ou
   - período inicial e final em que ocorreu a eleição
2. O sistema habilita as seguintes opções para o usuário:
   2.1. Inclusão de eleição
   2.2. Alteração de eleição
        2.2.1. Para alteração, o usuário deve pré-selecionar a eleição
   2.3. Exclusão de eleição
        2.3.1. Para exclusão, o usuário deve pré-selecionar a eleição
3. Para a opção de "Inclusão" ou "Alteração":
   3.1. O usuário informa/edita:
        3.1.1. objetivo da eleição
        3.1.2. data da eleição
4. Para a opção de "Exclusão":
   4.1. O sistema exibe os dados do item 3.1 desabilitados para edição.
   4.2. O usuário confirma a exclusão da eleição.
5. O usuário confirma a operação.
   5.1. O sistema atualiza o cadastro de eleições.

Cenários alternativos:
   Pesquisa de eleição
   1.a. A pesquisa de eleição deve desconsiderar caixa alta e
   baixa, acentuação e realizar a localização dos trechos em
   qualquer ordem que os mesmos apareçam no cadastro.
   1.b. A pesquisa de período deve ser feita considerando os
   períodos inicial e final, inclusive; podendo ser informado
   apenas um dos períodos.
   Permissão de "Alteração" da eleição
   2.a. Se já houver data que indique o início da votação, a
   eleição não poderá ser alterada. Exibir mensagem de erro e
   retornar ao passo 1.
   Permissão de "Exclusão" da eleição
   2.b. Se já houver data que indique o início da votação, a
   eleição não poderá ser excluída. Exibir mensagem de erro e
   retornar ao passo 1.
   Validação da data de eleição
   3.a. Se o usuário informar uma data de eleição que não seja
   futura, exibir mensagem de erro e retornar ao passo 3.
   3.a. Se o usuário informar uma data de eleição futura que já
   esteja cadastrada para outra eleição, exibir mensagem de erro
   informando a qual eleição a data já está cadastrada e retornar
   ao passo 3.
```

Listagem 2. UC Manter Candidatos

```
Descrição: Este caso de uso tem por objetivo manter o cadastro
de candidatos, permitindo a inclusão, alteração, exclusão ou
consulta de candidatos.
Atores: Coordenadoria de Eleições
Pré-condição: existir cadastro prévio de eleições.
Cenário principal:
1. O sistema prepara uma lista de eleições cadastradas, que
   ainda não tenham ocorrido a votação (início de votação em
   branco), exibindo para cada uma: objetivo, data da eleição,
   lista de candidatos cadastrados. Para cada candidato
   cadastrado, exibir: número e nome.
2. O usuário seleciona uma eleição.
3. O sistema habilita as seguintes opções para o usuário:
   3.1. Inclusão de candidato
   3.2. Alteração de candidato
        3.2.1.Para alteração, o usuário deve pré-selecionar
              também o candidato.
   3.3. Exclusão de candidato
        3.3.1. Para exclusão, o usuário deve pré-selecionar
               também o candidato.
4. Para a opção de “Inclusão” ou “Alteração”:
   4.1. O usuário informa/edita:
        4.1.1. número do candidato
        4.1.2. nome do candidato
        4.1.3. foto do candidato
5. Para a opção de “Exclusão”:
   5.1. O sistema exibe os dados do item 4.1 desabilitados para edição.
   5.2. O usuário confirma a exclusão da eleição.
6. O usuário confirma a operação.
   6.1 O sistema atualiza o cadastro de candidatos.

Cenários alternativos:
   Validação do número do candidato
   4.a. Se o usuário informar um número de candidato já cadastrado
   na eleição escolhida, exibir mensagem de erro informando a que
   candidato está associado e retornar ao passo 4.
```

Listagem 3. UC Iniciar Votação

```
Descrição: Este caso de uso tem por objetivo dar início à
votação de uma eleição.
Atores: Mesário
Pré-condição: existir uma eleição cuja data é igual à data
vigente e que ainda não tenha tido a votação iniciada.
Cenário principal:
1. O sistema busca a eleição cuja data é igual à data vigente.
2. O usuário confirma o início da votação.
   2.1. O sistema atualiza o cadastro de eleições, registrando a
        data e hora atual no campo “início da votação”.

Cenários alternativos:
Não se aplica
```

Listagem 4. UC Habilitar Eleitor

```
Descrição: Este caso de uso tem por objetivo validar o eleitor e
liberar a urna para votação.
Atores: Mesário
Pré-condição: existir uma eleição cuja data é igual à data
vigente e cujo início da votação já tenha sido preenchido.
Cenário principal:
1. O usuário informa a matrícula do eleitor.
   1.1. O sistema busca o eleitor, exibindo o seu nome.
2. O usuário libera a urna para votação.

Cenários alternativos:
   Validação da matrícula do eleitor
   1.a. Se o usuário informar um número de matrícula que não
   exista no cadastro, exibir mensagem de erro e retornar ao passo 1.
   1.b. Se o eleitor já tiver votado na eleição corrente, exibir
   mensagem de erro e retornar ao passo 1.
   Permissão para liberar a urna
   2.a. Se o eleitor anterior ainda não tiver finalizado a
   votação, exibir mensagem de erro e retornar ao passo 1.
```

Listagem 5. UC Votar

```
Descrição: Este caso de uso tem por objetivo permitir que o
eleitor registre o seu voto.
Atores: Eleitor
Pré-condição: a urna estar liberada para votação. Receber a
identificação da eleição e do eleitor habilitado para votação.
Cenário principal:
1. O sistema busca e exibe todos os candidatos associados à
   eleição identificada.
   1.1. Para cada candidato, o sistema exibe: número do
        candidato, nome do candidato e foto do candidato.
2. O sistema habilita as opções “Nulo” e “Branco”.
3. O usuário seleciona um dos candidatos ou uma das opções
   “Nulo” ou “Branco”.
4. O usuário confirma a votação.
5. O sistema atualiza o cadastro de votação.
   5.1. O sistema registra que o eleitor identificado já efetuou
        seu voto, sem associá-lo ao voto.
   5.2. O sistema computa 1 voto para o candidato selecionado ou
        para as opções “Nulo” ou “Branco”.
   5.3. O sistema libera a urna.

Cenários alternativos:
   Permissão de correção da votação
   4.a. O usuário pode solicitar a correção do seu voto, no
   momento de confirmar a votação. Nesse caso, o sistema deve
   retornar ao passo 3.
```

Modelando o primeiro Diagrama de Sequências

Ao se modelar um diagrama de sequências, em geral elaboramos ao menos um diagrama para cada caso de uso. Começaremos com o **UC Habilitar Eleitor**, por ser o mais simples para nossa primeira explicação.

Primeiro desenhamos o mesmo **ator** do caso de uso (*Mesário*), no canto superior esquerdo. No topo desenhamos, em seguida, um **objeto** para representar a interface do sistema e um objeto para cada classe envolvida nesse caso de uso. Nesse caso, as classes *Eleicao* e *Eleitor*. A partir de cada objeto, incluindo o ator, desenharemos uma linha tracejada que é a linha de vida desse objeto.

A **linha de vida** indica o tempo de vida desse objeto. Sendo colocada desde o início do diagrama é como se esse objeto fosse criado desde o início da rotina. Terminando no final do diagrama, indica que o objeto é destruído quando a rotina é encerrada. A maioria dos diagramas de sequências é desenhada dessa maneira. Contudo, se for necessário representar que um objeto é criado ou destruído durante a execução da rotina é possível fazê-lo. Veremos num tópico mais à frente. Veja a representação gráfica desse primeiro diagrama na **Figura 3**.

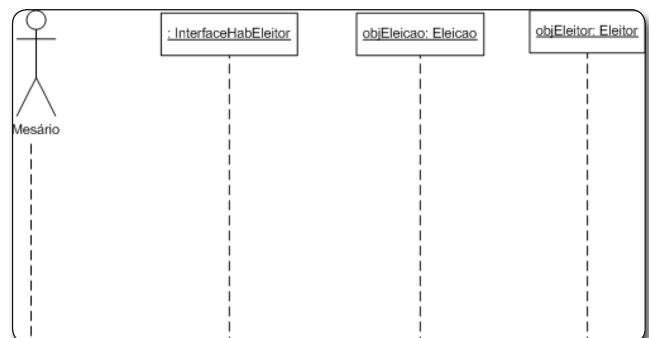


Figura 3. Primeira parte do desenho do Diagrama de Sequências do UC Habilitar Eleitor

Os objetos se diferenciam das classes por se apresentarem sublinhados. E podem ser desenhados com três notações diferentes:

<u>**nome do objeto : nome da classe**</u>

<u>**ou**</u>

<u>**: nome da classe**</u>

<u>**ou**</u>

<u>**nome do objeto**</u>

A primeira notação traz primeiro um nome aleatório de objeto que será útil se precisarmos usar esse objeto como parâmetro de um método.

A segunda notação indica um objeto anônimo, ou seja, desconhecemos o nome do objeto por ele ser irrelevante, mas conhecemos a classe da qual ele é instanciado.

A terceira notação é a menos indicada, pois desconhecemos o nome da classe. Nesse caso o nome do objeto deve ser claro o suficiente para que o leitor identifique de qual classe ele foi instanciado.

A partir da estrutura inicial, já podemos desenhar as **mensagens**, que partem sempre de uma origem para o objeto destino que contém o serviço que se está querendo chamar.

Assim começaríamos pelo item 1 do caso de uso. Contudo, como temos uma pré-condição, essa pré-condição pode representar apenas um parâmetro que a rotina receba ao ser iniciada, ou uma verificação que precisa ser verdadeira, para que a rotina tenha início. Como, nesse caso, representa uma verificação, a primeira mensagem de nosso diagrama de sequências será essa checagem.

A pré-condição determina que exista uma eleição cuja data é igual à data vigente e cujo início da votação já tenha sido preenchido. Isso significa que precisamos de um método na classe *Eleicao* que possa retornar se existe ou não uma eleição com a data atual. E depois, se existir, basta verificar se o atributo *inicioVotacao* está preenchido. Essa última verificação reside no processamento que se inicia com a mensagem que parte da *interface*. Se essa verificação for OK, a rotina terá início.

O retângulo que surge a partir de uma mensagem disparada é chamado de *caixa de ativação*. Ele dura o tempo de processamento daquela mensagem, tanto no objeto origem quanto no objeto destino.

O método criado na classe *Eleicao* é o *buscaEleicao*, passando como parâmetro a *dataAtual*, que será usada para comparação com as datas cadastradas das eleições.

Com a primeira verificação concluída, o ator está liberado para iniciar a interação com o sistema. Olhando o caso de uso, vemos no item 1 que o usuário informa a matrícula do eleitor. Essa representação é vista no diagrama de sequências com a mensagem partindo do ator (com o conteúdo *matriculaEleitor*) para o objeto *interface*. A partir dessa mensagem, no caso de uso, tem-se a ação do sistema buscando esse eleitor e exibindo seus dados, caso encontre. A busca do eleitor requer um novo método, nesse caso, o método *buscaEleitor* na classe *Eleitor*. Como argumento, esse método receberá a informação passada pelo ator, a *matriculaEleitor*.

Porém, repare que o cenário principal que faz a busca do eleitor (item 1) tem dois cenários alternativos associados. O primeiro trata a resposta falsa na busca do eleitor, o que já se resolve na mensagem de retorno. O segundo cenário alternativo verifica se esse eleitor já votou. Essa informação será obtida questionando-se à classe *Eleição* a existência de um relacionamento da eleição com eleitor, que só é criada quando da votação pelo eleitor. Assim, temos no mesmo processamento iniciado com a mensagem do ator, uma nova mensagem sendo disparada (sem que tenha havido interrupção do processamento) em direção à classe *Eleicao*, chamando pelo método *verificaVotacao*. Como argumento é passado o próprio objeto *Eleitor*, chamado *objEleitor*, que, nesse momento, estará preenchido com o eleitor identificado na mensagem anterior.

Com todas as validações efetuadas, o controle é repassado novamente ao usuário, para decidir o momento de liberar a urna para votação — item 2 do caso de uso. Porém, da mesma forma que ocorreu com o item anterior, esse item do cenário principal tem atrelado a ele um item do cenário alternativo. Então, logo depois da interface receber a mensagem do ator, ela, que representa o sistema, deve verificar se é possível liberar a urna. Para isso, é preciso questionar à classe *Eleicao* se a urna está liberada. Isso é feito pela chamada do método *urnaLiberada*. Aqui, fazemos uso do retorno *status* para colocar uma condição na mensagem que liberará a urna para o próximo eleitor.

Repare que, por convenção, só colocamos mensagem de retorno para o ator, no final do diagrama, indicando que a rotina foi concluída com sucesso.

Analisando-se esse diagrama, percebemos que identificamos quatro métodos da classe *Eleicao* (*buscaEleicao*(), *verificaVotacao*(), *urnaLiberada*() e *liberarUrna*()) e um método da classe *Eleitor* (*buscaEleitor*()) (ver **Figura 4**).

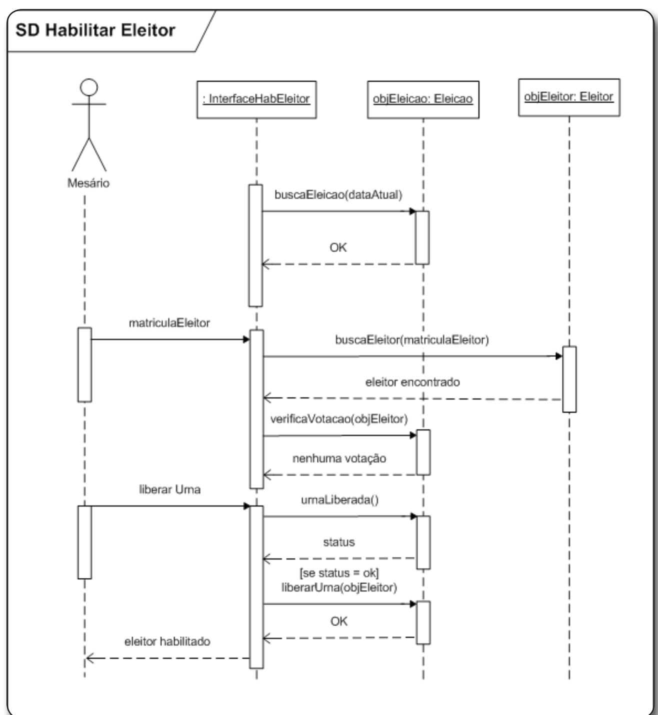


Figura 4. Diagrama de Sequências finalizado do UC Habilitar Eleitor

Modelando Diagrama de Sequências usando a nomenclatura da UML 2.0

Se podemos considerar um local no qual houve boas alterações na versão 2.0 da UML, esse local é o Diagrama de Sequências. A fim de obtermos maior legibilidade na separação das funcionalidades, de forma a permitir agrupamentos, condições relacionadas a um grupo de mensagens e não só a uma mensagem, trechos opcionais, etc., a UML disponibilizou os operadores e as ocorrências de interação.

Vejamos na prática como isso funciona modelando o Diagrama de Sequências do **UC Manter Candidatos**.

Começamos desenhando o ator *Coordenadoria de Eleições*. Em seguida, o objeto da *interface* e os objetos para as classes *Eleicao* e *Candidato*.

Analisando o caso de uso, vemos que a primeira providência a tomar é garantir a pré-condição. Então, traçaremos uma mensagem da interface (que representa o sistema) para a classe *Eleicao*, a fim de obter com o método *existeEleicaoCadastrada*() a garantia de que a pré-condição será cumprida.

Com a resposta positiva, o sistema continua com o controle, pois deverá trazer a lista de eleições cadastradas, que ainda não tenham ocorrido a votação. Nesse caso, criaremos o método *buscaEleicoesNaoRealizadas*() da classe *Eleicao*. Repare que o retorno dessa lista não traz apenas dados da eleição, mas dados do candidato também. Isso significa dizer que o método *buscaEleicoesNaoRealizadas*() deve ser responsável por obter, para cada eleição encontrada, os candidatos que já se encontram cadastrados. Essa dependência na busca é típica do relacionamento de agregação por composição que existe entre as classes *Eleicao* e *Candidato*.

Então, dentro do processamento do método buscaEleicoesNaoRealizadas() é feita uma chamada ao objeto Candidato, chamando o método buscaCandidato() que fará uma busca simples dos atributos desse objeto. Só que esse método não será chamado uma única vez, e, sim, tantas vezes quantas forem a quantidade de candidatos cadastrados. Por isso, aparece sobre a mensagem o símbolo "*" para indicar iteração (repetição) e a condição, colocada entre colchetes, para indicar a condição de parada.

Veja que só depois que essa busca é feita, o controle é entregue, pela primeira vez ao ator. Então ele faz a seleção de uma das eleições exibidas. Veja que todo esse processo de exibição fica dentro da caixa de ativação, ou seja, não é relevante mostrar no Diagrama de Seqüências, pois o detalhamento desse tipo de procedimento já está no caso de uso.

A partir da seleção da eleição, o controle volta ao sistema que habilita as opções. Isso também não é representado explicitamente no diagrama de sequências, ficando subentendido na caixa de processamento. A quebra da caixa de ativação indica que o controle volta para o usuário, para que o use no tempo desejado. Na nova intervenção do ator, há a passagem de uma mensagem com a seleção da sua opção.

Repare que, a partir desse ponto, temos blocos diferentes de processamento. Temos um bloco reservado à inclusão e alteração, e outro à exclusão. Os blocos são controlados pelo **operador alt**. Em cada bloco interno, coloca-se uma condição que indica que esse bloco só será executado se a condição for verdadeira. Assim, o primeiro bloco só será executado se a opção selecionada pelo usuário tiver sido a inclusão ou a alteração. No segundo bloco, somente se a opção selecionada tiver sido a exclusão.

Vemos que todas as mensagens que ficam dentro de um bloco são pertinentes só e somente só a esse bloco. Então, só haverá passagem da mensagem com o *número, nome e foto do candidato*, se o processamento estiver na inclusão ou na alteração. Uma vez informados esses dados, o diagrama representa o cenário alternativo que verifica se esse número é duplicado ou não. Assim surge a mensagem com chamada do método *buscaCandidato* da classe *Candidato.* Se nenhum candidato for encontrado com aquele número, o sistema estará habilitado para continuar o processo de edição, chamando então o método *gravaCandidato* da classe *Eleicao*, pois somente essa classe pode se responsabilizar pela gravação do candidato, visto que a classe *Eleicao* é a classe todo no relacionamento de agregação por composição.

No segundo bloco, há a mensagem de confirmação do ator. A partir do recebimento dessa mensagem, o sistema pode passar a mensagem *excluiCandidato* para a classe *Eleicao* para que ela se responsabilize pela chamada do método *excluiCandidato*() da classe *Candidato.* Veja como ficou esse diagrama na **Figura 5**.

Outros recursos do Diagrama de Sequências

Vejamos os outros recursos que o Diagrama de Sequências nos oferece para se atingir o melhor que for possível na modelagem que representará um caso de uso.

Vimos que, por padrão, criamos os objetos no início do diagrama e sua destruição é feita ao término do mesmo, quando a rotina se encerra. Mas se precisarmos criar um objeto no meio do diagrama, devemos colocar uma mensagem de criação chegando ao centro do objeto, e depois podemos incluir as outras mensagens que executarão os demais métodos. Veja a estrutura na **Figura 6**.

Da mesma forma, para excluir um objeto durante a execução do diagrama, devemos colocar uma mensagem de destruição chegando ao final da linha de vida, onde é colocado um X, para indicar esse fim "precoce". Veja a estrutura na **Figura 7**.

Outra necessidade que pode surgir durante a modelagem é a chamada de um método dentro da própria classe. Suponha que uma mensagem que chega no objeto Disciplina chame o método busca. Porém, ao encontrar a disciplina, temos que fazer a busca de todos os objetos que estejam ligados a essa instância. E se um desses objetos for a própria disciplina, que esteja como um atributo de pré-requisito, teremos que fazer uma nova busca, ao mesmo objeto. Essa chamada recursiva é denominada **auto-chamada** e é representada por uma caixa de ativação sobreposta à caixa anterior. Veja exemplo na **Figura 8.**

Para finalizarmos, vamos conhecer os outros operadores existentes e seus objetivos. É bom lembrar que esses operadores são colocados na aba de identificação do frame com o qual estamos trabalhando.

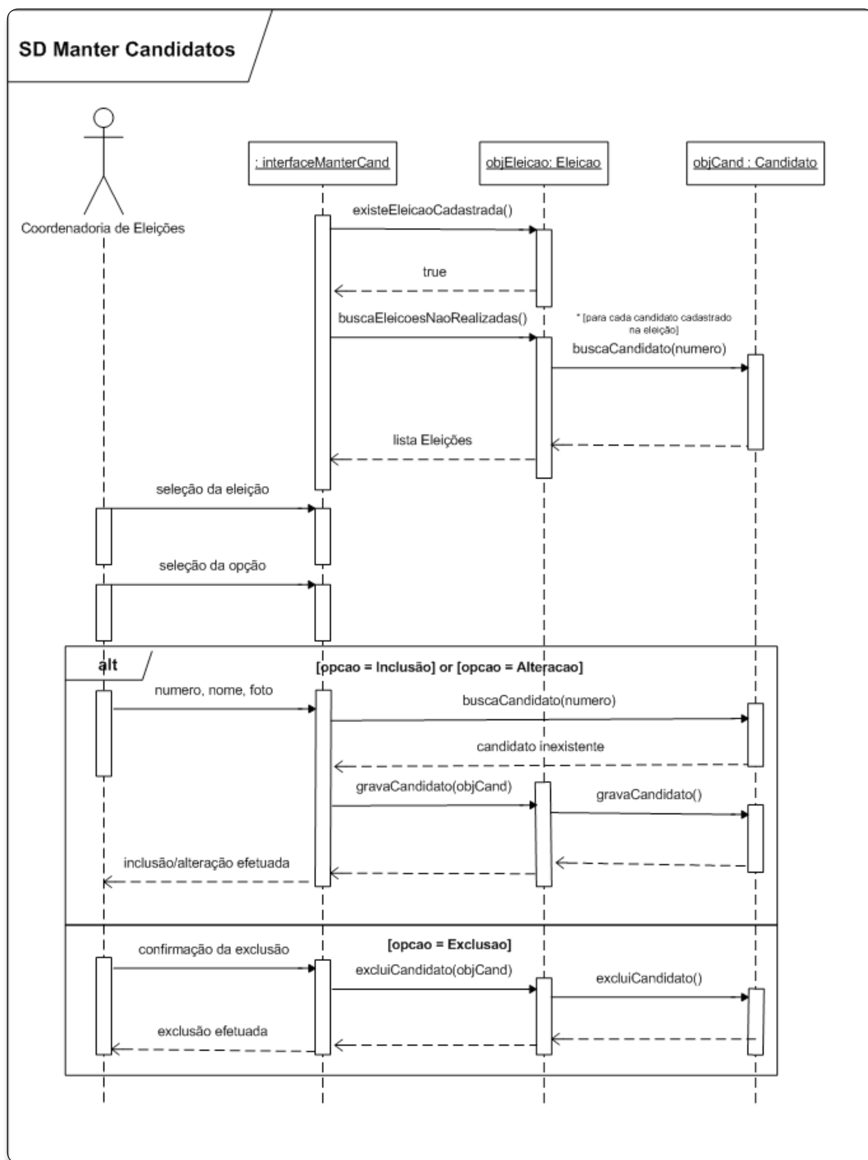


Figura 5. Diagrama de Sequências finalizado do UC Manter Candidatos

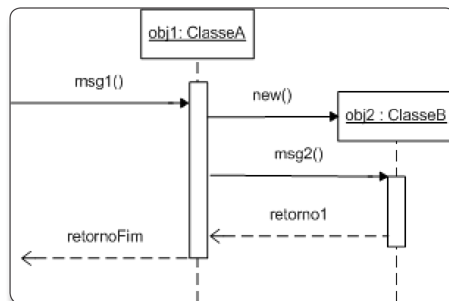


Figura 6. Sintaxe para criação do objeto durante a sequência de mensagens

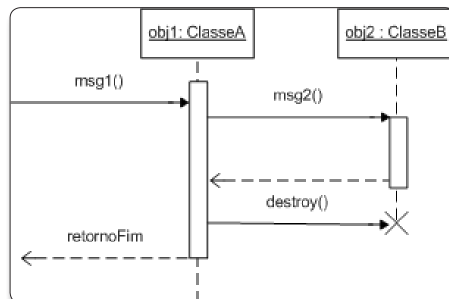


Figura 7. Sintaxe para destruição do objeto durante a sequência de mensagens

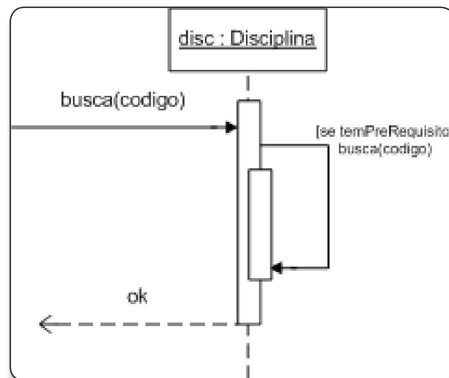


Figura 8. Exemplo de auto-chamada

- **Operador ref:** permite referenciar como ocorrência de interação uma outra interação que representa uma porção comum. O uso do operador ref permite que trechos de um diagrama de sequência sejam isolados em pequenos frames e chamados apenas pelo nome, evitando a duplicidade de modelagem. Não só a modelagem é evitada, como a manutenção é simplificada em um único lugar. Veja exemplo nas **Figuras 9** e **10** com o uso desse operador ref.

  Para entender esse exemplo, imagine um sistema de controle acadêmico, no qual na maioria das funções é necessário que se faça uma busca do aluno, por meio de sua matrícula. No nosso exemplo, esse trecho de busca (**Figura 9**) é representado apenas por uma mensagem da interface para a classe aluno, com a chamada do método busca, e depois a mensagem de retorno. Mas esse trecho poderia ser mais complexo, com chamadas a mais de uma classe, a mais de um método. Imagine esse trecho sendo repetido em vários diagramas de seqüência! Além do esforço em se reproduzir um mesmo trecho, em caso de manutenção, teríamos que procurar todos os diagramas para alteração. No momento em que extraímos esse trecho de interação que se repete em vários lugares, e o colocamos num único lugar, ganhamos reaproveitamento de modelo e agilidade de manutenção. Após o trecho ser separado, basta que façamos a sua chamada no diagrama de seqüências que precisará utilizá-lo, por meio da ocorrência de interação, com o operador ref. Em vez de se desenhar o trecho, desenha-se apenas um frame, colocando ao centro o nome da ocorrência de interação (**Figura 10**). Também é possível adicionarmos parâmetros de entrada e saída a essas ocorrências de interação. O default é o parâmetro de entrada, em que nada é preciso acrescentar. Para o parâmetro de saída, acrescenta-se a palavra-chave **out**. Veja exemplo:

```
sd BuscarAlunoMaiorMedia (objDisciplina, out media):
                objAluno
```

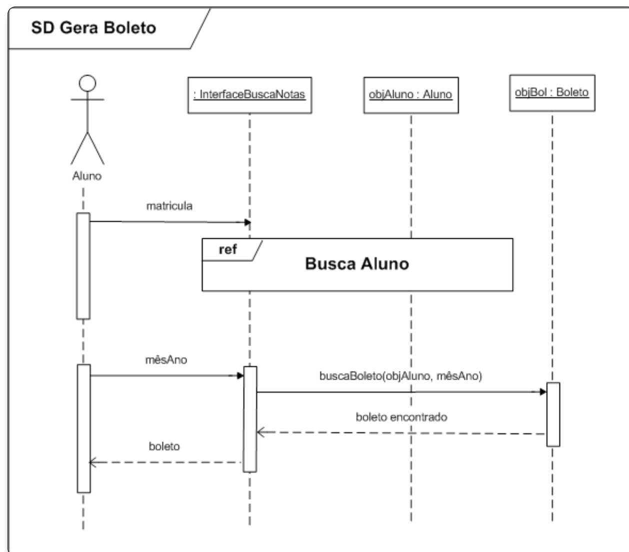


Figura 9. Exemplo de interação que será referenciada

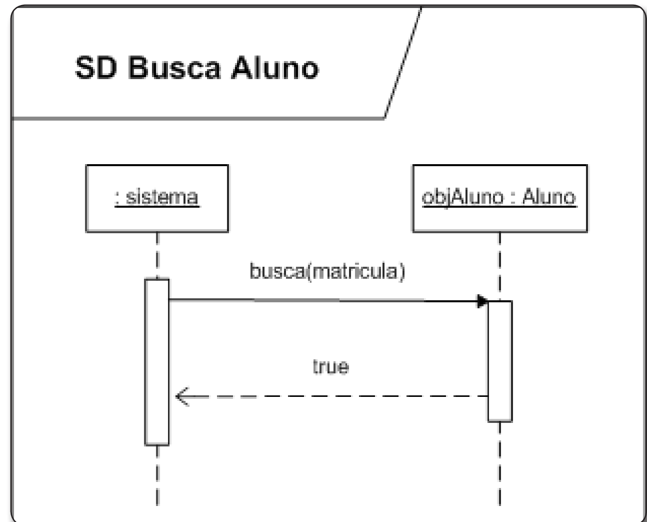


Figura 10. Diagrama de sequências com uma ocorrência de interação para o trecho "Busca Aluno"

No exemplo acima, a ocorrência de interação recebe dois parâmetros: objDisciplina e media. O parâmetro objDisciplina é apenas de entrada. O parâmetro media deve retornar preenchido, pois é um parâmetro de saída. O retorno da função será o objAluno que indicará quem é o aluno que tem a maior média. Assim, conseguimos que o retorno do Diagrama de Sequências terá dois valores de retorno: o objeto que representa o aluno com maior média e a própria média desse aluno.

• Operador opt: permite definir um trecho da interação como opcional. Similar ao que acontece com o operador alt, é colocado no topo, ao centro, uma condição que indicará se o bloco será executado ou não.

Suponha que temos um diagrama de sequências que recebe o valor vendido de um produto, para atualizar o estoque. Logo depois de chamar o método que faz a atualização do estoque, o sistema deve checar se o estoque do produto atingiu o valor mínimo. Se atingiu, deve emitir um pedido de compra, enviar um alerta para o gerente da área e um outro para o setor de compras. Imagine que são várias mensagens que dependem da resposta do teste:

```
[estoque.valorAtual <= estoque.valorMinimo]
```

Na versão antiga da UML, teríamos que repetir a condição em todas as mensagens, para que não houvesse dúvida de que todas elas eram dependentes da mesma condição.

Na versão nova, basta envolvermos todas essas mensagens num frame e identificá-lo com o operador opt. Como condição de guarda colocamos o teste citado acima. Isso significa que o trecho indicado pelo frame só será executado se o valor atual do estoque estiver menor ou igual ao valor mínimo de estoque.

Operador loop: permite a repetição de um trecho da interação. Nesse caso, indicamos um intervalo mínimo e máximo para repetição com uma condição de parada. Se não houver a condição de parada, que é opcional, o bloco será repetido tantas vezes quanto for o intervalo entre o valor mínimo e o máximo.

Veja exemplo:

```
loop 1, 3 [senha not OK]
```

O exemplo indica que o trecho marcado com o operador loop será executado no mínimo, uma, no máximo, três vezes. Contudo, o trecho só será executado enquanto a senha não estiver OK. Então, se após a primeira vez, a senha estiver OK, o trecho é abandonado, continuando o processamento logo após o fim do frame do operador loop. Se após a terceira vez, a senha continuar inválida, mesmo assim o trecho será abandonado, pois se atingiu o limite máximo de três tentativas.

Conclusão

Neste artigo foi apresentado o uso do diagrama de sequencia. Percebemos que integrando corretamente os casos de uso, diagrama de classes e o diagrama de sequências, temos um modelo completo para implementação da maioria dos sistemas orientados a objetos. ●

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

Cursos Online

A **Revista WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade **.Conheça abaixo os cursos já disponíveis.**

Curso de .net em destaque

Aplicação completa de orçamento Doméstico no Visual Studio 2005

Confira neste curso online como criar uma aplicação completa no Visual Studio 2005, usando ASP.NET, Web Services, Mobile e muito mais! Veja como criar uma aplicação de orçamento doméstico, usando diagrama de classes de uma forma muita produtiva, criar classes de negócios de acesso a dados.

Confira o plano de aula completo:
www.devmedia.com.br/domesticovs

Curso de Java em destaque

Introdução ao desenvolvimento para celulares com J2ME

Confira neste curso os principais recursos do J2ME. Aprenda também o passo a passo para criar sua primeira aplicação J2ME. Neste curso você irá aprender diversas funcionalidades desta tecnologia para desenvolvimento de dispositivos móveis.

Confira o plano de aula completo:
www.devmedia.com.br/celularesj2me

Assine a **WebMobile** e comece já seu treinamento!
www.devmedia.com.br/assine

A sua melhor opção de aprendizagem!

Em breve mais novidades para você! www.devmedia.com.br/curso

DevMedia GROUP **Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025**

Testes de Software, um processo criativo

Melissa Barbosa Pontes

melissa.pontes@cesar.org.br

Mestre em Engenharia de Software do Cesar.EDU. Certificada em testes de software pelo ISTBQ (International Software Testing Qualification Board), atualmente trabalha como engenheira de testes no CESAR (Centro de Estudos e Sistemas Avançados do Recife), onde desempenha atividades de definição de processos, liderança e consultoria. É integrante do GrIT (Grupo Independente de Testes do Cesar) através do qual realiza pesquisas e treinamentos em testes na organização. Possui artigos e trabalhos publicados em eventos internacionais. É membro integrante da comissão de organização de EBTS - Encontro Brasileiro de Testes de Software, que já se encontra na quarta edição.

Um importante fator de sucesso de um software é sua qualidade. Existem diversas maneiras de se avaliar a qualidade de um produto, uma dessas maneiras é a realização de atividades de testes de software por uma equipe especializada no assunto. O principal objetivo dessas atividades é descobrir se o produto está de acordo com as exigências do cliente. A introdução dessas atividades em uma organização deve ser feita de maneira estruturada, de forma que lhe permita uma boa definição de quais atividades devem ocorrer em um projeto, em que sequência e por quem estas devem ser realizadas.

O Processo de Testes de Software

As atividades de testes de software são diversificadas, e são mais bem desempenhadas se estiverem organizadas em um processo. Koomen, em seu livro chamado Test Process Improvement: A step-by-step guide to structured testing, estima que entre 25% e 50% dos custos e esforços de um projeto são gastos com testes. Daí a importância de investimentos em definição e melhoria de um bom processo de testes de software para uma organização.

De que se trata o artigo?

Este artigo aborda o tema processo de testes de software mostrando uma maneira de agrupar as atividades de testes em etapas, com objetivos bem definidos. Ele mostra que a organização das atividades pode ter um impacto positivo significativo na qualidade do produto gerado, e que muito mais do que burocracia é preciso criatividade para um bom desempenho de tais atividades.

Para que serve?

Uma maneira de se avaliar a qualidade de um produto é através da realização de atividades de testes. Porém, se essas atividades forem realizadas de maneira ad-hoc, ou seja, sem um planejamento nem estruturação, pode não ficar tão evidente a dimensão da contribuição que os testes podem trazer para a qualidade de um produto. Por isso, é fundamental a definição e utilização de um processo para guiar a equipe de testes na condução de suas atividades.

Em que situação o tema é útil?

Uma vez entendida a importância da realização de testes em um projeto, é prudente que estas comecem de maneira mais estruturada possível. A visão geral de um processo de testes fornecida por este artigo dá uma idéia de como implantar as atividades de testes em uma organização com um mínimo de organização.

Essas atividades devem iniciar o quanto antes em um projeto de desenvolvimento, se possível no primeiro dia do projeto. Dessa maneira, defeitos podem ser identificados o quanto antes no produto, minimizando os custos da sua correção. Isso nos leva a concluir que não é uma prática vantajosa tratar testes como uma única atividade de execução, e deixá-las para o final do projeto. Isso poderá gerar uma corrida por correções de defeitos, e estudos mostram que quanto mais modificado é um código, maior a propensão desse código ao aparecimento de mais defeitos. Esta constatação pode ser evidenciada pelo trabalho de Naggapan, publicado no artigo Use of relative code churn measures to predict system defect density, no ano de 2005.

De maneira simplificada, o processo de testes pode ser ilustrado como na **Figura 1**.

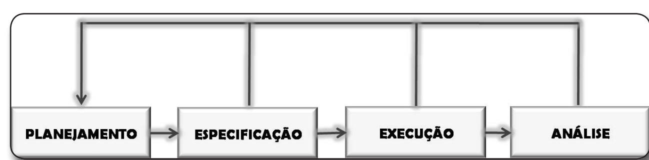


Figura 1. Processo de testes

Na maioria dos casos, as fases do processo acontecem em sequência, como apresentado na figura, e nada impede de voltarmos para uma fase anterior para melhorar suas atividades.

Trata-se de um processo com atividades e papéis bem definidos, e é necessária muita criatividade em quase todas as etapas. A equipe deve estar segura de que elaborou diversos cenários diferentes para avaliar a aplicação. Também deve ter a certeza de que os executores de teste souberam discernir entre um produto de qualidade e um produto que ainda não está pronto para ir ao mercado, e utilizaram sua percepção sobre o produto para influenciar decisões tomadas pela alta gerência.

Várias atividades do processo podem ser realizadas de forma automática, mas isso não é o foco deste artigo.

A seguir, falaremos um pouco sobre cada atividade, dando uma visão geral.

Planejamento de Testes

Objetivo Principal: Elaborar uma estratégia de testes para o projeto e definir como esta será implementada. Tudo isso deve ser documentado no Plano de Testes, artefato principal desta etapa.

Planejar começa com um bom entendimento de contexto. Antes de dar inicio às atividades de testes propriamente ditas, a equipe deve entender o objetivo do projeto em questão, e somente a partir daí, definir o objetivo dos testes, que pode ser: identificar novos defeitos, checar se novas implementações não quebraram algo que funcionava antes, homologar a aplicação junto ao cliente, etc.

Entender o objetivo do projeto significa estar alinhado com as necessidades do cliente, além de entender quais riscos este projeto apresenta e levantá-los com antecedência.

Começar o planejamento com a definição do escopo é fundamental. Para saber como testar, é necessário saber o que será testado. E também é importante saber o que não será testado, e o porquê. A partir daí é que a equipe consegue definir melhor o tempo que levará para realizar as atividades, quantas pessoas devem ser envolvidas e se será ou não necessária a utilização de ferramentas, por exemplo.

A estratégia é a alma do planejamento de testes, mas o planejamento envolve muito mais do que elaborar a estratégia. Nesse momento deve ser identificado de que maneira os testes vão ser organizados, quem vai escrevê-los, quem vai executá-los, e em que momento a equipe poderá parar de testar.

Devem ser definidos quais os níveis de testes que vão ser realizados no projeto. Uma boa definição de níveis de teste pode ser encontrada no artigo Introdução a Teste de Software, publicado na primeira edição desta revista, por Arilo Dias-Neto. Para cada nível de teste, deve-se definir quantos ciclos de teste serão executados. Um ciclo é um conjunto de testes pronto para execução.

Também faz parte da estratégia o nível de cobertura dos testes. Dentro do escopo definido, a equipe deve decidir se vai exercitar todo o código, ou se uma porcentagem dele já é suficiente. Esse critério de cobertura do código, ou dos requisitos, somado aos defeitos encontrados, ajudam a definir o critério de parada dos testes. Como a equipe sabe se testou o suficiente? A resposta para essa pergunta deve estar no plano de testes.

Deve-se buscar toda a informação que for possível em relação ao projeto. Então, documentação de requisitos e casos de uso, caso existam, ajuda muito, mas nem sempre são suficientes. E por que não são suficientes? Porque podem estar errados, ambíguos ou mal escritos. Se esses documentos estiverem mal escritos podem até atrapalhar o time de testes.

Se envolver em reuniões de planejamento do projeto ajuda a identificar o que é mais importante no produto e que áreas representam riscos para a gerência e para a equipe de desenvolvimento. Ainda, saber quando o software deve ser lançado no mercado, qual o seu público alvo, qual a estratégia de marketing e qual a mensagem que a organização quer passar para o público com o produto desenvolvido.

Se os usuários esperam um produto fácil de usar, os testes devem identificar se tudo está em conformidade com esse aspecto. Caso o produto tenha que dar respostas rápidas ao usuário, muda completamente o tipo de teste que verifica essa característica, porque nesse caso, o desempenho da aplicação deve ser analisado. E aí, a partir do que se espera do produto, vai se modelando a estratégia de testes, que consiste em identificar que tipos de testes vão ser realizados para cada área da aplicação, em qual momento, e com qual intensidade.

Todo o conhecimento adquirido em relação ao projeto vai compor o que na prática chamamos de oráculo. No livro Software Testing Foundations, Spillner explica que oráculo é um mecanismo de obtenção dos resultados esperados. É através do oráculo, dentre outras coisas, que a equipe de testes vai conseguir identificar com mais segurança se já realizou testes suficientes para poder emitir uma opinião sobre a qualidade do produto.

Assim como em todas as atividades de um projeto, as atividades de testes também podem ser impactadas por problemas que chegam a inviabilizar sua realização. Por isso, é importante que a equipe identifique riscos para minimizar o impacto desses incidentes no projeto.

Exemplos do que podem atrapalhar as atividades de testes são: atraso na entrega do código, pela equipe de desenvolvimento, falta de ferramentas e ambiente adequado, imaturidade da equipe de testes, e muito mais. Tudo deve ser registrado no plano de testes e algum direcionamento deve ser dado a esses problemas.

Especificação de casos de teste

Objetivo: Produzir uma suíte de testes, ou seja, um conjunto de casos de teste.

Essa fase também é conhecida como fase de projeto de testes. Este projeto pode ser de execução manual, semi-automática ou automática. Caso seja manual, os casos de testes serão escritos e podem ser armazenados em planilha excel, documento do Word, ou ferramenta de gerenciamento de casos de teste. Caso seja semi-automática ou automática, serão gerados scripts após a escrita manual, e serão desenvolvidos em linguagem de programação. Neste artigo, daremos foco aos casos de testes manuais.

De onde vêm os casos de teste? Eles podem ser elaborados a partir de diversas fontes, inclusive da imaginação de quem os está escrevendo. Geralmente os casos de teste são escritos com base nas documentações do projeto, e têm o objetivo de checar se o software funciona corretamente e principalmente se está de acordo com o que foi especificado pelo cliente.

Que documentações de projeto podem dar origem a casos de teste? Na maioria das vezes são os requisitos e casos de uso especificados, mas nada impede, e é até aconselhável, que toda a documentação existente seja analisada. Isso ajuda no entendimento do objetivo do projeto e da sua criticidade para o cliente.

Então, o projetista de testes, ou desenhista, como é chamada a pessoa que escreve os casos de teste, depois de entender o objetivo dos seus testes, a chamada missão de testes, deve imaginar os cenários através dos quais tentará encontrar os defeitos na aplicação.

Consultando o plano de testes, o projetista identifica para quais funcionalidades da aplicação ele deve escrever testes em dado momento. Na seção de escopo dos testes do plano devem estar definidos, através de um trabalho realizado na fase anterior do processo, quais são os requisitos a ser testados e que tipos de testes (funcionalidade, carga, desempenho) podem ser aplicados a aqueles requisitos. O trabalho agora é dizer como testar.

Serão elaborados os roteiros que vão ajudar os executores de testes a analisar o produto.

Com o que se parece um caso de teste?

Um caso de teste simples deve conter: um objetivo, que os testadores devem tentar alcançar; um passo a passo, mais conhecido como procedimento de teste, através dos quais o objetivo do teste pode ser alcançado. Além disso, o teste pode ter uma pré-condição, que indica o que é necessário para a sua realização.

Mais informações podem estar presentes em um caso de teste, mas no momento, ficaremos com um exemplo bem simples, somente para dar a idéia do que se trata. A **Figura 2** traz um exemplo de como se escreve um caso de teste para um requisito de login de uma aplicação.

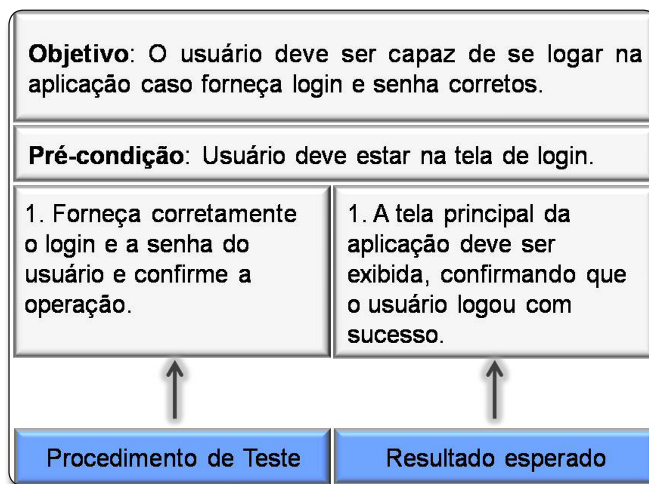


Figura 2. Exemplo de Caso de Teste

Quanto mais casos de testes, melhor!

Muita criatividade, além da documentação disponível, é necessária para se escrever bons e diversificados cenários de testes. É preciso imaginar o que outras pessoas não pensaram. Explorar mentalmente as diversas maneiras de utilizar a aplicação, tentando fazer com que a mesma levante todas as exceções, provocar para que ela dê mensagens de erro, imaginar de que maneira as funcionalidades poderiam apresentar um comportamento inesperado. Geralmente a forma como o sistema não deve funcionar não está especificado.

Um projeto de testes escrito sem criatividade fica muito parecido com os requisitos e casos de uso. O mais indicado é ter quantos casos de testes forem possíveis para cada requisito da aplicação.

Podemos então deduzir que quanto mais cenários forem elaborados, mais bem testada estará a aplicação. Cuidado com essa afirmação, pois a geração de muitos cenários, sem um direcionamento correto, pode trazer alguns problemas para a equipe:

1. Explosão de casos de teste. O projetista pode escrever tantos cenários que o time não será capaz de executá-los no tempo que dispõe;
2. Casos de testes repetidos. Pode ocorrer que alguns testes, aparentemente diferentes, exercitam o mesmo trecho de código do software, não agregando nenhuma informação nova para a equipe, e consumindo tempo de escrita, execução e análise de seu resultado;
3. Dificuldade de manutenção da suíte de testes. Gerenciar uma grande quantidade de casos de teste pode ser tornar

uma tarefa exaustiva e desmotivadora. Além de propiciar o aparecimento de erros humanos.

Como então minimizar o impacto das dificuldades mencionadas acima?

1. Definir prioridades. Projetar o que é mais crítico e mais importante primeiro. Além de classificar os casos de testes por níveis de prioridade. Casos de testes muito importantes podem receber uma identificação, o que vai fazer com que as pessoas sempre olhem para eles na hora de compor um ciclo de execução de testes;
2. Escolher os melhores cenários. De que maneira? Existem várias técnicas na literatura que ajudam os projetistas a identificar bons cenários para encontrar defeitos, e ainda ajudam a não repetir casos de teste. O artigo Introdução a Teste de Software, de Arilo Dias-Neto, já mencionado anteriormente, traz uma breve explicação sobre essas técnicas;
3. Elaborar uma matriz de rastreabilidade, que é uma correspondência entre os requisitos da aplicação e os casos de teste. Dessa maneira, cada vez que um requisito for modificado ou removido, fica mais fácil saber quais casos de testes são impactados pela mudança, facilitando a manutenção da suíte de testes.

Além da escolha do cenário, deve-se ter muito cuidado na escrita do caso de teste. Embora escrever um caso de testes pareça uma atividade simples, existem algumas armadilhas que devem ser evitadas porque podem causar problemas no futuro.

Começando pelo objetivo do caso de testes:

1. Este deve ser simples e direto, contendo uma única finalidade. Isso vai evitar que o testador se confunda em relação ao que deve fazer. Também vai evitar que um caso de teste tenha metade de seu objetivo passando, e metade dele falhando, o que provocaria confusão no momento de decidir se esse teste passou ou falhou;
2. Deve ter base nos requisitos, mas foco no usuário final. O que isso quer dizer? Que os requisitos não são perfeitos e o testador tem que por suas impressões, como pessoa, para avaliar se tudo está indo bem;
3. São mais importantes do que o procedimento. Porque o procedimento de testes escrito por um projetista, nem sempre é a única maneira de se atingir o objetivo. Às vezes dá para chegar ao mesmo lugar por diversos caminhos.

O que dizer sobre procedimentos de testes?

1. Não devem divergir do objetivo do caso de teste. Supondo que o objetivo do caso de teste seja alcançado, e somente um passo do seu procedimento, divergente do objetivo, esteja falhando, vai ser trabalhoso para quem vai analisar métricas de testes contabilizarem testes que estão passando ou falhando;
2. Devem ter a menor quantidade de passos possível. Um caso de testes com muitos passos pode ser um indicativo de que o seu escopo está muito grande, indo além do objetivo principal, ou que o objetivo não está claro;
3. Se possível, devem ter seu passo a passo escrito em alto nível. Em vez de escrever "Clique no botão OK", pode ser melhor escrever "Confirme a operação". Isso vai evitar muito retrabalho caso itens da tela sejam redefinidos, por exemplo. A não ser que a situação seja um teste especifico de rótulos da tela.

Se os casos de teste não forem simples e fáceis de gerenciar, fica difícil medir o progresso do projeto em relação à quantidade de testes passando.

Execução de testes

Objetivo: Executar os casos de testes especificados na fase anterior com o objetivo de encontrar defeitos na aplicação.

A primeira coisa a fazer então quando o projeto está na fase de execução de testes é preparar o cenário para que tudo aconteça como planejado nas fases anteriores. O plano de testes deve ser analisado para entender como será o ambiente dessa execução. E este ambiente deve ser preparado para os testes, caso contrário, essa atividade pode até chegar a ser suspensa, ou mesmo cancelada.

Em seguida, os ciclos de testes devem ser criados. Um ciclo de teste é uma maneira de definir o escopo da execução em determinado momento. Em um ciclo devem entrar somente os casos de testes que serão executados. Como organizar então um ciclo de execução de testes? Pelo contexto. Responder algumas perguntas pode ajudar, tais como: quais funcionalidades foram recentemente implementadas? Caso se deseje testar coisas novas; áreas antigas do sistema ainda continuam funcionando corretamente? Caso o time esteja querendo analisar o impacto da integração de novas funcionalidades no restante da aplicação. Esses são alguns exemplos de como escolher os testes para um ciclo.

Definido o ciclo, que deve estar de acordo com o contexto e com o plano de testes, chega o momento de começar a execução propriamente dita. Por onde começar então? Uma dica é organizar testes que tenham configurações semelhantes como pré-requisito para execução, e atribuí-los para a mesma pessoa da equipe. Com essa prática, a equipe ganha tempo configurando o ambiente apenas uma vez, e executando todos os testes de uma vez.

Em geral, os executores de testes devem pegar os roteiros definidos pelos projetistas e segui-los com o foco no seu objetivo. Caso a aplicação se comporte de maneira diferente do que o caso de teste especifica, pode ser que realmente haja um defeito no software, e esse caso de teste então deve falhar. Da mesma maneira, caso a aplicação se comporte de forma da mesma maneira que o caso de teste especifica, este teste deve passar. Um terceiro estado também pode ser atribuído como resultado de um caso de teste. Este pode ser bloqueado, se por algum motivo se encontrar impedido de ser executado. Os defeitos encontrados devem ser registrados para futura análise.

Quase sempre, é da maneira descrita acima que as execuções de teste acontecem, mas isso não é uma regra. Um caso de teste divergente do comportamento da aplicação pode não significar

que um defeito de software foi encontrado, mas sim que um defeito no caso de teste existe. A equipe deve estar atenta a esse tipo de situação para melhorar o produto e o processo.

Realizar testes com roteiro e ainda usar a criatividade. Isso é possível?

Em geral, testadores se baseiam nos casos de teste, e seguem o seu roteiro para testar a aplicação em questão. Mas, para a obtenção de um melhor resultado com a execução de testes, é recomendado que o roteiro sirva somente como um cenário básico. Seguir o roteiro sem limitar-se a eles é uma prática que agrega muito valor ao projeto, pois permite ao time contar com a criatividade dos executores de testes também. Executores devem sentir-se livres para testar sua imaginação no projeto e relatar tudo o que puderem observar.

O que se pode observar com essa prática é que muito mais defeitos são encontrados com um só caso de teste.

Pode haver defeitos sem casos de teste associados?

Em um projeto onde todos contribuem com sua imaginação, isso geralmente acontece. Isso comprova que nem sempre os projetistas de testes conseguem pensar em todas as situações possíveis de uso da aplicação. Esses novos cenários encontrados pelos executores de teste devem ajudar na definição de novos testes a serem incluídos na suíte para serem realizados nas próximas versões do código.

Análise de Resultados

Objetivo: Avaliar os resultados da execução dos testes e analisar se tudo está de acordo com o que se esperava do produto.

Uma vez que o ciclo de teste foi realizado e os defeitos encontrados foram registrados, um relatório de defeitos deve ser construído para relatar tudo o que aconteceu. Devem constar nesse relatório quais foram os casos de teste que entraram no ciclo, qual o resultado de cada um, quais defeitos foram abertos e a que caso de teste eles estão relacionados, qual a versão do software analisada, quem foram os testadores envolvidos nessa execução. Outras informações também podem ser definidas pela equipe do projeto.

Neste momento deve-se analisar se o produto está de acordo com o critério de aceitação definido no início do projeto, mas não podemos concluir muito sobre a qualidade do produto baseando apenas nos resultados dos casos de teste. A equipe deve se basear em todo o processo utilizado em testes para chegar a uma conclusão mais confiável. Um processo de qualidade dá mais confiança a equipe de testes na hora de opinar sobre o estado da aplicação.

A fase de análise deve ser um trabalho completo, analisando o produto e o processo. E o resultado dessa análise deve dar subsídios que ajudem em várias decisões, tais como:

1. Decidir se o produto está pronto para ser lançado no mercado, ainda que possua alguns defeitos;
2. Corrigir defeitos críticos antes de liberar o produto;
3. Executar mais testes, inclusive elaborando novos cenários;
4. Melhorar a escrita e / ou cobertura dos casos de teste.

Muitos testes falharam. O software está ruim?

Pode não ser esta a questão. Os casos de teste podem estar desatualizados, ou mal escritos, e isso pode causar problemas de entendimento na hora da execução. Em times pequenos, geralmente os casos de teste são escritos por uma pessoa só, que em fases iniciais do projeto pode ter se baseado em requisitos ainda imaturos.

O conhecimento de como a aplicação funciona verdadeiramente fica mais evidente quando o testador está vendo o produto, o que pode gerar alterações nos casos de teste ou mesmo nos requisitos do projeto. Essa questão deve ser avaliada com cautela antes de tirar conclusões sobre o produto.

Melhoria de um Processo de Testes

Por mais simples que seja um processo de testes, ele pode ser melhorado. Porém, Rex Black, em seu livro Advanced Software Testing, ressalta um ponto muito importante. Embora existam algumas frentes para a melhoria de processos de software, tais como: CMM (Capability Maturity Model), CMMi (Capability Maturity Model Integration) e SPICE (Software Process Improvement and Capability dEtermination), deve ficar claro que a busca por qualidade nesses modelos não se dá através de testes como o principal caminho.

Devido á esta lacuna que os modelos mencionados acima deixam, alguns modelos foram propostos para melhoria de processos de testes especificamente. Alguns deles são:

- Critical Testing Process (CTP);
- Systematic Test and Evaluation Process (STEP);
- Test Maturity Model (TMM);
- Test Process Improvement (TPI).

Não entraremos em detalhes em relação aos modelos acima, deixando este assunto para ser abordado em um trabalho futuro.

Conclusões

Apesar de termos um processo que permite a organização e sequenciamento das atividades de teste de software, é muito importante investir na capacitação das pessoas que vão desempenhá-las. A literatura está repleta de técnicas e práticas para a realização de testes, mas é preciso conhecimento do assunto para escolher o que é melhor utilizar, levando em consideração o contexto.

A certeza de uma boa estratégia de testes é um bom ponto de partida, seguida de um projeto de testes com boa cobertura dos requisitos da aplicação e definição de bons cenários.

Embora o objetivo das atividades de testes seja encontrar defeitos no produto, não significa que a fase mais importante do processo inteiro seja a execução dos testes. Todas as fases do processo têm grande importância em um projeto.

Diversas iniciativas existem na literatura com a finalidade de propor melhorias em processos de teste, isso deve ser analisado para que as organizações possam realizar testes de maneira cada vez melhor.

Agradecimento

Gostaria de agradecer à Marcelle Frazão D. C. de Araújo e Mariana Pinto Xavier pela leitura e contribuições pertinentes fornecidas para a melhoria deste artigo.

Dê seu feedback sobre esta edição!

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

www.devmedia.com.br/esmag/feedback

Referências

Foundations of Software Testing: ISTQB Certification -Dorothy Graham, Erik van Veenendaal,Isabel Evans, Rex Black (2007)

The Art of software Testing. Glenford J. Myers (1979)

Test Process Improvement: A step-by-step guide to structured testing. Tim Koomen, M. Pol (1999).

Use of relative code churn measures to predict system defect density. Nachiappan Naggapan, Tomas Ball (2005).

Software Testing Foundations: A Study Guide for the Certified Tester Exam. Andreas Spillner, Tilo Linz, Hans Schaefer. (2007).

Introdução a Testes de Software. Arilo Dias-Neto. (2008). Artigo publicado na edição 1 da revista Engenharia de Software.

Cursos Online

Assinatura

Mais conteúdo .NET por muito menos!

A Revista **.net Magazine** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade .

Conheça abaixo os cursos já disponíveis:
www.devmedia.com.br/curso/netmagazine

- **Crie uma loja Virtual completa**
- **Construindo relatórios com Crystal Reports e Visual Studio 2005**
- **Criando uma aplicação Web Completa**
- **Criando uma aplicação client/server no Visual Studio 2005**
- **Aprenda a criar um blog com ASP.NET**

Curso de C# * Curso em andamento

A sua melhor opção de aprendizagem!

Assine a **.net Magazine** e
Comece já seu treinamento!
www.devmedia.com.br/assine

DevMedia GROUP

Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025

Manutenção: Desafios para os Testes numa Fábrica de Software

Dificuldades e sugestões para ganhar produtividade e qualidade nas atividades de testes em uma fábrica de software

Daniel Scaldaferri Lages

dlages@gmail.com

Possui MBA em Gerência de Projetos pela Fundação Getúlio Vargas, é pós-graduado em Gerência de T.I. pela Universidade FUMEC e Bacharel em Ciência da Computação pela UFMG. Atualmente é Coordenador da equipe de Quality Assurance da CPM Braxis, filial BH. Sua experiência profissional inclui o cargo de Analista de Testes no Synergia (núcleo de engenharia de software do Departamento de Ciência da Computação da UFMG), Gerente da fábrica de software na Unitech (hoje CPM Braxis) e docência no curso de graduação de Sistemas de Informação na PUC-MG. Possui a certificação ITIL para gerenciamento de serviços de T.I. É certificado em testes de software pela ISTQB e em qualidade de software pela IBM.

Se uma pesquisa fosse realizada com os desenvolvedores e testadores de uma Fábrica de Software sobre a preferência entre participar de novos projetos (aqueles iniciados "do zero"), ou projetos de manutenção, seria possível arriscar que a grande maioria ficaria com a primeira opção. Apesar da atividade de manutenção de um software ser uma tarefa tão desafiadora quanto a criação de um novo, para muitos, participar de um projeto desde o início é uma tarefa mais motivante e prazerosa. Muitas vezes mais simples, pois quem participa desde o início do projeto tem mais tempo para assimilá-lo, obtendo um grau satisfatório de domínio das suas regras e arquitetura, diferentemente de um sistema a ser mantido.

O maior "pesadelo" para as fábricas de software são os sistemas que surgem de maneira não planejada. Eles nascem da seguinte forma: um funcionário de uma empresa cliente que possui conhecimentos em informática ou em alguma linguagem de programação, com o intuito de automatizar um processo para agilizar seu trabalho ou do seu chefe, cria um "sisteminha". Uma planilha do excel com macros, ou uma simples página web, o que seja. Esse "sisteminha" é apresentado para o chefe, que o aprova e solicita novas funcionalidades ao longo

De que se trata o artigo?

Neste artigo veremos dicas e sugestões para superar os principais desafios e entraves associados às atividades de testes na manutenção de softwares em um ambiente de Fábrica de Software.

Para que serve?

As dicas e as sugestões apresentadas servem para tornar as atividades de testes de software mais produtivas dentro do processo de manutenção de software, visto que é esperado que uma Fábrica de Software realize o serviço com qualidade no menor tempo possível.

Em que situação o tema é útil?

Empresas e profissionais que prestam serviços de manutenção em diversos sistemas, assim como as empresas clientes, que tenham interesse em melhorar o processo de desenvolvimento e de testes de software, obtendo resultados em médio e longo prazo.

do tempo. Chega um determinado momento que o "sisteminha" torna-se um "sistemão", com informações importantes sobre aquela área de negócio. Então decide-se que o servidor, normalmente localizado debaixo da mesa do funcionário, deve ser passado para a área de TI da empresa, que imediatamente repassa para a fábrica. Sem documentação, sem padronização, sem boas práticas de programação etc. O "Frankstein" está ali, pronto para ser entendido e mantido.

A visão sobre a manutenção de software não é algo positivo. Esse olhar negativo dos envolvidos possui vários motivos comumente vividos pelas Fábricas de Software que oferecem esse tipo de serviço. A manutenção exige a análise, na maioria da vezes, de códigos mal escritos, sem comentários, despadronizados, ou seja, com baixa manutenibilidade. Mas isso que a torna uma tarefa desafiadora! Em um curto espaço de tempo (pois geralmente o prazo para manutenção é bem mais curto em relação ao desenvolvimento de novos sistemas, devido à urgência), a capacidade analítica e investigativa deve entrar em ação com rapidez e qualidade para satisfazer as necessidades do cliente. Isso serve tanto para desenvolvedores quanto para testadores. Mesmo quando o código possui uma boa qualidade, não é bem visto, pois foi criado por outras pessoas, que possuem diferentes formas de pensar e codificar. É natural do ser humano querer fazer do seu jeito, da maneira como aprendeu e sente maior confiança.

Este artigo, baseado na experiência do autor, relata brevemente as principais dificuldades encontradas na manutenção de um software na perspectiva dos testes, levando em consideração a qualidade e agilidade a que uma Fábrica de Software se proprõe. Para cada problema são sugeridas ações para minimizar essas dificuldades.

Fábrica de Software

Para que seja possível entender os problemas relacionados aos testes de software em manutenções, deve-se conhecer o ambiente onde os testes serão realizados. Segundo [HERBSLEB], as Fábricas de Software são definidas como organizações que fornecem serviços de desenvolvimento de sistemas com **qualidade**, a **baixo custo** e de **forma rápida**. Utilizando um processo de desenvolvimento de software bem definido e com apoio de tecnologias de mercado, além de reconhecer e lidar com oportunidades de melhoria do seu processo.

Através do conceito de [HERBSLEB], fica clara a imensa expectativa dos clientes quanto ao retorno na contratação de uma Fábrica de Software, principalmente quanto à agilidade e qualidade dos serviços. Desta forma, para satisfazer o cliente, o processo de manutenção deve ser certeiro, e estar preparado para enfrentar sistemas "Frankstein".

As Fábricas de Software não desenvolvem apenas novos sistemas. Muitas delas, pelo contrário, realizam manutenções como sua atividade principal. Alteram sistemas desenvolvidos por elas mesmas, mas também sistemas legados. O primeiro grupo apresenta certas vantagens para fábrica em relação ao segundo, pois como ela desenvolveu, teoricamente possui o conhecimento de todo o sistema. Conhecimento este presente na forma de documentações ou de capital intelectual da empresa. Sistema legado, conforme [WARD e BENNET] pode ser definido como aquele que executa tarefas úteis para a organização, mas que foi desenvolvido utilizando-se técnicas atualmente consideradas obsoletas. Segundo [SERRA], o sistema legado nos passa a impressão de serem antigos e ultrapassados, de não possuir especificação, modelagem de solução, documentação de regras de negócio, além do código ser despadronizado.

A manutenção de sistemas legados é uma atividade altamente desempenhada pelas Fábricas de Software. Muitas delas são responsáveis por manter uma quantidade enorme de sistemas, de diversas tecnologias e diferentes domínios de aplicação. Devido à expectativa e cobrança sobre os resultados de uma Fábrica de Software, juntamente com os problemas enfrentados na manutenção desses sistemas de baixa manutenibilidade, torna-se um desafio oferecer um serviço de qualidade.

Manutenção do Software

A vida de um software não termina após a sua implantação. Ele ainda viverá durante muito tempo. Será utilizado por anos, e com certeza, terá muitas atualizações, gerando novas versões do sistema. De acordo com [IEEE], a manutenção é caracterizada pela modificação do software já entregue ao cliente, ou seja, a manutenção é qualquer alteração no software após sua entrada em produção.

Conforme [SPILLNER, LINZ e SCHAEFER], o propósito da manutenção de software é diferente da manutenção de produtos industrializados, pois a manutenção realizada não é feita para reparar danos causados pelo tempo de uso do software. Defeitos não são introduzidos pelo tempo e nem pela carga de utilização. Os defeitos encontrados já existiam, antes do software entrar em produção. Por algum motivo, não foram detectados em fases anteriores. Mas a manutenção não se caracteriza apenas por correções. De acordo com [LIENTZ], existem três tipos principais de manutenções em softwares, Adaptativas, Corretivas e Evolutivas (Perfectivas):

- **Adaptativas:** são alterações que visam adaptar o software a uma nova realidade ou novo ambiente externo, normalmente imposto. Um exemplo claro seriam mudanças de leis ou regras, definidas pelo governo e/ou órgãos reguladores;
- **Corretivas:** como o próprio nome diz, servem para eliminar as falhas encontradas em produção. É bastante comum, principalmente quando o processo de desenvolvimento não se preocupou de maneira adequada com a qualidade do software;
- **Evolutivas:** são alterações que visam agregar novas funcionalidades e melhorias para os usuários que as solicitaram. Não se deve confundir esse tipo de manutenção com as entregas programadas de um processo de desenvolvimento interativo. A integração com outros sistemas também é considerada um tipo de evolução.

As migrações para novas tecnologias, sejam de hardware ou software, podem causar dúvidas quanto à classificação do tipo de manutenção. Se a tecnologia foi imposta, a manutenção é classificada como adaptativa. Se a tecnologia é uma solução de

melhoria do sistema, é classificada como evolutiva. [PAULA FILHO] ainda cita mais um tipo de manutenção, a chamada preventiva. Esta procura localizar os defeitos antes que se manifestem em operação. Entretanto, é raramente praticada.

A proporção do esforço de manutenção, segundo [LIENTZ], após estudos com 487 organizações da área, é de 20% para corretivas, 25% adaptativas, 50% evolutivas e apenas 5% para preventivas.

A seguir serão apresentadas situações que dificultam os testes após as manutenções nos softwares terem sido realizadas.

Frequência das Manutenções

Este é um problema que ocorre principalmente em Fábricas de Software de grande porte, que possuem vários clientes, e que são responsáveis por vários sistemas distintos, implementados em diversas tecnologias. É normal que alguns sistemas sejam mais alterados que outros. Essa diferença impacta no desenvolvimento e nos testes do sistema.

O problema da frequência de manutenções realizadas pode ser percebida claramente quando se compara a qualidade do serviço de manutenção entre sistemas com alta e baixa frequências de alterações. Quando a frequência é alta, normalmente a equipe responsável pelo sistema é maior e mais capacitada. A Fábrica de Software identifica e se prepara para atendê-lo da melhor forma, logicamente. Com uma equipe grande, maior o número de responsáveis pelo sucesso e, o que é mais importante, o conhecimento é melhor disseminado. Dessa forma, os testes de software são mais fáceis, pois existe um conhecimento e um suporte (dos desenvolvedores aos testadores) adequado sobre o sistema. A prática e o conhecimento sobre o sistema se tornam sempre "recentes".

Mas quando a frequência é baixa, ou muito baixa, torna-se um grande problema, indo contra as características do serviço prestado em sistemas com alta frequência de manutenção. Como por exemplo, o conhecimento deixa a desejar, pois como não existe a prática, ele se perde ao longo do tempo. Também não existe um "guru" do sistema. O profissional que está disponível naquele momento é que será o responsável pelo serviço de manutenção, mesmo que não domine a tecnologia do sistema. Os testes são difíceis, pois não se tem conhecimento amplo do sistema, tornado-os pouco efetivos.

Esse problema é agravado pelo fator "rotatividade" dentro de uma empresa. A mesma pesquisa realizada por [LIENTZ], citada anteriormente, diz que quanto mais antigo o sistema, maior é o esforço que deve ser empregado em sua manutenção. A fábrica, também, torna-se mais sujeita a perder o conhecimento em torno deste sistema, devido à rotatividade dos profissionais. Isto é, como a frequência de manuteção é baixa, o profissional que realizou a última manutenção no sistema pode já não estar mais na equipe. Consequentemente, caso ações para reter o conhecimento não tenham sido tomadas, este conhecimento também não estará mais na empresa.

Uma das soluções, que não trará resultados imediatos, mas a médio e longo prazo, e manterá o conhecimento dentro da empresa é a elaboração e o reuso dos casos de testes. Tendo como premissa que uma fábrica preza pela qualidade dos serviços, especifica os casos de testes referentes à alteração realizada como parte integrante do seu processo, a preocupação recai sobre uma forma fácil de reusar os casos de testes.

O reuso dos casos de testes tornam a execução dos testes mais ágeis, pois evitam a elaboração dos casos de testes novamente, e já traz consigo o conhecimento daquela funcionalidade do sistema. Mas para conseguir esse reuso, os testadores devem armazená-los de forma adequada, para que a busca seja fácil quando forem solicitados os testes de um sistema com baixa frequência de manutenção. Outro ponto de atenção é sempre atualizar os casos de testes, pois, se desatualizados, podem atrapalhar, ao invés de ajudar.

O ideal é utilizar alguma ferramenta que possa servir como um repositório centralizado de casos de testes. A utilização de planilhas não é indicada, pois com o tempo, elas se espalham dentro do sistema de arquivos da fábrica, além de proporcionar possibilidade de redundâncias. A ferramenta Testlink, apresentada na edição número 3 desta revista – Ferramentas open source e melhores práticas na gestão de testes – pode ser utilizada para realizar o papel deste repositório.

Ao longo do tempo, os casos de testes tornam-se uma especificação do sistema para os testadores, uma vez que uma especificação de testes possui um conjunto de entradas, condições de execução e um critério de sucesso/falha, segundo [PEZZÈ e YOUNG]. Com essas informações, os testadores conseguem claramente entender o sistema e os testes a serem realizados.

A situação ideal seria que a fábrica realizasse um trabalho completo de especificação de casos de testes, para quando uma alteração fosse realizada em um sistemaa com baixa frequência de manutenção, os testes se tornassem uma tarefa mais ágil. Mas as fábricas, devido ao cenário apresentado que exige agilidade e qualidade a baixo custo, não acham a idéia compensadora, uma vez que não possuem garantia de que o sistema, ao menos, sofrerá alguma outra manutenção. Logicamente, para os sistemas com alta frequência, essa ação é um investimento com retorno garantido.

Efeitos colaterais das Manutenções

Quando se altera o software, seja para correção, evolução ou adaptação, existe a possibilidade de se introduzir novos defeitos ou reintroduzir erros que ocorriam anteriormente no mesmo. É o chamado efeito colateral. O ideal seria que todos os módulos ligados à parte alterada ou o sistema todo fossem retestados. Mas devido aos curtos prazos, principalmente em manutenções corretivas emergenciais, quase nunca isso é possível. Segundo [SPILLNER, LINZ e SCHAEFER], a falha causada através do efeito colateral pode aparecer em qualquer ponto do software, podendo ficar fora da cobertura dos testes daquela manutenção.

Isto é um desafio para os testadores, pois eles precisam garantir o correto funcionamento do sistema como um todo. Uma forma de prevenir este efeito seria a realização dos testes de regressão, que conforme [SPILLNER, LINZ e SCHAEFER], são realizados para garantir que os novos defeitos introduzidos ou desmascarados sejam encontrados e removidos.

Para atender aos curtos prazos, os testes de regressão devem ser automatizados. Realizá-los manualmente não seria viável. Entretanto, conforme [FEWSTER], automatizar é uma tarefa complexa e custosa, que exige grande esforço. Mas, uma vez automatizado, o teste de regressão torna-se muito mais econômico, correspondendo apenas a uma fração do tempo gasto caso fosse realizado manualmente. Uma sugestão para diluir o alto custo da automação dos testes de regressão, seria a automação de partes do sistema a cada nova demanda de manutenção, alcançando a situação ideal no médio ou longo prazo.

De acordo com [PEZZÈ e YOUNG], a automação é particularmente importante para obter um nível razoável de garantia em um ciclo de desenvolvimento muito rápido, como por exemplo, o ciclo das manutenções corretivas. Uma vez atingido esse cenário ideal, os testes serão rápidos e eficazes, mitigando os riscos das falhas causadas pelo efeito colateral.

Escopo dos Testes na Manutenção

Quando um software é desenvolvido do zero, ou seja, quando ele é totalmente novo, as atividades de testes tornam-se tarefas mais claras e objetivas. Muitas vezes mais rápidas de serem concluídas em comparação às mesmas atividades em um sistema que está passando por uma grande manutenção, seja ela do tipo adaptativa ou evolutiva, principalmente. Esta clareza e objetividade dos testes diz respeito ao seu escopo. Isto ocorre pelo simples fato de que se o sistema é novo, tudo tem que ser testado.

O mesmo não acontece para as manutenções de software. O escopo dos testes é restrito às funcionalidades que foram alteradas. Mas ao mesmo tempo, a fábrica deve garantir que tudo que funcionava antes continua funcionando após a manutenção. A identificação do escopo dos testes não é uma tarefa tão simples. Perguntas, como por exemplo, se a falha encontrada está ou não dentro do escopo daquele serviço de manutenção são frequentes. É comum nos testes de manutenções encontrar falhas em outras partes do sistema que não tenham sido alteradas. Por mais que o testador focalize no escopo da demanda, ele se depara com falhas no sistema, por exemplo, no caminho para alcançar a funcionalidade a ser testada. Logicamente, o bom testador não deixará de registrá-la.

Outras perguntas também são realizadas. Se a falha estiver fora do escopo, quem foi o responsável por introduzir o defeito? A fábrica atual ou a anterior? Sendo a anterior responsável, logicamente, a fábrica da vez não concordará com o ônus da correção da falha, e irá solicitar nova solicitação de manutenção para essa correção, pois irá querer receber por isso, uma vez que este é o seu negócio. Como os sistemas legados são antigos, podem ser mantidos por várias fábricas distintas ao longo do tempo e esse tipo de situação é inevitável de acontecer.

Sendo a fábrica atual a responsável, outra pergunta surge: O defeito está ou não dentro do período de garantia dos prévios serviços? Se estiver dentro do prazo, a correção será efetuada. Caso contrário, qual será a estratégia? Agregar valor ao serviço de manutenção e agradar o cliente corrigindo a falha, ou considerar uma oportunidade para uma nova demanda de manutenção?

No momento da execução dos testes, essas dúvidas surgem frequentemente entre testadores e desenvolvedores. Mas independentemene dessas questões, os testadores deverão registrar as falhas encontradas. O que deve ser pensado é na maneira de organizar e classificar esses tipos de falhas para auxiliar os desenvolvedores e gerentes de projetos a priorizarem as atividades de correção. É possível ocorrer os seguintes tipos de falhas:

- Falhas dentro do escopo da demanda;
- Falhas fora do escopo da demanda, de responsabilidade da fábrica, fora da garantia;
- Falhas fora do escopo da demanda, de responsabilidade da fábrica, dentro da garantia;
- Falhas fora do escopo da demanda, de responsabiliade de alguma fábrica anterior.

No momento dos testes é possível classificar se a falha está dentro ou fora do escopo da demanda, pois o testador tem idéia do que foi alterado e do que deverá ser testado, pois ele segue um plano de testes e uma especificação de requisitos. Mas algumas vezes surgem dúvidas sobre tal classificação, pois em alguns casos, em se tratando de manutenção, a fronteira do que está dentro ou fora do escopo é tênue. Quanto à responsabilidade da introdução do defeito e à garantia do serviço, é melhor que os líderes e gestores do serviço decidam. Não é responsabilidade técnica dos analistas de testes, e sim uma decisão gerencial e contratual.

Através das ferramentas de Gestão de Incidentes, como o Mantis e o Bugzilla, é possível registrar se a falha encontrada está ou não dentro do escopo daquele serviço. Essas ferramentas são customizáveis, permitindo a criação de campos. Dessa forma, caso o testador não saiba identificar se a falha está no escopo ou não, pode-se registrar "escopo a ser verificado", por exemplo. Sendo assim, é possível ajudar os desenvolvedores a priorizar a correção.

Uma forma da fábrica se resguardar quanto às questões da cobertura da garantia e da origem das falhas é através da utilização de um processo de controle de versão. Existem algumas boas ferramentas gratuitas que podem ser adotadas, e que satisfazem às necessidades, como exemplos, CVS e Subversion. A informação histórica mantida nestes tipos de aplicativos é útil para rastrear falhas ao longo de versões, segundo [PEZZÈ e YOUNG]. Eles proporcionam um controle apurado das alterações, indicando os autores, as datas e os motivos das manutenções realizadas.

É comum existirem manutenções onde o escopo é a migração da tecnologia do sistema, como por exemplo, a liguagem de programação. Quando o sistema implementado na nova tecnologia é testado e encontram-se falhas básicas, essas devem ser registradas. Não se deve considerar que, como a falha já ocorria no sistema antigo, ela não é problema. Este "nivelamento por baixo" de qualidade em relação à omissão de falhas encontradas por não serem de responsabilidade da fábrica atual, não pode acontecer. O mínimo que deve ser feito é registrá-las como falhas fora do escopo ou até mesmo como sugestões de melhorias.

Ambientes de Testes

A quantidade de sistemas e a diversificação das tecnologias utilizadas também é um desafio para as fábricas se a questão do ambiente de testes não for muito bem planejada. Essa diversidade, aliada à frequência de manutenção, pode tornar-se um gargalo no momento da execução dos testes. Manter o ambiente de um sistema com baixa frequência de manutenção é um desperdício de infra-estrutura. É caro manter um servidor preparado com um ambiente que será pouco utilizado. Ao mesmo tempo, montar um ambiente para os testes não é uma tarefa rápida que pode ser deixada para o último momento. Planejar a preparação do ambiente antes do momento da execução dos testes é um fator importante para evitar atrasos.

Solicitar o dump da base de dados para o cliente é um procedimento comum. Mas, comum também, pode ser a demora do retorno a essa solicitação. Algumas vezes o cliente não atende ao pedido, pois os dados são considerados confidenciais. Existem boas ferramentas para geração da massa de dados com o custo acessível. Com esse tipo de ferramenta é possível ganhar agilidade na geração da massa de dados, evitando solicitações aos clientes. Apesar disso, a massa de dados reais extraídas do ambiente de produção é bem mais "rica" e, portanto, mais benéfica aos testes. Os scripts de geração da massa de dados podem ser armazenados e reutilizados, ganhando produtividade. Esses artefatos, logicamente, devem fazer parte da baseline dos artefatos, juntamente com os planos e casos de testes, e devem ser administrados através de ferramentas de controle de versão.

Falta de Documentação

A falta de documentação é um problema que afeta não só as atividades de testes, mas todas as demais tarefas de desenvolvimento, como exemplo, a implementação. A ferramenta Rational Requisite Pro é bastante interessante no caso de manutenções. Ela funciona como um repositório de requisitos e casos de uso. Ao longo das manutenções, esses itens sofrem alterações. Através da ferramenta, a tarefa de encontrar e realizar as alterações torna-se bem produtiva. Com a utilização do SoDA, ferramenta responsável por extrair os dados do Requisite Pro, os documentos são gerados seguindo um template padrão, que pode ser customizável, agregando qualidade às documentações.

De forma semelhante ao uso do Testlink para casos de testes, no médio e longo prazo, o sistema sem documentação, após o povoamento do Requisito Pro com seus casos de uso, possuirá informações suficientes para atender o cliente com agilidade. Para as atividades de testes a utilização dessa ferramenta também é muito bem vinda, visto que as informações do Requisite Pro que servem de base para a elaboração dos casos de testes estarão padronizadas. Além disso, fica simples rastrear casos de testes para os casos de usos e requisitos.

Conclusão

Seguir um processo de testes em pequenas manutenções pode parecer, a princípio, uma tarefa burocrática e anti produtiva, principalmente em razão das dificuldades encontradas somadas à forte expectativa de retorno de uma fábrica de software em relação aos prazos e à qualidade. Não é aconselhável que as organizações abram mão dos testes dessas manutenções para conseguir cumprir o prazo de entrega. O tempo ganho ignorando as atividades de testes pode se reverter em grandes problemas após a alteração realizada entrar em produção, tornando o custo da manutenção bem maior, conforme [SPILLNER, LINZ e SCHAEFER]. Este é um princípio básico dos testes de software.

A centralização e o reuso dos artefatos são as palavras-chaves para ganhar agilidade nas atividades de testes dentro da manutenção de um sistema, assim como manter o conhecimento dentro da Fábrica de Software. Foram apresentados exemplos de ferramentas, como o Teslink e o Requisito Pro, capazes de centralizar e suportar o reuso dos artefatos de testes e de especificação, respectivamente, tornando a manutenção mais ágil.

Este artigo apresentou algumas questões que dificultam as atividades da equipe de testes, principalmente em relação à execução dos testes. As dicas e soluções propostas estão longe de resolverem os problemas em um curto espaço de tempo, mas são ações que irão trazer, no médio e longo prazo, agilidade aos serviços. Após este tempo, a fábrica estará bem mais preparada, alcançando a capacidade necessária para atender às expectativas dos seus clientes. ●

Links

Testlink: www.teamst.org/

Mantis: www.mantisbt.org/

Bugzilla: www.bugzilla.org/

CVS: www.nongnu.org/cvs/

Subversion: subversion.tigris.org/

Requisite Pro: www-01.ibm.com/software/awdtools/reqpro/

SoDA: www-01.ibm.com/software/awdtools/soda/index.html

Refer"encias

FEWSTER, Mark, "Common Mistakes in Test Automation". Grove Consultants, – Llandeilo UK, 2001.

HERBSLEB, J. D., Grinter, R. E. (1999). "Splitting the Organization and Integrating the Code: Conway's Law Revisited". In: Proceedings of ICSE. Los Angeles: IEEECSP, 1999. p. 85–95.

IEEE (1998) "Std 1219 – IEEE Standard for Software Maintenance" Institute of Electrical and Eletronic Engineers, New York, NY, USA.

LIENTZ, B.P; Swanson, E.B. (1980) "Software Maintenance Management", Reading, MA, Addison Wesley.

PAULA FILHO, Wilson de Pádua. Engenharia de Software: fundamentos, métodos e padrões. 2a ed. Rio de Janeiro: LTC - Livros Técnicos Científicos. 2003.

PEZZÈ, Mauro; YOUNG, Michal, Teste e Análise de Software. Edição Traduzida. Porto Alegre: Bookman 2008.

SERRA, Ana Paula Gonçalves. Reengenharia de Software – Uma visão Geral. Engenharia de Software Magazine, Número 11, 2009.

SPILLNER, Andreas; LINZ, Tilo; SCHAEFER, Hans; Software Testing Foundations – A Study Guide for the Certified Tester Exam. Rio de Janeiro: Fundação Getúlio Vargas, 2006.

WARD, M.P; BENNETT, K.H. Formal Methods for Legacy Systems. Journal of Software Maintenance: Research and Practice, v.7 n.3, 1995

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

Cursos Online

Assinatura
ClubeDelphi PLUS
Mais conteúdo DELPHI por muito menos!

A Revista **ClubeDelphi** oferece para seus assinantes uma série de **Cursos Online** de alto padrão de qualidade . Conheça abaixo os cursos já disponíveis..

Curso em destaque

Aplicações Client/Server com dbExpress e Firebird

Confira neste curso online como criar uma aplicação client/server completa no Delphi 7, utilizando dbExpress e Firebird.
Aprenda também: Como trabalhar com um driver dbExpress específico e gratuito para o Firebird ; Como utilizar Orientação a Objetos, e técnicas como herança visual de formulários e relatórios ; saiba como colocar regras de negócios no banco de dados, usando Triggers e Stored Procedures; E crie relatórios com o Quick Report, Rave Reports e Report Builder, e muito mais.

Confira o plano de aula completo:
www.devmedia.com.br/clienteserver

A sua melhor opção de aprendizagem!
Assine a **ClubeDeplhi** e comece já seu treinamento!
www.devmedia.com.br/assine

Outros cursos disponíveis: **www.devmedia.com.br/curso**

- Curso Online - Sistema SysCash
- Aplicações client/server com Windows Forms no Delphi 2006
- Criando uma Aplicação multi-camadas Completa com Delphi
- Aplicações WEB com Delphi 7 (Delphi Win32)

DevMedia GROUP **Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025**

Plano de Teste

Um 'Mapa' Essencial para Teste de Software

De que se trata o artigo?

Apresenta o plano de teste de software, destacando sua importância no processo de desenvolvimento de software, mostrando como elaborá-lo e exemplificando os itens que devem compor o referido documento.

Para que serve?

Orientar o gerente de projeto e/ou líder da equipe de teste na elaboração de um plano de teste para um sistema de software.

Em que situação o tema é útil?

Trata-se de uma prática regular na engenharia de software e ajuda na elaboração de um plano de teste, na definição dos itens que devem compor esse plano e justifica sua necessidade num processo de desenvolvimento de software.

Antonio Mendes da Silva Filho

antoniom.silvafilho@gmail.com

Professor e consultor em área de tecnologia da informação e comunicação com mais de 20 anos de experiência profissional, é autor do livros Arquitetura de Software e Programando com XML, ambos pela Editora Campus/Elsevier, possui diversos artigos publicados em eventos nacionais e internacionais, colunista para Ciência e Tecnologia pela Revista Espaço Acadêmico com mais de 90 artigos publicados, tendo feitos palestras em eventos nacionais e exterior. Foi Professor Visitante da University of Texas at Dallas e da University of Ottawa. Formado em Engenharia Elétrica pela Universidade de Pernambuco, com Mestrado em Engenharia Elétrica pela Universidade Federal da Paraíba (Campina Grande), Mestrado em Engenharia da Computação pela University of Waterloo e Doutor em Ciência da Computação pela Univesidade Federal de Pernambuco.

Uma atividade essencial no desenvolvimento de todo e qualquer projeto é o planejamento. Um plano tem o papel semelhante ao de um 'mapa'. Sem um mapa, um plano ou qualquer outra fonte de informação similar, você não conhecerá seus objetivos, nem aonde quer chegar e jamais terá a certeza de ter alcançado sua meta. Perceba que entender o propósito do planejamento é de suma importância a fim de monitorar a execução de atividades, sendo também importante conhecer o papel dos riscos no planejamento, bem como diferenciar estratégias de planos. Planejamento engloba três atividades principais:

1. Definir um cronograma de atividades: estabelecer as atividades que devem ser realizadas, as etapas a serem seguidas e a ordem cronológica de execução;
2. Fazer alocação de recursos: definir quem realiza as atividades e quais ferramentas/recursos a serem utilizados;
3. Definir marcos de projeto – estabelecer os marcos, ou milestones, a serem alcançados com objetivo de se fazer o acompanhamento.

Perceba que o planejamento é acompanhado da atividade de monitoração ou supervisão que visa avaliar se o progresso que tem sido alcançado está em conformidade com o que foi estabelecido no plano ou, em outras palavras, responder a questão: quão bem estamos indo no projeto?

Agora, dentro do contexto do desenvolvimento de software, você necessitará de vários documentos como, por exemplo, plano de projeto, documento de requisitos e plano de teste. Neste artigo, o foco recai sobre o último, isto é, plano de teste. Trata-se de um documento ou mapa no qual se definem escopo e objetivos, além de requisitos, estratégias e recursos a serem empregados nas atividades de testes de software. Nesse sentido, o artigo apresenta os itens que devem fazer parte de um documento de plano de teste, exemplificando e discutindo esses itens.

Teste de Software

Teste de software é uma das atividades do processo de desenvolvimento de sistema de software que visa executar um programa de modo sistemático com o objetivo de encontrar falhas. Perceba que isto requer verificação e validação de software. Nesse sentido, definir quando as atividades de verificação e validação iniciam e terminam, como os atributos de qualidade serão avaliados e como os releases do software serão controlados, são questões que devem ser acompanhadas ao longo do processo de software.

Vale ressaltar que teste não deve ser a última atividade do processo de desenvolvimento de software. Ela ocorre durante todo o processo, como exemplificado na visão geral do processo RUP (Rational Unified Process) mostrado na **Figura 1**.

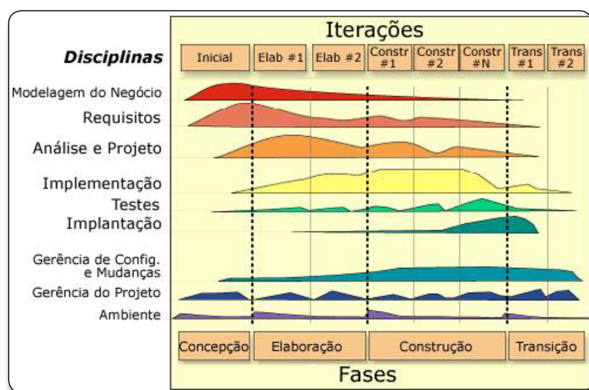


Figura 1. Visão geral do RUP.

E, além de encontrar falhas, testes objetivam aumentar a confiabilidade de um sistema de software, isto é, aumentar a probabilidade de que um sistema continuará funcionando sem falhas durante um período de tempo.

Embora seja desejável testar um sistema por completo, deve-se ter em mente que a atividade de teste assegura apenas encontrar falhas se ela(s) existirem, mas não asseguram sua ausência. Portanto, as atividades devem ser disciplinadas a fim de identificar a maioria dos erros existentes. Note que realizar os testes de software implica em responder às questões:

1. Quais atributos da qualidade deverão ser testados?
2. Quem realizará os testes?
3. Quais recursos serão utilizados?
4. Quais as dependências entre os atributos de qualidade?
5. Quais as dependências entre as atividades de desenvolvimento?
6. Como o processo e a qualidade do sistema de software serão acompanhados?

Na seção seguinte, um exemplo do conjunto de seções de um plano de teste é apresentado com o objetivo de ilustrar como as informações pertinentes ao teste de software poderiam ser tratadas e documentas. Não há, portanto, o objetivo de ser completo, pois cada sistema possui suas peculiaridades que devem ser consideradas caso a caso.

Plano de Teste

O plano de teste é um dos documentos produzidos na condução de um projeto. Ele funciona como:

- Um 'integrador' entre diversas atividades de testes no projeto;
- Mecanismo de comunicação para os stakeholders (i.e. a equipe de testes e outros interessados);
- Guia para execução e controle das atividades de testes.

O plano de teste, que pode ser elaborado pelo gerente de projeto ou gerente de testes, visa planejar as atividades a serem realizadas, definir os métodos a serem empregados, planejar a capacidade necessária, estabelecer métricas e formas de acompanhamento do processo. Nesse sentido, deve conter:

- Introdução com identificação do projeto (definições, abreviações, referências), definição de escopo e objetivos;
- Conjunto de requisitos a serem testados;
- Tipos de testes a serem realizados e ferramentas utilizadas;
- Recursos utilizados nos testes;
- Cronograma de atividades (e definição de marcos de projeto).

Em outras palavras, um plano de teste deve definir:

1. Os itens a serem testados: o escopo e objetivos do plano devem ser estabelecidos no início do projeto.
2. Atividades e recursos a serem empregados: as estratégias de testes e recursos utilizados devem ser definidos, bem como toda e qualquer restrição imposta sobre as atividades e/ou recursos.
3. Os tipos de testes a serem realizados e ferramentas empregadas: os tipos de testes e a ordem cronológica de sua ocorrência são estabelecidos no plano.
4. Critérios para avaliar os resultados obtidos: métricas devem ser definidas para acompanhar os resultados alcançados.

Perceba que o planejamento é necessário a fim de antecipar o que pode ocorrer e, portanto, provisionar os recursos necessários nos momentos adequados. Isto significa coordenar o processo de teste de modo a perseguir a meta de qualidade do produto (sistema de software).

Exemplificando o Plano de Teste

O plano de teste contém um conjunto de informações que permite ao gerente de projeto não apenas coordenar as atividades de testes de um projeto, mas também monitorar seu progresso e verificar se o executado está em conformidade com o planejado. A Tabela 1 apresenta uma relação dos itens consideradas imprescindíveis em um plano de teste. A relação de itens não pressupõe a intenção de ser completo, mas de apontar os itens considerados como obrigatórios num plano de teste de uma instituição.

Itens de um Plano de Teste	Conteúdo
1. Introdução	Contém uma identificação do projeto, descrição dos objetivos do documento, o público ao qual ele se destina e escopo do projeto a ser desenvolvido. Pode adicionalmente conter termos e abreviações usadas, além de informar como o plano deve evoluir.
2. Requisitos a serem testados	Esta seção descreve em linhas gerais o conjunto de requisitos a serem testados no projeto a ser desenvolvido, comunicando o que deve ser verificado. Exemplos de requisitos a serem testados são: Esta desempenho, segurança, interface de usuário, controle de acesso, funcionalidades.
3. Estratégias e ferramentas de teste	Apresenta um conjunto de tipos de testes a serem realizados, respectivas técnicas empregadas e critério de finalização de teste. Além disso, é listado o conjunto de ferramentas utilizadas.
4. Equipe e infra-estrutura	Contém descrição da equipe e da infra-estrutura utilizada para o desenvolvimento das atividades de testes, incluindo: pessoal, equipamentos, software de apoio, materiais, dentre outros. Isto visa garantir uma estrutura adequada para a execução das atividades de testes previstas no plano.
5. Cronograma de atividades	Contém uma descrição de marcos importantes (milestones) das atividades (incluindo as datas de início e fim da atividade). Apenas marcos relevantes devem ser listados, ou seja, aqueles que contribuirão nas atividades de testes. Por exemplo: projeto de testes, execução de testes ou avaliação de testes.
6. Documentação complementar	Apresenta-se uma relação dos documentos pertinentes ao projeto.

Tabela 1 – Relação de itens de um plano de Teste.

O conteúdo exato das seções que compõem um plano de teste, geralmente, difere de instituição para instituição. Entretanto, os itens acima apontados existem nas seções do documento. A subseções, destacados nos quadros a seguir (enumerados de 1 a 6), ilustram o conteúdo que compõe um plano de teste. O **Quadro 1** destaca a seção 1 do plano de teste.

O **Quadro 2** representa a seção 2 do documento e apresenta um conjunto de requisitos a serem testados.

Perceba que os objetivos do projeto são peculiares a cada um. Além disso, os critérios de aceitação final do projeto é resultado de acordo entre as parte envolvidas (i.e. empresa desenvolvedora e cliente). O **Quadro 3** caracteriza a seção dos requisitos do sistema, que apresenta uma visão geral do projeto trazendo objetivos, participantes e mecanismos de evolução do plano de teste e aceitação.

A motivação da seção anterior é caracterizar os principais testes a serem aplicados às funcionalidades do sistema e possíveis ferramentas a serem usadas. A seção representada no **Quadro 4** apresenta a equipe de teste com respectivas funções e infra-estrutura utilizada.

Neste ponto, pode-se definir o cronograma (ver **Quadro 5**).

Perceba que o cronograma apresentado na seção 5 destaca apenas as principais atividades, caracterizando os principais marcos do projeto. Não há, contudo, a intenção aqui em ser completo, mas a de ressaltar como as informações podem ser apresentadas no plano de teste.

1. Introdução

Este documento apresenta o planejamento das atividades de testes do sistema Exemplo o qual será utilizado como base para as atividades de acompanhamento, revisão, verificação e validação do projeto, desde seu início até sua conclusão, a fim de garantir a análise comparativa do resultado real versus planejado. Desta forma, ações corretivas e preventivas poderão ser tomadas, sempre que resultados reais desviarem significativamente do planejado.

1.1 Termos e acrônimos

Esta seção explica o conceito de um subconjunto de termos importantes que serão mencionados no decorrer deste documento. Estes termos são descritos na **Tabela 2**, estando apresentados por ordem alfabética.

Termo	Descrição
Artefato	Tudo que é produzido e documentado em uma atividade de qualquer fluxo do projeto. Por exemplo: documento de requisitos, diagrama de casos de usos e glossário.
Milestone	Ponto de checagem; marco que indica a conclusão de uma fase ou etapa.
NA	Não Aplicável
Patrocinador	Representante da empresa cliente ou contratada responsável pelo sucesso do projeto em instância superior, garantindo o cumprimento de responsabilidades estabelecidas.
Revisão	Apresentação de produtos de software para os interessados visando comentário e aprovação dos mesmos.
SQA	Software Quality Assurance, profissional ou grupo responsável por garantir a qualidade do produto de software e processo de desenvolvimento.

Tabela 2 – Termos e acrônimos do projeto.

Note que a **Tabela 2** identifica um subconjunto de termos que pode caracterizar um projeto. Todo e qualquer termo, convenção adotada ou abreviações deveriam ser apresentadas nessa tabela a fim de comunicar às partes envolvidas e interessadas (i.e. os stakeholders) o seu significado. Isto visa prover as denominações corretas empregadas no documento.

1.2 Objetivos

Esta seção contém o conjunto de objetivos que orientam as atividades de testes do sistema a ser desenvolvido. Esse documento do Plano de Testes do sistema Exemplo possui os seguintes objetivos:

- Levantar as informações de projeto pertinentes e os componentes de software a serem testados.
- Definir o conjunto de requisitos a serem testados (alto nível).
- Definir e detalhar as estratégias de teste a serem utilizadas.
- Definir os recursos necessários e obter uma estimativa dos esforços das atividades de teste.
- Identificar os artefatos resultantes das atividades de testes.

1.3 Sistema Exemplo

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

(Nota que o propósito desse sistema, usado aqui apenas com fins ilustrativos, é similar ao de um sistema como o de portais de jornais e revistas e outros provedores de conteúdo que permitem o acesso ao conteúdo apenas a clientes devidamente cadastrados no sistema).

1.4 Escopo

Este projeto aborda um sistema Exemplo de informação online, com foco em prover conteúdo online. Este sistema Exemplo necessitará fazer testes unitários, de integração e de sistema. Os

Quadro 1. Introdução

testes unitários e de integração visam tratar a qualidade funcional, a interface gráfica e controle de acesso. Por outro lado, os testes de sistema tratarão as questões de desempenho.

Para a execução dos testes serão utilizadas máquinas o mais idênticas possível, em termos de hardware, àquelas que serão implantadas no cliente (provedor de conteúdo online, como um portal de notícias), a fim de garantir a previsibilidade de desempenho e compatibilidade.

Outros testes como os testes de stress, de volume e de falha/recuperação não serão realizados por se considerar que o ambiente de implantação do sistema nao está sujeito a esse tipo de ocorrência, as quais podem ser facilmente previstas e tratadas pelo cliente.

1.5 Documentação do projeto

Esta seção contém informações sobre o conjunto de documentos disponíveis e respectiva situação de revisão, os quais são importantes no plano de teste. O quadro abaixo ilustra uma situação exemplo.

Documento	Disponível	Revisado
Especificação de Requisitos	■ Sim □ Não	■ Sim □ Não
Plano de Projeto	■ Sim □ Não	■ Sim □ Não
Modelo de Análise	■ Sim □ Não	■ Sim □ Não
Modelo de Projeto	■ Sim □ Não	□ Sim ■ Não
Documento de Arquitetura	□ Sim ■ Não	□ Sim ■ Não
Protótipo	□ Sim ■ Não	□ Sim ■ Não
Manual do Usuário	□ Sim ■ Não	□ Sim ■ Não
Lista de Riscos	■ Sim □ Não	■ Sim □ Não

Continuação Quadro 1. Introdução

2. Requisitos a serem testados

Esta seção apresenta um conjunto de requisitos funcionais e não funcionais que foram identificados durante o levantamento de requisitos e para os quais os testes abaixo são considerados como necessários (representam um extrato do que será testado).

Teste da Interface do Usuário

a) Navegue através de todas as funcionalidades, verificando que cada tela de interface gráfica pode ser rapidamente entendida e facilmente utilizada.

b) Verifique que se a ajuda online está funcionando adequadamente.

Teste Funcional

a) Verifique se as informações úteis obtidas pelo subsistema responsável são automaticamente e periodicamente atualizadas.

b) Verifique se qualquer usuário pode acessar as notícias e outras informações disponíveis no portal.

Teste de Desempenho

a) Verifique o tempo de resposta da rede interna e do servidor em relação aos terminais.

b) Verifique o tempo de consulta/atualização do subsistema de notícias e outras informações.

c) Verifique se o tempo de resposta para operações que envolvam dados multimídia (imagens, vídeos, etc.) não ultrapassam 15 segundos.

Teste do Banco de Dados

a) Verifique se as informações de conteúdos, notícias, categorias e demais informações podem ser inseridos, atualizados e consultados pelo administrador de sistema.

b) Verifique se as informações úteis obtidas pelo subsistema responsável podem ser atualizadas e que as mesmas podem ser apresentadas.

c) Verifique se as informações do portal de notícias possam ser consultadas pelos usuários.

d) Verifique se as informações úteis disponíveis no portal possam ser consultadas.

(Note que um sistema possui diversos requisitos que podem ser verificados através de outros tipos de testes como teste de carga, teste de instalação, teste de volume, teste de stress, teste de configuração. Por exemplo, num teste de segurança e de controle de acesso, você pode verificar se usuários não cadastrados podem acessar informações restritas aos cadastrados. Adicionalmente, você pode checar se, além do administrador, ninguém mais pode inserir, atualizar ou remover dados do sistema).

Quadro 2. Requisitos a serem testados

3. Estratégias e ferramentas de teste

Esta seção apresenta um conjunto de tipos de testes a serem realizados onde, para cada tipo de teste adotado, definem-se as técnicas utilizadas, o critério de finalização de teste e outras suposições ou considerações especificas para o teste. Adicionalmente, informa-se o conjunto de ferramentas empregadas.

Estratégias

As estratégias compreendem um conjunto de testes empregados com o objetivo de verificar aspectos específicos do sistema em desenvolvimento (abaixo, apresenta-se um extrato de possíveis testes que podem ser adotados).

Teste funcional (ver Tabela 3)

Objetivo do Teste:	Assegurar a funcionalidade adequada do teste, incluindo navegação, entrada de dados, processamento e recuperação.
Técnica:	Executar cada caso de uso e fluxo de caso de uso, usando dados válidos e inválidos a fim de verificar: • Se os resultados esperados ocorrem quando dados válidos são usados • Se mensagens de erro ou aviso apropriadas são exibidas quando dados inválidos são usados. • Se cada regra de negócio está sendo aplicada de maneira apropriada
Critério de Finalização:	Todos os testes planejados foram executados. Todos os defeitos identificados foram tratados.
Considerações Especiais:	Nenhuma

Tabela 3. Teste Funcional

Teste de interface de usuário (ver Tabela 4)

Objetivo do Teste:	Verificar se a navegação através das funcionalidades testadas refletem as funções e os requisitos do negócio apropriadamente, incluindo janela-a-janela, campo-a-campo, e o uso de métodos de acesso (movimentos do mouse, teclas aceleradoras) Checar se os objetos e características da janela, tais como menus, tamanho, posição, estado e foco conformam-se aos padrões.
Técnica:	Criar ou modificar os testes para cada janela a fim de verificar a navegação e os estados de objeto adequados para cada janela e objetos da aplicação.
Critério de Finalização:	É verificado que cada janela permanece consistente com a versão de comparação ou dentro de padrões aceitáveis de usabilidade.
Considerações Especiais:	Todas as propriedades para objetos devem ser acessadas.

Tabela 4. Teste de interface

Ferramentas

Um conjunto de ferramentas empregadas em atividades deste projeto compreende (ver **Tabela 5**):

Atividade	Ferramenta
Gerenciamento de Teste	Rational RequisitePro
Projeto de Teste	Rational Rose
Gerenciamento de Projeto	Microsoft Project
Ferramentas do SGBD	MySQL Control Center

Tabela 5. Ferramentas

Quadro 3. Estratégias e ferramentas de teste

4. Equipe e infra-estrutura

Esta seção apresenta a equipe de testes, identificando-se os papéis e responsabilidades. Além disso, a infra-estrutura existente é listada.

Equipe de teste

A **Tabela 6** descreve um conjunto de papéis e respectivas responsabilidades na equipe de teste.

Papel	Responsabilidades
Gerente do Projeto	• Fornece orientação técnica • Adquire recursos necessários • Elabora relatórios de gerenciamento
Projetista de teste	• Identifica, prioriza, e implementa os casos de teste • Gera o plano de teste • Cria o modelo de teste • Avalia o esforço de teste
Testador	• Executa os testes • Registra os resultados • Documenta as solicitações de mudança
Implementador de testes	• Implementa e faz os testes unitários das classes e pacotes de teste • Cria as classes e pacotes de teste implementados no modelo de teste
Projetista	• Identifica e define as operações, atributos, e associações das classes de teste • Identifica e define as classes e pacotes de teste

Tabela 6 - Papéis e responsabilidades da equipe de projeto.

Infra-estrutura

Abaixo, é listado o conjunto de recursos disponíveis para este projeto.

• Servidor de Banco de Dados - MySQL DataBase Server

• Repositório de Testes - 4 PCs de Desenvolvimento de Teste

Quadro 4. Equipe e infra-estrutura

5. Cronograma

O cronograma da **Tabela 7** contempla as atividades de testes, marcos, e respectivas datas de início e término. Este quadro não ilustra qualquer dependência entre as atividades. Caso exista, isso pode ser apresentado através de um diagrama de Gantt usando Microsoft Project.

Milestone	Data de Início	Data de Término
Planejar Teste	03/06/09	05/06/09
Projetar Teste	08/06/09	10/06/09
Implementar Teste	12/06/09	17/06/09
Executar Teste	17/06/09	20/06/09
Avaliar Teste	22/06/09	23/06/09

Tabela 7. Cronograma

Quadro 5. Cronograma

Perceba, por exemplo, que a atividade de projeto, implementação e execução de teste poderia ser decomposta em subsistemas, caso você estiver tratando de um sistema de médio a grande porte.

A seção apresentada no **Quadro 6** trata de informações que são detalhadas em outros documentos.

6. Documentação complementar

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

1. Documento de Requisitos do Sistema Exemplo, Versão 00.01 22/05/2009.
2. Ata de Reunião – Levantamento de Requisitos do Módulo A do Sistema Exemplo, 12/05/2009.
3. Ata de Reunião – Levantamento de Requisitos do Módulo B do Sistema Exemplo, 13/05/2009.
4. Ata de Reunião – Levantamento de Requisitos do Módulo C do Sistema Exemplo, 14/05/2009.
5. Ata de Reunião – Validação de Requisitos do Sistema Exemplo, 15/05/2009.
6. Plano de Projeto do Sistema Exemplo.

Quadro 6. Documentação complementar

O conjunto de seções apresentados acima servem para ilustrar pontos importantes num plano de teste. Não houve aqui a intenção de ser completo, mas de informar quais itens deveriam compor o plano de teste, bem como a de ilustrar o conteúdo que pode ser encontrado nesse documento.

Conclusão

Um projeto compreende um conjunto de atividades inter-relacionadas com datas de início e fim, além de metas específicas. Dentre essas atividades, uma essencial é a de teste cujo objetivo é encontrar sistematicamente falhas. Para tanto, faz-se necessário elaborar um plano de teste que servirá como um guia das atividades de teste que compreende o projeto, implementação, execução e avaliação de testes.

Trata-se de um documento que o gerente de projeto usa com objetivo de coordenar as atividades de teste. Você deve entender que a visibilidade do processo (de desenvolvimento e, especificamente, de teste) é essencial. Tal visibilidade permite monitorar a qualidade obtida em cada etapa e programar cada atividade a ser executada. Esse processo de acompanhamento contínuo das atividades de teste é de fundamental importância para obtenção dos resultados em termos de qualidade, custo e tempo. Portanto, sem esse 'mapa', torna-se muito difícil realizar com sucesso as atividades de teste.

Links

Introdução ao Teste (baseado no RUP)
http://www.wthreex.com/rup/portugues/index.htm
Teste de software na Wikipedia
http://pt.wikipedia.org/wiki/Teste_de_software
Processo de teste de software (Datasus)
http://pts.datasus.gov.br
Risk and Requirements -based Testing artigo de James Bach, IEEE Computer
http://www.satisfice.com/articles/requirements_based_testing.pdf
Software Testing Brain
http://www.testingbrain.com/
Software Errors Cost U.S. Economy $59.5 Billion Annually
http://www.nist.gov/public_affairs/releases/n02-10.htm

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

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

Para mais informações:
www.devmedia.com.br/central

Linhas de Produtos de Software

Introdução, conceitos e desafios para sua adoção

Pasqueline Scaico
pasqueline@ccae.ufpb.br

Professora da Universidade Federal da Paraíba (UFPB) – Litoral Norte, pesquisadora na área de Linhas de Produtos de Software. Possui mais de nove anos de experiência com Engenharia de Software. Membro do grupo de pesquisa Computação Aplicada (Applied). Atualmente, coordena um projeto do CNPq que visa a elaboração de um método para identificação de famílias de produtos chamado YANA.

Alexandre Scaico
alexandre@ccae.ufpb.br

Doutor em Engenharia Elétrica e Professor da Universidade Federal da Paraíba (UFPB) – Litoral Norte. Pesquisador na área de Interface com o Usuário e Linhas de Produtos de Software. Já publicou mais de vinte trabalhos, alguns deles em periódicos de referência na área de Computação, como o Lecture Notes on Computer Science (LNCS) e o IEEE- Transaction of Society for Computer Simulation.

Fillipe Lourenço da Cunha Lima
fillipe.lourenco@gmail.com

Graduando em Licenciatura em Ciência da Computação pela Universidade Federal da Paraíba (UFPB) – Litoral Norte. Premiado no XI Encontro de Iniciação a Docência realizado pela PRG/UFPB em 2008. Trabalha no projeto YANA.

Não seria legal se para desenvolver uma aplicação de software precisássemos apenas escolher artefatos para que no final tivéssemos a aplicação funcionando? Um dos temas em maior evidência nos últimos anos é a Engenharia de Linhas de Produtos de Software (LPS), um paradigma para o desenvolvimento de aplicações, que traz como bandeira uma filosofia audaciosa para o conceito amplo de reuso. Aqui são apresentados conceitos introdutórios sobre este assunto e mencionados os desafios existentes nesse complexo universo. Neste artigo será tratado com mais ênfase a análise de domínio orientada a *features*.

A origem das Linhas de produtos

O conceito tradicional das linhas de produtos foi criado na indústria automobilística para substituir o modelo de fabricação existente na época no qual um produto era inteiramente produzido por uma única pessoa ou por um pequeno grupo delas. As linhas de produção introduziram o conceito de modularização, já que o trabalho poderia ser feito em "pedaços", e possibilitava que os produtos fossem desenvolvidos em um tempo menor devido ao paralelismo das tarefas.

De que se trata o artigo?

Linhas de Produtos de Software é um novo paradigma da Engenharia de Software voltado à extrema utilização do reuso para a construção de famílias de produtos. A ênfase do artigo é dada aos conceitos iniciais da abordagem e a análise de domínio de características, uma das etapas principais para o estabelecimento das plataformas de produção.

Para que serve?

As Linhas de Produtos permitem que se estabeleça a infra-estrutura necessária para que uma família de produtos, aplicações que fornecem serviços parecidos, contudo, diferenciados, seja constituída para a construção rápida das aplicações.

Em que situação o tema é útil?

Para empresas que atuam em um segmento específico de mercado e, dentro dele, querem oferecer produtos mais diferenciados de acordo com o perfil de consumidores que possuem.

Este modelo de fabricação permitiu que mais clientes fossem atendidos, mas teve que ser adaptado para permitir a diferenciação dos produtos, já que até então, todos os carros tinham a mesma aparência e serviam a um determinado propósito. Era preciso adequar o modelo de fabricação para que fosse possível oferecer produtos diferenciados com a mesma rapidez de produção. O conceito de customização em massa (**Figura 1**) representou a possibilidade de atender a produção de bens em larga escala com base em perfis de consumidores. As fábricas passaram a projetar automóveis levando em consideração os segmentos de clientes e suas necessidades, o que ocasionou versões diferentes de um mesmo modelo de carro. Surge, então, a idéia das famílias de produtos.

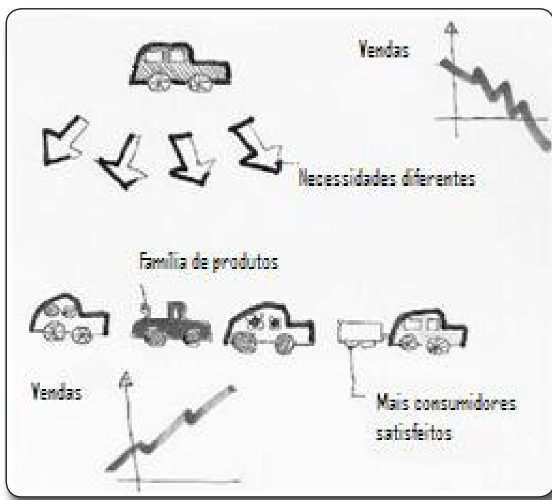


Figura 1. Customização em massa

Para atingir esse grau de customização, plataformas de produção foram estabelecidas e passaram a ser usadas como moldes para aqueles componentes que eram comuns à maioria dos produtos: chassis, motor e câmbio. Os componentes que eram diferentes, a exemplo da carroceria e itens de conforto, eram responsáveis por permitir as variações dentro da família de produtos. Logo, a maneira como o reuso das peças foi utilizado permitiu que se produzissem diversos modelos de automóveis.

Os primeiros conceitos de Linhas de Produto de Software (LPS) surgiram da observação dos processos industriais na década de 70, contudo, apenas nos anos 90 foram completamente introduzidos através das contribuições do projeto FODA (Feature-Oriented Domain Analysis) e das pesquisas realizadas pelo SEI (Software Engineering Institute) no desenvolvimento de aplicações para o governo. Uma LPS segue os mesmos princípios das linhas encontradas na indústria, sendo que constituem um conjunto de aplicações de software relacionadas entre si que oferecem funcionalidades parecidas e que satisfazem as diferentes necessidades de um determinado segmento ou missão de mercado.

Conceitos básicos

Normalmente os termos família de produtos e linha de produtos são citados quando se fala deste tema. Basicamente, as linhas de produtos se referem ao paradigma de desenvolvimento, enquanto as famílias de produtos representam o conjunto de sistemas, os quais são muito parecidos, mas que possuem características diferenciadas que atendem especificamente a um grupo de clientes de um setor do mercado, contudo, muitos autores as usam indistintamente. Estas características que diferem são chamadas de pontos de variação ou de variabilidades. São elas que ditam a versatilidade e (em muitas vezes, a complexidade) de uma linha de produtos.

Para entender a idéia por traz das linhas de produtos, imagine um software de segurança de ambientes. Dependendo do tipo de local, a aplicação deverá oferecer diferentes possibilidades para atender as necessidades dos usuários. Todo sistema de segurança possui um mecanismo de travamento de portas e de alarme para o caso de intrusão. Contudo, outro serviço que pode ser oferecido opcionalmente é o desligamento automático de luzes, que eventualmente ficaram acesas no fechamento das portas. Perceba que as características presentes apenas em algumas "versões" deste tipo de sistema são as chamadas variabilidades.

Numa abordagem de desenvolvimento convencional é muito provável que todos estes serviços sejam especificados e projetados em função de uma única aplicação que atenderá da mesma maneira (ou com certos ajustes) os diversos perfis de usuários. Em LPS, o paradigma para a concepção da solução é outro de forma que, a partir de um estudo sistêmico de todas as características (comuns e variáveis) do domínio em questão será montada uma plataforma de artefatos reutilizáveis que ao serem combinados, permitirão o desenvolvimento de várias instâncias diferentes do sistema, dedicadas, cada uma, a um grupo de clientes. Estes artefatos também são chamados de ativos. Nas próximas seções são apresentados mais detalhes sobre estes conceitos.

Muitos benefícios são decorrentes do uso de linhas de produtos. Podem ser citados como exemplo a diminuição do *time-to-market*, já que com o alto grau de reusabilidade dos artefatos a implementação decorre da junção de componentes; a redução dos custos; a melhoria na qualidade final do produto, já que a conduta de realizar os testes se torna mais rigorosa para que seja possível garantir a manutenção e evolução da plataforma de artefatos; a redução dos riscos já que, em geral, a empresa se apega um domínio de problema cuja a tendência é a de que com o tempo seja mais explorado e conhecido; e por fim, o posicionamento da empresa no mercado já que atuará com estratégias e abordagens mais refinadas e aprimoradas.

Contudo, quando se pensa em adotar o desenvolvimento de aplicações sob esta perspectiva, é necessário tomar ciência das mudanças que precisam ocorrer na organização. Há de ser considerado:

- Os investimentos iniciais para a adoção decorrentes da preparação de competências, dos treinamentos, da contratação de especialistas na área de negócios e da aquisição de infra-estrutura de desenvolvimento, entre outros;

- A necessidade de que os processos internos e organizacionais tenham um certo grau de maturidade para que seja possível identificar, controlar e continuá-los sem um gasto considerável de tempo;
- As mudanças organizacionais, já que os produtos dentro de uma família não podem mais ser vistos como independentes e, por isso, a equipe precisa ser reorganizada. Um time responsável pelo estabelecimento da linha de produção deve trabalhar em sintonia com a equipe desenvolvedora;
- Um novo esforço deve ser despendido para a elaboração da plataforma que começa com a padronização de processos, fluxos de trabalho, tecnologia e artefatos utilizados;
- Como será elaborada a plataforma da linha de produção, de maneira que uma estratégia conveniente seja estipulada para descobrir as características comuns da grande maioria dos produtos e depois as suas especificidades;
- A flexibilidade que os componentes devem ter para que possam ser adequados a diferentes sistemas para permitir o reuso: é importante que seja possível mapear os pontos em que os produtos diferem para que se determinem as variabilidades. Produtos que compartilham a mesma plataforma e apresentam características semelhantes pertencem a uma mesma família de produtos.

Portanto, o uso por conveniência de reuso não caracteriza a utilização de linhas de produtos. Este é o caso do uso de APIs, de frameworks ou de componentes. Como será observado adiante, começando pelo ciclo de vida, grandes mudanças acontecem para o uso desta abordagem.

Ciclo de vida das linhas de produtos

Normalmente, quando se fala em reuso o primeiro artefato a se pensar é o código. O aprimoramento de padrões arquiteturais, dos padrões de projeto e a construção de frameworks refletem esta preocupação. Em linhas de produtos, a dimensão que o reuso toma é muito maior porque se procura maximizar o reuso em todos os artefatos do ciclo de desenvolvimento, de forma que seja possível escolher pedaços de artefatos mais gerais para compor um produto da linha.

Assim, as etapas do ciclo de vida tradicional não são suficientes para este contexto. Ao contrário da abordagem tradicional em que se espera especificar requisitos de software, em LPS os esforços iniciais estão voltados para identificar as características dos produtos na linha, ou seja, para o entendimento do domínio do problema no qual a linha de produtos estará inserida. Este processo é chamado de Engenharia de Domínio. Como todos os artefatos serão reaproveitados ao longo dos projetos, um alto grau de flexibilidade precisa existir para que um mesmo artefato possa carregar informações úteis para outros artefatos e que seja de serventia para os múltiplos produtos da linha.

Na Engenharia de Domínio o desenvolvimento se dá para o reuso. Os artefatos que vão sendo criados deixam explícita a questão da variabilidade nos mesmos. Um exemplo é que o documento de requisitos pode conter uma descrição explícita de que alguns requisitos se aplicam apenas a um subconjunto de produtos, ou são opcionais a outros. Veja um exemplo na **Tabela 1**, onde em uma família de produtos para bibliotecas, o documento de requisitos poderia ser descrito como se segue. A característica comum na família de sistemas para bibliotecas é o empréstimo. Apenas alguns desses sistemas permitem a reserva dos exemplares, que pode ser feita com o pagamento de uma taxa de reserva **ou** não.

Requisitos
OBRIGATÓRIOS
OBR 1 – Empréstimo de livros
OPCIONAIS
OPC 1 – Reservas
ALTERNATIVAS
OPC 1 – ALT 1 – Reserva com pagamento de taxa
OPC 1 – ALT 2 – Reserva sem pagamento de taxa

Tabela 1. Trecho de especificação de requisitos em um LPS

Observando a **Figura 2** fica claro que quando se usa linhas de produtos não se tem a especificação de um sistema, e sim, a especificação de todas as características daquele domínio e para um conjunto de sistemas. Os artefatos da Engenharia de Domínio são especificados genericamente e contemplam as características do universo que está sendo explorado. A próxima seção deste artigo trata mais detalhadamente das atividades necessárias para a representação de domínio.

Figura 2. Engenharia de Domínio

Depois de representadas as características, a criação dos produtos tem início, os quais terão artefatos mais específicos. Esta etapa do ciclo chama-se Engenharia de Aplicação e tem como finalidade tirar proveito das variabilidades na linha, combinando as variações extraídas pela Engenharia de Domínio, para que produtos diferenciados sejam construídos. É neste processo que acontece o desenvolvimento com o reuso. A infra-estrutura da linha oferece a maioria das funcionalidades para o produto seja construído. Além disso, os demais artefatos são simplesmente instanciados a partir da base e encaixados uns com os outros para formar a nova aplicação (**Figura 3**).

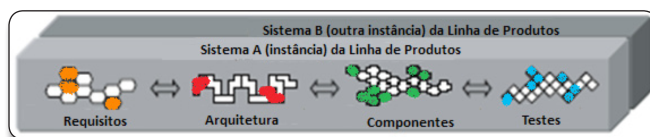


Figura 3. Engenharia de Aplicação

A idéia é que o documento de requisitos, a arquitetura, as classes de software e os testes que deverão ser utilizados, por exemplo, sejam simplesmente escolhidos da plataforma de artefatos especificadas pela Engenharia de Domínio.

Análise de Domínio Orientada a Features

Muitos métodos de análise do domínio existentes são extensões ou derivações do modelo proposto pelo FODA que indicou pioneiramente a importância da investigação sistêmica do ambiente no qual estarão inseridas aplicações, para a identificação das variáveis importantes no domínio em estudo. Uma *feature* é uma característica-chave distintiva de um produto, visível pelo usuário e que é relevante para os *stakeholders*. Cada produto da linha é definido a partir de uma seleção delas. São essas combinações que diferem uns produtos dos outros na linha. Em geral, a análise do domínio é uma atividade investigativa que visa delimitar o escopo da linha de produtos através da identificação e representação das *features*.

As características variáveis podem estar associadas ao tempo ou ao espaço. O primeiro caso acontece quando há mais de uma versão de um mesmo artefato em diferentes momentos (é comum que a evolução tecnológica cause este tipo de estado). Quando há diferentes versões que podem ser usados em um dado momento temos a presença da variabilidade no espaço. As *features* podem ser identificadas e classificadas em termos dos serviços que podem ser oferecidos, das tecnologias empregadas, do tipo de rede utilizado, do nível de segurança requerido, do tipo de interface, das técnicas de implementação aplicáveis, do hardware que será utilizado e do ambiente operacional onde o sistema será implantado, por exemplo.

Depois de identificadas, as *features* são representadas em um modelo hierárquico para que as relações e dependências sejam mais facilmente evidenciadas. Tais modelos são muito úteis no levantamento de informações com os *stakeholders*. Ainda não há uma linguagem padrão, como a UML (*Unified Modeling Language*), para modelar *features*. A definição de alternância ou exclusividade de uma *feature* em um modelo é o grande fator variante entre uma notação e outra. A primeira notação foi proposta pelo projeto FODA e é mostrada na **Figura 4**.

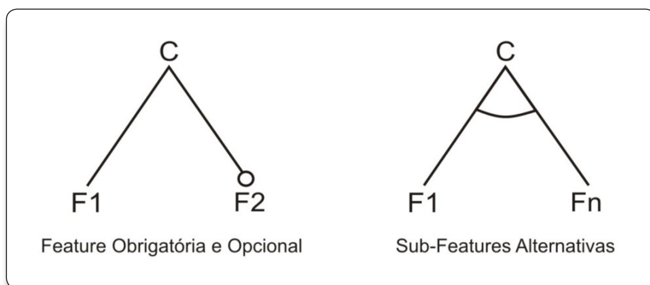


Figura 4. Notação Original FODA

A primeira derivação da Original FODA é a notação Czarnecki-Eisenecker a qual além de definir *features* obrigatórias e opcionais, alternativas e alternativas exclusivas, representa *sub-features* alternativas e opcionais. Em todos os nodos possui a indicação de obrigatória e opcional, seja em uma *feature* ou em uma sub-*feature*. Na **Figura 5** temos essa representação.

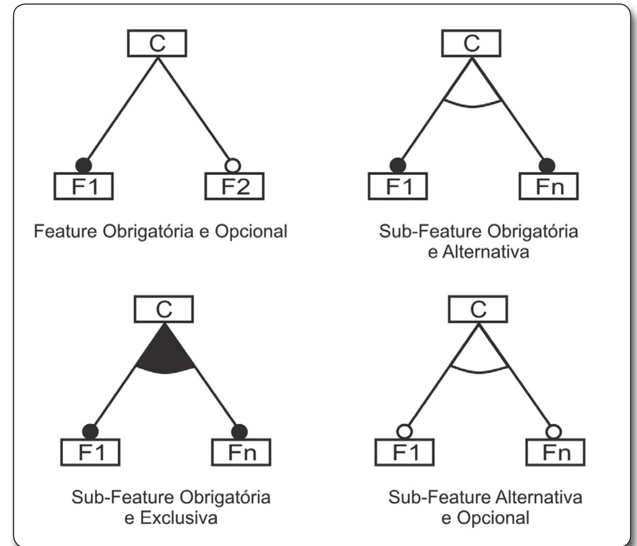


Figura 5. Notação Czarnecki-Eisenecker

Outra notação, a Extended Czarnecki-Eisenecker, como o nome já propõe, estende da Czarnecki-Eisenecker. Sua principal característica é a ausência dos indicadores de obrigatória e opcional em seus nodos de *sub-features*, em contrapartida utiliza-se a cardinalidade para representar tais relacionamentos (**Figura 6**).

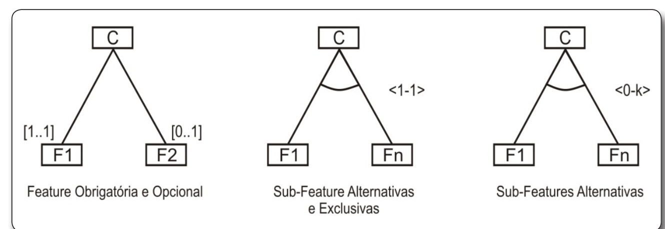


Figura 6. Notação Extended Czarnecki-Eisenecker

A notação FeatuRESB (**Figura 7**) segue todo padrão da Original FODA, sua divergência está na representação de sub-features onde classifica como pontos de variação dinâmico ou estático e obrigatório. Esta representação, porém já está superada pela notação Gurp-Bosch-Svahnberg (**Figura 8**).

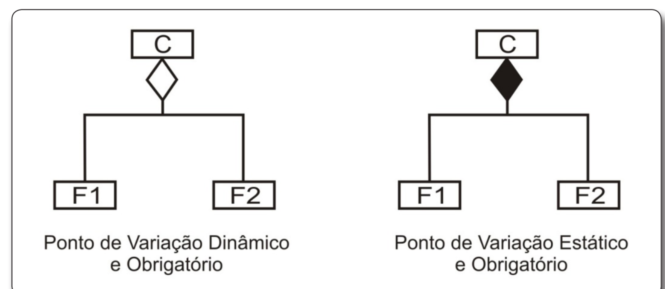


Figura 7. Notação FeatuRESB

Diferentemente da FeatuRESB, a notação Gurp-Bosch-Svahnberg volta a apresentar as *sub-features* como alternativa e alternativa exclusiva. Ela adiciona a questão dinâmica à

sub-fetures que possam ser adicionadas após atingir um certo estado de utilização do software ou uma ação que venha a adicionar uma funcionalidade ao produto depois de concluído.

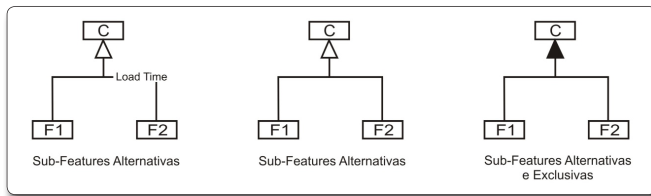


Figura 8. Notação Gurp-Bosch-Svahnberg

Como apresentado, embora possuam representações diferentes, as notações apresentadas são bastante parecidas quanto as suas características gráficas, pois todas derivam da notação FODA. Para exemplificar, um modelo de *feature* simplificado, que representa uma família de celulares, é apresentado na **Figura 9**. A notação utilizada é a de Czarnecki-Eisenecker. Percebe-se que há características comuns e específicas para diferentes celulares. Fazendo uma breve leitura, note que os círculos preenchidos indicam que todos os celulares da linha terão uma porta para entrada e saída de dados do sistema (1). É obrigatório escolher uma forma de interatividade que, de acordo com (2), pode ser por teclado, por uma tela sensível ao toque ou ambos. Todos os celulares terão um sistema operacional que é único, como indicado em (3), que pode ser um sistema específico, suportar o Windows CE ou Linux ME, mas nunca mais de um ao mesmo tempo. Nem todos os celulares têm suporte a cartão de memória (4). Pode-se observar que diferentes combinações das *features* demonstradas ocasionam versões diferentes de um celular da mesma marca. Dessa forma estando bem representado o domínio garante-se o reuso e uma produção em paralelo de forma eficaz.

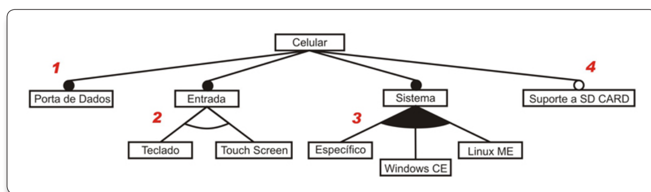


Figura 9. Exemplo de modelo de features para uma linha de produção de celular

A utilização de apoio ferramental para a criação dos modelos é muito importante. A ferramentas XFeature e Gears servem a este propósito. Alguns plug-ins para o IDE (Integrated Development Environment) Eclipse também podem ser utilizados como alternativa. É o caso do FeaturePlugin, fmp e o fmp2rsm e o pure::variant.

Aspectos econômicos

Um dos primeiros aspectos que precisa ser considerado na adoção é a maturidade da empresa em relação aos negócios. Enquanto as técnicas da Engenharia de Software voltadas para a melhoria de custos e qualidade possuem uma frágil conexão com as questões diretamente relacionadas ao negócio, as Linhas de Produtos influenciam naturalmente esta perspectiva. Quando se estabelece uma plataforma de produtos, a primeira coisa que se deve perceber é se existe um segmento de mercado a ser atacado, o que muitas vezes não está aflorado. O alinhamento das plataformas de produção com as estratégias de negócio podem conduzir a uma presença de mercado muito forte. Optar por implantar uma linha requer uma análise organizacional que responda a questões importantes para justificar o investimento necessário à adoção.

Em primeiro lugar, quais são as estratégias que a empresa usa para determinar como novos produtos serão criados? Diferentes abordagens existem e se baseiam em modelos orientados ao cliente (sob demanda), orientados ao fornecedor (de prateleira), orientados a tecnologia ou orientados a mercado (quando a empresa aposta na inovação e sempre busca as oportunidades para novos projetos). Dependendo do modelo que a empresa utiliza, o processo de adoção das linhas poderá ter um custo muito elevado. Este é o caso de contratos orientados ao cliente, que sugere que a organização não está atenta ao mercado em potencial, já que adota uma postura mais passiva diante dele esperando que os clientes iniciem os novos projetos.

Qual é a estratégia de mercado que a empresa utiliza para ser reconhecida no mercado? Existem três principais tipos de estratégias: liderança de preço (definem o preço mais baixo chegando a ficar próximo aos custos de produção), a diferenciação (oferecem serviços ou funcionalidades diferenciadas dos concorrentes) e o foco (especialista em um nicho de mercado).

Como é o ciclo de vida da linha de produtos no mercado? Para sistemas "individuais" os estágios são representados pela introdução do produto no mercado, crescimento (novos requisitos ainda são adicionados), maturidade (ápice do seu desempenho face aos clientes), saturação (o produto começa a não atender algumas demandas) e degeneração (abandono). Numa linha de produtos é preciso levar em conta o conjunto de produtos. Duas perspectivas são importantes: a variação e o tempo. Na primeira, onde existem variações de produtos na linha, pode ocorrer um processo conhecido como canibalização onde os sistemas competem entre si. Na variação do tempo, que geralmente ocorre quando as variações surgem em decorrência da mudança da tecnologia, pode resultar na renovação dos produtos da linha.

Uma linha de produtos é capaz de dar suporte às estratégias de mercado da empresa (liderança de preço ou inovação, por exemplo) porque considera o reuso como a base do processo de desenvolvimento, com isso é possível atingir resultados que favoreçam a redução dos custos do desenvolvimento, o aumento da qualidade, a redução do *time-to-market*, dos esforços com manutenção e da melhoria nas estimativas. Após avaliar se os aspectos relacionados ao negócio estão afinados com a filosofia da Engenharia de Linhas o processo de adoção pode passar para a próxima etapa.

Conclusão

Apesar de trazer muitos benefícios, a filosofia de Linhas de Produtos impõe muitos desafios. Em primeiro lugar porque as empresas de software precisam mudar a maneira de como elas enxergam os clientes, a sua postura de mercado e como

elas gerenciam o seu portfólio de produtos já que irão mudar o foco para uma visão de negócios mais ampla. Atualmente, muitas empresas ainda concentram uma parcela considerável de tempo, esforço e investimento na aquisição e aprendizado de tecnologias, deixando em segundo plano as estratégias para captação de novos projetos.

O paradigma de LPS oferece a oportunidade para que estas empresas reestruturem seus processos internos. Mas, o fato é que elas não conhecem esta abordagem. Além disso, considerando que ela é inerentemente complexa pela riqueza de informações que contém e que grande parte do conhecimento ainda está concentrado no meio acadêmico, é quase impossível considerar a implantação de LPS sem o auxílio de especialistas. Os custos relacionados aos investimentos necessários para adotar a estratégia, o tempo que durará a implantação e os efeitos decorrentes com o rompimento das práticas e condutas do modelo tradicional de desenvolvimento são fatores desmotivadores para a adoção.

Além disso, pouco consenso existe para o uso de terminologia e para a definição de padrões na área o que causa entrave nas discussões. Um grande desafio ainda a ser vencido está relacionado à modelagem da variabilidade das *features*. Apesar da modelagem de domínio ser bastante útil para documentar em que as instâncias dos produtos em uma linha diferem, é necessário propagar a variabilidade ao longo dos artefatos tradicionais no ciclo desenvolvimento. A variabilidade precisa ser representada em diferentes níveis de abstração, para que os usuários possam visualizar características específicas para alguns sistemas e finalmente, em mais baixo nível a equipe de desenvolvimento possa perceber os pontos de variação nos artefatos do projeto. Contudo, ainda não há uma abordagem unificada para este tipo de especificação.

Atualmente na UFPB, Campus Litoral Norte, está sendo desenvolvido um projeto que auxiliará as organizações que queiram usar a abordagem de LPS. Para isso, um método para identificação de famílias de produtos, chamado YANA, avaliará a organização e indicará se existem potenciais famílias a serem derivadas em função das aplicações que a empresa já desenvolve.

Links

Site sobre Linhas de Produtos de Software - Software Engineering Institute (SEI)
http://www.sei.cmu.edu/productlines/index.html

Monografia "Comparação Entre Ferramentas para Linha de Produtos de Software", escrita Rogério Aguiar de Lima Júnior
http://dsc.upe.br/~tcc/20081/RogeriomonografiaFinal.pdf

Livro "Generative Programming: Methods, Tools, and Applications. Addison-Wesley" escrito por Krysztof Czarnecki e Ulrich Eisenecker.
http://www.pdf-search-engine.com/generative-programming-methods-tools-and-applications--pdf.html

Relatório técnico "Feature-oriented domain analysis (FODA) feasibility study", escrito por Kyo C.Kang, Sholom G. Cohen, James A. Hess, William A. Novak, Spencer Peterson.http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/epmd/2008/bib/foda.pdf

Artigo "Integrating Feature Modeling with the RSEB" escrito por Giss, M. L. Favaro, J. d'Alessandro
http://www.favaro.net/john/home/publications/rseb.pdf

Artigo "Exploring the Commonality in Feature Modeling Notations" escrito por Miloslav Sipka.
http://www2.fiit.stuba.sk/iit-src/2005/22-sipka.pdf

Artigo "Managing Variability in Software Product Lines"escrito por Tommi Myllymäki
http://practise2.cs.tut.fi/pub/papers/VarMgnFinal.pdf

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

Cursos Online

A revista **Java Magazine** oferece para seus assinantes uma série de **Cursos Online** de alto padrão de qualidade .
Conheça abaixo o curso já disponível.

Curso em destaque

Introdução ao desenvolvimento para celulares com J2ME

Confira neste curso os principais recursos do J2ME. Aprenda também com o passo a passo para criar sua primeira aplicação J2ME. Neste curso você irá aprender diversas funcionalidades desta tecnologia para desenvolvimento de dispositivos móveis.

Confira o plano de aula completo:
www.devmedia.com.br/celularesj2me

Assine a **Java Magazine** e comece já seu treinamento!
www.devmedia.com.br/assine

A sua melhor opção de aprendizagem!

Outros cursos disponíveis: **www.devmedia.com.br/curso**

DevMedia GROUP **Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025**

Governança de Tecnologia de Informação

Uma Visão Integrada à Engenharia de Software

De que se trata o artigo?

Este artigo apresenta os principais conceitos e padrões relacionados à Governança de TI e oferece uma visão para integrar a Engenharia de Software a esse contexto.

Para que serve?

Fornecer conhecimento essencial sobre Governança de TI e o posicionamento da Engenharia de Software nesse contexto para organizações, pesquisadores, estudantes e profissionais de software.

Em que situação o tema é útil?

Organizações que realizam ou desejam realizar Governança de TI e desejam integrar suas práticas de Engenharia de Software a essa abordagem. Pesquisadores, estudantes e profissionais de TI que buscam entendimento sobre Governança de TI, bem como sua relação com Engenharia de Software.

Monalessa Perini Barcellos

monalessa@inf.ufes.br

É Doutoranda em Engenharia de Sistemas e Computação (COPPE/UFRJ), Mestre em Engenharia de Sistemas e Computação (COPPE/UFRJ), Bacharel em Ciência da Computação (UFES). Professora do Departamento de Informática, área Engenharia de Software, da Universidade Federal do Espírito Santo (UFES). Atuante desde 1999 em projetos, consultorias, treinamentos e pesquisas da área de Engenharia de Software.

Alex Sandro Barreto Rodrigues

asbrodrigues@yahoo.com.br

É profissional da área de Governança de TI, Master Business Administrator em Gerência de Projetos (FGV), Foundation Certificate in IT Service Management - ITIL, Professor da graduação e pós-graduação das Unidades de Negócio e de Sistemas da FAESA (Vitória - ES), Professor de pós-graduação da UCL (Faculdade do Centro Leste) (Vitória - ES), Criador e Coordenador do LIG (ITSMF) do Espírito Santo. Atuante desde 1997 em projetos, consultorias e treinamentos relacionados a Tecnologia da Informação.

Nos dias de hoje, é quase impossível pensar em uma organização que não faça uso da Tecnologia da Informação. Se há alguns anos a Tecnologia da Informação era privilégio das grandes empresas e centros de pesquisa, hoje ela pode ser considerada uma necessidade comum a praticamente todos os tipos de organizações, desde a 'vendinha do Seu Joaquim' até as grandes empresas multinacionais.

No início de sua utilização nas organizações, a Tecnologia da Informação teve sua aplicação focada principalmente no apoio à realização de processos operacionais. Com o passar do tempo e com as evoluções tecnológicas que ocorreram em ritmo acelerado, a Tecnologia da Informação passou a apoiar processos de todos os níveis e áreas das organizações.

Mas, o que levou as organizações a destinarem grande parte de seus orçamentos para investimentos em Tecnologia da Informação? Modismo? Definitivamente não.

A utilização da Tecnologia de Informação de maneira adequada mostrou-se como um acelerador do desempenho organizacional, o que, entre outros, contribuiu fortemente para o aumento da competitividade das organizações. Percebeu-se, então, que quanto melhor a Tecnologia de Informação fosse aplicada

e gerida, melhores seriam os resultados obtidos. Em outras palavras, enxergou-se na Tecnologia de Informação uma aliada fundamental para as organizações alcançarem seus objetivos de negócio.

Apesar dessa percepção, ainda é muito comum organizações investirem milhões em tecnologia e não obterem o retorno esperado. Ainda são comuns projetos de Tecnologia da Informação que estouram os prazos e custos e que não atendem aos requisitos de qualidade desejados. Ainda são comuns os serviços de Tecnologia da Informação prestados de forma insatisfatória.

Considerando esse cenário e a dependência cada vez maior dos negócios em relação à Tecnologia da Informação, torna-se necessária sua gestão de forma alinhada aos objetivos estratégicos das organizações. Nesse sentido, surge a Governança de Tecnologia da Informação, que trata basicamente do alinhamento das ações relacionadas à Tecnologia da Informação com os objetivos do negócio, a fim de que a utilização da Tecnologia da Informação seja capaz de agregar valor ao negócio como um todo.

E onde entra a Engenharia de Software? Para responder a essa questão basta lembrar que desenvolver e manter software são atividades relacionadas à Tecnologia da Informação e, no contexto da estrutura organizacional, normalmente a área de desenvolvimento de sistemas de uma organização é parte da área de Tecnologia de Informação. Ou seja, uma vez que o desenvolvimento e a manutenção de sistemas fazem parte da Tecnologia da Informação em uma organização, estes também são considerados na Governança de Tecnologia da Informação.

Para entender melhor a Governança de Tecnologia da Informação e o posicionamento das práticas de Engenharia de Software nesse contexto, nas seções seguintes são apresentados alguns conceitos e padrões relacionados ao tema para, então, ser possível localizar a Engenharia de Software no âmbito da Governança de Tecnologia de Informação.

As Organizações e a Tecnologia de Informação

Organizações são compostas por diferentes níveis e áreas funcionais. Apesar de existirem diversas nomenclaturas propostas por diferentes autores, comumente diz-se que as organizações são estruturadas em três níveis (nível estratégico, nível gerencial ou tático e nível operacional) e que esses níveis são 'atravessados' pelas áreas funcionais da organização (como vendas e marketing, recursos humanos, produção e finanças, por exemplo). À estrutura formada pelos níveis e áreas funcionais dá-se o nome de Arquitetura Organizacional, conforme mostra a **Figura 1**.

No nível operacional encontram-se os processos cuja execução garante o funcionamento diário da organização. Em um banco, por exemplo, depósitos e transferências são processos do nível operacional.

No nível gerencial (também chamado nível tático) encontram-se os processos relacionados à monitoração, controle, tomada de decisões e procedimentos administrativos. No exemplo do banco, a monitoração e controle do alcance das metas diárias são processos do nível gerencial.

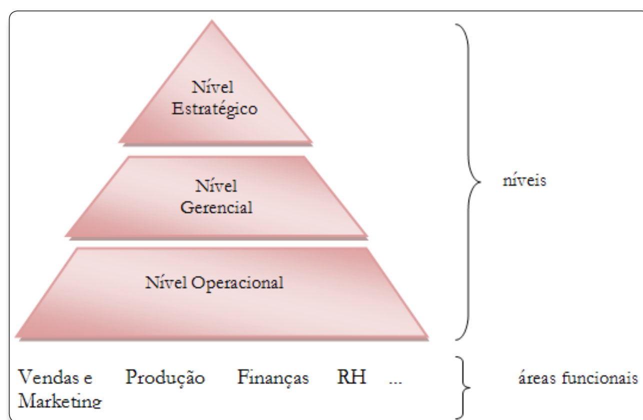


Figura 1. Arquitetura Organizacional.

No nível estratégico estão os processos que consideram uma visão mais ampla da organização, incluindo seu posicionamento em relação ao mercado. A análise de viabilidade para abertura de uma nova agência bancária ou lançamento de um novo produto é exemplo de um processo do nível estratégico.

Em cada nível, também estão as pessoas responsáveis pela execução dos processos: funcionários em geral, gerentes, diretores e executivos.

O funcionamento e a integração entre os processos e as pessoas que compõem a Arquitetura Organizacional podem ser descritas, macroscopicamente, da seguinte forma: no nível estratégico é definido o Planejamento Estratégico da organização, onde constam os objetivos de negócio, estabelecidos com foco na competitividade organizacional e que devem ser alcançados em períodos de tempo determinados (longo, médio e curto prazo). Para esses objetivos são definidos Planos de Ação, que são executados (em sua maior parte) no nível operacional, sendo monitorados e controlados pelo nível gerencial.

A palavra-chave que guia os processos do nível estratégico é competitividade. A questão básica é o que deve ser feito para que a organização seja sempre interessante para o mercado. Por outro lado, a palavra-chave que guia os processos do nível gerencial é sobrevivência. A questão básica passa a ser o que deve ser feito para que a organização atinja as metas estabelecidas e, assim, continue funcionando. Por fim, a palavra-chave que guia os processos do nível operacional é funcionamento, uma vez que o nível operacional é a 'água que move o moinho', ou seja, faz a organização funcionar.

A utilização da Tecnologia da Informação em uma organização deve ser realizada a fim de apoiar a execução dos processos da Arquitetura Organizacional. Ao integrar a Tecnologia da Informação à Arquitetura Organizacional, os processos da organização ficam sobre a estrutura de Tecnologia da Informação e quanto melhor apoiados forem os processos, maiores são as oportunidades de competitividade para a organização. A **Figura 2** apresenta uma evolução da **Figura 1**, evidenciando a relação entre a Arquitetura Organizacional e a estrutura de Tecnologia da Informação.

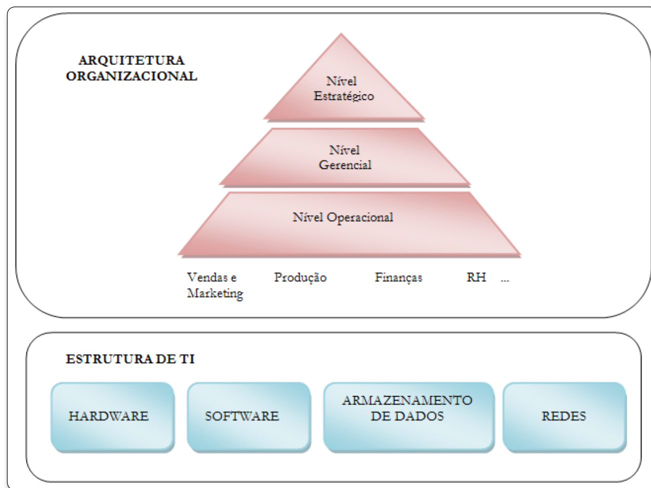


Figura 2. Arquitetura Organizacional e estrutura de TI.

Conforme ilustrado na **Figura 2**, a estrutura de Tecnologia da Informação refere-se à estrutura de hardware, software, armazenamento de dados e redes de comunicação que apóiam tecnologicamente os processos das organizações. Sendo assim, aplicativos de integração de processos e/ou tomada de decisão, bem como, sistemas operacionais, serviços de redes e correio eletrônico são partes integrantes da TI.

Vale destacar que a Governança de TI considera além da estrutura de TI que foi descrita, os recursos humanos envolvidos em seu desenvolvimento e operação.

Governança de TI

Apesar de seu destaque associado à Tecnologia de Informação ser relativamente recente, o termo governança não é novo para a Administração. A palavra governança significa ato ou efeito de governar. Governar, por sua vez, significa administrar, dirigir. No contexto da Administração, pode-se dizer, então, que de certa forma, a governança tem estado presente desde o final do século XIX, quando o foco era a gestão da produção (tratada principalmente por Taylor, Ford e Fayol), mantendo sua presença durante as evoluções da gestão, que culminaram no controle da qualidade e produtividade de produtos e processos.

No contexto da Tecnologia da Informação, o termo governança pode ser considerado novo, mas, como tudo o que é relacionado à tecnologia, vem evoluindo e ganhando espaço rapidamente.

A Governança de TI faz parte da Governança Corporativa, que surgiu nos Estados Unidos e na Inglaterra no final dos anos 90 e está relacionada à forma como as empresas são dirigidas e controladas. A Governança Corporativa é um conjunto de práticas de relacionamento entre acionistas, cotistas, conselho de administração, diretoria e conselho fiscal, que tem a finalidade de otimizar o desempenho da organização e facilitar o acesso ao capital.

A Governança de TI é um conjunto de práticas que visam à utilização e gestão da Tecnologia da Informação alinhada aos objetivos estratégicos e é de responsabilidade da alta administração (incluindo diretores e executivos de negócio e de TI), que deve atuar para garantir que a Tecnologia da Informação da organização seja capaz de sustentar e estender seus objetivos estratégicos, através do gerenciamento de serviços e produtos de TI de forma dinâmica e competitiva. Para isso, a Governança de TI possui cinco focos principais:

- Alinhamento Estratégico: busca integrar TI e negócios.
- Agregação de Valor: busca alcançar benefícios com custos otimizados.
- Gerenciamento de Recursos: busca otimizar os investimentos em recursos de TI, bem como a utilização desses recursos.
- Gerenciamento de Riscos: busca incorporar à TI análise e resposta aos riscos, bem como conformidade de processos.
- Mensuração de Desempenho: busca avaliar e divulgar o desempenho dos aspectos tratados pela TI.

A utilização da Governança de TI tem sido impulsionada por diversos fatores. Como já comentado neste artigo, a intensa competição mercadológica tem exigido que as organizações realizem grandes investimentos em TI, o que tem tornado o sucesso dos negócios cada vez mais dependente do sucesso da aplicação da Tecnologia da Informação. Consequentemente, as organizações têm buscado melhorar a gestão dos recursos destinados à TI, sendo necessárias práticas que assegurem que tanto os gestores quanto a estrutura de Tecnologia da Informação estejam adequados para agregar valor para a organização e que o capital destinado para a TI não será investido em ações de baixo valor estratégico.

Se por um lado, as próprias organizações têm percebido a necessidade de melhorar a gestão da Tecnologia da Informação, por outro, leis e regulamentações dos negócios têm imposto essa necessidade de melhoria. Empresas com ações na bolsa de valores norte-americana, por exemplo, são obrigadas a atender aos requisitos do Ato Sarbanes-Oxley. Bancos, por sua vez, precisam atender aos requisitos de gestão de riscos operacionais e de créditos previsto no Acordo da Basiléia II.

Uma maneira interessante de notar a importância e a responsabilidade da Governança de TI é relembrar alguns escândalos que ocorreram em empresas norte-americanas.

Em 2001, organizações como a Enron, a Tyco International e a WorldCom foram denunciadas por fraudes contábeis e fiscais, pois divulgaram lucros fictícios em seus relatórios financeiros, a fim de tornarem-se mais atraentes aos investidores de capitais, omitindo sua real situação financeira. A Enron, na época supostamente 7ª colocada no ranking da economia americana, faliu em dezembro de 2001 e a WorldCom demitiu cerca de 20.000 funcionários. Ambas, após vários processos e investigações, tiveram seus principais executivos presos e responsabilizados pelas fraudes.

Essa onda de escândalos trouxe à tona questões como transparência, ética nos negócios e conflitos de interesses que geraram desconfiança e instabilidade entre os investidores do mercado de capitais. Como resposta à onda de fraudes, em Julho de 2002, os senadores Paul Sarbanes e Michael Oxley formularam o Ato Sarbanes-Oxley que responsabiliza os executivos por estabelecer, avaliar e monitorar os controles

internos relacionados aos relatórios financeiros da organização. O objetivo principal do ato é proteger os investidores do mercado de capitais americano de fraudes contábeis e financeiras, bem como responsabilizar os envolvidos com uma série de penalidades.

Mas, o que isso tem a ver com Governança de TI?

Em síntese, o Ato Sarbanes-Oxley prevê controles internos sobre os relatórios financeiros das empresas. Os relatórios produzidos devem ser atestados pelos principais executivos da organização. E, normalmente, de onde vêm esses relatórios? Esses relatórios são resultado de várias transações que dependem de operações de TI, como sistemas de informação, acessos autorizados a redes de dados, armazenamento e recuperação de informações, entre outros. Ou seja, a figura do gestor de TI ou executivo de TI passa a ter uma conotação equivalente aos demais executivos da organização, uma vez que somente uma gestão transparente e alinhada aos objetivos de negócio da organização pode ser capaz de justificar os investimentos em TI e atestar a qualidade dos produtos desenvolvidos e serviços prestados.

Estrutura da Governança de TI

Tradicionalmente, a estrutura da Governança de TI é representada por uma figura que se assemelha a uma casa, conforme mostra a **Figura 3**.

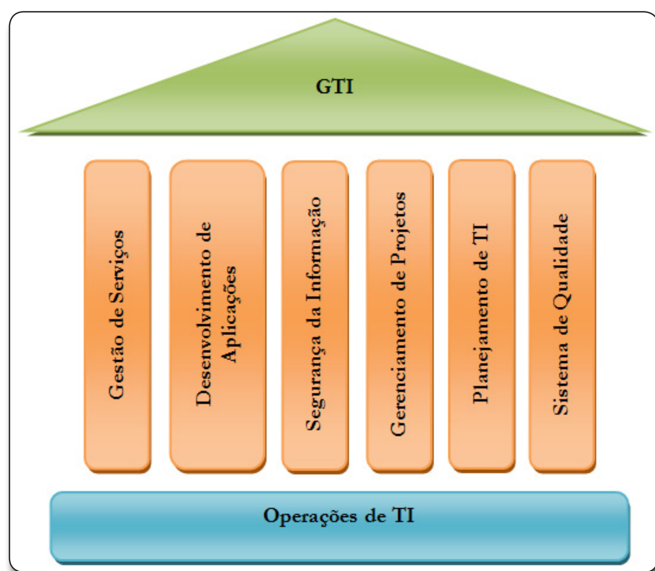


Figura 3. Estrutura da Governança de TI.

O telhado de uma casa, como se sabe, deve cobrir toda sua estrutura. Analogamente, no telhado da estrutura da Governança de TI encontram-se processos, práticas e diretrizes específicos da Governança de TI, ou seja, que definem o que deve ser feito para que a gestão de TI da organização seja alinhada aos seus objetivos estratégicos.

Sob o telhado estão os pilares da Tecnologia da Informação da organização, ou seja, todos os conjuntos de processos que estão relacionados à TI e são responsáveis por sustentá-la na organização. Conforme mencionado, no telhado é definido o que deve ser feito para que seja possível obter uma gestão de TI alinhada aos objetivos estratégicos. Nos pilares, por sua vez, o que é transformado em como, ou seja, as diretrizes fornecidas pela Governança de TI são incorporadas aos processos definidos em cada pilar.

Na base da casa, as operações de TI fornecem a fundação para que os processos definidos nos pilares possam ser corretamente realizados, produzindo resultados aderentes aos princípios da Governança de TI.

Apesar da estrutura de Governança de TI incluir vários componentes, nem sempre é necessário que uma organização implemente todos. O que define o que deve ou não ser incluso na Governança de TI em uma organização são suas necessidades, objetivos e recursos. Além disso, uma organização pode, também, optar por uma abordagem gradativa, trabalhando inicialmente nos pilares mais críticos e, aos poucos, inserindo os demais pilares.

Modelos, Normas e Melhores Práticas

Para apoiar a implementação da Governança de TI nas organizações, há no mercado, um conjunto de modelos, normas, práticas e frameworks que podem ser adotados, em conjunto ou não, para atender às demandas da Governança de TI. Esses modelos, normas, práticas e frameworks são recomendados, pois foram elaborados por órgãos específicos com grande conhecimento em suas respectivas áreas. Salvo algumas exceções, não faz sentido, por exemplo, reinventar a roda do gerenciamento de projetos, uma vez, que existe um guia de conhecimento mantido pelo PMI (Project Management Institute) bastante utilizado e aceito pelo mercado.

Na **Figura 4** os principais modelos, normas, práticas e frameworks são relacionados aos componentes da estrutura de Governança de TI que melhor apóiam. Em seguida é apresentada uma breve descrição de cada um.

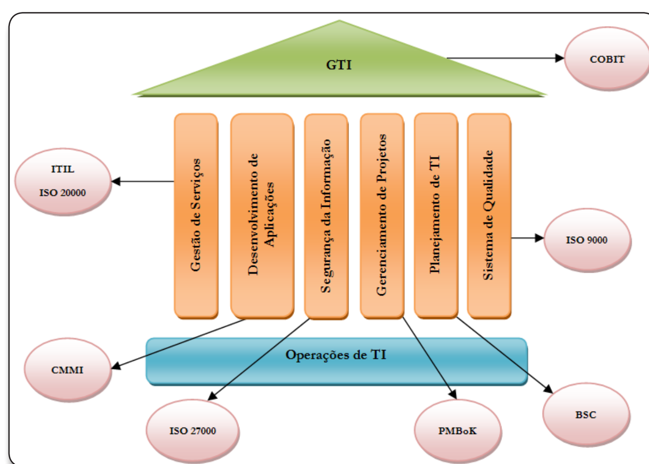


Figura 4. Frameworks de apoio à Governança de TI.

COBIT (Control Objectives for Information and Related Technology)

É considerado o modelo mais abrangente de Governança de TI. O Information Technology Governance Institute (ITGI) é atualmente o responsável pelo COBIT. É composto por 34

processos que estão organizados em quatro domínios que espelham os agrupamentos de TI usuais em uma organização: Planejamento e Organização; Aquisição e Implementação; Entrega e Suporte; e, Monitoração e Avaliação. As interações entre esses grupos de processos definem um ciclo de vida que contribui para o alinhamento da TI aos objetivos estratégicos da organização. COBIT foca o sucesso da entrega de produtos e serviços de TI, a partir da perspectiva das necessidades do negócio, com um foco mais acentuado no controle que na execução. Ou seja, preocupa-se mais com o que deve ser feito que como deve ser feito.

ITIL (Information Technology Infrastructure Library)

Descreve um conjunto de melhores práticas para gestão dos serviços de TI. Foi criada no final dos anos 80 pela CCTA (Central Computing and Telecommunications Agency) para o governo britânico e recebeu esse nome devido à quantidade de livros gerados para descrever as melhores práticas. Atualmente, a ITIL está em sua terceira versão e apresenta um framework para gerenciar o ciclo de vida dos serviços de TI. Inclui livros com melhores práticas para: definição e execução da estratégia de serviços, projeto e desenvolvimento de serviços, transição de serviços, operação de serviços e melhoria contínua dos serviços.

ISO/IEC 20000 – Information Technology Service Management

A norma ISO/IEC 20000, formulada a partir da BS 15000 (British Standards Instituition's Standard for IT Service Management), foi publicada em 2005 para responder às necessidades de entendimento comum sobre gerenciamento de serviços de TI. Caracteriza o gerenciamento de serviços de TI baseando-se nas melhores práticas reunidas na ITIL e promovendo a adoção de um processo integrado para a prestação e o gerenciamento eficaz dos serviços de TI que respondem aos requisitos do negócio e dos seus clientes. A norma está dividida em duas partes. A Parte 1 contém as especificações para o gerenciamento de serviços de TI e fornece os requisitos que devem ser cumpridos para obtenção da certificação ISO 20000. É relevante para os responsáveis pela preparação, implementação ou gerenciamento continuado dos serviços de TI em uma organização. A Parte 2 apresenta o código de prática e fornece orientação para auditores internos, bem como assistência aos prestadores de serviços que planejam melhorias.

CMMI (Capability Maturity Model Integration)

Criado pelo SEI (Software Engineering Institute) com o objetivo de ser um modelo integrado de práticas para apoiar a Engenharia de Software. O CMMI apóia a Governança de TI uma vez que guia a melhoria dos processos e habilidades organizacionais que cobrem o ciclo de vida de produtos e serviços.

Série ISO/IEC 27000 - Information Security Management Systems

A ISO/IEC 27001 trata do estabelecimento, implantação, operação, monitoração, revisão, manutenção e melhoria de um Sistema de Gestão da Segurança da Informação. Pode ser usada na avaliação da conformidade de um Sistema de Gestão da Segurança da Informação por partes interessadas internas e externas. A ISO/IEC 27002 estabelece os princípios gerais para iniciar, manter e melhorar a gestão da segurança da informação em uma organização, provendo, diretrizes sobre as metas geralmente aceitas nesse âmbito. Pode ser utilizada como um guia para elaborar os procedimentos de segurança da informação e práticas eficientes de gestão de segurança em uma organização.

PMBoK (Project Management Body of Knowledge)

É um documento que contém o conhecimento considerado relevante ao gerenciamento de projetos. É mantido pelo PMI (Project Management Institute), uma organização não governamental que trata das práticas do gerenciamento de projetos. O PMBoK define um conjunto de processos necessários ao gerenciamento de projetos, relacionando-os a nove áreas de conhecimento (escopo, tempo, custos, recursos humanos, qualidade, riscos, comunicações, aquisição e integração) e organizando-os em grupos de processos (iniciação, planejamento, execução, controle e encerramento) que compõem o ciclo da gerência de projetos.

BSC (Balanced Scorecard)

É um mecanismo que auxilia as organizações no alinhamento de suas ações em relação a seus objetivos estratégicos. Está organizado considerando quatro perspectivas do negócio: perspectiva financeira, perspectiva de processos internos, perspectiva de aprendizado e crescimento e perspectiva do cliente. A relação entre os objetivos do negócio e as perspectivas é demonstrada por meio de mapas estratégicos, que representam a criação de valor em uma organização. O BSC pode ser utilizado para apoiar as organizações no Planejamento Estratégico da TI, uma vez que permite desdobrar os objetivos estratégicos de TI em iniciativas que contribuam para os objetivos estratégicos da organização. Essas iniciativas podem ser realizadas em projetos que podem ser acompanhados, medidos e verificados até que sejam concluídos.

Série ISO 9000:2000 - Sistema de Gestão da Qualidade

É uma série de normas que utiliza princípios da gestão da qualidade total e tem como principal objetivo a geração de produtos e/ou serviços em conformidade com requisitos estabelecidos. O ciclo PDCA (Plan, Do, Check, Act) é usado na ISO 9000:2000 para tratar a melhoria contínua do Sistema de Gestão da Qualidade. Pode ser utilizada em conjunto com outras práticas para garantir a qualidade na entrega dos produtos/serviços. Vale ressaltar que o ciclo PDCA presente na ISO 9000:2000 também é utilizado em outras normas como, por exemplo, na ISO/IEC 20000 onde o foco é a melhoria contínua dos serviços de TI.

A Engenharia de Software na Governança de TI

Conhecidos os principais conceitos e a estrutura da Governança de TI, a relação entre Engenharia de Software e Governança de TI torna-se bastante clara. Não?

Analisando-se a **Figura 3**, é possível perceber que práticas de Engenharia de Software podem ser encontradas pelo menos nos pilares Desenvolvimento de Aplicações, Gerenciamento de Projetos e Sistema de Qualidade. Mas, exatamente onde?

Em Desenvolvimento de Aplicações são construídos os sistemas de informação necessários à organização. Para desenvolver software devem ser utilizadas práticas de Engenharia de Software. Além disso, é comum que o desenvolvimento de sistemas seja realizado por meio de projetos, e projetos devem ser gerenciados (no pilar Gerenciamento de Projetos da estrutura de Governança de TI). Para gerenciar projetos de desenvolvimento e/ou manutenção de software, as práticas de Engenharia de Software concernentes à gerência dos projetos precisam ser realizadas. Por fim, os softwares produzidos e os processos que os produzem precisam atender aos requisitos de qualidade de produto e processo de software estabelecidos, o que também envolve práticas da Engenharia de Software, como práticas de garantia da qualidade, por exemplo, que na estrutura de Governança de TI estão presentes no pilar Sistema de Qualidade.

Como argumentado no início deste artigo, o desenvolvimento e a manutenção de software são atividades relacionadas à Tecnologia da Informação. E, uma vez que no contexto da estrutura de Governança de TI, elas encontram-se sob os princípios dessa forma de gestão, é importante que suas práticas sejam condizentes com as demais práticas relacionadas a TI, guiadas pela Governança de TI. Em outras palavras, falar em Governança de TI envolve falar em Engenharia de Software. Planejar adequadamente a Governança de TI em uma organização exige que os aspectos relacionados à Engenharia de Software também sejam considerados e corretamente estruturados na estratégia de Governança de TI definida. Para isso, é necessário que os papéis, responsabilidades, entradas, saídas e interações sejam claramente estabelecidos ao longo de toda a estrutura de Governança de TI adotada.

Conclusão

A Governança de TI foi inicialmente impulsionada pela necessidade das organizações atenderem requisitos definidos em legislações específicas de seus segmentos. Percebidos os benefícios gerais alcançados, a Governança de TI passou a ser atraente para organizações em geral e vem conquistando cada vez mais seu espaço.

Na Governança de TI, os tradicionais gerentes de TI têm novas responsabilidades e uma nova visão organizacional é deles exigida, a fim de que se transformem em gestores de TI.

Ao contrário do que se imagina, gestores de TI, gerentes de projetos e engenheiros de software não são concorrentes na área de TI. Na verdade, estão todos no mesmo barco e, se querem chegar ao destino que representa o sucesso da organização como um todo, precisam remar de maneira sincronizada e na mesma direção. ●

Referências

FERNANDES, A. A., ABREU, V. F., 2008, "Implantando a Governança de TI: da Estratégia à Gestão dos Processos e Serviços", 2ª. Edição, Brasport, Rio de Janeiro.

LAUDON, K. C., LAUDON, J. P., 2004, "Sistemas de Informação Gerenciais – Administrando a Empresa Digital", 2ª. Edição, Pearson - Prentice Hall, São Paulo.

MAGALHÃES, I. L.; PINHEIRO, W. B., 2007, "Gerenciamento de Serviços de TI na Prática: Uma Abordagem com Base na ITIL", Novatec, São Paulo.

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

A EDIÇÃO QUE VOCÊ PRECISA ESTÁ ESGOTADA?

SEUS PROBLEMAS ACABARAM!!!

Seja um assinante Gold!

Com a assinatura Gold você já pode consultar online todos os artigos publicados na sua revista desde a edição nº 1.

Saiba Mais! Acesse:
www.devmedia.com.br/assgold

Para mais informações:
www.devmedia.com.br/central

Assinatura Gold

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